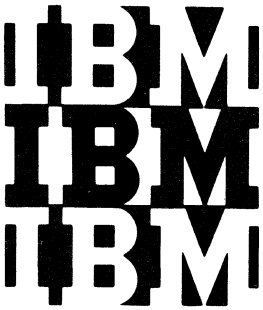


File Number S360-25
Form Z28-6620-0
Page Revised 3/15/66
By TNL Z31-5008-0

Program Logic

IBM System/360
Basic Programming Support
FORTRAN IV

Program Number 360P-FO-031

This manual provides information on the internal logic of the IBM System/360 Basic Programming Support FORTRAN system. The contents are intended for technical personnel who are responsible for analyzing system operations, diagnosing them, and/or adapting them for special usage.

Form Z28-6620-0
Page Revised 3/15/66
By TNL Z31-5008-0

PREFACE

Effective use of this Program Logic Manual (PLM) requires an understanding of the contents of the following manuals:

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

IBM System/360 Basic Programming Support FORTRAN IV, Form C28-6504

IBM System/360 Basic Programming Support FORTRAN Programmer's Guide, Form C28-6583

ORGANIZATION OF THE MANUAL

The manual is divided into five parts. The first part contains an introduction that describes the overall structure of the IBM System/360 Basic Programming Support FORTRAN IV system. This introduction is required reading for a basic understanding of the system. The second part describes the control segments for the system, while the remaining three parts reflect the three functions performed by the system.

Reference material for the PLM is contained in the appendices.

DEPTH OF DETAIL

This PLM provides a comprehensive understanding of the FORTRAN IV system down to the routine/subroutine level.

USING THE MANUAL

A user of this manual should read the introductory section to obtain an understanding of the overall structure of the system. From the material presented in that section, the user can determine the functions accomplished by the various segments of the system.

The introduction to each segment gives the overall logic of that segment, and indicates the routine/subroutines associated with the different functions of the segment.

Each routine/subroutine description within a given segment provides the user with a definition of the function and a description of the method employed to implement that function. A routine/subroutine description, when necessary, is accompanied by a corresponding flowchart. Where possible, a name for the associated portion of coding in the program listing is placed on an individual block in the flowchart. This name gives a direct relationship between the flowchart and the program listing.

In addition to flowcharts for routines/subroutines, flowcharts are provided at the introductory levels to supplement the discussion of concepts and overall logic.

Copies of this and other IBM publications can be obtained through IBM Branch Offices. A form has been provided at the back of this publication for readers' comments. If the form has been detached, comments may be directed to:

IBM Programming Publications, Rochester, Minnesota 55901

ILLUSTRATIONS

FIGURES

TABLES

CHARTS

PART 1: INTRODUCTION

This part contains a concise description of the Basic Programming Support FORTRAN IV system.

IBM System/360 Basic Programming Support FORTRAN IV operates independently of any other programming system. The system is comprised of segments that reside on a system tape. The segments are read into main storage and executed, depending on the function to be performed. The three system functions are:

1. Compilation.
2. Object-time execution.
3. System modification.

The segments that are always required, irrespective of the type of processing performed by the FORTRAN system, are the FORTRAN System Director and the Control Card routine.

In addition, the segments of the system used for compilation are Phases 10, 12, 14, 15, 20, 25, and 30; for object-time execution, the FORTRAN relocating loader and IBCOM; and for system modification, the editor.

Chart 00 represents the overall logic flow for the system and Figure 1 represents the input/output flow for the system.

SYSTEM INITIALIZATION

The system is initiated by operator action; pressing the IPL key. Thus, the operator causes the initial program load (IPL) to be read. IPL reads in the FORTRAN System Director from, and passes control to, the system.

FORTRAN SYSTEM DIRECTOR

The FORTRAN System Director (FSD) controls the various functions of the system. It remains in storage during compilation, object-time execution, and system modification. Initially, the FSD reads in the Control Card routine.

CONTROL CARD ROUTINE

The Control Card routine reads in control cards and determines, among other things, whether:

1. A source program is to be compiled.
2. An object program is to be executed.
3. The system is to be modified.
4. A combination of functions is to be performed (e.g., compile and execute).

SOURCE PROGRAM COMPILATION

Source programs written in the IBM System/360 Basic Programming Support FORTRAN IV language are compiled by the segments on the system tape that constitute the Basic Programming Support FORTRAN compiler.

The compiler segments are the FSD, the Control Card routine, and the seven phases (10, 12, 14, 15, 20, 25, and 30).

The FORTRAN compiler analyzes the source program statements and transforms them into an object program compatible to IBM System/360. In addition, if any source program errors exist, the FORTRAN compiler produces appropriate messages. At the user's option, a complete listing of the source program is produced and/or an object deck is punched.

FORTRAN SYSTEM DIRECTOR (COMPILATION)

The FORTRAN System Director performs the following functions during a compilation:

1. Handles the initialization required for a compilation.
2. Loads each phase of the compiler for execution.
3. Fills the input/output (I/O) requests of the various phases of the compiler.
4. Determines the point at which control is to be returned to a phase after an I/O request of that phase is filled.

Because a compilation is to be performed, the FSD reads in Phase 10 and passes control to it.

PHASE 10

Phase 10 reads in each statement of the source program and converts the statement (unless it is a COMMON or EQUIVALENCE

statement) into intermediate text which is used as input to subsequent phases of the compiler. To allow this intermediate text to be properly processed, certain information must be known about the symbols in the source statements. This information is maintained in a dictionary and an overflow table. For COMMON and EQUIVALENCE statements, Phase 10 produces another type of text which remains in storage to be processed by Phase 12.

Upon completion of Phase 10 processing, control returns to the FSD, which reads in and passes control to Phase 12.

PHASE 12

Phase 12 primarily allocates storage to symbols entered in the dictionary, overflow table, COMMON text, and EQUIVALENCE text. The storage allocated at this time dictates where the various symbols will reside in main storage during the execution of the object program. The main storage reserved for COMMON and EQUIVALENCE text is then made available for subsequent phases.

Phase 12 punches loader input cards for the object program and text cards for all constants used by the program, if the DECK option is specified. It writes these cards on the GO tape (a temporary tape containing any object program produced), if the GOGO or COMPILE and GO options are specified. If the MAP option is specified, all symbols and their relative addresses are printed as part of a storage map, as the addresses are being assigned.

Upon completion of Phase 12 processing, control returns to the FSD, which reads in and passes control to Phase 14.

PHASE 14

Phase 14 reads the intermediate text created by Phase 10 and replaces any pointers to dictionary information with information accessed from the dictionary. Phase 14 converts intermediate text for FORMAT statements to an internal code. At object-time execution, this internal code is used by the IBCOM routine (an object-time I/O control program) to place input and output records into the specified format. If requested, the code is written on the GO tape and/or punched on text cards.

The main storage reserved for the dictionary is then made available for subsequent phases.

Upon completion of Phase 14 processing, control returns to the FSD, which reads in and passes control to Phase 15.

PHASE 15

Phase 15 primarily translates arithmetic expressions into approximate machine code; that is, it produces the data necessary to allow the text word to be converted to a machine instruction by Phase 25.

Upon completion of Phase 15 processing, control returns to the FSD, which reads in and passes control to Phase 20.

PHASE 20

Phase 20 increases the efficiency of the object program by decreasing the amount of computation associated with subscript expressions. Phase 20, if requested via the DECK option, punches loader input cards for any required library exponentiation subprograms, for any references to IBCOM, and for literals that are generated during the phase in connection with array displacement.

Upon completion of Phase 20 processing, control returns to the FSD which, in turn, reads in and passes control to Phase 25 or 30 depending on whether:

1. The COMPILE and GO, GOGO, or NOGO option is specified.
2. Any source program errors are found.

If the GO option is specified and source program errors are found, the FSD passes control to Phase 30. If no source program errors are found, the FSD passes control to Phase 25.

If the GOGO option is specified, the FSD passes control to Phase 25, irrespective of whether source program errors are found.

If the NOGO option is specified and source program errors are found, the FSD passes control to Phase 30. If no source program errors are found, the FSD passes control to the Control Card routine.

PHASE 25

Phase 25 analyzes the text produced by the preceding phases of the compiler and transforms that text, wherever necessary,

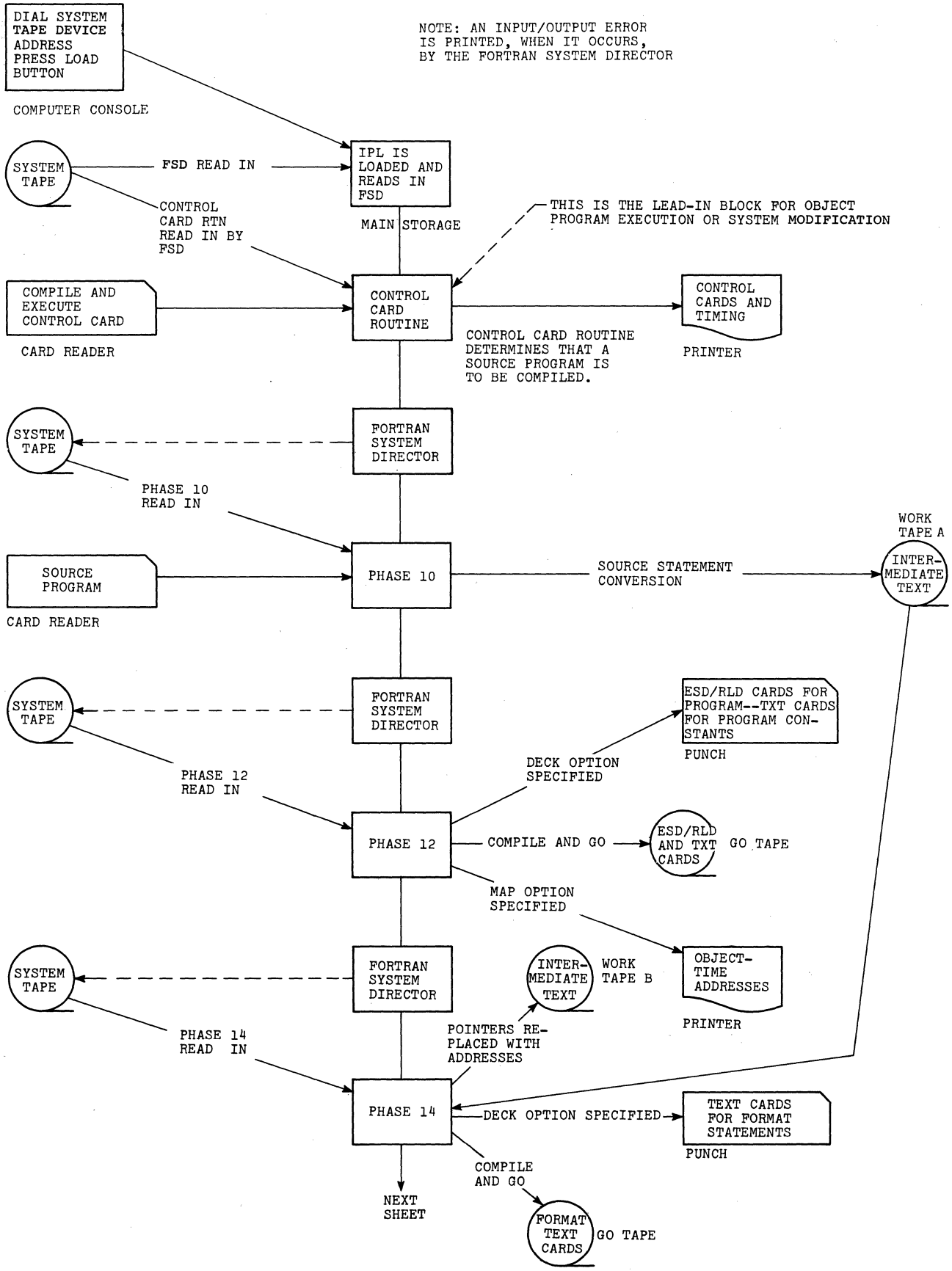
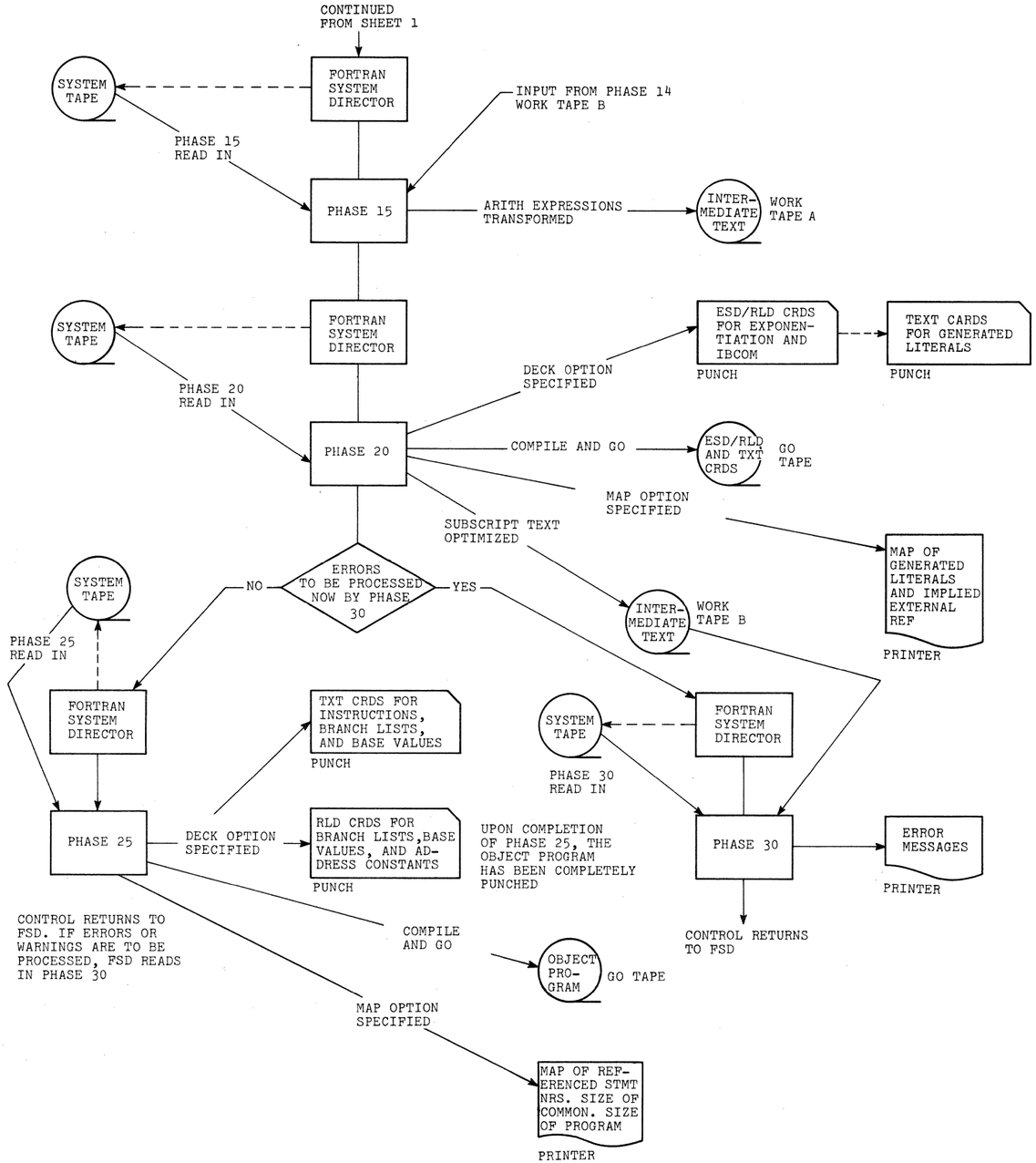
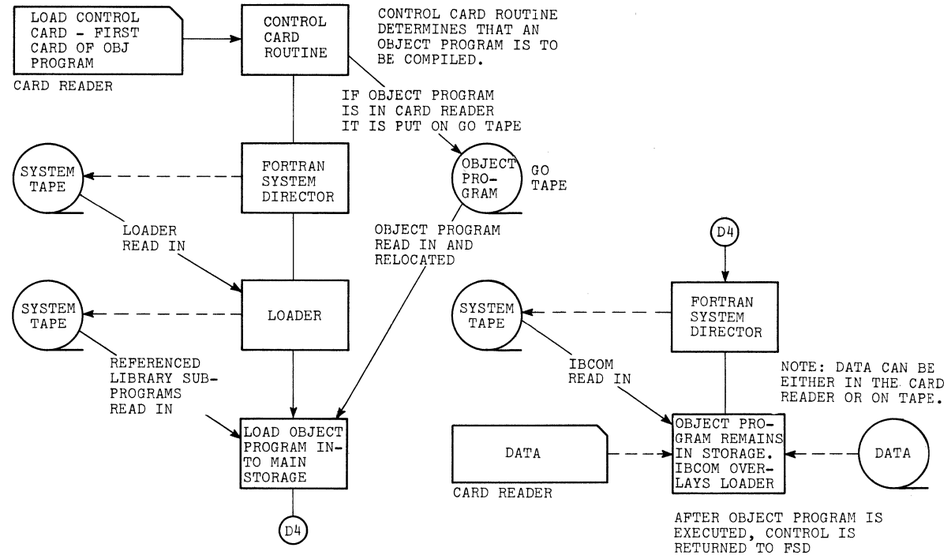


Figure 1. I/O Flow for IBM System/360 BPS FORTRAN (sheet 1 of 2)



OBJECT PROGRAM EXECUTION



SYSTEM MODIFICATION

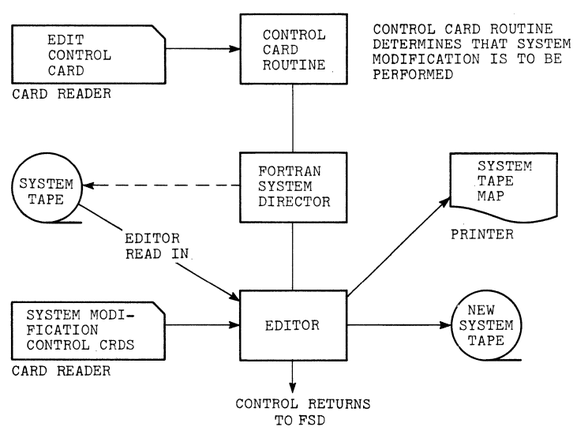


Figure 1. I/O Flow for IBM System/360 BPS FORTRAN (sheet 2 of 2)

into the desired object code. It assembles the entire transformed text into a card format that is acceptable to the Basic Programming Support FORTRAN loader. Thus, the output of Phase 25 (and the compiler) is an object program in the form of loader input cards.

Upon completion of Phase 25 processing, control returns to the FSD, which reads in and passes control to Phase 30 if source program errors are found. If no source program errors are found, control is passed to the Control Card routine.

PHASE 30

Phase 30 produces error and warning messages signalled by error/warning indicators set in the output text of any preceding phase.

If no error or warning conditions are encountered during the compilation, Phase 30 is bypassed. Upon completion of Phase 30 processing, control returns to the FSD.

COMPLETION OF COMPILATION

At the completion of a compilation, the FSD passes control to the Control Card routine to read in any additional cards for processing. If there are no additional cards (i.e., another source program to be compiled), the FSD either reads in the relocating loader and passes control to it, or displays an end of job message, and then goes into a wait status, depending on the option specified. If the GO or GOGO options are specified, control is passed to the loader. If the NOGO option is specified, an end of job message is displayed, and a wait status is entered.

OBJECT PROGRAM EXECUTION

An object program generated by the FORTRAN compiler is executed through the use of certain segments on the system tape. These segments are the FORTRAN System Director, the FORTRAN Relocating Loader, and the IBCOM routine.

FORTRAN SYSTEM DIRECTOR (EXECUTION)

The FORTRAN System Director performs the following functions during object-time execution:

1. Handles the initialization required for an execution.
2. Loads the FORTRAN loader into main storage.
3. Loads IBCOM into main storage after the FORTRAN loader performs its duties.
4. Fills the I/O requests of the FORTRAN loader and the IBCOM routine.

FORTRAN RELOCATING LOADER

The FORTRAN loader loads the main object program and any associated object subprograms into main storage from the GO tape (or from the card reader). In addition, it loads the required out-of-line subprograms from the library on the system tape. This produces a storage map of each object program that is loaded, if the MAP option is specified. Upon completion of the loading, control passes to the FSD.

IBCOM

After the FORTRAN loader has been used, the FSD loads the IBCOM routine from the system tape over the FORTRAN loader. The IBCOM routine serves as the hub of the FORTRAN input/output object code statements. It is used by the object program as an interface with the I/O routines in the FSD.

Although the I/O routines in the FSD perform the actual I/O operations, IBCOM sets up all required information. For example, IBCOM converts any data to be read or written by the FSD to its specified format. IBCOM remains in main storage until the conclusion of object-time execution.

COMPLETION OF EXECUTION

At the completion of object-time execution, control returns to the FSD from the object program.

SYSTEM MODIFICATION

The Basic Programming Support FORTRAN system may be tailored to fit the programming requirements of a particular installation.

The editor, a segment of the FORTRAN system, enables the user to revise one or more segments of the system tape. This revision (the addition, replacement, or deletion of features as desired) is accomplished through the use of control cards (also referred to as control statements).

FORTRAN SYSTEM DIRECTOR (MODIFICATION)

The FORTRAN System Director performs the following functions during a system modification:

1. Handles the initialization required for a system modification.
2. Loads the editor into main storage.
3. Fills the I/O requests of the editor in reading in the segments of the system to be modified and writing out the modified segments on the new system tape.

EDITOR

After the FSD loads the editor into main storage, the editor reads in the system maintenance control cards (and any object decks associated with them) and modifies each segment of the FORTRAN system as specified. The editor has control throughout the editing process. The editing process ends when there are no more control cards to be read or when the editor encounters a control card indicating that no more editing is to be done. Control is then returned to the FSD.

COMPLETION OF SYSTEM MODIFICATION

At the completion of system modification, control returns to the FSD from the editor. If there is additional processing to be performed in the job (e.g., compiling a source program using the new system tape), the FSD gives control to the Control Card routine. Otherwise, the FSD enters a wait status.

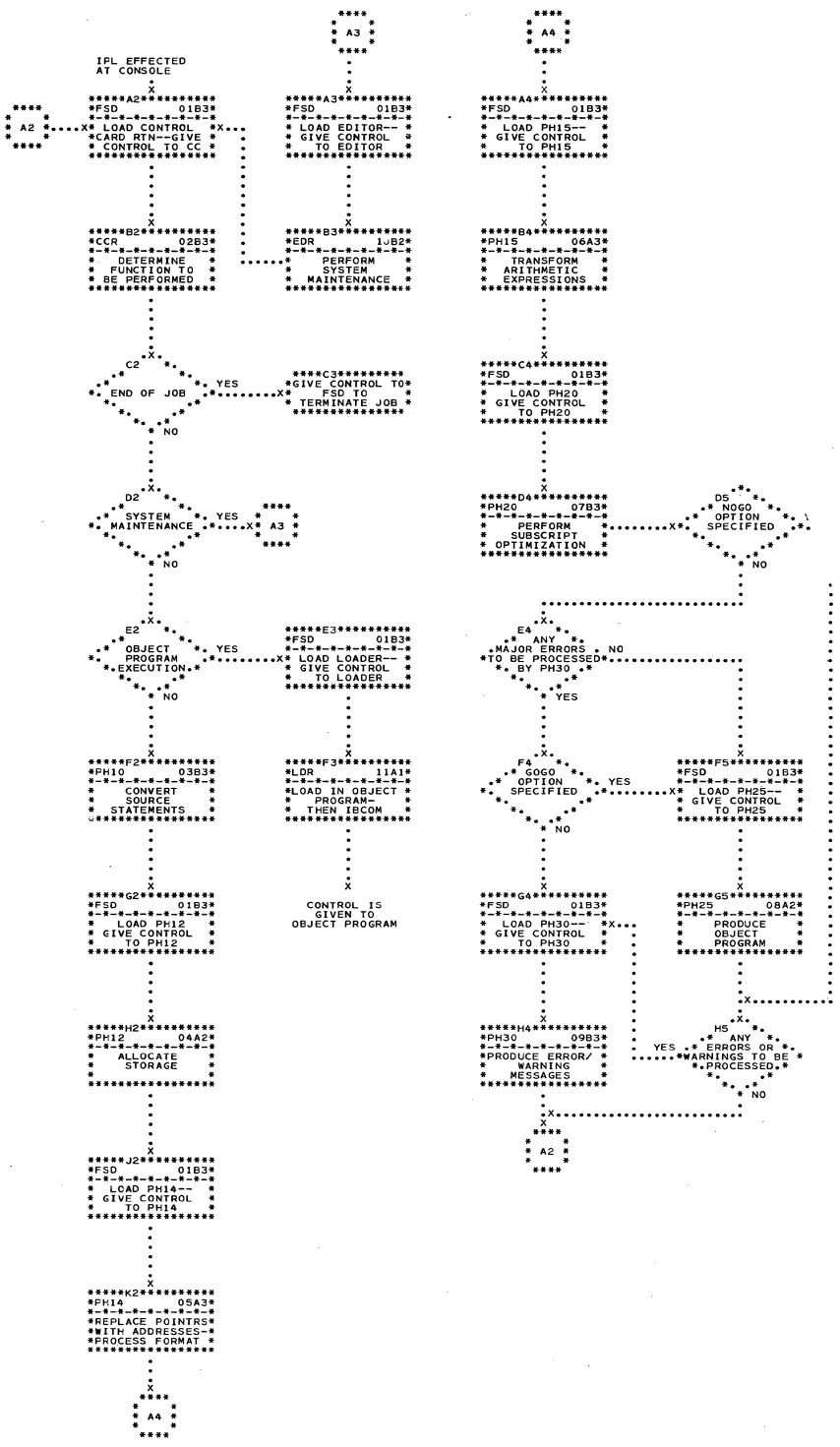


Chart 00. FORTRAN System Overall Logic Diagram

Control of the various functions of the Basic Programming Support (BPS) FORTRAN IV system resides within the FORTRAN System Director (FSD). During the system functions (compilation, object-time execution, and system modification), the FSD remains in storage.

Initially, the FSD reads in the Control Card routine to determine which system function is to be performed.

The FORTRAN System Director (FSD) controls the functions of the FORTRAN system. The FSD remains in storage during compilation, object-time execution, and/or system modification.

All communication between the various segments of the system and the FSD is by supervisor call (SVC) instructions. An SVC instruction requests the FSD to perform a certain operation. One SVC instruction is reserved for the I/O operations of the FSD. (These operations include such things as reading tape, writing tape, printing, and punching.) Loading of the various segments is also initiated by an SVC instruction.

A communications area exists within the FSD. This area serves as a central gathering point for common information. The contents of the communications area are specified in the program listing supplied by IBM for the FSD.

Chart 01, the FSD Overall Logic Diagram, indicates the entrance to and exit from the FSD and is a guide to the overall functions of the FSD.

I/O OPERATIONS

The FORTRAN System Director (FSD) transfers control to the I/O routines whenever an SVC instruction, requesting an I/O operation, is encountered. The I/O operations are explained in accordance with:

1. The functions supported.
2. SVC I/O formats.
3. Data set designation.
4. Return to the user's program.

In general, the flow within the I/O routines begins with an SVC instruction. The co-ordination of I/O devices and functions is controlled by the device assignment (referred to as a unit table on the program listing provided by IBM for the I/O routines). The I/O routines set up for (SIODIR) and initiate (SIOGO) all the I/O operations. They handle all I/O interrupts (SNTPIN), provide for tape read and/or write retry procedures (SRETRY), and under certain conditions allow error recovery procedures (SERP). The routines provide the initial location of the device assignment table if it is not already known (SD1), set up for a set mode operation code and CAW (SETMD), and set up for a check operation (SD2). They set up simple control operations (SD5), data operations (SD7), and print operations (SD74, SD741).

The I/O routines construct a model for the current call. This model consists of 12 bytes that contain all the information necessary to process the current call. After the model is fully developed, it contains the CCW for the current I/O operation.

I/O FUNCTIONS

The BPS FORTRAN I/O routines support the functions defined in Figure 2.

FUNCTION	EXPLANATION & CONSIDERATIONS
Write	This operation provides an output facility for areas that are to be handled as data. Modifiers are specified in the call parameters.
Read	This operation provides an input facility. Modifiers are specified in the call parameters.
Control 3	This facility is for operations not involving read or write, such as immediate space, stacker select, set mode, etc. (operation code 011).
Control 7	This facility is for operations not involving read or write, such as rewind, space, write tape mark, etc. (operation code 111).

Figure 2. I/O Functions (continued)

(continued)

FUNCTION	EXPLANATION & CONSIDERATIONS
Print	This operation provides a facility for areas intended for graphic material. It is similar to the write operation except that it allows carriage control specification for off-line work (as determined by the class of device). The carriage control character is located in the first byte of the data area. When the call is to a graphic device, this character controls the insertion of the appropriate System/360 modifiers. When a call is to a non-graphic device, the control character located in the first byte is written out as the first data byte and the modifiers specified in the call parameter are used.
Check (Wait)	This operation provides the facility to examine a designated unit for a busy condition, waiting if the unit is busy, and interrogating the result. (This function is automatically included in WAIT calls.) Return is made to Normal return, Unit Exceptional Condition return, or Error return, as indicated in a data set control block within the device assignment table and determined by current conditions. If no operation has been initiated on the designated unit since the last check of the unit, direct normal return is made. (See "Return to User's Program.")
Note: The print facility is divided into two subfunctions, PRINT A and PRINT B (see Figures 12 and 13 respectively).	

Figure 2. I/O Functions

SVC I/O FORMATS

The operation and data set desired by the user is specified in an SVC instruction. One SVC format is used as the basis of all I/O calls; however, additional I/O routine capabilities can be introduced by parameters contained in an expansion of the basic format. These capabilities include the use of modifiers to the I/O command operation code, specification of data parameters in indicated registers (rather than in the data set control block), and temporary cancellation of overlapped operation on the data set designated by the call. The structure of the SVC I/O formats is defined in Figure 3.

NAME	FORMAT	BYTES	EXPLANATION
Base Format			This format is used for all simple data operations, that is for operations that do not involve command operation modifiers; this format is used for Check calls.*
	SVC I/O	2	Specify the type of SVC.
	T DS	1	Tag and data set. Bits 4-7 give the data set reference number (0 through 15).
	SPEC	1	All I/O functions are defined for the routines in this byte. See Figure 4 for a definition of this field.
	Start of Return		See "Return to User's Program" for a definition of this field.
Expansion A			This format is used for Control 3, Control 7, and for any operations requiring command modifiers.
	SVC I/O	2	Same as for the base format.
	T DS	1	Same as for the base format.

Figure 3. SVC I/O Formats (continued)

(continued)

NAME	FORMAT	BYTES	EXPLANATION
	SPEC	1	Same as for the base format.
	MODS	1	This byte supplies the command modifiers for the current operation.
	A B	1	A and B are any pair of registers containing the buffer address (in A) and the byte count (in B). Both fields must be supplied if either is supplied, and register contents will replace current data parameters in the data set control block. An A,B of 0 indicates that the current data parameters are to be used.
	Start of Return		See "Return to User's Program" for a definition of this field.

* Data parameters (address of buffer area and byte count) must exist in the UCB.

Figure 3. SVC I/O Formats

A detailed discussion of those fields of the SVC formats peculiar to the BPS FORTRAN I/O routines is presented in the following sections.

	BITS	HEXA-DECIMAL	SIGNIFICANCE
FLAGS (bits 0-3)	0000	0	(Reserved)
	0001	1	Wait on this operation. This flag may be combined with any other flag.
	0010	2	Disregard incorrect length indication (ILI) now. May be combined with any other flag.
	0100	4	Use data group now.
	1100	C	(Illegal)
	1101	D	(Illegal)
	1110	E	(Illegal)
	1111	F	(Illegal)
OPERATIONS (bits 4-7)	0000	0	(Reserved)
	0001	1	Write (data)
	0010	2	Read
	0011	3	Control 3
	0100	4	(Reserved)
	0101	5	(Reserved)
	0110	6	(Reserved)
	0111	7	Control 7
	1000	8	(Reserved)
	1001	9	PRINTA (write graphic data)
	1010	A	(Reserved)
	1011	B	(Reserved)
	1100	C	(Reserved)
	1101	D	PRINTB (write graphic data)
	1110	E	(Reserved)
	1111	F	Check

Figure 4. Contents of the Specifier Byte

Operation Specification

Figure 3 indicates that all I/O functions are defined for the routines in the specifier byte. This byte is structured as follows: bits 0-3 are used for flags; bits 4-7 specify the operation. Figure 4 defines the contents of this byte.

Tag and Data Set Byte

The tag and data set byte indicates whether modifiers and/or the use of registers for data parameters are present in the current call; it provides the unit reference number. Figure 5 defines the contents of this byte.

BITS	HEXA-DECIMAL	SIGNIFICANCE
0000	0	(Reserved)
0001	1	(Reserved)
0010	2	Indicates whether or not modifers are supplied and/or data parameters for the data group are contained in registers (Expansion A).
0011	3	(Reserved)
0100 thru 0111	4 thru 7	Contain the data set reference numbers (0 through 15).

Figure 5. Contents of Tag and Data Set Byte

DATA SET DESIGNATION

The correlation of I/O devices and functions is controlled through the use of the device assignment table (DAT). This table is comprised of two sections: data set table (DSTAB) and data set control blocks (DSCB). In the program listing provided by IBM for the I/O routines, DSTAB is referred to as UTAB and DSCB as UCB.

DSTAB -- Data Set Table

DSTAB is an open end list, referenced from the I/O routines, and composed of one 6-byte block for each data set. The initial entry in DSTAB is a 4-byte header block; the last entry is a 2-byte message data set identity block (see Figure 6).

Each 6-byte block holds one assigned physical device address, an amount representing the byte offset of the associated DSCB from the head DSCB, and the device type identification in hexadecimal digits (3 bytes).

DATA SET REFERENCE NUMBER	NAME	STRUCTURE			BYTES
	DSTAB	n+1 | Address of DSCB0			4
0	DSTAB0	Device 0 Address	Offset 0	Type	6
1	DSTAB1	Device 1 Address	Offset 1	Type	6
.					
.					
.					
n	DSTABn	Device n Address	Offset n	Type	6
m	$DSTAB_n$+6	Data Set | Dev Addr			2

NOTE: The "Type" field is further illustrated as follows:

X	X	X	X	D	M

Where: XXXX is four hexadecimal digits defining the type of devices, such as 2400 for a 2400 series tape.

D is one hexadecimal digit for a service type subclass, such as 24009 for a 9-track tape.

M is reserved.

Figure 6. Data Set Table Format

The 4-byte header block holds the number of data sets in DAT (1 byte) and the location of the start of the DSCB section (3 bytes). The 2-byte message data set identity block holds the data set reference number in the high order 5 bits and the device address in the low order 11 bits. Bit position 1 is reserved.

DSTAB is arranged in sequence according to data set reference number, 0 through n, and is so referenced by any SVC instruction requesting an I/O operation.

DSCB -- Data Set Control Block

Each entry in DSTAB requires an associated DSCB. The DSCB can vary in size from a minimum of 22 bytes to a maximum of 44 bytes.

The DSCB describes the associated data set (identified with the physical device address in DSTAB) and the extent of operations to be performed on that device. The DSCB also provides space for retaining any history requisite to the progress or control of those operations being performed on the device. The DSCB may also provide the optional capabilities of overlapped operation recognition, and separate indication (return) for unit exceptional condition, and/or error conditions.

Figure 7 presents a general description of the contents of the DSCB. Discussion of fields that require further explanation are presented immediately following the figure.

DEVICE CODE BYTES: The bit configurations of these bytes are as follows: Bits 0-4 contain the set mode modifier pattern for 7-track tape: ddmmm. Bits 5-7 contain the expansion code for this unit: 001, Expansion B. Bits 8-14 specify the device code as follows:

Bit 8 Tape
Bit 9 Printer
Bit 10 Punch
Bit 11 Reader
Bits 12-14 are used for a subclass of one of the above unit types.

Bit 15 contains the multiplex mode flag.

Figure 8 illustrates the device code assignment.

	NAME	BYTES	EXPANSION	TOTAL	CONTENTS
(DSCB0)	Device Code	2			Multiplex mode flag, device code, expansion code, set mode modifier pattern for 7-track tape.
+2	Flags	2			Extent of operations to be performed.
+4	Specifier	2			The contents of these bytes is the same as the SVC specifier byte.
+6	CCW	8			
+14	Check	1			DSCB check byte.
+15	Byte Count 1	2			This is the byte count for data group entries.
+17	Buffer Address 1	3			This is the buffer address for data group entries.
+20	Error	2	A 22	22	Error mask bytes.
+22	Sense Bytes	6			Note: Minimum requirements.
+28	I/O Old PSW	8			
+36	CSW	8	B 22	44	Note: Requirement for overlapped operation.

Figure 7. DSCB Format

DEVICE	MODE	BITS 8-14	BIT 15	HEXA-DECIMAL CODING
1052 Printer	Multi-plex	0001 001	1	13
1402 Reader	Multi-plex	0010 000	1	21
1402 Punch	Multi-plex	0010 001	1	23
1442 Reader	Multi-plex	0011 000	1	31
1442 Punch	Multi-plex	0011 001	1	33
1443 Printer	Multi-plex	0100 001	1	43
1403 Printer	Multi-plex	0100 011	1	47
2400 9-Track Tape Read	Burst	1000 000	0	80
2400 9-Track Tape Write	Burst	1000 001	0	82
2400 7-Track Tape Read	Burst	1000 010	0	84
2400 7-Track Tape Write	Burst	1000 011	0	86

Figure 8. DSCB Device Code Assignment

DSCB FLAG BYTES: Figure 9 illustrates the structure of the DSCB flag bytes.

BYTE	BIT	SIGNIFICANCE
$DSCB_n$+2	0	Operation not checked; last operation not yet interrogated.
	1	Wait-Check
	2	Reserved
	3	Chaining Flag
	4	Retry complete; all retries resulted in failure.
	5	Reserved
	6	Reserved
	7	Reserved

Figure 9. DSCB Flag Bytes (continued)

(continued)

BYTE	BIT	SIGNIFICANCE
$DSCB_n$+3	8	Reserved
	9	No overlap: zero overlap permissible and requires expansion B.
	10	SILI: disregard all incorrect length indications from this unit.
	11	Reserved
	12	Reserved
	13	Reserved
	14	Unit exceptional condition return; user will accept unit exception return.
	15	Error return; user will accept error return.

Figure 9. DSCB Flag Bytes

DSCB CHECK BYTE: The bit configuration of the DSCB check byte is illustrated in Figure 10.

BYTE	BIT	SIGNIFICANCE
$DSCB_n$+14	0	Program control interrupt (PCI)
	1	Attention
	2	Incorrect length record
	3	Error
	4	Exceptional condition
	5	Status report applies to the previous call
	6	Reserved
	7	Busy; current operation has not received device end, reject, error, or exceptional condition

Figure 10. DSCB Check Byte

ERROR MASK BYTES: The significance of the error mask bytes is explained in Figure 11.

BYTE	BIT	SIGNIFICANCE
$DSCB_n$+20	0-3	Second level retry count
	4-7	First level retry count
	8-9	Reserved
	10	Previous read error
	11	Not first entry
	12-15	Reserved

Figure 11. Error Mask Bytes

CALLS TC A PRINTER

Due to the peculiarities of a FORTRAN print command (which is actually a write graphic data command) the user should be familiar with the following material on FORTRAN printer carriage control characters and data parameters for print calls.

FORTRAN PRINTER CARRIAGE CONTROL CHARACTERS

Figures 12 and 13 define the carriage control characters and their effect. PRINTA writes after performing the indicated carriage function (see Figure 12); PRINTB writes before the carriage function is performed (see Figure 13).

CHARACTER	EFFECT	ACTION
0 (zero)	Double Space	Immediate space 2; Write*
(blank)	Single Space	Immediate space 1; Write*
+ (plus)	Print without spacing	Immediate NOP; Write*
1	Print on first line of next page	Immediate skip to line 1 of the next page; Write*
*Write has no integral carriage motion.		

Figure 12. FORTRAN Printer Carriage Control Characters (PRINTA)

CHARACTER	EFFECT	ACTION
0 (zero)	Double Space	Write*, space 2
(blank)	Single Space	Write*, space 1
+ (plus)	Print without Spacing	Write*
1	Print last line then go to first line of next page	
*Write has no integral carriage motion.		

Figure 13. FORTRAN Printer Carriage Control Characters (PRINTB)

DATA PARAMETERS FOR PRINT CALLS

In a print call, the data address points to the carriage control character which is contained in the byte immediately preceding the graphic data bytes. The byte count includes the carriage control character byte. In the following example:

```
120 characters for a print line
  1 carriage control character
---
121
```

a byte count of 121 is supplied to the I/O routines.

Error Routines

If there is an error during a tape read or write operation, a given number of retries will be performed (according to IBM standards). If the retries are successful, processing will continue. If they are not, control may be returned to the user's program (see "Return to User's Program") or a wait PSW may be loaded (see "SRETRY Routine").

Error recovery procedures may or may not enable the user to recover the error manually from the console. For a discussion of the conditions governing this procedure, see "SERP Routine."

RETURN TO USER'S PROGRAM

Returns to the user's program from an I/O routine are made starting at the location immediately following the SVC block. The return can occur in any one of three formats depending on the capabilities built into the DSCB. Indication is given in the DSCB if the return is a result of the previous call rather than the current one.

Figure 14 defines the types of returns.

ROUTINES

The routines of the FSD are:

1. FSD Initialization routine (DINT) Chart AA.
2. FSD Load Segment routine (LDPH) Chart AB.
3. Exit routine (EXIT) Chart AC.
4. I/O routines (see "I/O Operations") Charts AE through AR.

TYPE	RETURN AT	LOCATION	BYTES	WHEN USED
Type 1	Return	End of SVC block All returns		Used when neither error nor exceptional condition return is provided for in the DSCB.
Type 2	Return +4	End of SVC block Unusual return Normal return	4	Used when either error or exceptional condition is provided for in the DSCB, but not when both are provided for.
Type 3	Return +4 +8	End of SVC block Error return Exceptional Condition return Normal return	4 4	Used when both error and exceptional condition returns are provided for in the DSCB.

Figure 14. Return to the User's Program

DINT Routine: Chart AA

The DINT routine performs the required initialization.

ENTRANCE: The DINT routine receives control from IPL.

CONSIDERATION: The DINT routine performs the following initialization:

1. Associates the device upon which the system tape resides with data set reference number 0. (The system tape is always referenced as data reference number 0.)
2. Clears lower storage and the general registers.
3. Sets up the program, machine check, and supervisor program status words (PSWs).
4. Sets an indicator in the communications area that the FSD has control.

OPERATION: To establish the system tape device as data set reference number 0, the system tape device is placed into the data set reference number 0 entry of the device assignment table.

The system tape device address is determined when IPL is effected. This device address is compared against each device address in the device assignment table. The following conditions can occur:

1. The system tape device is already associated with data set reference number 0.
2. The system tape device compares with a device address associated with a data set reference number other than 0. The two device addresses are, therefore, switched.
3. The system tape device is not present in the device assignment table. The device address is, therefore, entered in the data set reference number 0 entry of the device assignment table.

Lower storage and the general registers are then cleared.

The FSD constructs the program, machine check, and supervisor PSWs and places them in their appropriate lower storage locations. Included as elements in the various PSWs are the following:

1. Program PSW: address of the routine to be branched to if a program interrupt occurs.
2. Machine Check PSW: address of the routine to be branched to if a machine check interrupt occurs.
3. Supervisor PSW: address of that portion of the FSD to be branched to when one of the phases requests a certain function of the FSD.

The FSD indicates that it currently has control by setting a specific indicator in the communications area.

EXIT: The DINT routine exits to the LDPH routine.

LDPH Routine: Chart AB

The LDPH routine loads a segment of the system, as required, for execution and determines the point at which control is to be received.

ENTRANCE: The LDPH routine initially receives control from the DINT routine. Subsequent to this initial entry, the LDPH routine receives control from one of the various segments of the system.

OPERATION: The load segment function of the FSD is initiated by an SVC instruction that can call for the load of a segment. After the load operation is complete, the FSD passes control to that segment.

Included in the load segment function of the FSD for the compiler is a check to insure that the punch device used to punch the output of a particular phase is not busy. If busy, the read of the next phase is not issued by the FSD until the punch is free. This insures that the contents of the output buffers of a given phase are not destroyed until the contents of the buffer have been punched.

EXIT: The LDPH routine exits to the newly-loaded segment.

Exit Routine: Chart AC

The EXIT routine determines the point of return within a segment so the FSD can return to the appropriate place after an I/O operation is performed.

ENTRANCE: The EXIT routine receives control form an I/O routine within the FSD, after that routine has fulfilled the request for some segment.

CONSIDERATION: The return address is determined from the address of the byte following the SVC instruction that requested the I/O operation. This address, which was saved in the supervisor old PSW, may or may not be the return address.

If a parameter list follows the SVC, the saved address is the address of the first parameter. If no parameter list follows the SVC, the saved address is the return address within the segment after its I/O request has been fulfilled.

OPERATION: After an I/O routine performs its specified function, it returns control to the EXIT routine to access the saved address. The EXIT routine adds to that address the number of bytes, if any, which the parameter list following the SVC instruction occupies. The resulting address is the return point to the segment that originated the I/O request.

EXIT: The EXIT routine exits to the segment that originated the I/O request.

SIODIR Routine: Chart AD

The SIODIR (I/O Director Base) routine completes the initialization steps necessary for all I/O operations.

ENTRANCE: This routine is entered whenever an SVC instruction requesting an I/O operation is encountered.

CONSIDERATIONS: The SIODIR routine is required for all I/O functions.

This routine requires that a specified symbolic register hold the address of the DSTAB header block, if the table is not compiled with the I/O routines.

OPERATION: The SIODIR routine sets up the I/O base register, return PSW, and gets the initial DSTAB location.

The routine then determines if it is being entered for the first time during the current I/O operation (external entry), or for the second time (internal entry). If entry results from an external call, the routine saves the entry registers and call return PSW, and sets the internal switch. If entry results from an internal call or when the operations resulting from an external call have been performed, the SIODIR routine extracts and saves the SVC specifier byte and the data set reference number, determines the DSTAB and associated DSCB locations, and sets the DSCB references.

If a check operation or any operations other than those essential to all I/O functions are requested, the SIODIR routine branches to the appropriate routine to set up those operations.

When all required I/O operations are set up, the SIODIR routine sets the suppress incorrect length indication (SILI) flag (if specified in the DSCB) into the CCW model and sets up the I/O interrupt new PSW.

EXITS: The SIODIR routine exits to the SIOGO routine.

ROUTINES CALLED: During execution this routine references the following routines: SD1, SD2, SD5, and SD7.

SIOGO Routine: Chart AE

The SIOGO (I/O Initiator Base) routine initiates all I/O calls.

ENTRANCE: The SIOGO routine is entered from the SIODIR routine when that routine

completes its set-up functions; it may also be entered from the SNTPIN and SRETRY routines.

CONSIDERATIONS: The SIOGO routine is required for all I/O functions.

OPERATION: The SIOGO routine determines if the physical device has been checked and, if not, branches to the SNTPIN routine to check it.

The routine initiates a series of tests to guard against an early burst mode device. If the routine is operating in a multiplex mode on a multiplex channel with a multiplex device, and the new device is not a multiplex device, the routine sets the CCW model in reserve and branches to the SNTPIN routine.

When a path is available, the CCW model is brought in and the DSCB set up. The SIOGO routine sets up the CCW and CAW, and issues the Start I/O (SIO) command to the device. After the SIO command is issued, a series of operations, based on the condition codes set after the command is issued, are performed.

Condition code 2 or 3 causes the SIOGO routine to set the CCW model into reserve and transfer control to the SNTPIN routine.

If condition code 1 is found and the busy bit is not present, control is transferred to the SNTPIN routine. If condition code 1 is found and the busy bit is present, the SIOGO routine sets the CCW model in reserve and transfers control to the SNTPIN routine.

If condition code 0 is indicated and the routine is not to wait for device end (in which case control is transferred to the SNTPIN routine), the SIOGO routine clears the internal flag, restores the original call return and entry registers, sets up to return control to the user's program, and returns control to it.

When a path is not available and the CSW has not been stored, the CCW model is set in reserve and control is transferred to the SNTPIN routine.

EXITS: The SIOGO routine exits to either the user's program or the appropriate location in the SNTPIN routine.

ROUTINES CALLED: During execution, the SIOGO routine references the SNTPIN and SETMD routines.

SNTPIN Routine: Chart AF

The SNTPIN (I/O Interrupt Entry) routine performs the analytic functions necessary to handle I/O interrupts.

ENTRANCE: The SNTPIN routine is entered at its initial location whenever an I/O interrupt occurs. It is entered at various symbolic locations from the SIOGO, SRETRY, SERP, and SD2 routines.

OPERATION: This routine establishes an I/O base register (saving the environment if the current entry is not internal) and sets the DSTAB and DSCB references.

After storing the latest I/O PSW and latest CSW, the routine determines if the operation has ended. If it has, a sense command is issued to the current device; the busy and multiplex flags are cleared; and, if a retry is specified at this time, control is transferred to the SRETRY routine.

When the retry indications have been cleared (i.e., no retry specified) or if the operation has not ended, tests for minor interrupt conditions (attention bit, program control interrupt-PCI, incorrect length record, or unit exceptional condition) are performed and the flag for the appropriate indication(s) is set.

The SNTPIN routine then performs a series of tests to establish the check operation status. These tests will ultimately result in transferring control to the appropriate location in the SIOGO routine or enabling a wait. The following paragraphs describe the possibilities.

When an immediate check is specified and the device is still not busy, the residual bit count is saved. Control is then transferred to the SERP routine to check for any class of errors.

After control returns from the SERP routine or if an immediate check was not specified, a check is made for any error or unusual condition which forces an immediate return to the user. If any exist, result area pointers are set in the communication registers.

If the operation is ended and return to the user is to be made, the internal flag, the device wait-check flag, the not-yet-checked flag, and the device usage flag are cleared. Checks are made for the presence of a wait or reserve operation. If either operation is present, the wait state will be entered until termination of the current operation. Return is made to the call that requested the current I/O operation, if

neither a wait nor reserve operation is present.

EXITS: This routine exits to either the SRETRY, SERP, or the appropriate location of the SIOGO routine.

ROUTINES CALLED: During execution the SNTPIN routine references the SIODIR, SIOGO, SRETRY, and SERP routines.

SD1 Routine: Chart AG

The SD1 routine extracts and saves four items: the DSTAB header block location, the address that points to the first physical device, the number of devices, and the initial DSCB location.

ENTRANCE: This routine is entered from the SIODIR routine.

CONSIDERATIONS: The SD1 routine is entered only during the initial entry to the SIODIR routine. After this first and only use, the entire routine is eliminated and cannot be used without reloading the entire program.

OPERATION: The SD1 routine determines if the initial DSTAB location is already present in the area designated to hold that address, and exits if it is.

Otherwise, the routine obtains and saves the DSTAB header block location. It then extracts the number of data sets and the initial DSCB location from the header block and saves them. The header block location is incremented by 4 and the result is saved as the initial DSTAB location.

The SD1 routine makes a final test to make certain that the initial DSTAB location is present, and then exits.

EXITS: This routine exits to the SIODIR routine.

SETMD Routine: Chart AH

The SETMD routine performs the set-up functions for I/O operations that require a set mode operation code and CAW. It also performs the set-up for I/O operations involving the use of FORTRAN printer carriage control characters at the start of the data stream and for immediate eject operations on the IBM 1442 punch.

ENTRANCE: The SETMD routine is entered from the SIOGO and SD74 routines.

CONSIDERATIONS: The use of this routine requires the presence of the SIODIR routine.

OPERATION: After setting up the standard FORTRAN I/O CAW, the SETMD routine determines if seven track tape is being used.

When it is not, this routine effects a series of branches to set up the printer immediate control chain and the repetitive punch on the IBM 1442 Card Read-Punch, which does not have an automatic ejection when punching is complete.

When seven track tape is being used, the SETMD routine sets up the set mode modifiers for the DSCB and CCW chain.

When the set mode modifiers have been set up or after setting up for punch eject operations on the 1442, the set mode FORTRAN I/O CAW is set up and the CCW operation code is cleared and replaced by the set mode operation code.

EXITS: This routine exits to the routine that called it.

ROUTINES CALLED: During execution the SETMD routine references routines SD741 and SD743.

SD2 Routine: Chart AI

The SD2 routine determines if the current operation is a check operation and, if so, whether the device has already been checked.

ENTRANCE: The SD2 routine is entered from the SIODIR routine.

OPERATION: The SD2 routine determines if the current operation is a check operation and exits if it is not. If it is a check operation, but the device has already been checked, the SD2 routine branches to that part of the SNTPIN routine that establishes an exit path; otherwise it branches to that part of the SNTPIN routine that checks for minor interrupt conditions.

EXITS: This routine exits to the appropriate location in the SIODIR or SNTPIN routine.

SD5 Routine: Chart AJ

This routine sets up the model for all simple control operations; that is, for all control operations whose entire function is

defined in the operation byte of the command.

ENTRANCE: The SD5 routine is entered from the SIODIR routine.

CONSIDERATIONS: Control operation modifiers are moved in from the SVC parameters.

OPERATION: After clearing the data group flags, the SD5 routine determines if the current I/O call has an operation code of 3 (all operations not involving read or write, such as immediate space or select stacker) or 7 (all operations not involving read or write, such as rewind tape). If neither operation code is found, control is returned to the SIODIR routine.

When either operation code 3 or 7 is found, the appropriate operation code is placed into the CCW model.

The operation modifiers for simple control operations are then moved into the model, along with a count of 1 and the SILI flag.

EXITS: The SD5 routine exits to the SIODIR routine.

SD7 Routine: Chart AK

This routine sets the proper parameters for data operations into the CCW model.

ENTRAN E: The SD7 routine is entered from the SIODIR routine.

CONSIDERATIONS: Data operations include read, write, and print.

OPERATION: The SD7 routine sets the data group flags and the data parameters (the storage location at which the data is found, and the byte count).

The appropriate data operation is then set up in the CCW model by inserting whatever modifiers are necessary, the proper operation code, and making whatever adjustments are necessary for a particular device (such as the special FORTRAN carriage control characters for the print routine).

EXITS: This routine exits to the SIODIR routine.

ROUTINES CALLED: During execution the SD7 routine references the SD74 and SD72 routines.

SD72 Routine: Chart AL

The SD72 routine extracts data parameters through SVC pointers.

ENTRANCE: This routine is entered from the SD7 routine.

CONSIDERATIONS: This routine requires the presence of the SD7 routine.

Although this routine is not essential for I/O operations, it must be included if any SVC formats include expansion A for providing data parameters in registers.

OPERATION: When the current call does not include expansion A, or when it does but no pointers are supplied, the SD72 routine transfers control to the SD7 routine.

Otherwise, a work register is cleared and the identities of the two registers containing the pointers to the data parameters (one register containing the buffer address and the other the byte count) are loaded into the work register.

The SD72 routine extracts the data address span and the storage span; the data address span is then reserved.

The routine then positions the byte count span, loads the save area pointer into another register, and forms the pointer to the byte count area by adding the save area pointer to the byte count span.

The SD72 routine sets the byte count into the CCW model, forms the pointer to the data address area, and sets the data address into the model.

EXITS: The SD72 routine exits to the SD7 routine.

SD74 Routine: Chart AM

The SD74 (Print Operation Base) routine sets up for a print operation.

ENTRANCE: This routine is entered from the SD7 routine.

OPERATION: The appropriate print operation is set up using the SD742 routine for a PRINTB operation and the SD743 routine for a PRINTA operation. If the operation is not on a graphic device, there is no further processing.

If the operation is on a graphic device, the SD741 routine is used to adjust for the FORTRAN control characters; the printer

carriage control character is then set into the model. If the unit is not a printer, the console printer carriage control character is set into the model before the data parameters are adjusted to omit the control characters.

EXITS: This routine returns control to the SD7 routine.

ROUTINES CALLED: During execution, the SD74 routine references the SD741 routine, the SD742 routine, and the SD743 routine.

SD741 Routine: Chart AN

This routine performs an adjustment operation for any FORTRAN print control characters.

ENTRANCE: This routine is entered from the SD74 routine at entry point SD741 or from the SETMD routine at entry point SD741B.

CONSIDERATIONS: This routine requires the presence of the SD74 and SETMD routines.

All FORTRAN control characters are entered in a FORTRAN control character list. This list consists of 2 chains, one for PRINTA and one for PRINTB.

OPERATION: When the SD741 routine is entered from the SD74 routine, the FORTRAN control character list is searched sequentially until the FORTRAN control character is found. A pointer is then set to that control character. If the control character is not found, the last character in the list is used. A return is then made.

When the SD741 routine is entered from the SETMD routine, CCW chains and/or chaining flags are set up according to the following scheme.

A 4-CCW chain and chaining flag for:
PRINTA console printer

A 3-CCW chain for:
PRINTA printer skip
PRINTA console printer single space
PRINTA console printer double space
PRINTB console printer skip

A 2-CCW chain and chaining flag for:
PRINTA printer single space
PRINTA printer double space
PRINTB console printer double space

A special CCW for:
PRINTB printer skip

A 1-CCW for:
all others

EXIT: The SD741 routine returns control to the calling routine.

SD742 Routine: Chart AO

The SD742 routine sets up the FORTRAN print control character for the PRINTB function.

ENTRANCE: This routine is entered from the SD74 routine.

CONSIDERATIONS: This routine requires the presence of the SD741 routine.

OPERATION: The SD742 routine determines if the current request is for PRINTB. If it is not, the routine returns control to the SD74 routine. If it is, a pointer is set to the PRINTB character list. A return is then made.

EXIT: This routine returns control to the SD74 routine.

SD743 Routine: Chart AO

This routine sets up the FORTRAN print control character for the PRINTA function.

ENTRANCE: The SD743 routine is entered from the SD74 routine.

CONSIDERATIONS: This routine requires the presence of the SD741 routine.

OPERATION: The SD743 routine determines if the current request is for PRINTA. If it is not, the routine returns control to the SD741 routine. If it is, a pointer is set to the PRINTA character list. A return is then made.

EXIT: This routine returns control to the SD74 routine.

SRETRY Routine: Chart AP

This routine performs error retry procedures for tape devices.

ENTRANCE: The SRETRY routine is entered from the SNTPIN routine.

CONSIDERATIONS: The SRETRY routine maintains two command chains. The clean chain consists of three backspace commands, two forward space commands and a transfer into the fix chain. The fix chain first issues a backspace command, then, depending on the conditions found, sets an erase gap command or request track in error into the FIXCCW; it terminates by a TIC instruction to the set mode CCW.

OPERATION: For illustrative purposes, the operation of this routine has been divided into three paths.

Path 1: If the current device is not a tape, control is returned to the SNTPIN routine.

The SRETRY routine determines if there is a data check. If there is no data check, processing continues at Path 2.

When the current routine entry is not the first error, the retry counts are restored and processing continues at Path 3. Otherwise, a test is made to determine if the tape is in write status; when it is, the retry count is set to 3, and processing continues at Path 3.

If the tape is not in write status, the routine determines if a noise record has been read and exits if it has. If a noise record has not been read, it sets the read indicator, sets the retry count to 10, sets the tape clean count to 10, and continues processing at Path 3.

Path 2: If this is not the first retry attempt, or if this is the first retry attempt but the load point indicator is not set, control is returned to the SNTPIN routine.

Otherwise, the SRETRY routine determines if the third backspace in the clean chain has been attempted. When it has not, the tape clean count is reset to 10, and processing continues at Path 3.

When the third backspace has been attempted, the control areas are cleared and the CAW is set to forward space one data record. The retry count is set to 10, the retry and tape clean counts saved, and the routine exits.

Path 3: The control areas are cleared. When both the retry and tape clean counts are exhausted, an indication that the retry is completed is made, and control is returned to the SNTPIN routine.

When the retry count is exhausted, but the tape clean count is not, the CAW is set to the clean chain and the retry count to 10. If the retry count is not exhausted, the CAW is set to the fix chain and a test is made to determine if the previous read error indicator is on. When that indicator is not on, the fix CCW is set to "erase gap."

When the previous read error indicator is on, or after setting the CAW to the clean chain, the fix CCW is set for "request track in error."

After the fix CCW has been set (for request track in error or erase gap), the retry and tape clean counts are saved and the routine exits.

EXITS: This routine returns control to that portion of the SIOGO routine that issues the Start I/O Command.

ROUTINES CALLED: During execution of the SRETRY routine the SIOGO routine is referenced.

SERP Routine: Charts AQ and AR

This routine prepares the system to perform error recovery procedures. It may be used to check for error conditions or for a condition code 3 occurring after an SIO operation.

ENTRANCE: The SERP routine is entered from the SNTPIN, SIOGO, or SRETRY routines.

CONSIDERATIONS: This routine is optional. Its use requires the presence of the SRETRY routine.

Four messages may be issued by this routine. They are; FIA, FIC, FID, and FIS.

OPERATION: When the routine is used for error detection, the channel failure and unit check indicators are tested. If none are on, a normal return is made. If any are on, the error indicated is processed.

When an error is known to exist, the processing takes one of three general paths. The first (Chart AQ) is for the condition code 3 occurring after an SIO operation, the

second (Chart AQ) is for a channel failure indication, and the third (Chart AR) is for a unit check indication. All three paths use a common routine procedure in case of error. This procedure consists of setting up and printing the proper message (FIA, FIC, FID, or FIS), setting up the entry for the external PSW and for SEREP, and moving the unit address to a position in the FSD area used by SEREP.

During object time, 2540 punch equipment check retries are handled differently than other error retries. 2540 punch equipment checks require the repunching of the last two cards punched. Other errors require reprocessing only one record, which can be done within FSD. For 2540 punch equipment checks, the return is set to the IBCOM IB2540 routine for the retry.

EXIT: If the SERP routine does not return control to the calling routine, an FIA, FIC, FID, or FIS message is issued. The routine then loads the wait PSW.

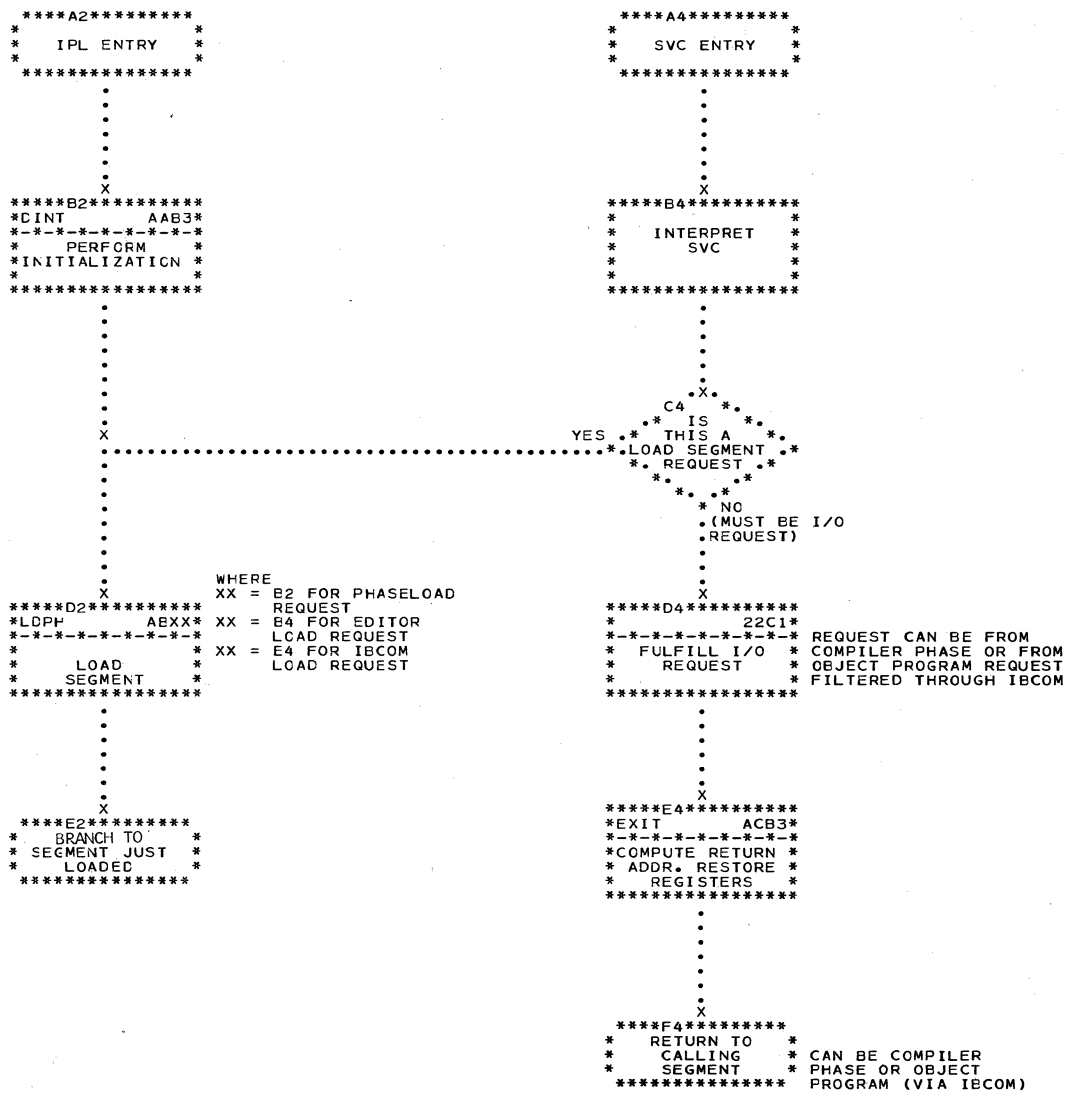


Chart 01. FSD Overall Logic Diagram

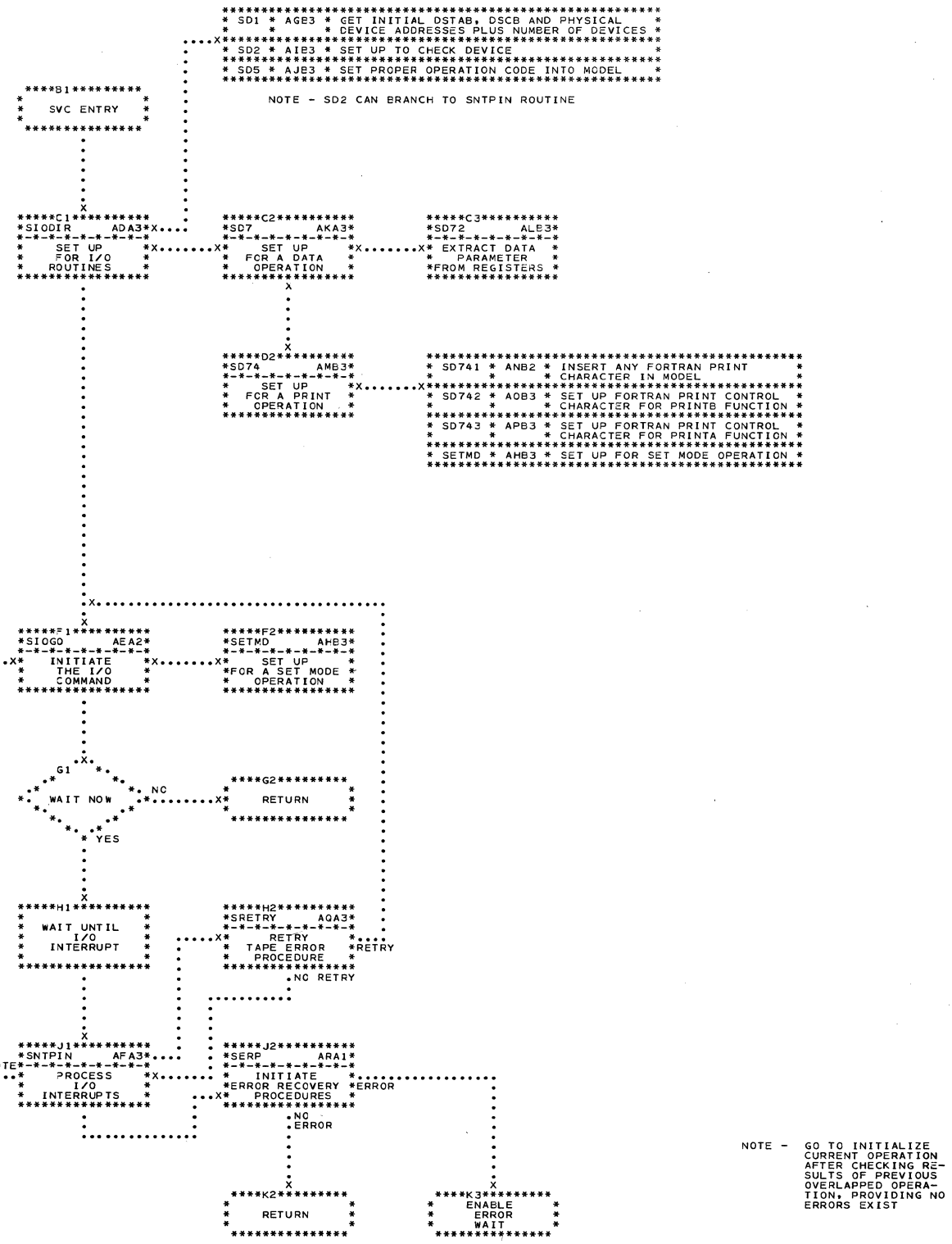


Chart 22. Overall Logic-I/O Routine

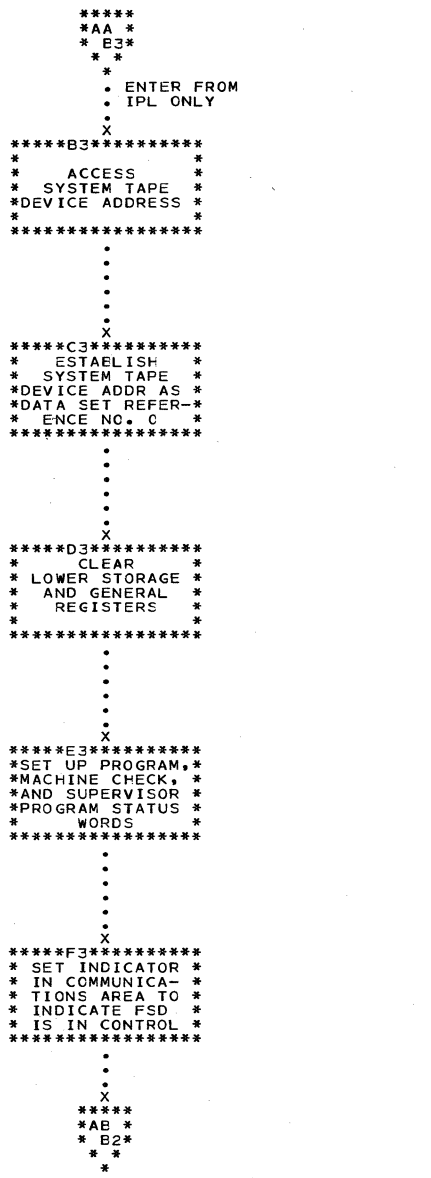


Chart AA. DINT Routine

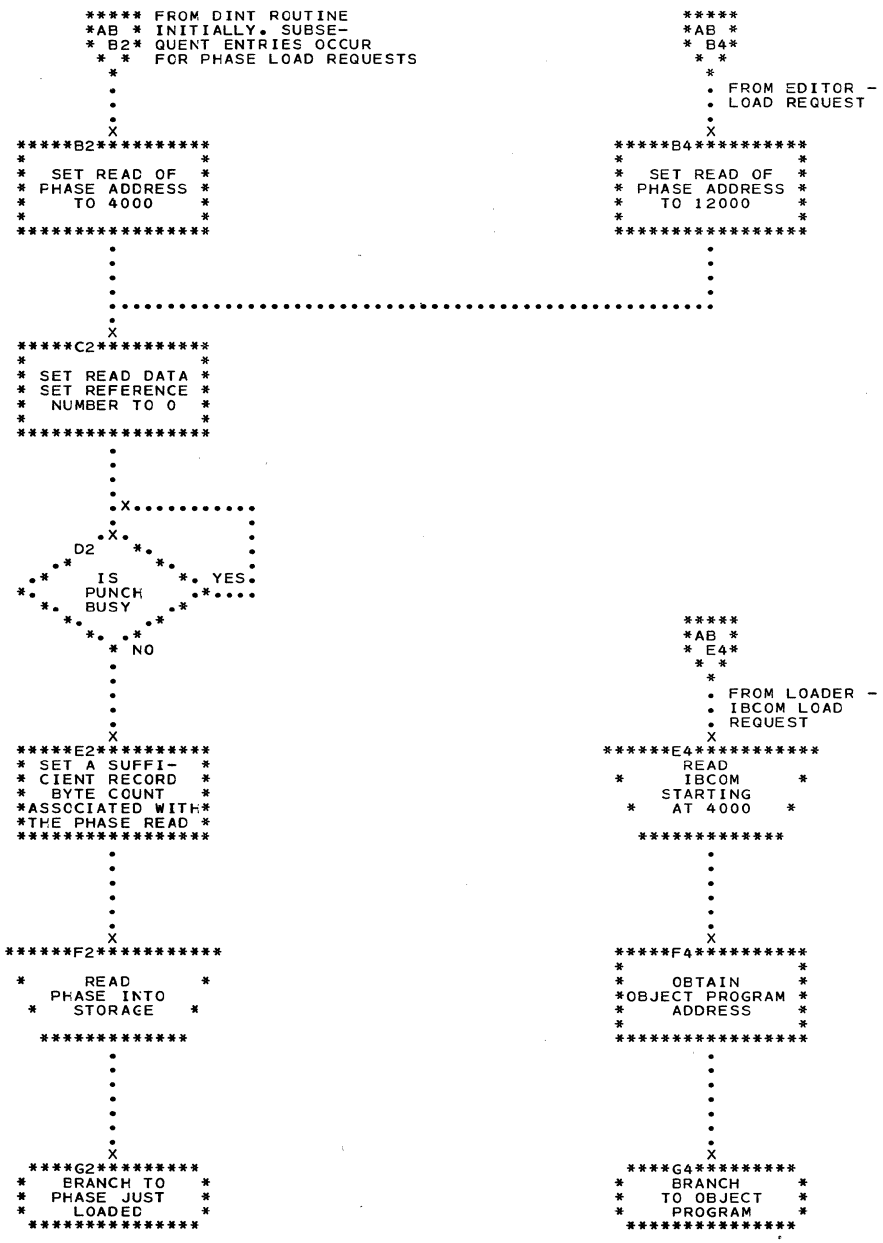


Chart AB. LDPH Routine

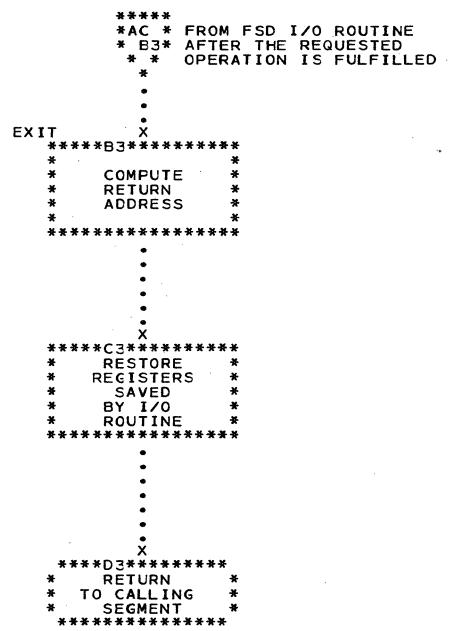


Chart AC. EXIT Routine

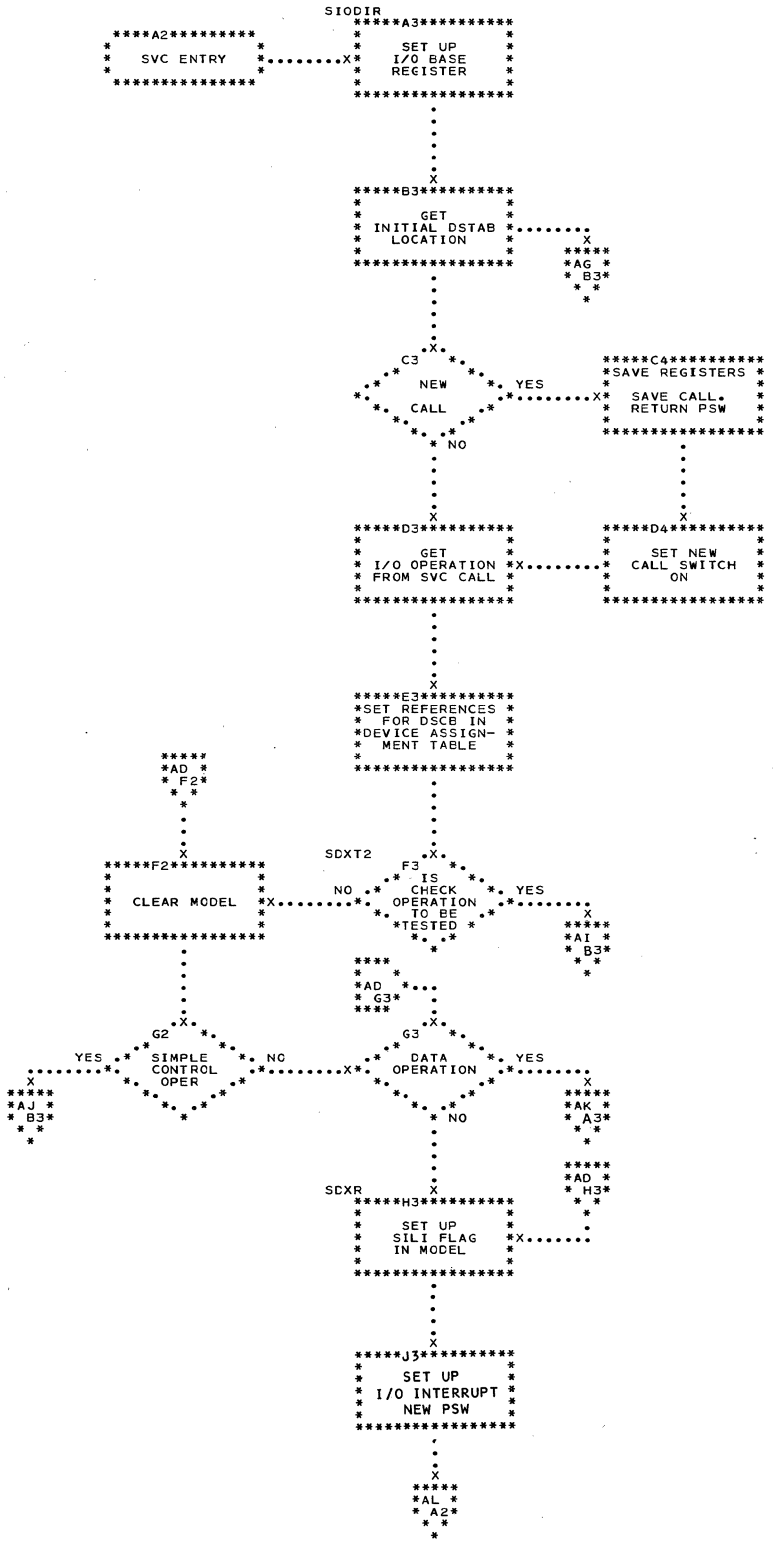


Chart AD. SIODIR Routine

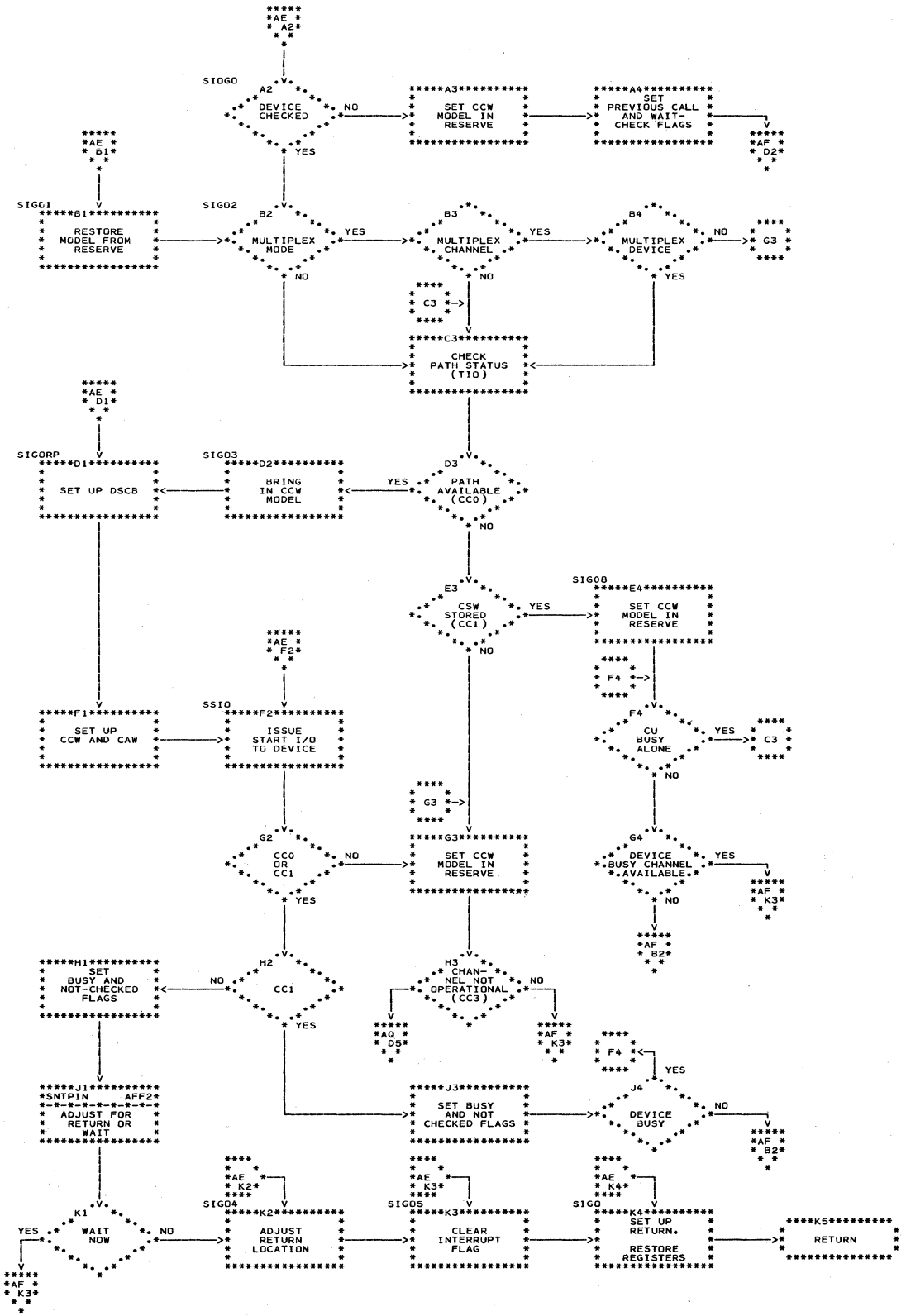


Chart AE. SIOGO Routine

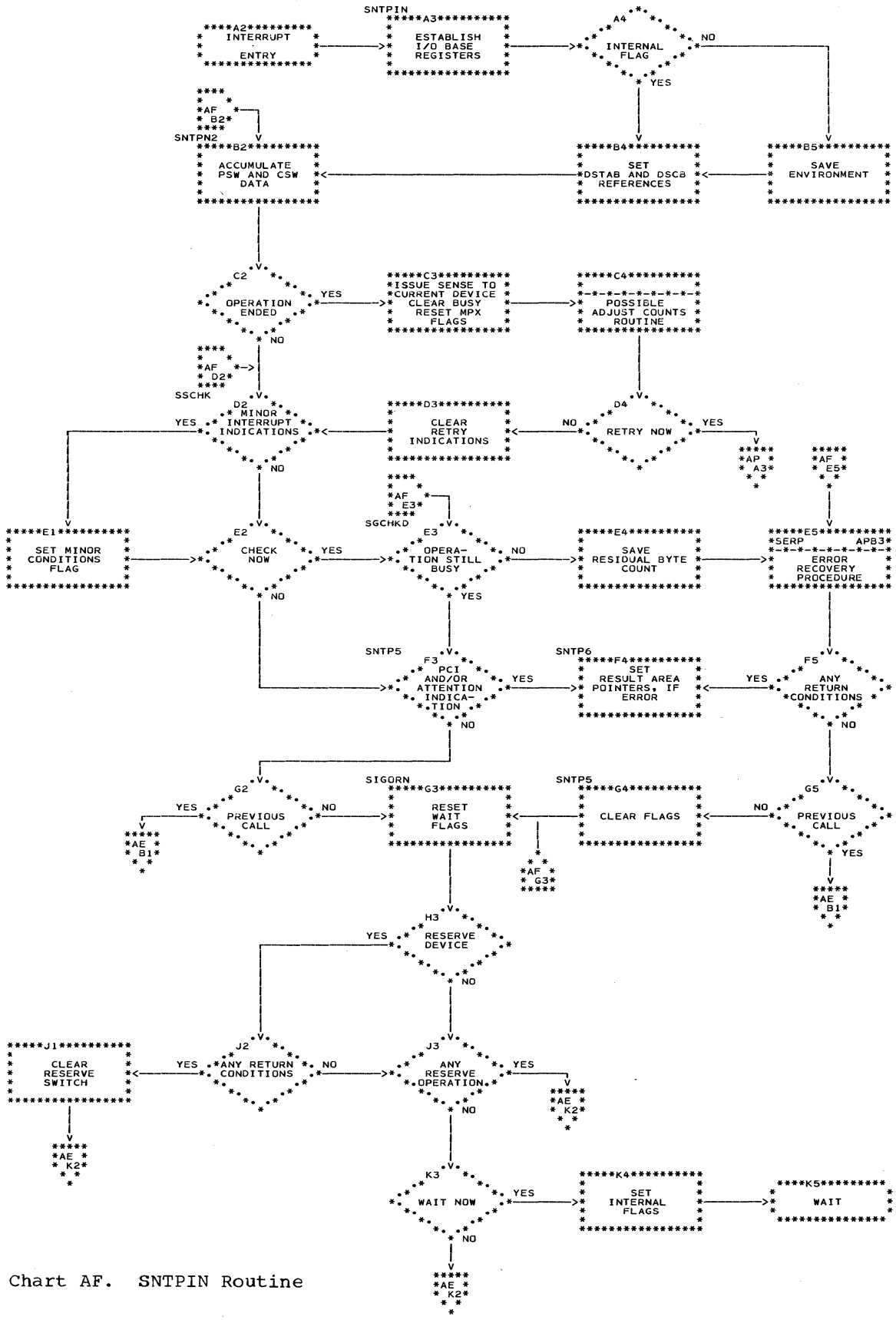


Chart AF. SNTPIN Routine

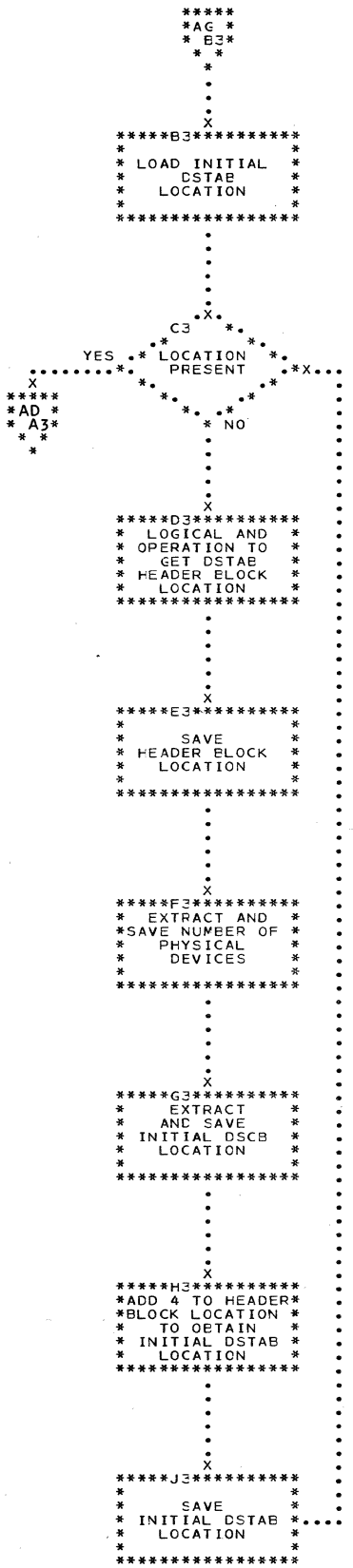


Chart AG. SD1 Routine

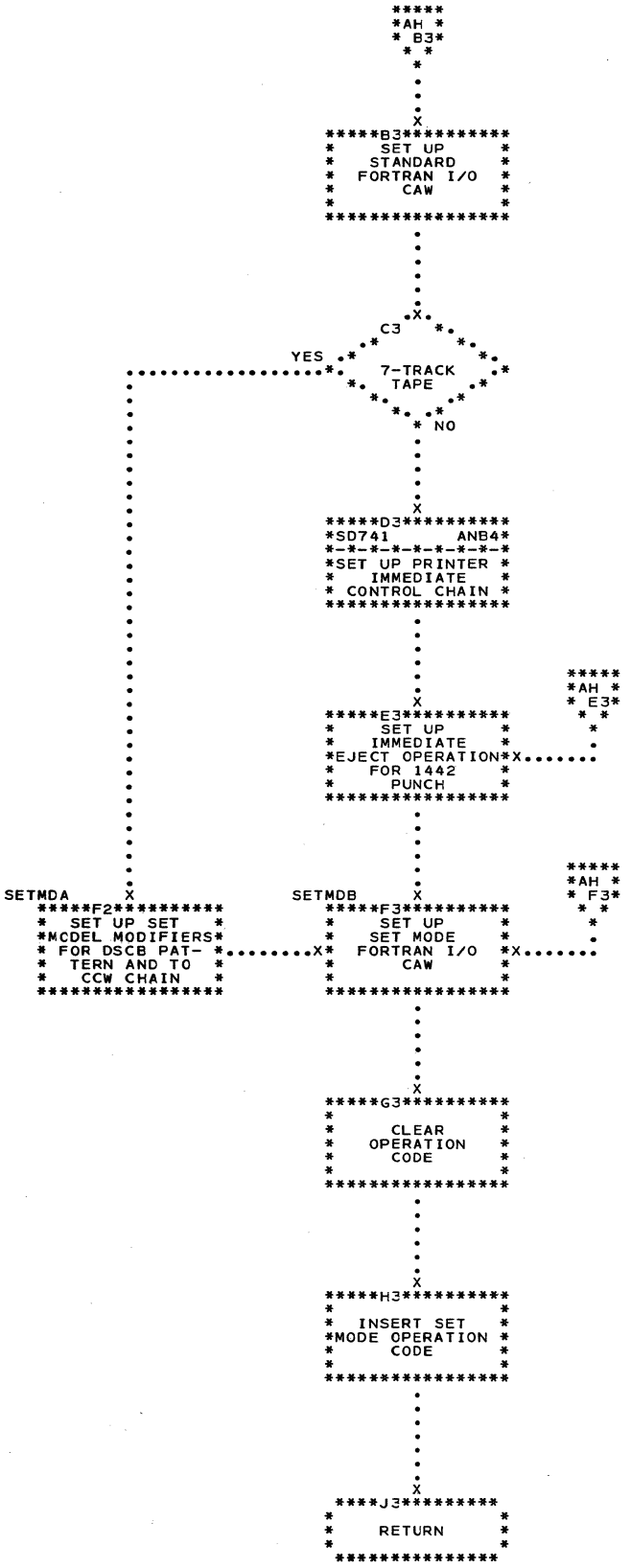


Chart AH. SETMD Routine

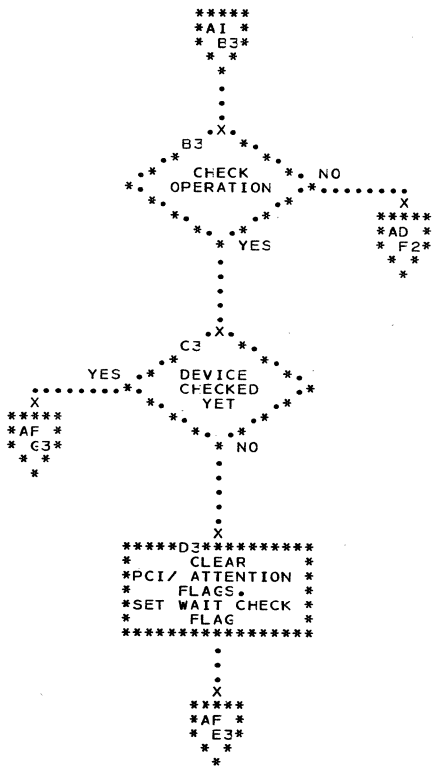


Chart AI. SD2 Routine

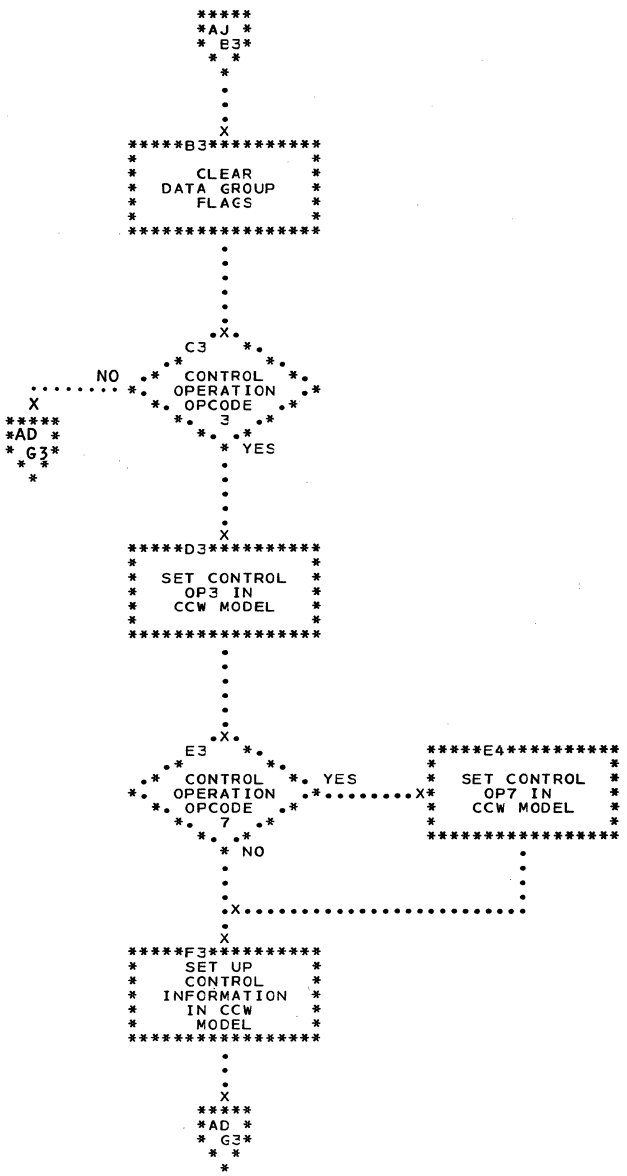


Chart AJ. SD5 Routine

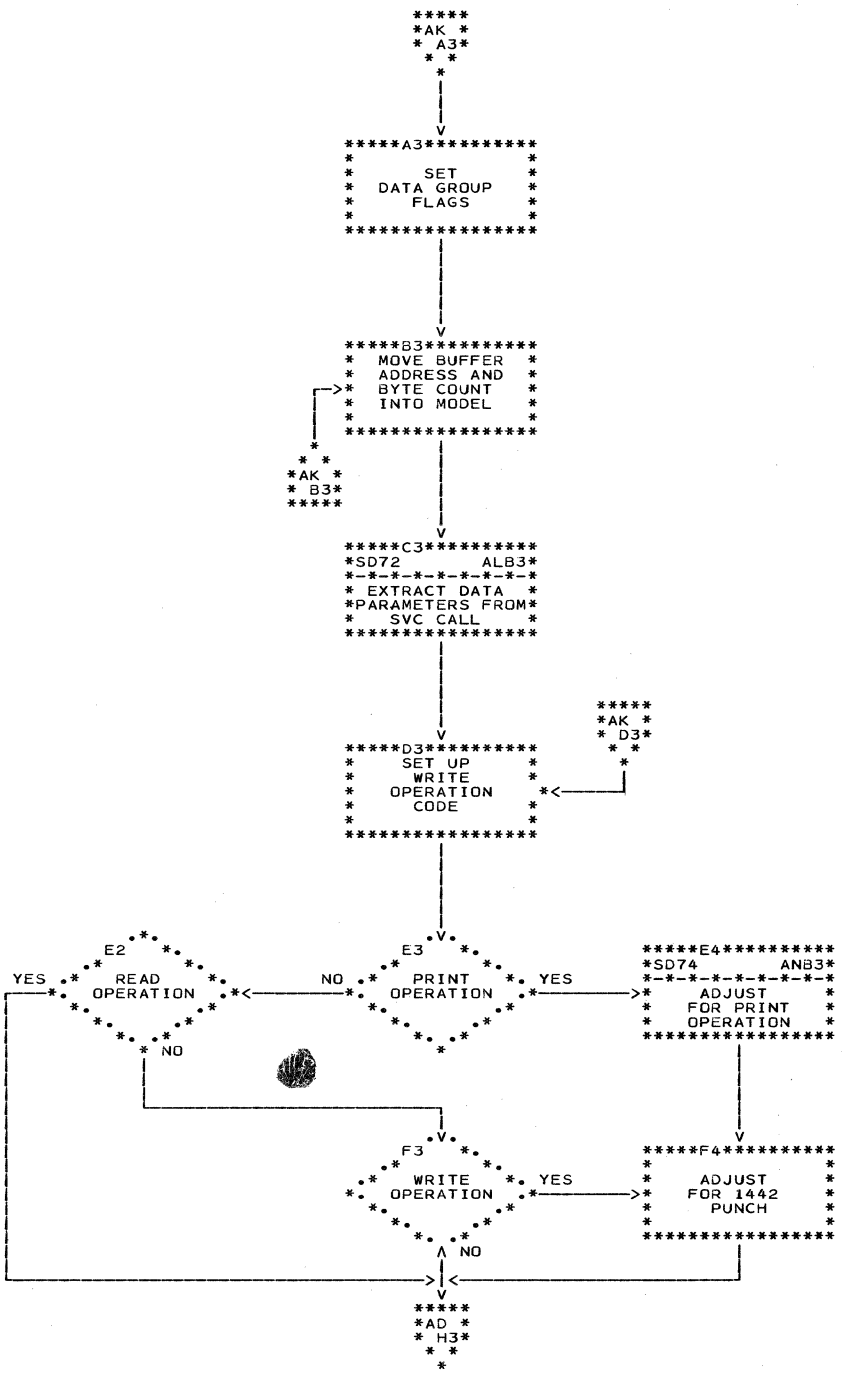


Chart AK. SD7 Routine

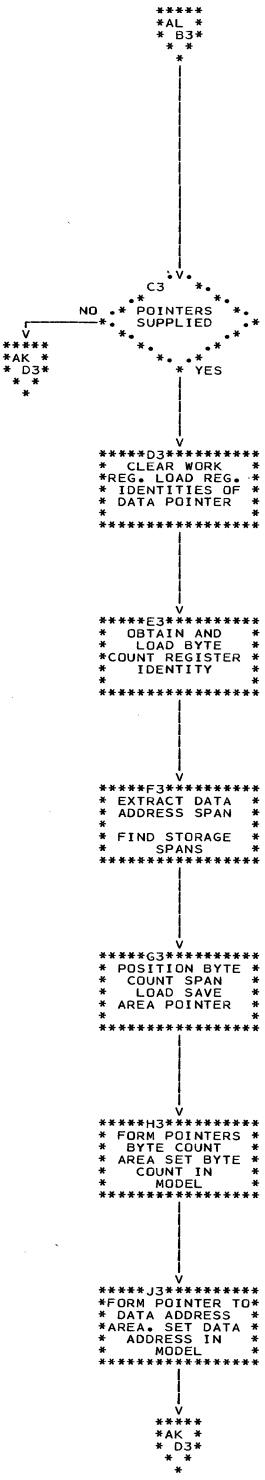


Chart AL. SD72 Routine

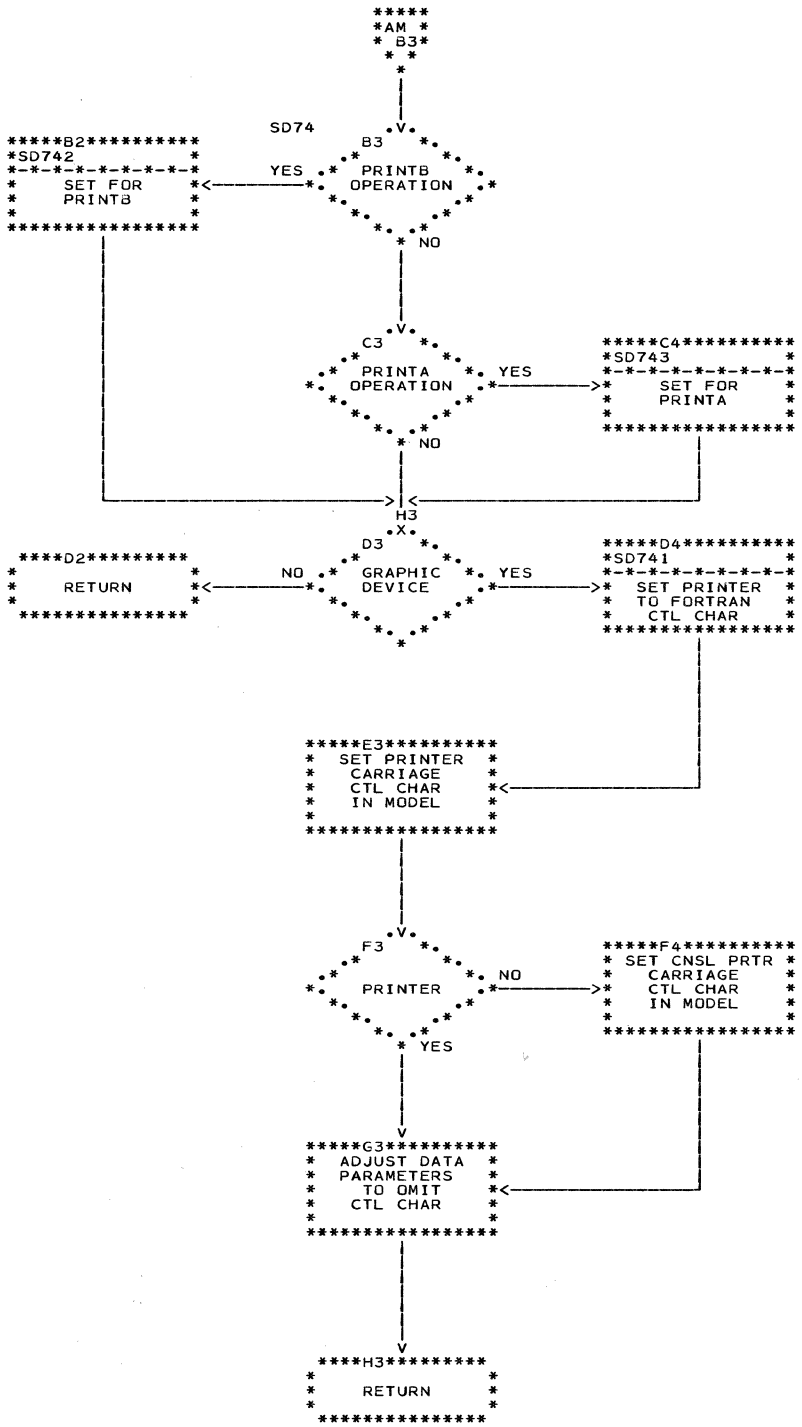


Chart AM. SD74 Routine

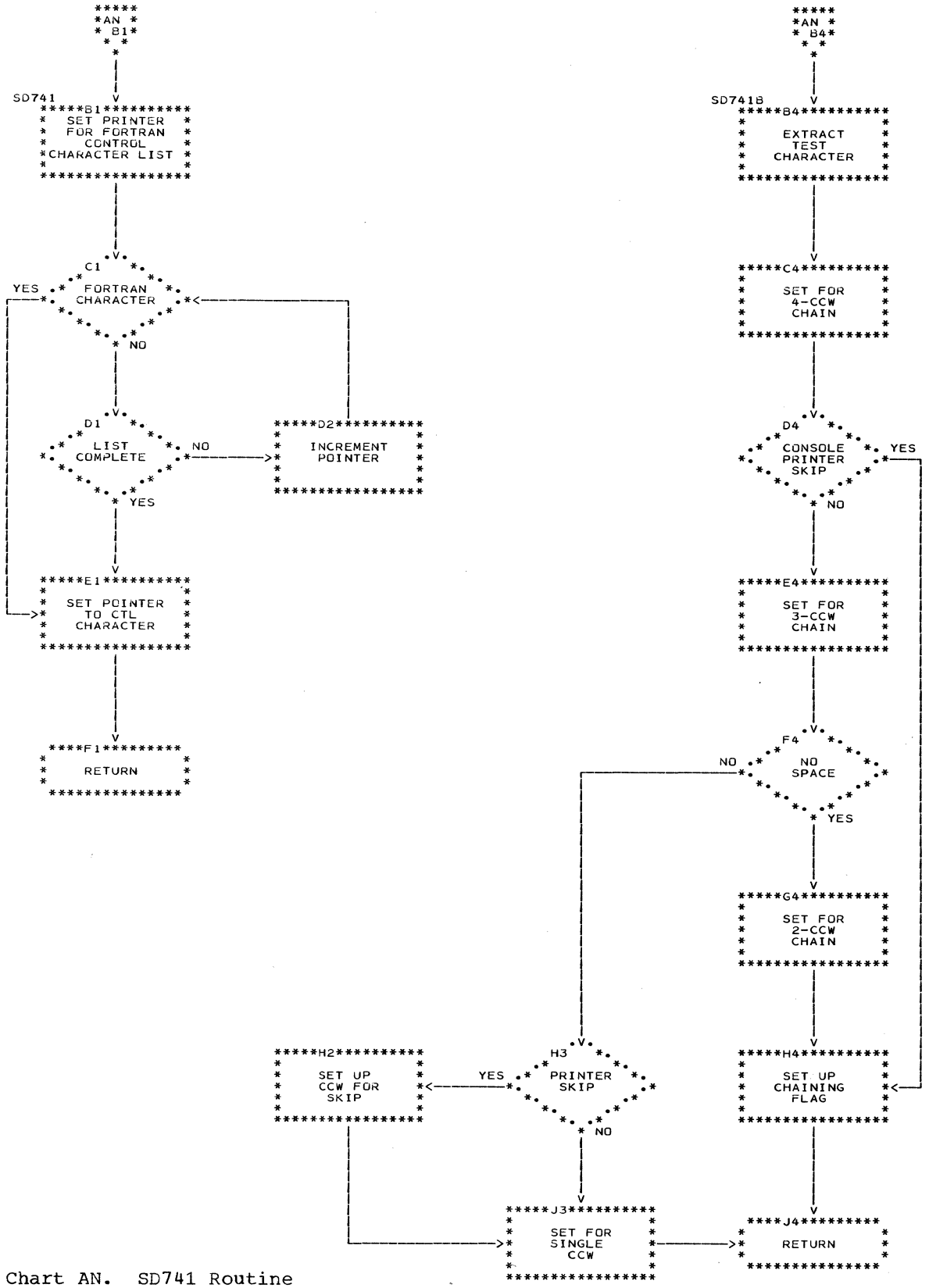


Chart AN. SD741 Routine

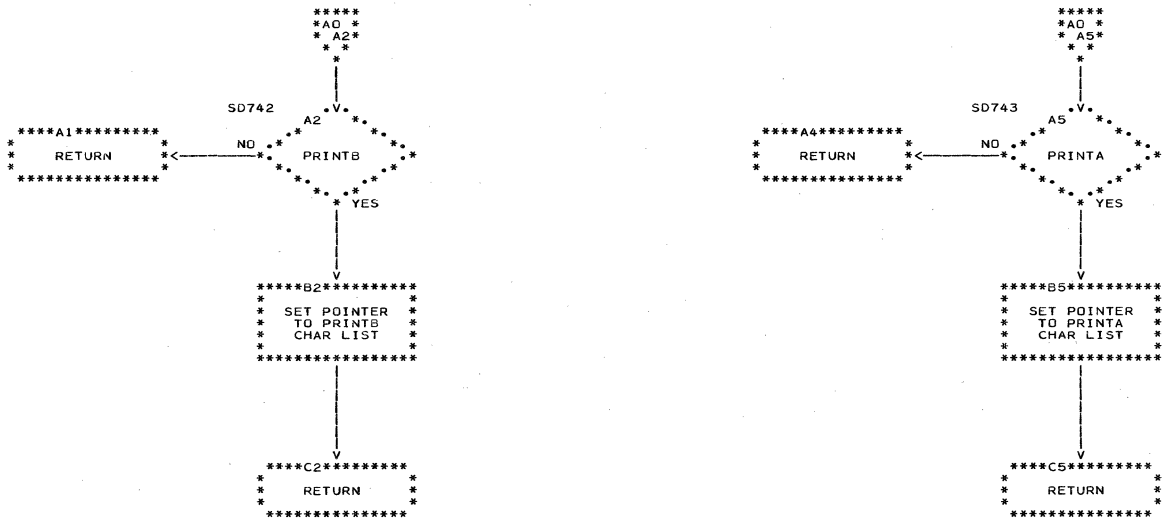


Chart AO. SD742 and SD743 Routines

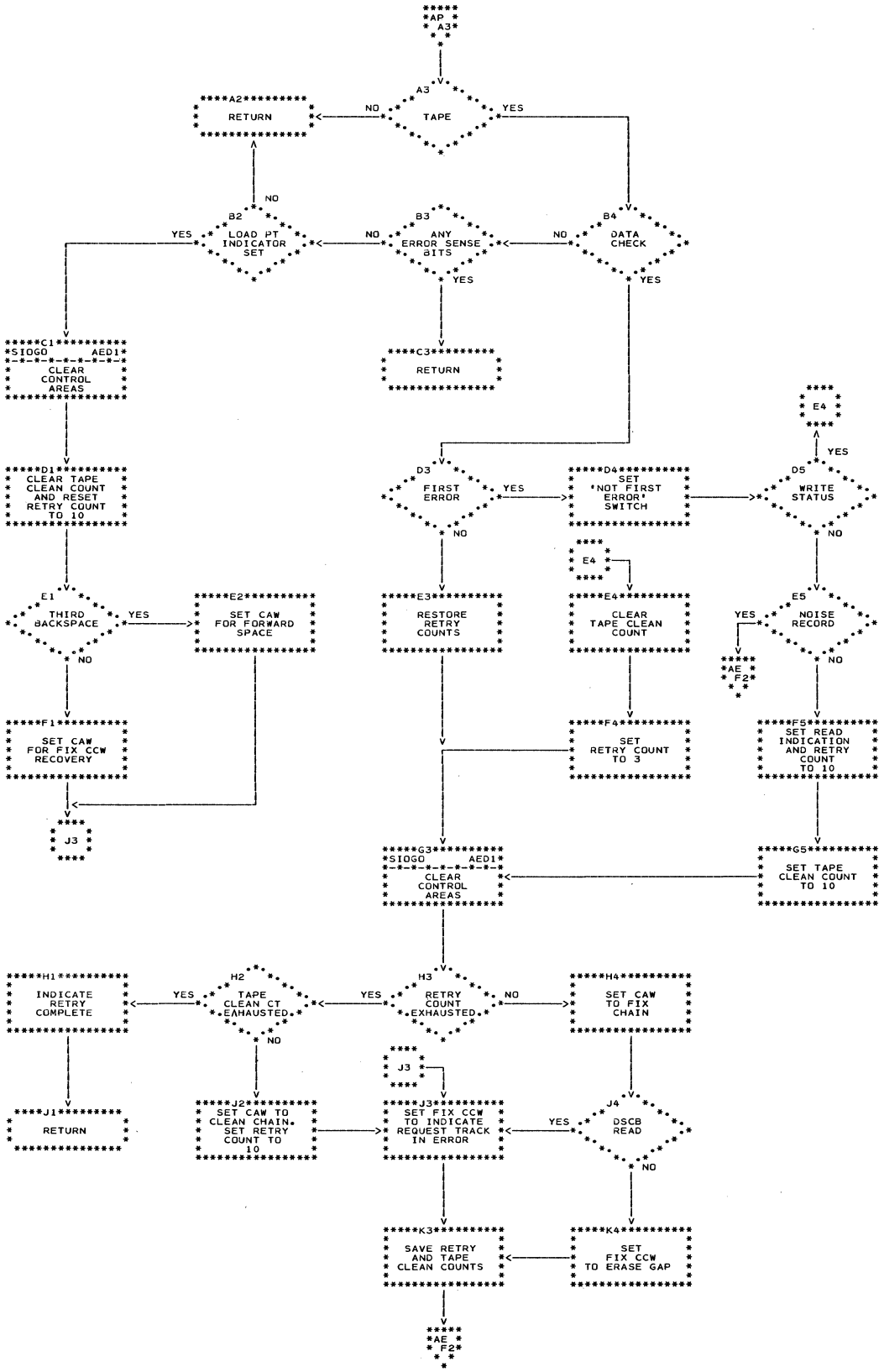
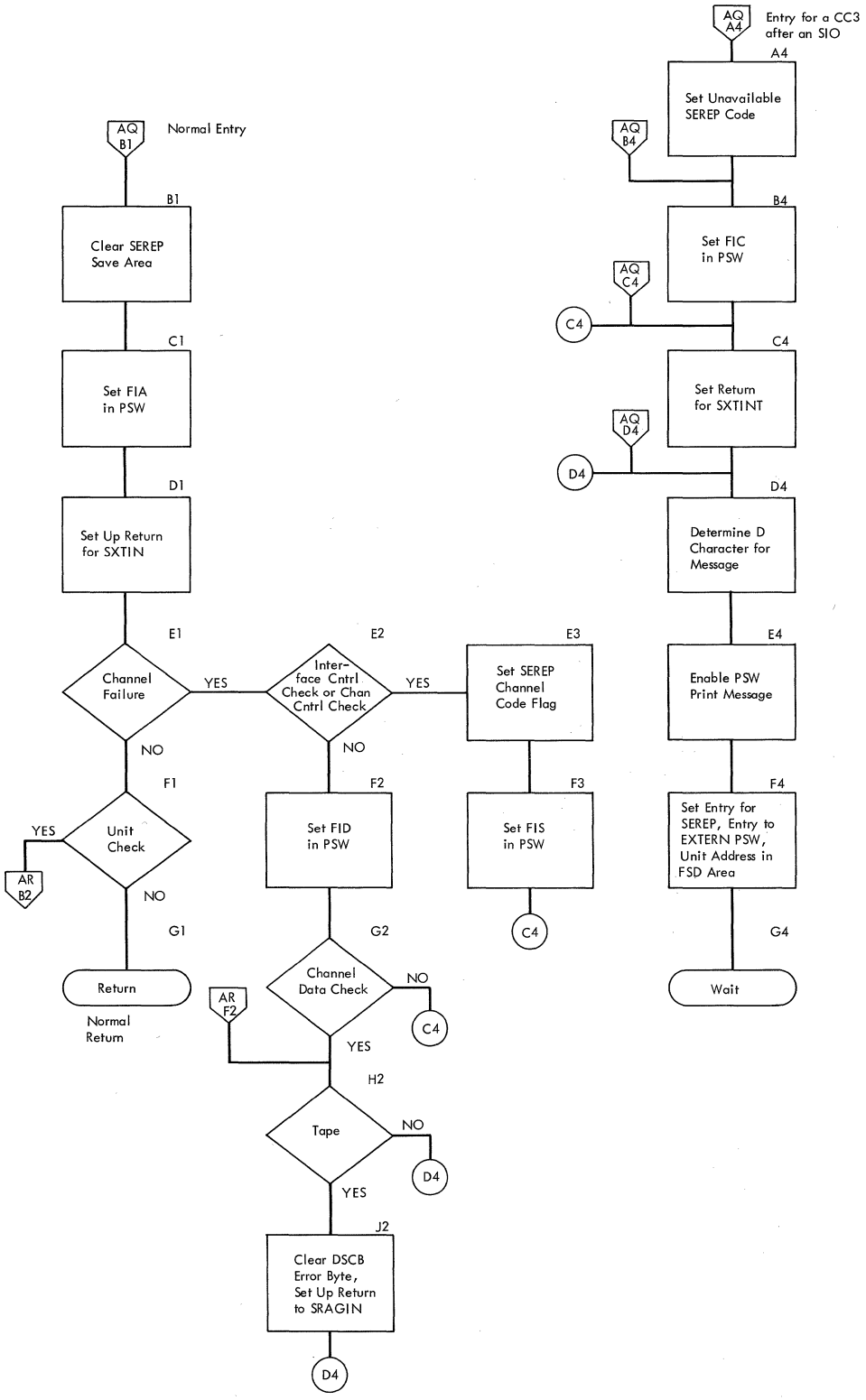
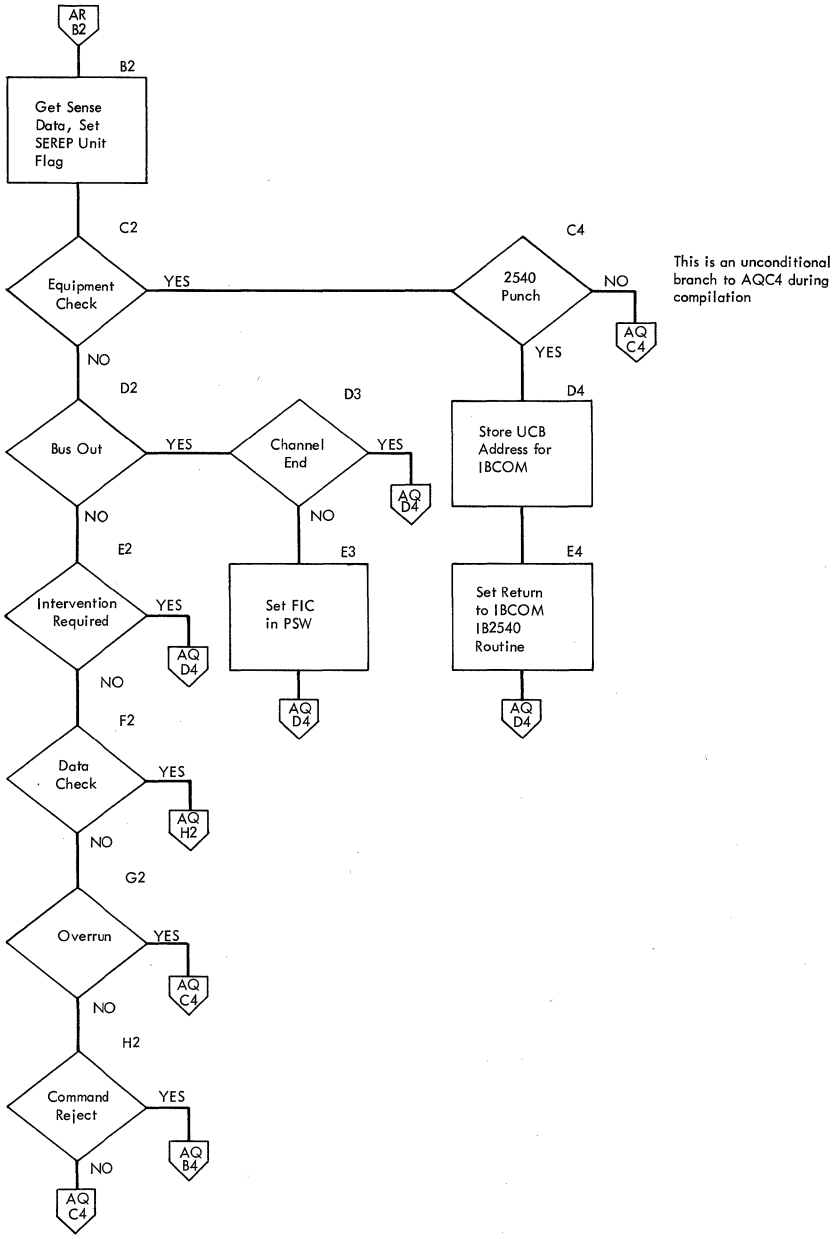


Chart AP. SRETRY Routine



● Chart AQ. SERP Routine (Part 1 of 2)



● Chart AR. SERP Routine (Part 2 of 2)

At the beginning of any job (whether a single job or a job within a multiple job), the FORTRAN System Director (FSD) reads in and gives control to the Control Card routine. The Control Card routine reads in control cards (also referred to as control statements), sets appropriate indicators in the communications area, and determines whether:

1. The system is to be modified.
2. An object program is to be executed.
3. A source program is to be compiled.
4. A combination of functions is to be performed (e.g., compile and execute).

Chart 02, the Control Card Routine Overall Logic Diagram, indicates the entrance to and exit from the Control Card routine and is a guide to the overall functions of the routine.

ROUTINES

The CCLASS routine (Chart AT) reads control cards and determines the type of function to be performed (e.g., a compilation). Charts AU through AZ represent the various routines that process the FTC, EDIT, LOAD, SET, JOB, and DATA control cards.

CCLASS Routine: Chart AT

The CCLASS (Control Card Classification) routine controls the processing of control cards (also referred to as control statements) used in conjunction with compilation and execution.

ENTRANCE: The CCLASS routine receives control from the FSD.

CONSIDERATION: Any job is terminated by either a DATA control card or an end of data set. The DATA control card is used as a flag by the CCLASS routine. In a compile and execute job, there may be blank cards between the source program and the input data to be used during execution. In this case, the CCLASS routine can quickly step through the blank cards because a DATA control card precedes any object-time data cards.

For compile and execute jobs, all compiled source and object programs to be executed are placed on the GO tape.

OPERATION: The operation of the CCLASS routine is discussed in accordance with the relative position of an input card to one or more jobs.

Initial Entry Into CCLASS Routine: Any entry into the CCLASS routine prior to the first compilation or execution immediately causes a card to be read. The card may be either a blank card, a control card, or the first card of an object or source program.

The CCLASS routine ignores any blank card and proceeds to read another card.

A control card causes the CCLASS routine to give control to the appropriate Control Card routine, which interprets the information on the card and sets up the proper directives for subsequent action.

The first card of an object program causes the CCLASS routine to take one of two options: (1) if the object program is to be executed, each object program card is written on the GO tape, or (2) if the object program is not to be executed, each object program card is ignored.

The encounter of any other type of card causes the CCLASS routine to assume the card is the first card of a source program. Therefore, the FSD is called to load Phase 10 and begin compilation.

Subsequent Entry Into CCLASS Routine: After a source program has been compiled, the FSD returns control to the CCLASS routine which determines if a single or multiple job is currently specified.

For a single job, the CCLASS routine, upon receiving control after the compilation, looks for a DATA control card. When a DATA control card is read, control is given to the CCDATA routine.

If the end of data set, or a card other than a blank card, is encountered without finding the DATA control card, the CCLASS routine performs the following functions. It prints a warning message indicating that the DATA control card is missing, simulates the DATA control card, and then returns control to the FSD. The FSD either terminates the job or calls the FORTRAN loader, depending on whether or not the job is to be executed.

For a multiple job, the CCLASS routine immediately reviews the contents of the last card that was read to determine its type. (The last phase has read a card that now becomes the first card to be processed by the CCLASS routine. This card is saved in a buffer in the communications area, which remains resident in main storage.)

The subsequent operation of the CCLASS routine is similar to the operations performed upon the initial entry into the CCLASS routine.

EXIT: The CCLASS routine exits to the FSD or to one of the control card routines.

CCJOB Routine: Chart AU

The CCJOB (Job Control Card) routine interprets information supplied on the JOB control card and transforms that information into appropriate directives for the FSD.

ENTRANCE: The CCJOB routine receives control from the CCLASS routine when a JOB control card is encountered.

CONSIDERATIONS: The information on the JOB control card is subsequently used by the CCLASS routine to determine: (1) whether a single or multiple job is specified, and (2) the action taken following compilation.

OPERATION: The JOB control card is scanned to determine the desired option. When an option is determined, a corresponding indicator is set in the communications area.

EXIT: When a blank field is encountered, indicating that all specifications have been examined, the CCJOB routine returns control to the CCLASS routine. The CCJOB routine may also be terminated if a specified option does not correspond with an available option. If this occurs, an error message is written, and control is returned to the CCLASS routine.

CCFTC Routine: Chart AV

The CCFTC (FTC Control Card) routine interprets information supplied on the FTC control card and transforms that information into directives for the FSD.

ENTRANCE: The CCFTC routine receives control from the CCLASS routine when a FTC control card is encountered.

CONSIDERATION: The information in the FTC control card is used by the FSD to determine action taken during and/or following compilation.

OPERATION: The FTC control card is scanned, field by field, by the CCFTC routine to determine any specified option or options. Appropriate indicators are set in the communications area for performance of the desired functions.

EXIT: When a blank field is encountered, indicating that all options have been examined, the CCFTC routine returns control to the CCLASS routine. The CCFTC routine may also be terminated if a specified option does not correspond with an available option. If this occurs, an error message is written, and control is returned to the CCLASS routine.

CCSET Routine: Chart AW

The CCSET (SET Control Card) routine uses information supplied on the SET control card to temporarily modify the device assignment table in main storage. The device assignment table is used in the compilation of the source program. The device assignment table on the system tape is not modified by this routine.

ENTRANCE: The CCSET routine receives control from the CCLASS routine when a SET control card is encountered.

CONSIDERATIONS: The device assignment table in the FSD contains a list of data set reference numbers and a list of corresponding addresses of the physical devices to which these data set reference numbers refer. (An input/output device is referenced by a data set reference number which in no way implies a particular device.) A specification within the device assignment table specifies the type of device and its physical address.

An arbitrary I/O configuration, understood by the compiler, exists for each installation. Unless specific changes are to be made to this configuration, the SET card does not have to be specified. Different installations have different physical addresses and therefore call for different changes in the device assignment table.

The SET option, requested by a control card at compile time, is performed only for that job. If permanent changes in the device assignment table are required, the EDIT option may be used. The EDIT option recognizes the same SET card; however, a

new system tape with an altered device assignment table is generated.

Another SET card option, LINE NUMBER, is used to specify a line longer than 120 characters. During compilation, a count is kept of the number of characters indicated in the FORMAT statement. If the count exceeds 120, a warning message is issued. If a printer with a 132-character line is being used, the LINE NUMBER option allows the normal 120-character line to extend to 132 characters.

OPERATION: The CCSET routine checks the first option field. If the field is blank, a return is made in the CCLASS routine to read the next card. If the field is not blank, a check is made for a line count. If the field contains a line count, the number is converted to binary and placed in the communications area. If the line count is not specified, the data set reference number on the card is stored, and the device assignment table in main storage is searched for that data set reference number. If the number is not found, the new data set refernce number is invalid and an error message is produced.

The next option field is examined. If this field is blank, the routine returns to the CCIASS routine to read another card. If the field is not blank, checks are made for a line number and/or data set reference number, and the information is processed as previously described. If the word LINE is misspelled, the routine assumes that the field contains a data set refernce number. If the assumed number is not found in the device assignment table, the misspelled word is treated as an invalid data set refernce number, and an error message is produced.

EXIT: When a blank field is encountered, indicating that all specifications have been processed, the CCSET routine returns control to the CCLASS routine.

CCLOAD Routine: Chart AX

The CCLOAD (LOAD Control Card) routine interprets information on the LOAD control card and transforms that information into directives for the CCLOAD routine.

ENTRANCE: The CCLOAD routine receives control form the CCLASS routine when a LOAD control card is encountered.

CONSIDERATION: The information on the LOAD control card is scanned by the CCLOAD routine to determine certain object-time information.

OPERATION: The LOAD control card is scanned by the CCLOAD routines to determine any specified option or options. Appropriate indicators are set in the communications area to indicate the specifed object-time information.

EXIT: The CCLOAD routine passes control to the FSD after any specified options have been processed. The FSD either terminates the job or calls the FORTRAN loader, depending upon the condition of the GO or NOGO flags.

CCEDIT Routine: Chart AY

The CCEDIT (EDIT Control Card) routine allows permanent changes in certain aspects of the system tape as contrasted to temporary alterations in system conditions caused by the CCSET routine. The CCEDIT routine accomplishes this by supplying user information on altered conditions to the editor.

ENTRANCE: The CCEDIT routine receives control from the CCLASS routine when an EDIT control card is encountered.

OPERATION: The EDIT control card causes the FORTRAN System Director to search the existing system tape for the editor, load the editor into main storage, and then pass control to the editor. Other than its identification, the EDIT control card is not examined in this routine.

EXIT: The CCEDIT routine is terminated when control is passed to the editor.

CCUNIT Routine: Chart AY

The CCUNIT (UNITS Control Card) routine prints out a description of the device assignment table indicating the logical unit number and its associated address.

ENTRANCE: The CCUNIT routine receives control from the CCLASS routine when a UNITS control card is encountered.

OPERATION: The UNIT control card indicates that a description of each of the 16 units used is to be printed with appropriate heading information. This printing occurs each time a UNIT control card is encountered.

EXIT: The CCUNIT routine returns control to the CCLASS routine.

CCDATA Routine: Chart AZ

The CCDATA (DATA Control Card) routine recognizes the end of the input data set for the compiler and determines if an execution is to be performed. It prepares directives for the CCLOAD routine or for the FORTRAN System Director job termination, accordingly.

ENTRANCE: The CCDATA routine receives control from the CCLASS routine when a DATA control card is encountered.

CONSIDERATIONS: The DATA control card indicates that there are no more cards to be processed by the CCLASS routine. Further, the DATA control card immediately precedes any data cards the user may have accompanying his program.

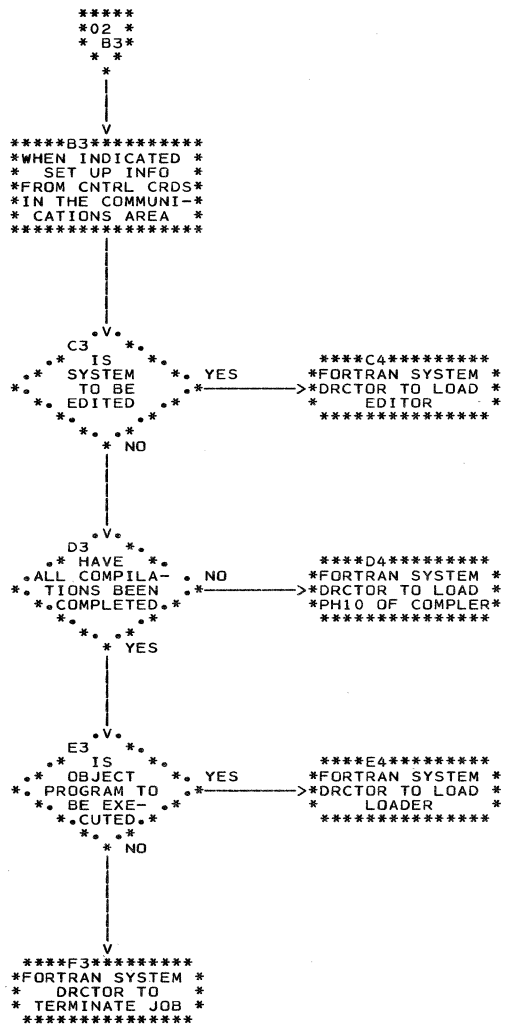
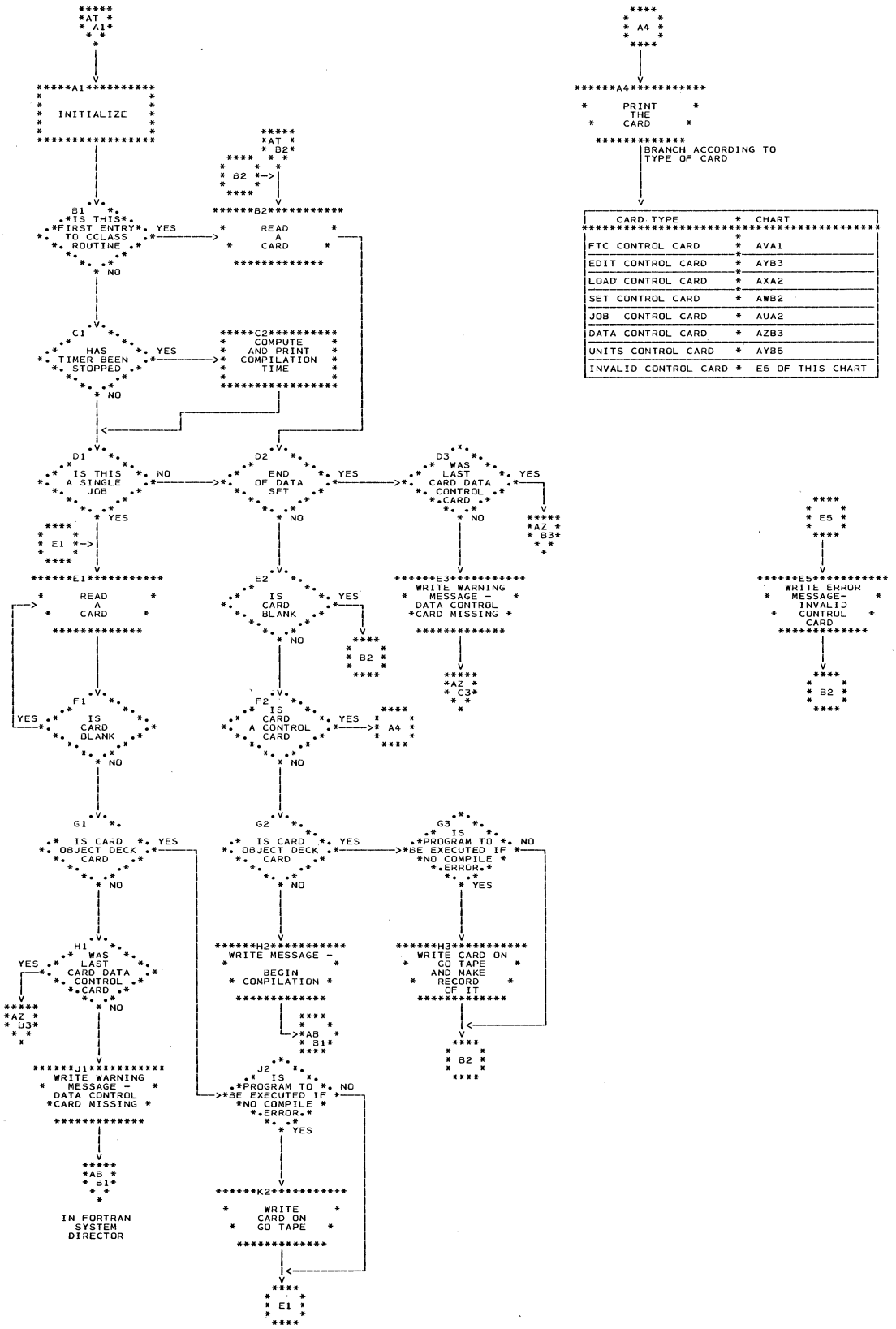


Chart 02. Control Card Overall Logic Diagram



CARD TYPE	CHART
FTC CONTROL CARD	AVA1
EDIT CONTROL CARD	AYB3
LOAD CONTROL CARD	AXA2
SET CONTROL CARD	AWB2
JOB CONTROL CARD	AUA2
DATA CONTROL CARD	AZB3
UNITS CONTROL CARD	AYB5
INVALID CONTROL CARD	E5 OF THIS CHART

Chart AT. CCLASS Routine

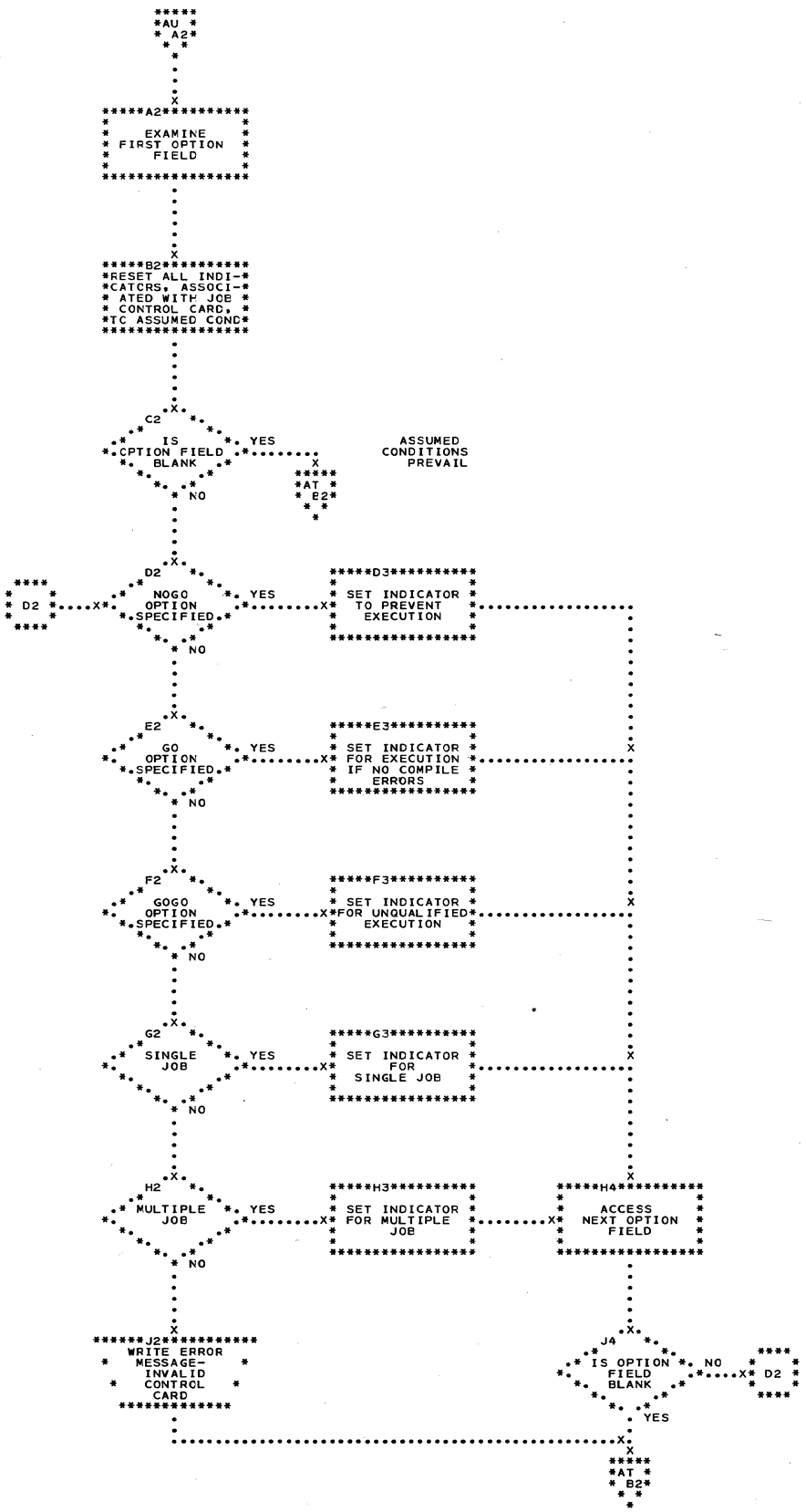


Chart AU. CCJOB Routine

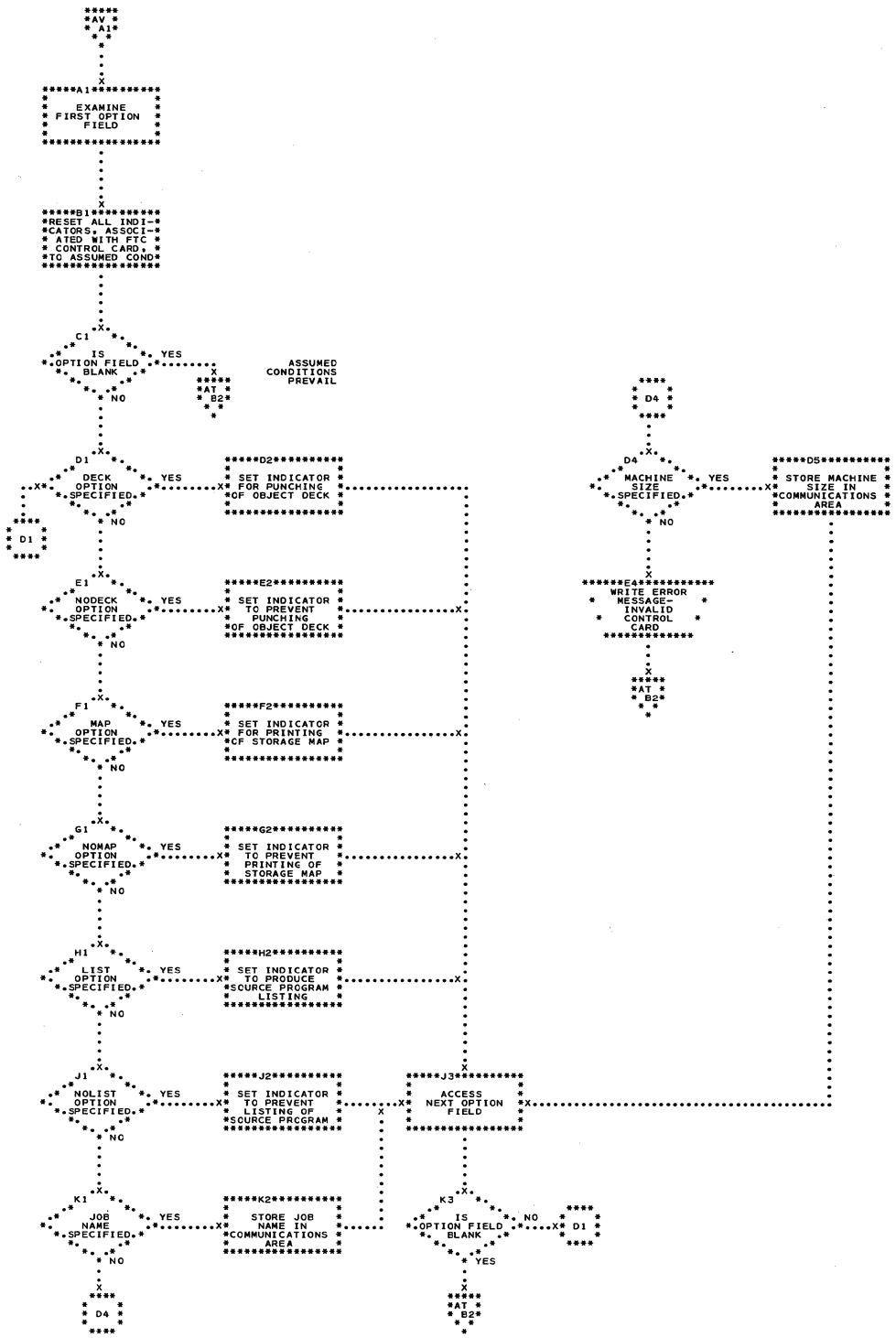


Chart AV. CCFTC Routine

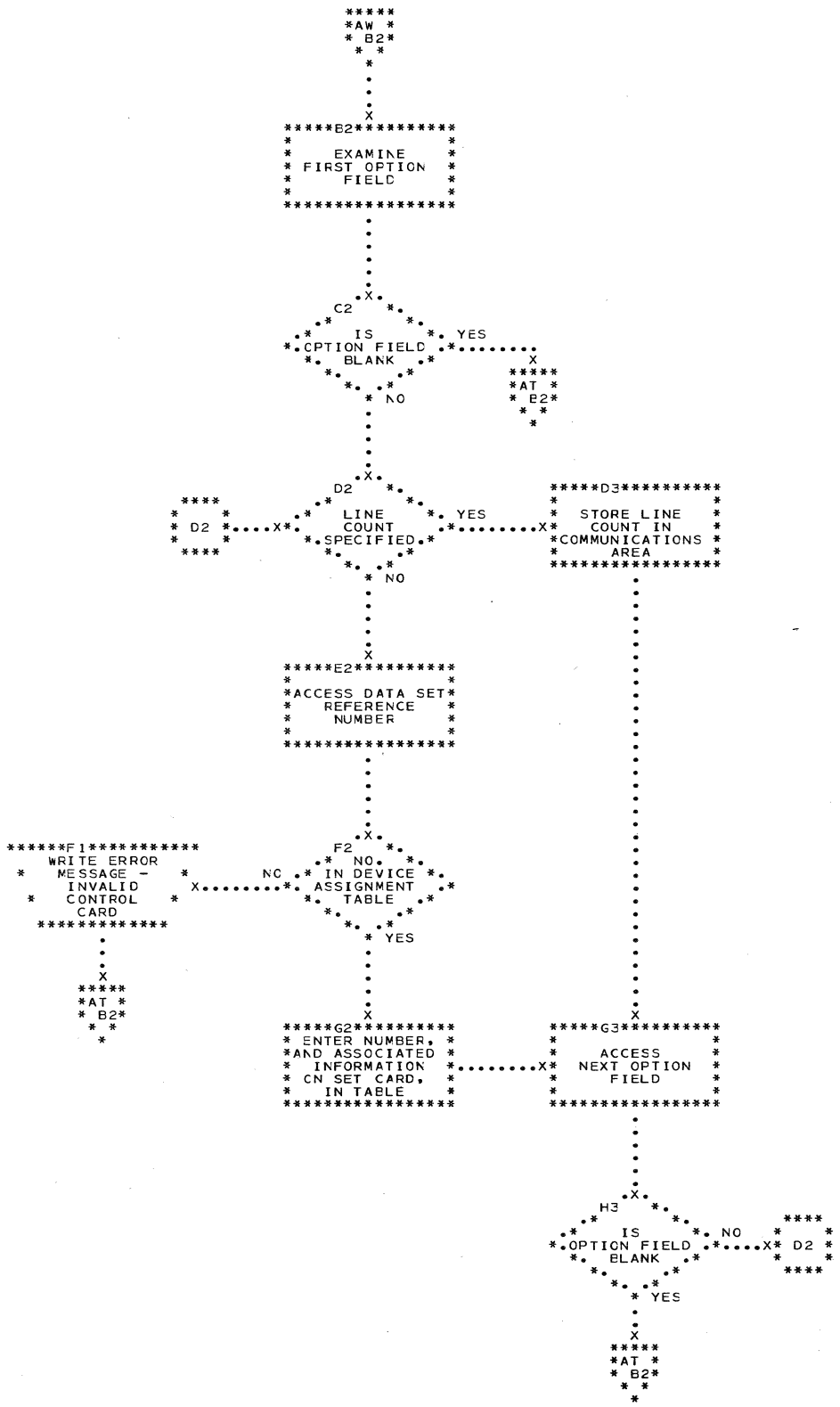


Chart AW. CCSET Routine

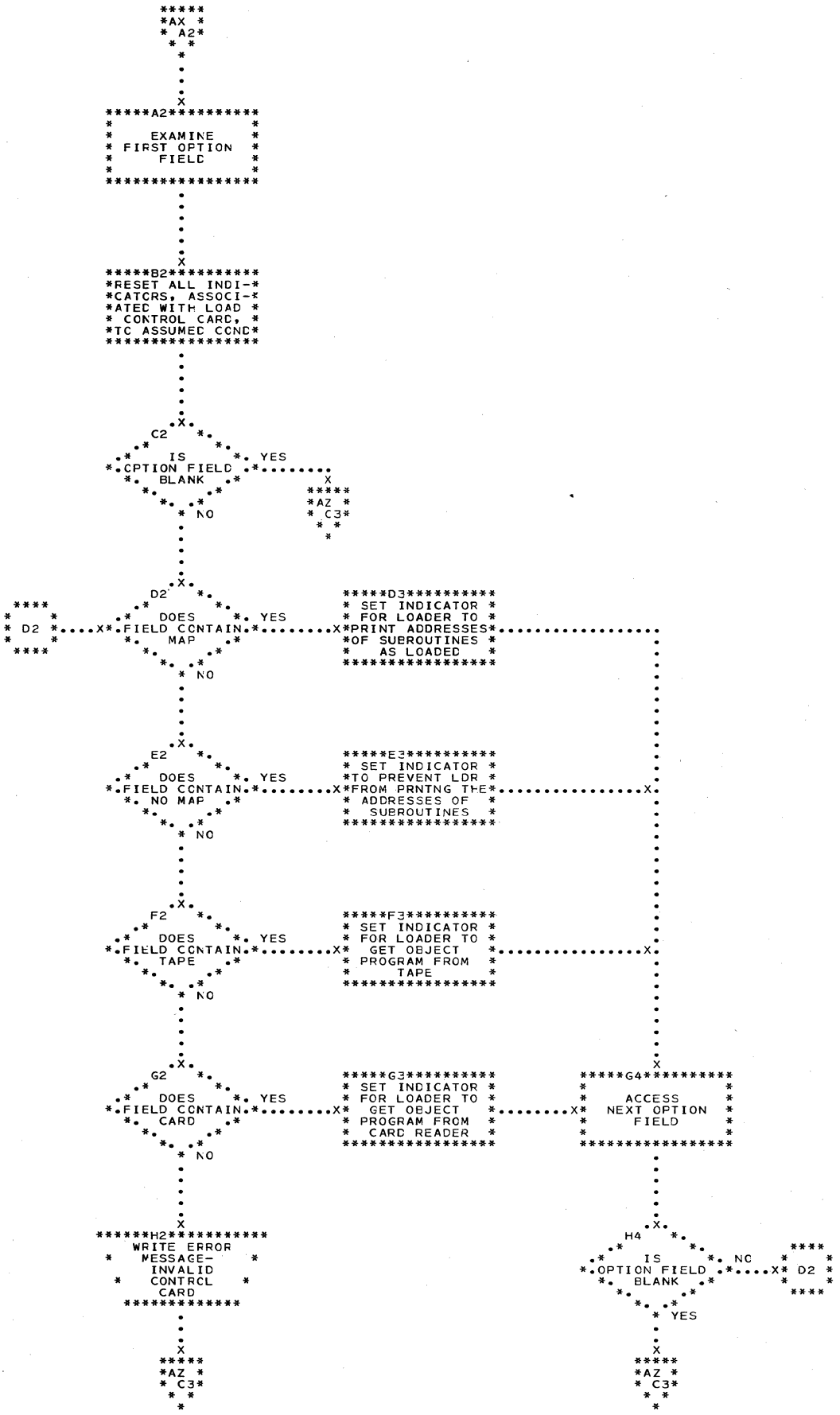


Chart AX. CCLOAD Routine

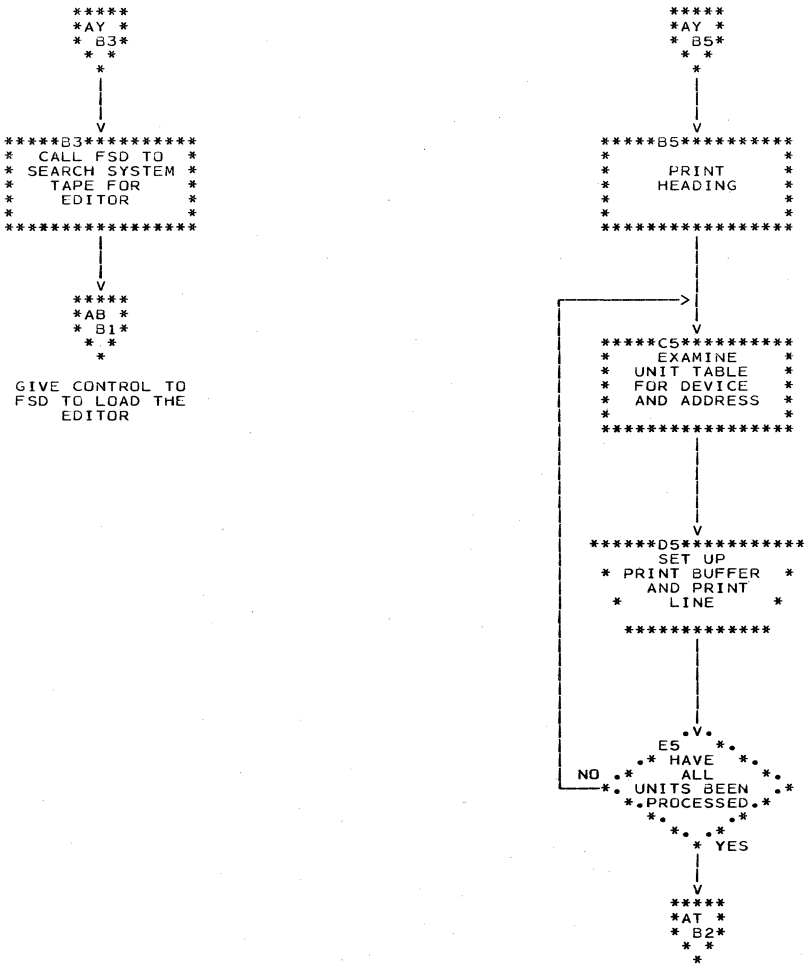


Chart AY. CCEDIT and CCUNIT Routines

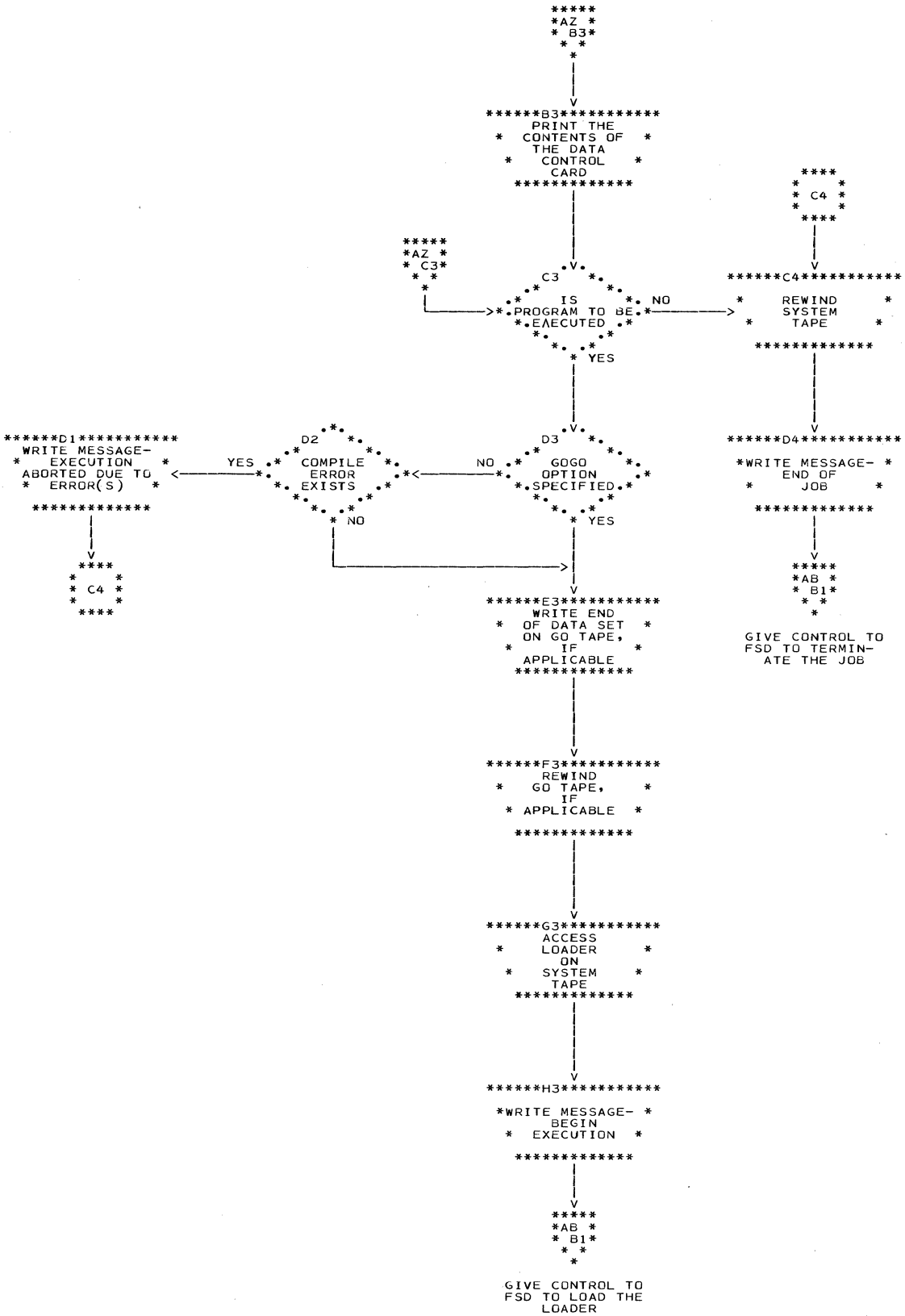


Chart AZ. CCDATA Routine

Source programs written in the IBM System/360 Basic Programming Support FORTRAN IV language are compiled by the segments on the system tape that constitute the Basic Programming Support FORTRAN compiler.

The FORTRAN compiler analyzes the source program statements and transforms them into an object program compatible to IBM System/360. In addition, if any source program errors exist, the FORTRAN compiler produces appropriate messages. At the user's option, a complete listing of the source program is produced and/or an object deck is punched.

The compiler segments consist of the two control segments discussed in Part 2 and the seven phases (10, 12, 14, 15, 20, 25, and 30) to be discussed in this part of the manual.

Phase 10 converts FORTRAN source statements into input for subsequent phases of the Basic Programming Support FORTRAN compiler. This input consists of intermediate text, the dictionary, overflow table, COMMON text, and EQUIVALENCE text.

Chart 03, the Phase 10 Overall Logic Diagram, indicates the entrance to and exit from Phase 10 and is a guide to the overall functions of the phase.

Intermediate text provides a format that can be easily converted to machine language instructions. This conversion requires coded information about variables, constants, arrays, statement numbers, in-line functions, and subscripts. This coded information, derived from the source statements, is contained in the dictionary and overflow table and referenced within the intermediate text.

The COMMON text is a table of variables which are assigned to the COMMON area by the source program in COMMON statements. The EQUIVALENCE text is a table of EQUIVALENCE groups assigned by EQUIVALENCE statements. The COMMON and EQUIVALENCE text contain references to the dictionary.

Each FORTRAN statement is classified as either a keyword statement, arithmetic statement function, or arithmetic statement.

The first symbol in the FORTRAN statement is checked against a list of keywords contained in the dictionary. If this symbol is in the dictionary, control is passed to a subroutine whose address is in the dictionary with the keyword. The keyword subroutine makes entries to the intermediate text to indicate that this statement requires special processing.

After these entries have been made, control is passed to either an arithmetic subroutine which processes arithmetic expressions or a subroutine which gets the next source statement.

If the FORTRAN statement does not begin with a keyword, Phase 10 determines whether the statement defines an arithmetic statement function. If it does, control passes to a subroutine which makes special entries to the intermediate text and dictionary for that statement function. Control is returned to the arithmetic subroutine which processes the arithmetic expression in the statement.

If the FORTRAN statement neither begins with a keyword nor defines an arithmetic statement function, it is an arithmetic statement. Control is passed directly to the arithmetic subroutine that makes the necessary entries for an arithmetic statement to the intermediate text, dictionary, and overflow table.

The errors checked in Phase 10 are only flagged in the intermediate text. No error messages are transmitted to the operator during Phase 10.

CHAINING

The technique used by the FORTRAN compiler to arrange and retrieve items entered in the dictionary and overflow table is called chaining. Items are chained so less time is required to locate the necessary information.

A chain is composed of a number of related entries. Each entry consists of an item and its related fields. One specific field within each entry points to some related entry, but not necessarily to the one that is physically adjacent in storage (See Figure 15).

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
ENTRY	A	AB	C	DC	EFG	HIJ	NO	POR	X
POINTER	3	4	9	7	6	8			

Figure 15. Example of Chaining

The lower line in Figure 15 is a pointer to the next entry in the chain. For example, entries 1, 3, and 9; 2, 4, and 7; and 5, 6, and 8 form separate chains. The common characteristic of these chains is that each item in the specific chain is composed of the same number of characters. Items can be grouped and entered on a separate chain by any characteristic (e.g., alphabet, length, or number) that would divide the table. Division of a long chain into several smaller chains saves time that would be used to search one long chain.

The thumb index, which contains the addresses of the first entry for each

chain, is directly related to the characteristic that separates items into chains. There is an entry in the thumb index for each characteristic that defines a separate chain. By first determining the characteristic of an item, that item is found or placed in the proper chain by use of the thumb index.

DICTIONARY

The dictionary contains names, constants, and data set reference numbers. A name is a string of alphabetic and numeric characters, the first of which must be alphabetic. A name can be any of the following:

1. Variable. In the statement:

   ALPHA=BETA+GAMMA-2.0-X123X

   the variables ALPHA, BETA, GAMMA, and X123X are names.

2. Keyword. In the statement:

   DIMENSION A (10,5)

   the keyword DIMENSION is a name.

3. Array. In the preceding statement, A is the name of the array.

4. In-line function. In the following statement:

   A=ABS (B)

   the in-line function ABS is a name.

In the statement:

REWIND J

J is entered into the dictionary as a variable name, not as a data set reference number.

In the statement:

REWIND 3

3 is a data set reference number. The compiler distinguishes a data set reference number from an integer constant by the context in which it is used. For example, in the statement:

I = I-3

3 is an integer constant because it is used in an arithmetic expression.

Operation

The dictionary is organized as a series of 15 chains and a thumb index. Each address in the thumb index points to the beginning of a different chain. There are 11 chains organized on the basis of name length. For example, all names with a length of one Binary Coded Decimal (BCD) character are placed in the first chain, all names with a length of two BCD characters in the second chain, and so on.

Keywords and in-line functions are names and the dictionary includes them as permanent residents of their respective chains. Keywords and in-line functions are present when the dictionary is first established in main storage. Names assigned by the user are placed in their respective dictionary chains as the source program is processed by Phase 10.

Chains 7 through 11 are reserved for keywords that range in length from 7 through 11 characters, (e.g., FUNCTION, DIMENSION, EQUIVALENCE, etc.). No user name is placed in these chains.

The four remaining chains in the dictionary are used for real, integer, and double precision constants and data set reference numbers; each has its own chain.

The search for a constant or data set reference number entry in the dictionary is accomplished by determining what the symbol is. If the symbol is a constant, Phase 10 determines the mode (real, integer, or double precision) and finds the proper address in the thumb index. This address directs Phase 10 to the beginning of the correct chain. If the symbol is a data set reference number, the address in the thumb index takes the compiler to the data set reference number chain. After the correct chain is determined, the compiler can follow the chain addresses in the dictionary to search for the correct entry.

DICTIONARY ENTRY: Each dictionary entry contains from five to seven fields. The address and size fields are optional (see Figure 16).

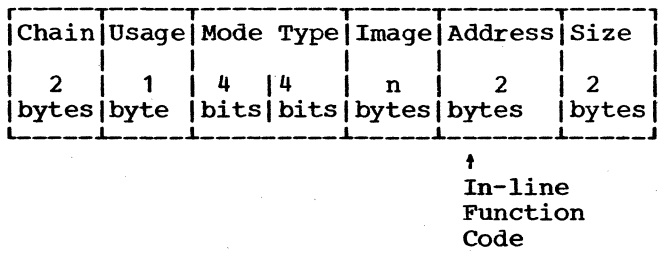


Figure 16. Dictionary Entry Format

Chain: During Phase 10 this field contains the address of the next entry in the chain. The value 0001 in this field indicates the last entry in the chain. By following the chain, a search is made to see if there is a dictionary entry for the current item. If no dictionary entry is made, one is assembled for this item and appended to the proper chain.

An illustration of chaining in the dictionary is shown in Figure 17. All keywords and in-line function names are the first entries in their respective chains and precede names assigned by the user. In the chain for length 2, the keywords IF, GO, and DO precede any entry of names assigned by the user. The thumb index for the length 2 chain points to the entry for IF. The chain address for the IF entry points to the entry for GO.

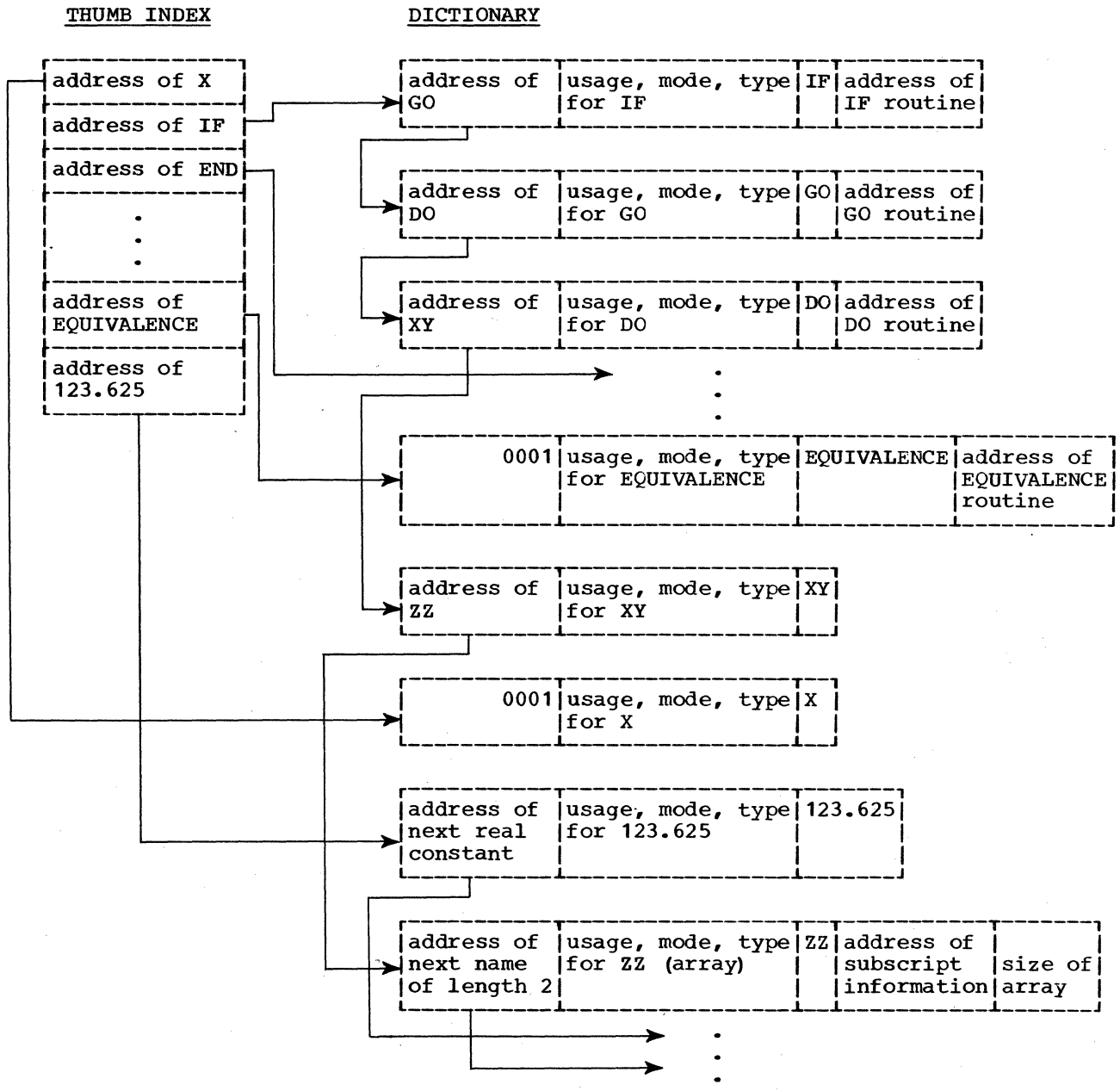


Figure 17. Dictionary and Thumb Index Format

Usage: The usage field indicates characteristics of the symbol. Usage is discussed later in this section.

Mode: The mode field contains a hexadecimal character denoting integer, real, or double precision mode. The codes for these modes are:

5 : integer
6 : double precision
7 : real

Type: The type field contains a hexadecimal character denoting a constant, array, function, or variable. Refer to Figure 25 for the code for these items. The mode and type codes are contained in one byte and used together in most cases.

Image: The image field is the BCD card image of the symbol. It ranges in length from 1 through 11 characters.

Address: This optional field contains the address of a subroutine if the symbol is a keyword. It may also be a pointer to the dimension entry in the overflow table if the symbol is an array. If the symbol is an in-line function, the first byte of the address field contains the code for the particular in-line function. The second byte of the address field is not entered for in-line functions.

Size: This optional field is used only for arrays and contains the size, in bytes, of an array. This size is found by multiplying the dimensions of the array by the length, in bytes, of each item. The length is 4 for real or integer mode and 8 for double precision mode.

The fields in a dictionary entry contain the mode/type, address, and size associated with a symbol, plus the symbol itself. Still, if the compiler is to produce machine language programs, other information is necessary.

The usage field contains a bit code to indicate characteristics of each item to the compiler. (See Figure 18).

Usage Field Bits	Condition	Bit Status
0	Mode not defined	0
	Mode has been defined	1
1	Type not defined	0
	Type has been defined	1
2	Variable not in COMMON	0
	Variable is in COMMON	1
3	Variable not equated	0
	Variable is equated	1
4	Not used in Phase 10	
5	Root Indicator for Equate	1
6	No Double Precision	0
	Double Precision	1
7	Punch ESD Card	1

Figure 18. Format of Usage Field

Bit 0 indicates the mode of a symbol has been defined. The mode of a symbol is defined only when:

1. The name is mentioned in an explicit mode definition statement.
2. The name is entered for the first time into the intermediate text.

Any time a variable is used in a FORTRAN statement, its mode is determined and a mode code inserted in the dictionary. If the mode has not been defined, it may change. The mode cannot be redefined if bit 0 has been set to 1. When the symbol is encountered again, its entry is found in the dictionary and the mode bit is checked. Assume the following statements occur in sequence:

```
REAL A, B, C,
   .
   .
   .
INTEGER I, J, A
```

In the first statement, A is explicitly defined as a real symbol. In its dictionary entry, the mode field contains the code for real. Bit 0 in the usage field is set to 1, indicating that the mode has been defined. The second statement attempts to redefine A as an integer. The mode bit (bit 0) is again tested to determine if the mode has been defined previously. Because it has been defined, an error condition is noted.

Bit 1 indicates whether the symbol type has been defined. Type is defined when:

1. An array is defined by a DIMENSION statement.
2. The names in COMMON or explicit specification statements are dimensioned.
3. A name is included in an EXTERNAL statement.
4. A subprogram name is defined in a SUBROUTINE or FUNCTION statement; the type for dummy variables in these statements is not defined.
5. A variable is entered for the first time in the intermediate text.

Assume the following statements occur in sequence:

```
X = A(I,B)
    .
    .
    .
XYZ = A
```

In the first statement, A is defined as the name of a FUNCTION subprogram. If this is the first time A is encountered in the program, the code for a FUNCTION subprogram is inserted in the type field of its dictionary entry. At the same time, bit 1 is set to 1 indicating that the type of A has been defined. The second statement indicates that A is a variable. Because bit 1 is set to 1, Phase 10 does not attempt to redefine A, but merely uses the type code that was established in the first statement. An error condition does exist because the program attempts to use A as both a FUNCTION subprogram name and a variable. The error condition, however, will not be noted until Phase 15.

Bit 2 indicates whether the variable is in the COMMON area. This bit is required for Phase 12 when storage is allocated.

Bits 3, 5, and 6 are not set during Phase 10. They are set and used by Phase 12 when EQUIVALENCE and COMMON statements are processed. Bit 6, the double precision bit, is set only for equated variables. The function and operation of these bits is explained in Phase 12.

Bit 7 is set to 1 by Phase 10 for symbols used as in-line functions or external references. If bit 7 is set to 1 and the type code denotes an external symbol, an ESD card is punched in Phase 12. ESD cards are not punched for in-line functions.

OVERFLOW TABLE

The overflow table produced by Phase 10 contains dimension, subscript, and statement number information.

Operation

The overflow table is constructed with the same chaining technique as the dictionary. The overflow table is composed of 11 chains. Three chains are reserved for array information; the first chain contains all arrays with one dimension, the second with two dimensions, and the third with three dimensions. Three additional chains are reserved for subscripted variables; the first chain contains information for all variables with one subscript, the second with two subscripts, and the third with three subscripts.

The last five chains contain statement number information. All statement numbers ending in 0 and 1 are contained in the first chain. The remaining chains contain statement numbers ending in 2 and 3, 4 and 5, 6 and 7, and 8 and 9, respectively.

Dimension Information

The format for these entries, (see Figure 19) is different from the dictionary format. The general form for defining 1-, 2- and 3-dimensional arrays is:

```
ARRAY (D1)
ARRAY (D1,D2)
ARRAY (D1,D2,D3)
```

where D1, D2, and D3 are integer constants.

The dimension information for 1-, 2-, and 3-dimensional arrays is:

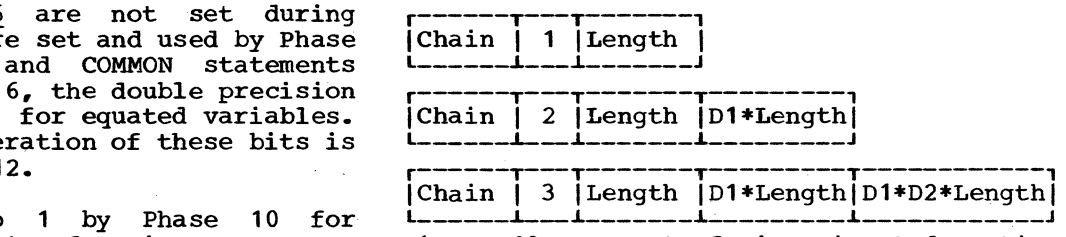


Figure 19. Format of Dimension Information in Overflow Table

Every entry made in the overflow table for dimension information has a chain field with the same function as the chain field in the dictionary. It links the entries with the chain. The second field in a dimension entry contains the number of dimensions in that array. A 1-dimensional array has the number 1 in this field. The third field contains the length of each element in the array. If the entries in an array are double precision, this field contains the number 8 because a double

precision number is exactly 8 bytes. If the array is real or integer, this field contains the number 4. These three fields are the only entries for 1-dimensional arrays, but are entered for all arrays.

For 2- and 3-dimensional arrays, another field is added. D1 represents the first dimension. The product, D1*Length, is an indexing factor used in the later phases of the compiler and in the object program. The use of this factor is explained in Appendix C. If a real array is defined with the statement:

```
DIMENSION A (20,10)
```

this field contains the product 80 (4*20 = 80). The length of a real number is 4; the first dimension is 20.

If the array is 3-dimensional, an additional field, D1*D2*Length is added. This field is another indexing factor used in later compiler phases and the object program. If the array is A(20, 10, 5) and is again composed of real numbers, this field contains the number 800 since D2 represents the second dimension.

When a DIMENSION or explicit specification statement that defines an array is read from the source deck, Phase 10 makes entries to both the dictionary and the overflow table. The name of the array is entered in the dictionary along with a pointer to an entry in the overflow table. The size of the array is entered and the type code is set to represent an array or a dummy array.

Assume the name ARRAY is defined as real and the statement:

```
DIMENSION ARRAY (4,3,2)
```

is read. Phase 10 makes entries in the dictionary and overflow table, as illustrated in Figure 20.

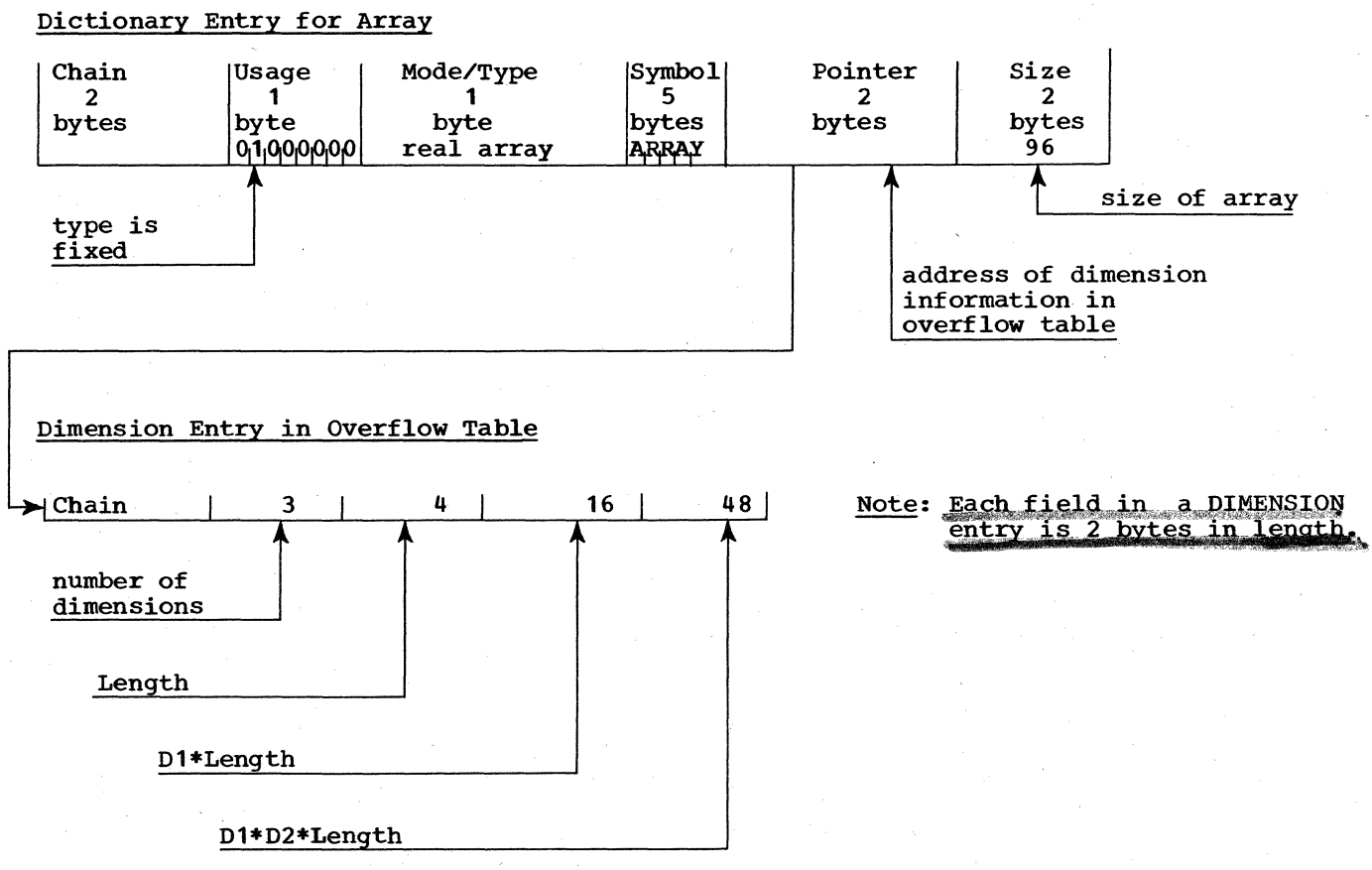


Figure 20. Entries to Dictionary and Overflow Table

Subscript Information

The second type of information entered in the overflow table is subscript entries for subscripted variables (see Figure 21). Each field is two bytes in length. These subscript variables can be in any one of the following forms, for 1-, 2-, and 3-dimensional variables, respectively.

```
VAR (C1*V1+J1)
VAR (C1*V1+J1,C2*V2+J2)
VAR (C1*V1+J1,C2*V2+J2,C3*V3+J3)
```

In the general form above, C1, C2, C3, J1, J2, and J3 are integer constants; V1, V2, and V3 are integer variables. VAR is any array defined either by a DIMENSION, COMMON, or explicit mode specification statement.

The entries in the overflow table bear a resemblance to the format in a subscript.

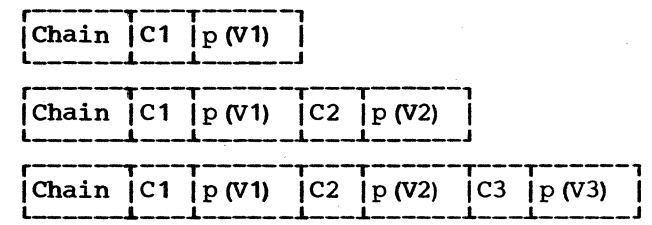


Figure 21. Format of Subscript Information

The symbols, p(V1), p(V2), and p(V3) represent pointers to the integer variables, V1, V2, and V3, which are entered in the dictionary. The offset, a constant indexing factor used to find the correct element in an array for a particular subscript expression, is computed using the integer constants J1, J2, and J3 and is placed in text. (Refer to Appendix C for an understanding of Array Displacement Calculation.) These constants are not entered in the dictionary or the overflow table.

Assume the subscripted variable, ARRAY (2,2*I-1,J) is encountered in a source statement. Furthermore, assume that the names ARRAY, I, and J have already been entered in the dictionary, and ARRAY is defined as DIMENSION ARRAY (4,3,2). For the subscripted variable ARRAY (2,2*I-1,J), the following entry (see Figure 22) is made to the overflow table in Phase 10.

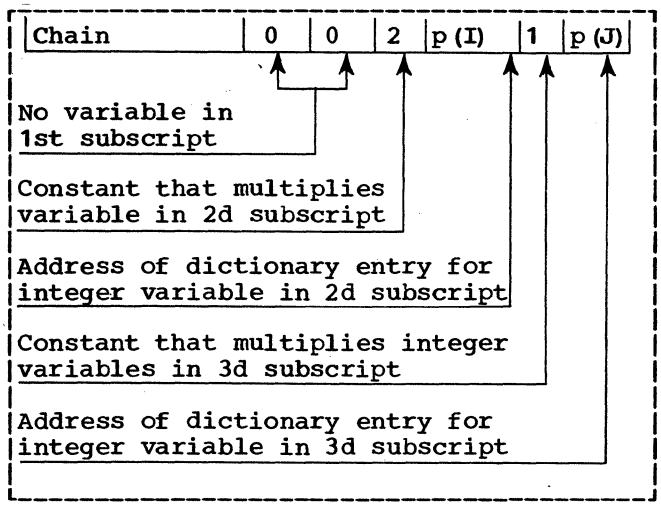
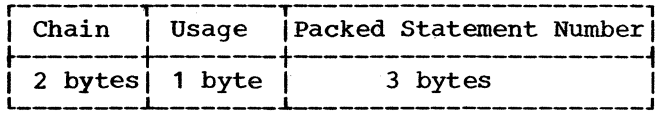


Figure 22. Overflow Table Entry

Only subscripts that contain at least one integer variable in a subscript parameter are entered in the overflow table. No integer variable is used to compute the first subscript parameter; consequently, the entries referring to the first subscript parameter in the overflow table are both zero. Notice that the names for the integer variables in the second and third subscript parameters are not included, but the addresses of their dictionary entries are inserted in the entry. If the subscripted variable is ARRAY (2,1,1), the indexing is completely taken care of by the offset and no entry is made to the overflow table.

Statement Number Information

The third type of entry made to the overflow table is for statement numbers. Any statement number encountered in the source statements is entered in the overflow table. The format of the entry is:



The statement number is obtained from the source statement and its Extended Binary Coded Decimal Interchange Code (EBCDIC) format is changed to the packed decimal format. A search is made of the proper chain. The first time the statement number is encountered, an entry is made in the overflow table and certain bits are set in the usage field (see Figure 23). The

usage field is primarily used for error checking in Phases 12 and 14.

Usage Field Bits	Condition	Bit Status
0	Statement No. undefined Statement No. defined	0 1
1	Statement No. not referenced Statement No. referenced	0 1
2	End DO	1
3	Statement No. of specification (e.g.; DIMENSION)	
4	Statement No. of FORMAT	1
5	Statement No. denotes DO nesting errors	1
6	Not used in Phase 10	
7	Not used in Phase 10	

Figure 23. Statement Number Information in Usage Field

Bits 0 and 1 denote whether the statement number is defined by a statement, and if the statement number is referenced, respectively. The statement:

```
112 A=B
```

sets bit 0 to 1. It has no effect on bit 1. The statement:

```
GO TO 112
```

sets bit 1 to 1. It has no effect on bit 0.

Bit 2 is the indicator set to define the end of a DO loop. This bit is set for error checking. GO TO, COMPUTED GO TO, PAUSE, RETURN, STOP, IF, FORMAT, and DO statements are not permitted to end a DO loop. This condition is checked in Phases 10 and 14. Bit 2 is also used in later phases to check for proper nesting of DO loops.

Another statement number error checked in Phase 10 is a "backward DO" (i.e., the statement ending the DO loop is sequentially in front of the statement that defines the DO). The program would be written as follows:

```
10 CONTINUE
   .
   .
   .
DO 10 I=I,1000
```

The statement that defines the DO loop follows the statement that is supposed to end the loop. If Phase 10 tries to set bit 2 to 1 (denoting an END DO) and bit 0 is set to 1 (denoting that the statement number has been defined) an error exists.

Bit 3 is set to 1 to indicate that this statement number defines a specification statement.

Bit 4 indicates the statement number of a FORMAT statement. If the statement number defines a FORMAT statement, bit 4 is set to 1. No statement except a FORMAT statement will set this bit to 1.

Bit 5 is set by Phase 15 to indicate DO nesting errors.

Bits 6 and 7 are not used.

OFFSET CALCULATIONS

The offset, a constant, is computed by Phase 10 and used as an indexing factor by Phase 25. The offset is not entered in the dictionary or overflow table. It is computed using the following formulas:

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

for one, two, and three subscripts, respectively.

Length is the length of each element in the array. If the elements of the array are integer or real, Length equals four. If they are double precision, Length equals eight. Assume ARRAY is dimensioned as ARRAY (4,3,2) and is real, therefore its Length is four. Then, the offset computation for the subscripted variable ARRAY (2,2*I-1,J) is:

```
Offset = [(2-1) + (-1-1) *4+ (0-1) *4*3] *4
Offset = [1+ ( (-2) *4) + ( (-1) *12) ] *4
Offset = [1-8-12 *4]
Offset = [-19] * 4
Offset = -76
```

In the example, ARRAY (2,1,1) with ARRAY composed of real numbers, the offset is different even though the two subscripts refer to the same array.

```
Offset = [(2-1) + (1-1) *4+ (1-1) *4*3] *4
Offset = [1+0+0] *4
Offset =  4
```

The offset is contained in the intermediate or EQUIVALENCE text. The offset is used then in the computation of an indexing factor to find the correct element in the array for a particular subscript expression.

Intermediate Text

Intermediate text is written in Phase 10 as input to the other phases of the Basic Programming Support FORTRAN compiler. The format for the intermediate text consists of three fields which contain an adjective code, a mode/type code, and a pointer to information in the dictionary or overflow table.

The following example illustrates the intermediate text entry format:

Adjective Code	Mode/Type Code	Pointer
1 byte	1 byte	2 bytes

The basic entry in the text is generally four bytes or one word long. This format is modified for the following special entries:

1. Subscripted variables.
2. FORMAT statements which do not conform to this basic entry and are discussed later in this phase.
3. Array size.
4. Number of arguments.
5. STOP or PAUSE statements.

The adjective code (see Figure 24) indicates the type of statement within the intermediate text. If the first symbol in a FORTRAN statement is a keyword, control is given to a subroutine which processes that statement. The keyword must be flagged so that subsequent phases can compile the correct machine language instructions. The subroutine that processes the keyword statement moves the adjective code for this keyword to the intermediate text.

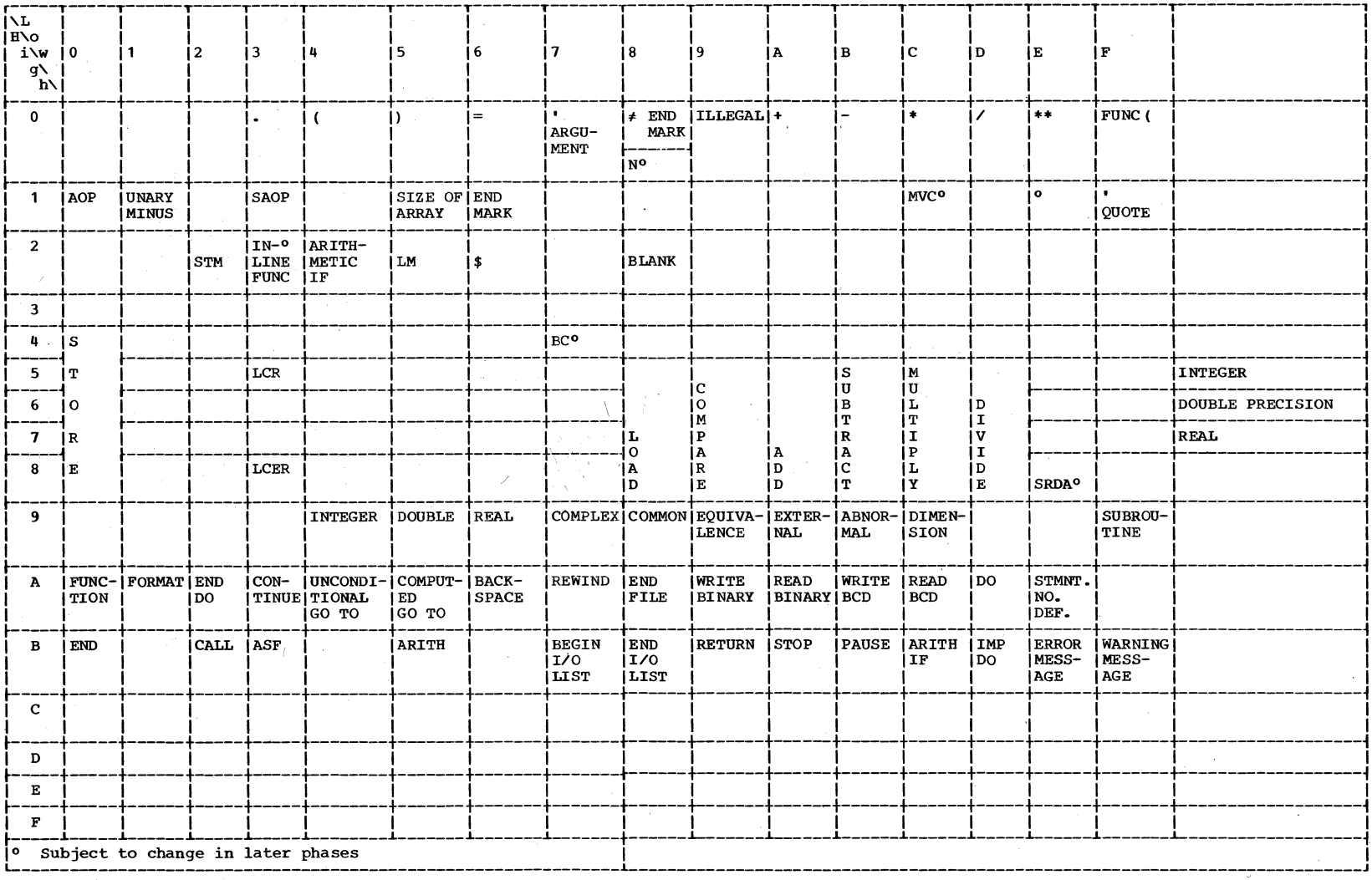
If the first symbol does not indicate a keyword statement, Phase 10 determines whether this statement defines an arithmetic statement function or an ordinary FORTRAN arithmetic statement and moves the proper adjective code to the intermediate text. Figure 24 contains the adjective codes and their use.

Adjective codes are also used to represent delimiters in a FORTRAN statement. Delimiters such as:

```
+  -  /  *  **  (  )  or  ,
```

are individually assigned a unique adjective code which denotes the type of operation to be performed.

The second byte in an intermediate text entry is the same as the mode/type code inserted in the dictionary to describe a symbol. The mode/type code (see Figure 25) denotes the mode of the symbol and the manner in which it is used.



\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	≠ END MARK / N°	ILLEGAL	+	-	*	/	**	FUNC (	
1	AOP	UNARY MINUS		SAOP		SIZE OF ARRAY	END MARK						MVC°		°	' QUOTE	
2			STM	IN-° LINE FUNC	ARITH-METIC IF	LM	$		BLANK								
3																	
4	S							BC°									
5	T			LCR						C		S U B T R A C T	M U L T I P L Y				INTEGER
6	O									O M P A R E	A D D			D I V I D E			DOUBLE PRECISION
7	R								L O A D								REAL
8	E			LCER											SRDA°		
9					INTEGER	DOUBLE	REAL	COMPLEX	COMMON	EQUIVA-LENCE	EXTER-NAL	ABNOR-MAL	DIMEN-SION			SUBROU-TINE	
A	FUNC-TION	FORMAT	END DO	CON-TINUE	UNCONDI-TIONAL GO TO	COMPUT-ED GO TO	BACK-SPACE	REWIND	END FILE	WRITE BINARY	READ BINARY	WRITE BCD	READ BCD	DO	STMNT. NO. DEF.		
B	END		CALL	ASF		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																	

° Subject to change in later phases

Figure 24. Adjective Code

Figure 25. Mode and Type Codes

TYPE CODE

MODE CODE (High \ Low)		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																	
1		STATEMENT NUMBER		UNIT			°IMMEDIATE CONSTANTS			°SUBPROGRAM	°DUMMY SUBPROGRAM						
2 3 4 5 6 7 8	5 FULL INTEGER 6 DOUBLE PRECISION 7 REAL					°GENERATED WORK AREA	°CONSTANT	°STATEMENT FUNCTION	°BUILT IN FUNCTION	°EXTERNAL & LIBRARY FUNCTION	°DUMMY FUNCTION	°VARIABLE	°DUMMY VARIABLE	SUBSCRIPTED VARIABLE	DUMMY SUBSCRIPTED VARIABLE	ARRAY	DUMMY ARRAY
9																	
A																	
B																	
C																	
D																	
E																	
F																	

° Subject to change in later phases

The third and fourth bytes in an intermediate text entry constitute an address which points to a symbol in the dictionary or to a statement number entry in the overflow table. These bytes may also contain an integer constant (less than 4096) of a DO statement and not the address of its dictionary entry.

For example, a typical entry for a FORTRAN arithmetic statement is the intermediate text for this statement:

```
CIRCUM=2.0*3.14159*RADIUS
```

Phase 10 would write the intermediate text shown in Figure 26.

Adjective Code	Mode/Type Code	Pointer
arithmetic statement (B5)	real variable (7A)	p (CIRCUM)
= (06)	real constant (75)	p (2.0)
* (0C)	real constant (75)	p (3.14159)
* (0C)	real variable (7A)	p (RADIUS)
end mark (16)	00	internal statement number

Figure 26. Format of Intermediate Text Entries

In Figure 26, the numbers in parentheses from the first two columns refer to the actual adjective and mode/type codes that appear in the intermediate text. The end mark entry indicates to other phases that this is the end of the intermediate text entry for this FORTRAN statement. The hexadecimal characters 00 represent a blank text entry. The items in the pointer field point to the dictionary. An internal statement number is assigned to each FORTRAN statement before it is processed.

If there are two delimiters together in a FORTRAN statement, zeros are used to fill the entries in the intermediate text. For example, in the statement:

```
A=-B
```

the intermediate text entries are as shown in Figure 27.

Adjective Code	Mode/Type Code	Pointer
arithmetic statement	real variable	p (A)
=	00	0000
unary -	real variable	p (B)
end mark	00	internal statement number

Figure 27. Intermediate Text Entries for a Unary Operation

Zeros are used to fill the second entry because no symbol follows the equal sign.

Statement Number Entries

Statement numbers are entered in intermediate text in a manner similar to entries for a variable in a statement (see Figure 10.14). When the statement is defined by a statement number, the statement number entry is entered in the intermediate text before any other entries are made for the statement itself. The statement:

```
101 A=B
```

is the data used to form the intermediate text shown in Figure 28.

Adjective Code	Mode/Type Code	Pointer
statement number	statement number	p (101)
arithmetic statement	real variable	p (A)
=	real variable	p (B)
end mark	00	internal statement number

Figure 28. Intermediate Text Entries for Statement Numbers

If a statement number is used in a statement itself, such as:

```
DO 101 I=M,N,3
```

the text entries are as shown in Figure 29.

Adjective Code	Mode/Type Code	Pointer
DO	statement number	p(101)
blank	integer variable	p(I)
=	integer variable	p(M)
,	integer variable	p(N)
,	immediate constant	3
end mark	00	internal statement number

Figure 29. Intermediate Text Entries for a DO Statement

The third parameter in this DO statement is the integer constant 3. The constant 3 is not entered in the dictionary, but is inserted in the address field of the intermediate text as the number 3. This is done to save dictionary space and to optimize instructions in the object program.

Subscripted Variable Entries

When a subscripted variable is used, the format of the intermediate text entries changes; three pointers are needed instead of the usual one. The second pointer points to the subscript information for the variable; the third points to dimension information for the array. For example, in the statement:

```
A(I,J,K)=B*2.0
```

the entries shown in Figure 30 are made to the intermediate text.

The first line contains the pointer to the dictionary entry for the subscripted variable A. In the second line SAOP is a special adjective code which is inserted in the intermediate text to indicate to other phases of the FORTRAN compiler that a subscript calculation is necessary. The pointer field in the second line contains the offset.

The third line contains two pointers to entries in the overflow table. The first points to the subscript information for this subscripted variable A; the second points to the dimension information for the array A. This information is necessary for processing subscripts in other phases.

Adjective Code	Mode/Type Code	Pointer
arithmetic statement	real subscripted variable	p(A)
SAOP	00	Offset
p(subscript A)		p(dimension A)
=	real variable	p(B)
*	real constant	p(2.0)
end mark	00	internal statement number

Figure 30. Intermediate Text Entries for Subscripted Variables

If the subscripts do not contain any variables, only the extra pointer to DIMENSION information is included. For example, the statement:

```
B=A(2,1,1)
```

where A is dimensioned as A(4,4,4), would be entered in the intermediate text as shown in Figure 31.

Adjective Code	Mode/Type Code	Pointer
arithmetic statement	real variable	p(B)
=	real subscripted variable	p(A)
SAOP	00	Offset
0000		p(dimension A)
end mark	00	internal statement number

Figure 31. Intermediate Text Entries for Constant Subscripts

After the initial calculation of the offset, no additional information is necessary because the offset represents a constant indexing factor. No pointer to subscript information is necessary because the

subscript information is used to calculate an indexing factor for variable subscripts.

Format Entries

Another change in the format of the intermediate text is caused by the FORMAT statement. Phase 10 makes little change to the FORTRAN card image of a FORMAT statement. Every FORMAT statement must have a statement number, which is converted to intermediate text and an entry for it must be made to the overflow table. The keyword FORMAT is then encountered and control is given to the keyword subroutine which processes FORMAT statements for Phase 10. This subroutine:

1. Inserts the adjective code for a FORMAT statement in the intermediate text.
2. Gets the entire card image, excluding the statement number and identification field (columns 73 through 80), and the EBCDIC image for that card in the intermediate text.
3. Inserts the EBCDIC image for that card in the intermediate text.

An end mark entry is then made with an internal statement number in the pointer field.

The statement:

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

would then produce the intermediate text illustrated in Figure 32.

Adjective Code	Mode/Type Code	Pointer	
statement number	statement number	p(12)	
FORMAT	(	F	2
0	.	5	,
I	6	)	blank
ALL CARD COLUMNS TO COLUMN 72			
end mark	00	internal statement number	

Figure 32. Intermediate Text Entries for a FORMAT Statement

The FORMAT information is held in BCD characters for the intermediate text. All of the characters on the FORMAT card, immediately after the keyword FORMAT through card column 72, are moved to intermediate text.

Errors

Any errors or warnings detected in Phase 10 are flagged in the intermediate text. The second byte in the end mark entry in the intermediate text is reserved for this action. If Phase 10 detects an error in a statement, the hexadecimal 01 is inserted in this byte. The next entry in the intermediate text contains an error/warning adjective code (see Figure 33). An error number is placed in the mode/type field by a general error subroutine. The pointer field contains the same internal statement number as the end mark entry.

Adjective Code	Mode/Type Code	Pointer
end mark	01	internal statement number
error code	error number	internal statement number

Figure 33. Intermediate Text Entries for an Error

Internal Statement Numbers

Phase 10 assigns an internal statement number to each FORTRAN statement as it is read into main storage. This number, which differs from the user statement number, is assigned whether or not intermediate text is written for this statement. If the statement is DIMENSION, REAL, INTEGER, DOUBLE PRECISION, or EXTERNAL, no intermediate text is written. However, these statements are assigned an internal statement number, and gaps may exist in the internal statement numbers of the intermediate text. Similarly, if an error occurs, two successive entries in the intermediate text may have the same internal statement number.

Intermediate Text Output

The intermediate text is written on a tape output data set and used as input to subsequent phases of the FORTRAN compiler. The buffer size is computed in the Control

Card routine. Both the dictionary and the overflow table remain in main storage for subsequent phases. The overflow table begins with the highest available location and extends down toward low addressed main storage. The dictionary origin depends on the size of the buffers and extends up toward the overflow table.

COMMON and EQUIVALENCE Text

For COMMON and EQUIVALENCE statements, Phase 10 writes another type of text which remains in main storage to be processed by Phase 12. The following COMMON text is composed of two fields, each two bytes in length for each variable entered in the COMMON area at object time.

2 bytes	2 bytes
pointers	length

The first field contains the address of the dictionary entry for that variable, and the second field contains the length of its name in EBCDIC characters. For example, the statement:

```
COMMON A,B,CON4Z
```

would cause this COMMON text:

Pointer	Length
p(A)	1
p(B)	1
p(CON4Z)	5

The length is needed to determine in what chain the variable is entered in the dictionary.

A FUNCTION or SUBROUTINE subprogram name must be defined in the first card of a FORTRAN source program. The Phase 10 subroutine for processing FUNCTION and SUBROUTINE statements is not required after the first card is processed. The COMMON text is written in the area that would have been occupied by this Phase 10 subroutine.

The EQUIVALENCE text is composed of three fields, each two bytes long. Every group of equated symbols is preceded by a header entry. The first field contains an adjective code representing an EQUIVALENCE statement. The second field contains binary zeros, and the third contains the number of equated variables.

The format for the header entry is:

2 bytes	2 bytes	2 bytes
adjective code	0000	number of entries

A detail entry is made for each variable in an EQUIVALENCE group. The first field is a pointer to the dictionary entry for the variable. The second field holds 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 zeros if it is not subscripted.

The format for the detail entry is:

2 bytes	2 bytes	2 bytes
pointer	size	offset or 0000

For example, the statements:

```
EQUIVALENCE (A(2,1),B(1),C), (M(3,2),I)
EQUIVALENCE (XEM,Y,ZETA)
```

where A, B, C, XEM, Y, and ZETA are real; M and I are integers; and the arrays are dimensioned as A(5,5), B(10), and M(10,5) to produce the EQUIVALENCE text shown in Figure 34.

All arrays must be defined before they can be used in an EQUIVALENCE statement or any other statement. The DIMENSION routine is overlaid with the text for the EQUIVALENCE statements.

Entry	Entry	Reason	Entry	Reason
99	0	(header entry)	3	(number of entries)
p(A)	100	(detail entry array size in bytes)	4	(offset)
p(B)	40	(detail entry array size in bytes	0	(offset)
p(C)	4	(detail entry size of a real)	0	(no subscript)
99	0	(header entry)	2	(number of entries)
p(M)	200	(detail entry size of array)	80	(offset)
p(I)	4	(detail entry size of an integer)	0	(no subscript)
99	0	(header entry)	3	(number of entries)
p(XEM)	4	(detail entry size of real)	0	(no subscript)
p(Y)	4	(detail entry size of real)	0	(no subscript)
p(ZETA)	4	(detail entry size of real)	0	(no subscript)

Figure 34. EQUIVALENCE Text Entry for EQUIVALENCE Statements

STORAGE MAP

The storage map for Phase 10 is shown in Figure 35.

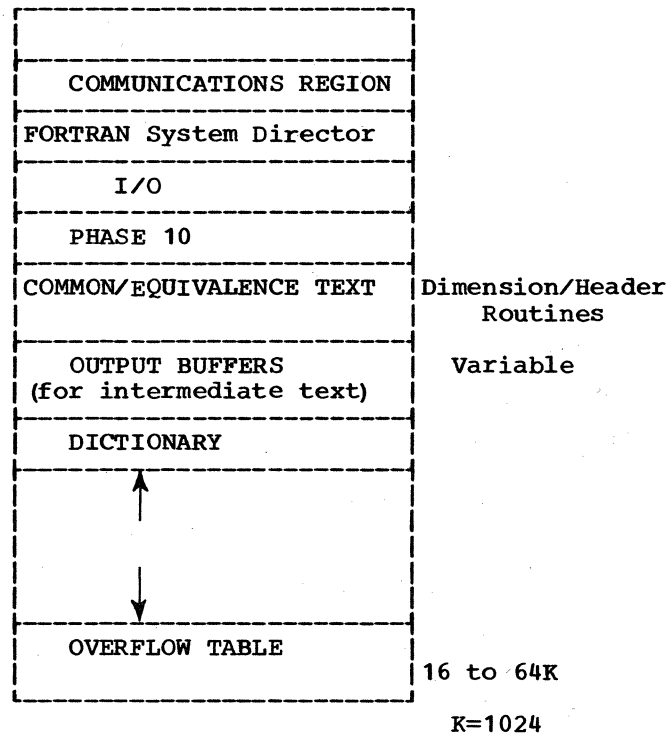


Figure 35. Storage Map for Phase 10

SUBROUTINES

The introduction to Phase 10 has discussed the input and output in Phase 10.

Five forms of data are developed from source statements:

1. Dictionary.
2. Overflow table.
3. Intermediate text.
4. COMMON text.
5. EQUIVALENCE text.

To develop this data, 3 types of subroutines (mainline, keyword, and utility) are used in Phase 10.

The mainline subroutines divide statements into three classes: arithmetic statements, keyword statements, and arithmetic statement functions. The mainline subroutines also process arithmetic expressions and statements and define arithmetic statement functions. These are covered in charts BB through BF.

The keyword subroutines supervise the processing of statements beginning with FORTRAN keywords. In fact they may process the entire keyword statement. These are covered in charts BJ through BW.

The utility subroutines called by mainline or keyword statements enter symbols in the dictionary, overflow tables, and intermediate text; convert numbers; call input/output devices; process subscripts,

and end marks, etc. These subroutines are covered in charts CB through CR.

Subroutine CLASSIFICATION: Chart BB

Subroutine CLASSIFICATION performs the following functions:

1. Initializes program switches to process another statement.
2. Processes any statement number that may define this statement.
3. Determines if this statement begins with a keyword. If it does, gives control to the correct keyword subroutine.
4. Gives control to subroutine ARITH if the statement does not begin with a keyword.

ENTRANCE: Subroutine CLASSIFICATION receives control from:

1. Phase 10 HOUSEKEEPING subroutine.
2. Subroutine ARITH after the entire FORTRAN statement has been processed.
3. A keyword subroutine if it has processed the entire statement.

OPERATION: In initialization, the byte called FUNISW is set to binary zero, 1 is added to the internal statement number, and the parentheses count is set to zero.

The bits in FUNISW represent the following:

0 : READ/WRITE statement
1 : Subscripted variable
2 : Immediate DO parameter
3 : Literal DO parameter
4 : IF statement
5 : Statement number
6 : Arithmetic statement function
7 : I/O unit

Subroutine GETWD retrieves a new statement from the buffer area. If the first five columns contain a statement number, the overflow table is scanned and the address of statement number overflow table entry is returned to CLASSIFICATION. This address is entered in the intermediate text along with the statement number adjective and mode/type codes.

If the first symbol (other than any possible statement number) is a keyword, control is passed to the subroutine which processes that particular keyword statement. If the statement does not begin with a keyword, control is passed to subroutine ARITH.

EXIT: Subroutine CLASSIFICATION exits to:

1. Any of the various keyword subroutines that process the statement under consideration.
2. Subroutine ARITH.
3. Subroutine ERROR if an error has been detected.

SUBROUTINES CALLED: During execution, subroutine CLASSIFICATION references subroutines GETWD, LABLU, CSORN, and WARNING.

Subroutine ARITH: Charts BC, BD, BE

Subroutine ARITH determines whether the statement defines an arithmetic statement function, and if a function is called with the statement. ARITH makes the entries for arithmetic expressions to the dictionary, intermediate text, and overflow table by calling other subroutines.

ENTRANCE: Subroutine ARITH is entered from:

1. Subroutine CLASSIFICATION if the statement does not begin with a keyword.
2. Subroutine ASF after the delimiter = is encountered in an arithmetic statement function.
3. Keyword subroutines if the keyword statement contains an arithmetic expression.
4. Subroutine READ/WRITE to analyze the I/O list.

CONSIDERATION: Subroutine ARITH is divided into three parts. ARITH Part 1 scans the statement for symbols and delimiters and enters the symbols in the dictionary. ARITH Part 2 determines if there are any subscripted variables or referenced functions on the right of the = sign. ARITH Part 3 makes the entries to the intermediate text. ARITH Part 2 and Part 3 process the delimiters that are found by ARITH Part 1.

Each of the three flowcharts associated with subroutine ARITH represents the specific functions performed by subroutine ARITH.

EXIT: Subroutine ARITH exits to:

1. Subroutine ASF.
2. Subroutine GO TO.
3. Subroutine END MARK CHECK.
4. Subroutine ERROR if an error is detected.

Subroutine ARITH Part 1

Subroutine ARITH Part 1 prepares to insert the adjective code for an arithmetic statement in the intermediate text. It determines if this statement defines an arithmetic statement function or if a subscripted variable appears left of the equal sign. ARITH Part 1 scans the statement for symbols and delimiters.

ENTRANCE: Subroutine ARITH Part 1 is entered from:

1. Subroutine CLASSIFICATION if the statement does not begin with a keyword.
2. Subroutine ASF to process the arithmetic expression to the right of the = sign.
3. Subroutine subroutines if this keyword statement contains an arithmetic expression or an I/O List.
4. Subroutine ARITH Part 3 to fetch another symbol.

OPERATION: The adjective code for an arithmetic statement is moved to a buffer by subroutine PUTX, which fills the intermediate text output buffers.

An arithmetic statement function is defined by checking the following conditions:

1. The following delimiter is a left parenthesis.
2. The name has not been dimensioned.
3. No executable statements have been processed. If these conditions are satisfied, ARITH Part 1 calls subroutine ASF.

The first entry for this statement, if it is not defining an arithmetic statement function, is entered in the intermediate text using subroutine PUTX in ARITH Part 1. ARITH Part 1 alternately scans the statement using a translate and test instruction issued by subroutines GETWD and SKPBLK. Any symbol in the arithmetic statement other than a delimiter is scanned by ARITH Part 1 and entered in the dictionary. If the symbol is a delimiter, control is passed to ARITH Part 2.

CONSIDERATION: Phase 10 determines if any executable statements have been processed by checking the executable switch. The switch is set on when the first executable statement of the program is processed. It is not set off for the remainder of the phase.

EXIT: Subroutine ARITH Part 1 exits to:

1. Subroutine ASF, if an arithmetic statement function is defined.
2. Subroutine ERROR, if an error is detected.
3. Subroutine ARITH Part 2 to begin processing a delimiter.

SUBROUTINES CALLED: During execution ARITH Part 1 calls subroutines:

1. SKPBLK to get a delimiter.
2. SUBS to process a subscripted variable.
3. PUTX to make entries to the intermediate text.
4. GETWD to access a symbol.
5. CSORN to make entries to the dictionary.

Subroutine ARITH Part 2

ARITH Part 2 determines if any subscripted variables or referenced functions follow the = sign and calls the appropriate subroutine. ARITH Part 2 then gives control to the correct routine in Part 3 to process the current delimiter.

ENTRANCE: Subroutine ARITH Part 2 is entered from subroutine ARITH Part 1.

OPERATION: If the delimiter following the symbol is a left parenthesis, the symbol must either be an array or function name. If the dictionary entry for the symbol indicates an array, the symbol is assumed to be a subscripted variable. Otherwise, it is assumed to be the name of a function.

ARITH Part 2 uses the byte placed in register 2 by the translate and test instruction in Part 1 to index a branch list in order to get to the correct delimiter routine in ARITH Part 3. This same byte is inserted in the adjective code to represent the delimiter.

CONSIDERATION: The first delimiter (not a blank) that appears after the first variable is assumed to be the = sign. It is checked by subroutine ARITH.

EXIT: ARITH Part 2 exits to:

1. Subroutine ARITH Part 3 to process a delimiter.
2. Subroutine ERROR if an error is detected.

SUBROUTINE CALLED: During execution ARITH Part 1 calls SUBS if a subscripted variable is recognized.

Subroutine ARITH Part 3

Subroutine ARITH Part 3 processes delimiters and makes entries to intermediate text.

ENTRANCE: Subroutine ARITH Part 3 is entered by subroutine ARITH Part 2.

OPERATION: If a decimal point is used as a delimiter, the symbol must be a floating-point constant. This subroutine must then get and convert the number.

A left parenthesis increments the parentheses count.

An equal sign is usually found as the first delimiter in an arithmetic statement after a subscripted or nonsubscripted variable. An = sign used in this manner is merely inserted in the intermediate text in ARITH Part 2 and is never processed by ARITH Part 3. If the statement contains an = sign in any other position, it is the delimiter for an implied DO in a READ/WRITE statement.

A right parenthesis decrements a parentheses count, and checks whether the count has reached zero or become negative.

An end mark gives control to subroutine END MARK CHECK. If the last symbol processed was a subscripted variable, control is given directly to END MARK CHECK because the subroutine SUBS has made all entries to the intermediate text.

Asterisk checks whether an asterisk immediately preceded this asterisk to supply the adjective code for exponentiation operation.

A plus, minus checks if the + or - sign follows another delimiter. If it does, it is assumed to be a unary operation.

CONSIDERATIONS: In ARITH Part 3, entries are made to the intermediate text through the logical block ARIT30. If the last variable is subscripted, no entries are made in the subroutine ARITH for the last variable. The subscript subroutine SUBS makes all the intermediate text entries necessary for the subscripted variable.

EXIT: Subroutine ARITH Part 3 exits to the following subroutines:

1. ARITH Part 2 after all entries have been made.
2. ERROR WARNING if an error is detected.
3. GOTO after processing the arithmetic expression in an IF statement.
4. DO after processing an implied DO in an I/O statement.
5. END MARK CHECK if an end mark is detected.
6. ARITH Part 1 to get another symbol.

SUBROUTINES CALLED: During execution subroutine ARITH Part 3 calls subroutines:

1. GETWD to access a number for a literal.
2. CSORN to enter a decimal number in the dictionary.
3. PUTX to make entries in the intermediate text.

Subroutine ASF: Chart BF

Subroutine ASF processes the parameters of an arithmetic statement function definition until the delimiter = is encountered. All symbols and delimiters which follow are processed by subroutine ARITH (see Figure 36).

ENTRANCE: Subroutine ASF is entered from:

1. Subroutine ARITH Part 1 if it determines that the current statement defines an arithmetic statement function.
2. Subroutine END MARK CHECK to complete processing an arithmetic statement function definition.

CONSIDERATIONS: An arithmetic statement function must be defined in a user program before it is called in an arithmetic statement.

The symbols used to define the parameters in an arithmetic statement function may

Statement	SUM	(A,B,C) =	A+B+C+2.0	≠
Subroutines that process the statement	CLASSIFICATION ARITH	ASF	ARITH	END MARK CHECK ASF (processes end mark text entry)

Figure 36. Arithmetic Statement Function Processing

be used in the main program. They do not carry the same meaning as they do in the statement function definition. For example:

```
DIMENSION A(20)
    .
    .
    .
SUM(A) = (A+2.3) *3.14
```

Both statements contain the name A. In the first statement A is defined as an array by use of a DIMENSION statement. In the second statement A is a dummy variable used to define the operations on the parameter passed to the function SUM, when the user calls the function SUM in a normal arithmetic statement. For all other statements in the main program, A is an array with 20 elements. However, the mode can be set by a specification statement.

OPERATION: Before subroutine ARITH determines that a statement defines an arithmetic statement function, it sets an adjective code to represent an arithmetic statement. If ARITH determines that the statement is an arithmetic statement function definition, subroutine ASF changes the adjective code to represent an arithmetic statement function.

Subroutine ASF searches the dictionary for a symbol that defines a parameter to determine if that symbol has been defined previously. If it has, ASF defaces the previous entry so that it can not be recognized. The address of this previous entry is saved, and the name of the parameter used to define the function is entered in the dictionary.

If that symbol has not been defined previously, it is entered in the dictionary and the mode/type field is set to indicate that the symbol is a dummy variable.

Subroutine END MARK CHECK returns to ASF if the switch indicating the processing of an arithmetic statement function definition is set on. The switch is then turned off. All defaced entries are restored to their original image using the previously stored addresses. The original entries are then recognized by subsequent searches of the dictionary. The entries made for defining the operations in the arithmetic statement function are then defaced so subsequent searches of the dictionary will not recognize the symbols used to define parameters.

EXIT: Subroutine ASF exits to the following subroutines:

1. ARITH Part 1 to process the statement to the right of the = sign.
2. END MARK CHECK after the original entries in the dictionary are restored.
3. ERROR if an error is detected.

SUBROUTINES CALLED: During its execution subroutine ASF references subroutines:

1. CSORN to search for and make entries in the dictionary.
2. GETWD to access symbols.
3. SKPBLK to access delimiters.

Subroutine GOTO: Chart BJ

Subroutine GOTO determines if the statement is an unconditional or computed GOTO statement. If the statement is an unconditional GOTO, the statement number is entered in the intermediate text. If the statement is a computed GOTO, the list of statement numbers is scanned and entered in the intermediate text and overflow table.

ENTRANCE: Subroutine GOTO receives control from the following subroutines:

1. CLASSIFICATION.
2. ARITH Part 3 to process the statement number for IF statements.

CONSIDERATION: A GOTO statement may begin with either the words GO TO, or the word GOTO. Subroutine CLASSIFICATION recognizes either form.

OPERATION: After the keyword has been checked, subroutine GOTO determines whether this statement is a computed GOTO. If the delimiter following the letters TO is a left parenthesis, subroutine GOTO assumes the statement is computed GOTO. Any delimiter, other than a blank, between the letters TO and the (first) statement number is not accepted.

A library subroutine is used by the object program to execute a computed GOTO statement. The first intermediate text entry for a computed GOTO contains the adjective code for a computed GOTO, the type code for a library function, and the library identification number for the computed GOTO subroutine in the pointer field.

Subroutine GOTO is entered after the arithmetic expression for the IF statement is processed by subroutine ARITH to process the list of statement numbers in the IF statement. The IF switch tells subroutine GOTO that the statement currently being processed is an IF statement.

EXIT: Subroutine GOTO exits to the following subroutines:

1. END MARK CHECK when the end mark is encountered.
2. ERROR if an error is detected.

SUBROUTINES CALLED: During execution subroutine GOTO references subroutines:

1. GETWD to access symbols in the source statement.
2. LABLU to process statement numbers.
3. PUTX to make entries to the intermediate text.
4. CSORN to make entries in the dictionary.
5. WARNING/ERRET if a warning is detected.

Subroutine DO: Chart BK

Subroutine DO scans the DO statement and makes entries to the intermediate text, dictionary, and overflow table. Subroutine DO also processes the parameters for an implied DO in a READ/WRITE statement.

ENTRANCE: Subroutine DO receives control from subroutines:

1. CLASSIFICATION.
2. ARITH Part 3 to process an implied DO in an I/O list.

OPERATION: An error, a backward DO, occurs if the statement number definition for the end of the DO loop precedes the DO statement. If statements occurred in the FORTRAN program in the following sequence, this condition would exist.

```
20 A=B
   .
   .
   .
DO 20 I=1,10
```

The switches for an immediate DO parameter are set on and off when subroutine DO calls CSORN (see description of subroutine CSORN, Chart CG).

Subroutine DO is also entered when subroutine ARITH is processing a READ/WRITE statement. If an implied DO is found in a READ/WRITE statement, subroutine DO will process the parameters. An implied DO is formed when a program attempts to perform an I/O operation on a number of elements from an array without listing all of them in an I/O list. For example, the statement READ (3,1), (A(I),I=1,100) contains an implied DO. The elements of the array A are to be read with this I/O statement.

EXIT: Subroutine DO exits to the following subroutines:

1. END MARK CHECK when the end mark is sensed.
2. ARITH if subroutine DO is processing an implied DO of a READ/WRITE statement.
3. ERROR if an error is detected.

SUBROUTINES CALLED: During execution subroutine DO references subroutines:

1. GETWD to access symbols and delimiters.
2. LABTLU to process overflow table entries for statement numbers.
3. PUTX to make entries in the intermediate text.
4. CSORN to process dictionary entries.
5. SKPBLK to access delimiters.

Subroutine SUBIF: Chart BL

Subroutine SUBIF enters the IF adjective code in the intermediate text and gives control to other subroutines to process the arithmetic expression and the list of statement numbers following the expression.

ENTRANCE: Subroutine SUBIF is entered from subroutine CLASSIFICATION.

OPERATION: Subroutine SUBIF sets the IF switch to control the processing of other subroutines which process portions of the IF statement. The IF switch is contained along with other switches in the byte called FUNISW (see CLASSIFICATION).

Subroutine ARITH processes the arithmetic expression, and subroutine GOTO processes the list of statement numbers.

EXIT: Subroutine SUBIF exits to:

1. Subroutine ARITH to process the arithmetic expression in an IF statement.
2. ERROR, if an error is detected.

SUBROUTINES CALLED: During execution subroutine SUBIF calls:

1. Subroutine SKPBLK to access a delimiter.
2. Subroutine PUTX to make entries in the intermediate text.

Subroutines CALL, FUNCTION/SUBRTN: Chart BM

Subroutine CALL

Subroutine CALL gets the name of the subprogram and passes control to subroutine ARITH to process the arguments.

ENTRANCE: Subroutine CALL is entered from subroutine CLASSIFICATION.

OPERATION: Since the arguments in a CALL statement may be arithmetic expressions, subroutine CALL uses subroutine ARITH to process the parameters that are passed to the user SUBROUTINE during the execution of the object program.

EXIT: Subroutine CALL exits to:

1. Subroutine ARITH Part 2 to process the arguments of a CALL statement.
2. Subroutine ERROR, if an error is detected.

SUBROUTINES CALLED: During execution subroutine CALL references the following subroutines:

1. PUTX to enter the CALL adjective code in the intermediate text.
2. GETWD to access the subprogram name.
3. CSORN to enter the name in the dictionary.

Subroutine FUNCTION/SUBRTN

Subroutine FUNCTION/SUBRTN processes the header cards for FUNCTION and SUBROUTINE subprograms. It makes entries to the intermediate text and dictionary for the subprogram name and the parameters passed to the subprogram.

ENTRANCE: Subroutine FUNCTION/SUBRTN is entered from subroutines:

1. CLASSIFICATION.
2. INTEGER/REAL/DOUBLE if a statement such as REAL FUNCTION A(B,C) is used.

OPERATION: Subroutine FUNCTION/SUBRTN is entered at two points. The first entry point is used if the program defines a SUBROUTINE. If FUNCTION/SUBRTN is entered at this point a switch is set to indicate that the statement is the header card for a user SUBROUTINE. The logic flow for FUNCTION and a SUBROUTINE at this time become the same. The second entry point is used if the subprogram is FUNCTION.

A test is made to check whether this is the first card of the source program. If not, an error condition exists. The internal statement number must equal 1 if this is the first card processed. TINE

The switch for a SUBROUTINE is tested in order to enter the correct adjective code. If the switch indicates that this is a SUBROUTINE subprogram definition, the delimiter following the subprogram name is tested. If it is an end mark, the subprogram definition is valid, because a SUBROUTINE subprogram may have no parameters. A FUNCTION subprogram definition is not valid if it has no parameters.

Subroutine FUNCTION/SUBRTN scans the list of arguments and enters them into the dictionary and intermediate text. It uses subroutines GETWD and CSORN. The only valid delimiters are a comma to separate the arguments and a right parenthesis to conclude the scan. The type codes for arguments in the subprogram are not entered here; they are defined implicitly or explicitly in the subprogram. Since arguments can be arrays, space is reserved in the dictionary entries for array information.

EXIT: Subroutine FUNCTION/SUBRTN exits to the following subroutines:

1. END MARK CHECK if an end mark is encountered.
2. ERROR if an error is detected.

SUBROUTINES CALLED: During execution subroutine FUNCTION/SUBRTN references subroutines:

1. GETWD to access symbols in the statement.
2. CSORN to make entries in the dictionary.
3. SKPBLK to access delimiters in the statement.
4. PUTX to make entries in the intermediate text.

Subroutine READ/WRITE: Chart BN

Subroutine READ/WRITE analyzes the symbol representing the data set reference number and the FORMAT statement number in a READ/WRITE statement, and enters them in the intermediate text, dictionary, and overflow table. Subroutine ARITH then processes the list in the READ/WRITE statement.

ENTRANCE: Subroutine READ/WRITE receives control from subroutine CLASSIFICATION.

OPERATION: When subroutine READ/WRITE is entered, it assumes the read/write operation will be in BCD mode. It enters the BCD adjective code into ADJ, and later it determines if the operation is in BCD or binary.

Subroutine READ/WRITE sets a switch to indicate that this is a READ/WRITE statement. This switch is tested in subroutine ARITH Part 3 when subroutine ARITH processes the READ/WRITE list. It is used in

connection with the implied DO for the READ/WRITE statement.

Subroutine READ/WRITE does not check the validity of the data set reference number specified in the READ/WRITE statement. It blindly enters it into the dictionary if it has not already been entered, and enters its dictionary address into the intermediate text.

Subroutine READ/WRITE determines if the operation is BCD or binary by the manner in which the statement is formed. The statement:

READ(3,1) HOG,TOAD,SHARK,LUNCH

is a statement instructing the object program to read in the BCD mode, while the statement:

READ(3) PAUL,CHUCK,FOO

directs that the read be in the binary mode.

When the subroutine READ/WRITE determines that the second delimiter is a right parenthesis instead of a comma, it changes the BCD adjective code entered in ADJ to a binary adjective code.

EXIT: Subroutine READ/WRITE exits to subroutines:

1. ERROR if an error has been detected.
2. ARITH Part 1 to begin processing the READ/WRITE variable list.

SUBROUTINES CALLED: During execution subroutine READ/WRITE references the following subroutines:

1. SKPBLK to access delimiters.
2. GETWD to access symbols in the statement.
3. CSORN to enter symbols in the dictionary.
4. PUTX to make entries in the intermediate text.
5. LABLU to process the FORMAT statement number.

Subroutine CONTINUE/RETURN, STOP/PAUSE: Chart BO

Subroutine CONTINUE/RETURN

Subroutine CONTINUE/RETURN makes the single intermediate text entry for the CONTINUE and RETURN statements.

ENTRANCE: Subroutine CONTINUE/RETURN is entered by subroutine CLASSIFICATION.

OPERATION: The entrance to subroutine CONTINUE/RETURN for a CONTINUE statement checks for a statement number. If there was no statement number, a warning is issued.

A RETURN statement is used to return control to the main program from a FUNCTION or a SUBROUTINE subprogram. If this statement is in a main program, an error condition exists.

Neither the CONTINUE nor the RETURN statement enters a pointer in intermediate text. Both the pointer and mode/type fields for their intermediate text entries are set to 0.

EXIT: Subroutine CONTINUE/RETURN exits to subroutine END MARK CHECK.

SUBROUTINES CALLED: During execution subroutine CONTINUE/RETURN references subroutines GETWD, WARNING, PUTX, and SKTEM.

Subroutine STOP/PAUSE

Subroutine STOP/PAUSE enters the adjective code and any number used to identify the halt into the intermediate text. This number is not entered in the dictionary.

ENTRANCE: Subroutine STOP/PAUSE is entered from subroutine CLASSIFICATION.

There are two intermediate text entries made for STOP and PAUSE statements. The first entry contains the STOP or PAUSE adjective code with zero in the entries for the mode/type and pointer fields. If there is a halt number , it is entered in the pointer field of the second intermediate text entry. If there is no halt number the second entry will contain zeros.

EXIT: Subroutine STOP/PAUSE exits to subroutines:

1. END MARK CHECK.
2. ERROR if an error is detected.

SUBROUTINES CALLED: Subroutine STOP/PAUSE calls subroutine:

1. GETWD to access symbols and delimiter.
2. PAKNUM to pack the halt number.
3. PUTX to make entries to the intermediate text.
4. SKPBLK to get the end mark.

Subroutine BKSP/REWIND/END/ENDFILE: Chart BP

Subroutine BKSP/REWIND/END/ENDFILE makes the intermediate text and dictionary entries for the REWIND, BACKSPACE, END, and ENDFILE statements.

ENTRANCE: BKSP/REWIND/END/ENDFILE receives control from subroutines:

1. CLASSIFICATION if the keywords BACKSPACE, REWIND, END, or ENDFILE are recognized.
2. END MARK CHECK if the end of data set in the card reader is sensed and the END card has not been read.

OPERATION: The intermediate text entries for the BACKSPACE, REWIND, END FILE, and ENDFILE statements are the same except for the adjective codes. The END FILE and ENDFILE keywords mean the same. The compiler accepts either form. The subroutine enters either the address of the dictionary entry for a data set reference number or the address of a name symbolizing the data set reference number in the intermediate text for the I/O statements.

When subroutine CLASSIFICATION recognizes the keyword END, it is not determined whether this statement is an END or END FILE statement. When CLASSIFICATION passes control to this subroutine after recognizing the word END, subroutine BKSP/REWIND/END/ENDFILE checks if the next symbol is the word FILE.

If this was an END statement signifying end of the program, subroutine BKSP/REWIND/END/ENDFILE sets a switch to indicate to subroutine END MARK CHECK that the END card has been read. When END MARK CHECK senses an end of data set at the input device, it gives control to BKSP/REWIND/END/ENDFILE to set a switch simulating that an END card has been read.

EXIT: Subroutine BKSP/REWIND/END/ENDFILE exits to subroutines:

1. END MARK CHECK.
2. ERROR if an error is detected.

SUBROUTINES CALLED: During execution subroutine BKSP/REWIND/END/ENDFILE references subroutines:

1. CSORN to process entries in the dictionary.
2. GETWD to access symbols and delimiter.
3. PUTX to make entries to the intermediate text.

Subroutine DIMENSION: Chart BQ

Subroutine DIMENSION scans the list of symbols for DIMENSION, COMMON, INTEGER, REAL, and DOUBLE PRECISION statements. It determines if variables are subscripted, calls subroutines to process the subscript, and changes the mode in the dictionary when an explicit mode defines the mode of a variable. In any one of the above statements, an array may be defined and subroutine DIMENSION makes entries in the overflow table and dictionary for the array.

ENTRANCE: Subroutine DIMENSION is entered by subroutines:

1. CLASSIFICATION.
2. COMMON to process the list of variables placed in COMMON.
3. INTEGER/REAL/DOUBLE PRECISION to process the list of variables.

OPERATION: A sequence error occurs if an executable or EQUIVALENCE statement is processed before the DIMENSION statement is read.

The scan is similar to the scan used in subroutine ARITH, but it is much simpler because there are only three legal delimiters, the comma and the left and right parentheses.

A multiple switch is set to determine the type of statement being processed. It is set in the subroutines COMMON and INTEGER/REAL/DOUBLE which transfer control to subroutine DIMENSION.

EXIT TO: Subroutine DIMENSION exits to subroutines:

1. END MARK CHECK.
2. COMMON in order that entries may be made in the COMMON text.
3. ERROR if an error is detected.

SUBROUTINES CALLED: During execution subroutine DIMENSION references subroutines:

1. GETWD to access symbols and delimiters.
2. RCOMMA to skip redundant commas.
3. SKPBLK to access delimiters.
4. CSORN to process dictionary entries.
5. DIMSUB to calculate array sizes.
6. WARNING/ERRET to process warnings.

Subroutine EQUIVALENCE: Charts BR, BS

Subroutine EQUIVALENCE creates the EQUIVALENCE text. The two flow charts associated with subroutine EQUIVALENCE rep-

resent specific functions performed by subroutine EQUIVALENCE. Each is discussed separately.

ENTRANCE: Subroutine EQUIVALENCE receives control from subroutine CLASSIFICATION.

EXIT: Subroutine EQUIVALENCE exits to subroutines:

1. END MARK CHECK.
2. ERROR if an error is detected.

Subroutine EQUIVALENCE Part 1

This part of subroutine EQUIVALENCE scans the EQUIVALENCE statement, getting the variable names and delimiters. It also makes both header and detail entries for the EQUIVALENCE text.

ENTRANCE: Subroutine EQUIVALENCE Part 1 is entered by subroutines:

1. CLASSIFICATION.
2. EQUIVALENCE Part 2 when a name is subscripted.

OPERATION: The EQUIVALENCE text is written in the area that orginally contained subroutine DIMENSION. When an EQUIVALENCE statement is processed, a switch is set to forego the processing of any DIMENSION statements that follow the EQUIVALENCE statement. Subroutine DIMENSION is overlaid by EQUIVALENCE text. This switch is never reset during Phase 10.

The EQUIVALENCE text contains a header entry and a detail entry for each element in the EQUIVALENCE group. The entire group must be scanned before the header entry is made because it contains a count equal to the number of variables in the EQUIVALENCE group. After the right parenthesis defining the end of this group is encountered, the element count is inserted in the header entry.

EXIT: Subroutine EQUIVALENCE Part 1 exits to subroutines:

1. END MARK CHECK.
2. ERROR if an error is detected.
3. EQUIVALENCE Part 2 to process a subscripted variable.

SUBROUTINES CALLED: During execution subroutine EQUIVALENCE Part 1 references subroutines:

1. SKPBLK to access delimiters
2. GETWD to access symbols and delimiters
3. CSORN to process dictionary entries
4. WARNING/ERRET to process warnings.

Subroutine EQUIVALENCE Part 2

EQUIVALENCE Part 2 processes the subscript information for any subscripted variable which is a member of an EQUIVALENCE group.

ENTRANCE: Subroutine EQUIVALENCE Part 2 is entered from subroutine EQUIVALENCE Part 1 to process a subscripted variable.

OPERATION: The offset is computed using the numbers in the subscripted variable in the EQUIVALENCE statement and the information for the array entered in the overflow table by subroutine DIMENSION.

The offset for 3-dimensional variables is computed using the following formula:

Offset=[(J1-1) + (J2-1) D1+ (J3-1) D1*D2] *Length

where: J1, J2, and J3 are constants in the subscripted variable A (J1, J2, J3) entered in the EQUIVALENCE group. The constants Length, D1*Length, and D1*D2*Length, are computed when the DIMENSION statement is processed by subroutine DIMENSION and stored in the overflow table. When EQUIVALENCE Part 2 collects each subscript, it subtracts 1 from each and multiplies it by the appropriate constant in the overflow table. The products of this multiplication are added into an accumulator until all subscripts for this variable are exhausted.

Valid subscripts for variables in an EQUIVALENCE group contain no variables. Subscripted variables may have one subscript for 1-, 2-, and 3-dimensioned variables, or the same number of subscripts as there are dimensions in its DIMENSION statement. For example, in the array A(5,5,5), A(2,2,2) and A(32) represent the same element.

EXIT: Subroutine EQUIVALENCE Part 2 exits to subroutines:

1. EQUIVALENCE Part 1 to enter the subscripted variable into the EQUIVALENCE text.
2. ERROR if an error is detected.

SUBROUTINES CALLED: During execution subroutine EQUIVALENCE Part 2 references subroutines:

1. GETWD to access symbols and delimiters.
2. INTCON to convert EBCDIC numbers to binary numbers.
3. SKPBLK to access delimiters.

Subroutine COMMON: Chart BT

The function of subroutine COMMON is to direct the processing of COMMON statements and to enter names in the COMMON text.

ENTRANCE: Subroutine COMMON receives control from subroutines CLASSIFICATION or DIMENSION.

CONSIDERATION: Subroutine COMMON uses subroutine DIMENSION to do the bulk of the processing for the COMMON statements. It sets a program switch to indicate to subroutine DIMENSION that this is a COMMON statement. After subroutine DIMENSION collects the symbols, COMMON enters them in the COMMON text. COMMON statements may be used in place of a dimension statement to define an array.

OPERATION: The executable switch is checked to see if any executable statement or EQUIVALENCE statement is processed. If there has been an executable or EQUIVALENCE statement processed, a sequence error is detected.

After subroutine DIMENSION retrieves each name in the COMMON statement from the dictionary, control returns to subroutine COMMON. The address of the entry in the dictionary along with the length of the name are entered in the COMMON text. The length of the name is entered in order to search the chain in the dictionary for the name.

EXIT: Control is passed from subroutine COMMON to subroutines:

1. DIMENSION to process entries in the COMMON statement.
2. ERROR if an error is detected.

Subroutine FORMAT: Chart BU

Subroutine FORMAT enters the adjective code for a FORMAT statement in the intermediate text. Then the card image immediately beyond the word FORMAT, extending through column 72, is moved in one byte BCD characters to the intermediate text. If a continuation card is required to complete the FORMAT statement, the image from column 7 of the continuation card through column 72 is moved to the intermediate text.

ENTRANCE: Subroutine FORMAT receives control from subroutine CLASSIFICATION.

OPERATION: Subroutine GETWD sets an end mark in the first column beyond the last non-blank character in the card. Because subroutine FORMAT moves all the characters beyond the word FORMAT through column 72 to the intermediate text, the end mark set by GETWD is blanked, and an end mark is placed in column 73 by subroutine FORMAT.

Subroutine CLASSIFICATION has made an entry in the overflow table for the statement number that refers to the FORMAT statement, but it did not adjust the usage field to indicate that this was the statement number for a FORMAT statement. Subroutine FORMAT, using the pointer that was supplied when the statement number was found or entered in the overflow table, adjusts the usage field to indicate that this statement number refers to a FORMAT statement.

Subroutine FORMAT then moves the image of the FORMAT card, byte by byte, to the intermediate text. It moves the characters up to and including the end mark which is placed in column 73. When the end mark is encountered, it is moved to the intermediate text. But the output pointer is not updated, so that if a continuation card were required to complete this statement, the character in column 7 would overlay the end mark. If there are no continuation cards or if this is the last one, the end mark remains in the intermediate text to signal to other phases that it is the end of the image of the FORMAT statement.

If there are no more continuation cards (or if none exist), the end of the FORMAT statement has been reached. When entries are made byte by byte, as in subroutine FORMAT, there is a good possibility that the entries did not stop on a full word boundary. All other intermediate text entries must begin on a full word boundary. The output pointer is then adjusted to a full word boundary to satisfy the format for the intermediate text and the end statement entry is made.

EXIT: Subroutine FORMAT exits to subroutine END MARK CHECK, where the entry for the end statement is made.

SUBROUTINE CALLED: During execution, subroutine FORMAT references subroutines

1. PUTX to make entries to the intermediate text.
2. GET to read cards.
3. WARNING/ERRET if a warning is detected.

Subroutine EXTERNAL: Chart BV

Subroutine EXTERNAL scans the card, placing each name on the card in the

dictionary and typing it as an external symbol. It sets the appropriate bit in the usage field of the dictionary, indicating that an ESD card must be punched for this symbol.

ENTRANCE: Subroutine EXTERNAL receives control from subroutine CLASSIFICATION.

OPERATION: All external symbols must be defined before any executable statements are encountered. If the executable switch is on, subroutine EXTERNAL cannot define external symbols. All symbols entered as external symbols must be names, otherwise subroutine EXTERNAL detects an error. A constant cannot be an external symbol.

EXIT: Subroutine EXTERNAL exits to subroutines:

1. END MARK CHECK when the end mark is encountered.
2. ERROR if an error is detected.

SUBROUTINES CALLED: During execution subroutine EXTERNAL references subroutines

1. GETWD to access symbols and delimiters.
2. CSORN to enter names in the dictionary.
3. RCOMA to bypass redundant commas.

Subroutines INTEGER/REAL/DOUBLE: Chart BW

Subroutine INTEGER/REAL/DOUBLE sets the mode for the statement and exits to subroutines DIMENSION or FUNCTION/SUBRTN.

ENTRANCE: Subroutine INTEGER/REAL/DOUBLE receives control from subroutine CLASSIFICATION.

OPERATION: The mode for this statement is inserted in a work area. Any variable that appears later in the statement being processed is assigned the mode explicitly stated in the first name in this statement. The first name always is REAL, INTEGER, or DOUBLE.

The first of two switches set in subroutine INTEGER/REAL/DOUBLE indicates to subroutine FUNCTION/SUBRTN that it was entered from INTEGER/REAL/DOUBLE and the mode is explicitly defined. When subroutine FUNCTION/SUBRTN enters the mode of the subprogram it checks this switch to see if an explicit mode has been defined. The second switch is set for subroutine DIMENSION, indicating that it was entered from subroutine INTEGER/REAL/DOUBLE.

Subroutine INTEGER/REAL/DOUBLE may exit to one of two subroutines. If the next symbol is the word FUNCTION, it exits to subroutine FUNCTION/SUBRTN to define the function that follows the word FUNCTION. For example:

```
REAL FUNCTION RAF (A,B,I)
```

defines the function RAF as a real function. The mode and type of the parameters are not defined until they are used in a statement other then the header card. If the next symbol is not the word FUNCTION, control is passed to subroutine DIMENSION, because an array may be defined explicit mode statement. For example:

```
DOUBLE PRECISION A, TOAD, HERBIE (20)
```

defines a double precision array HERBIE composed of 20 elements.

EXIT: Subroutine INTEGER/REAL/DOUBLE exits to the following subroutines:

1. DIMENSION.
2. FUNCTION/SUBRTN if the name after the specification keyword is the keyword FUNCTION.
3. ERROR if an error is detected.

SUBROUTINES CALLED: During execution subroutine INTEGER/REAL/DOUBLE references subroutines:

1. GETWD to access symbols and delimiters.
2. WARNING/ERRET to process warnings.

Subroutine HOUSEKEEPING: Chart CB

Subroutine HOUSEKEEPING enters information into the communication area, primes input buffers, and sets the beginning addresses for the dictionary and overflow table.

ENTRANCE: Subroutine HOUSEKEEPING is entered from the FORTRAN System Director after the FSD has loaded Phase 10.

OPERATION: Subroutine HOUSEKEEPING enters the following information:

1. Indication to FSD that Phase 10 is in control.
2. Address of output buffer, COMMON text area, EQUIVALENCE text area, and thumb index.

The dictionary is located initially at the end of the Phase 10 subroutines. Subroutine HOUSEKEEPING must move it to allow storage for the intermediate text output

buffers. The size of the output buffers is calculated from information in the communications area (supplied by the Control Card routine); then, the beginning address of the dictionary is calculated.

The resident dictionary is moved to this location, and the addresses in the thumb index are modified to reflect the new location of the dictionary.

Subroutine HOUSEKEEPING sets the address of the dictionary and overflow table in registers. It primes the input buffer by calling the FORTRAN System Director to read in the first two cards in the card reader while the communications area is being initialized.

EXIT: Subroutine HOUSEKEEPING exits to subroutine CLASSIFICATION to process the first source statement.

SUBROUTINES CALLED: Subroutine HOUSEKEEPING references the FORTRAN System Director to read the first two source cards.

Subroutine GETWD: Chart CC

Subroutine GETWD scans the card for names, constants, data set reference numbers, and delimiters. If the end mark for a card is sensed, GETWD reads a new card, prints it, sees if the card is a continuation card, and adjusts the pointers and register to process the continuation card.

ENTRANCE: The utility subroutine GETWD is referenced by subroutines EXTERNAL, REAL/INTEGER/DOUBLE, CLASSFICATION, ARITH, GOTO, CONTINUE/RETURN, STOP/PAUSE, BKSP/REWIND/END/END FILE, SUBS, EQUIVALENCE, DO, ASF, READ/WRITE, CALL, FUNCTION/SUBRTN, DIMENSION, DIM90, END MARK CHECK, SKPBLK, SKTEM, FORMAT

OPERATION: When subroutine GETWD is entered, it assumes that Phase 10 has already started processing a card. A pointer, set for the card, checks for a blank card position. If it is blank, the pointer is advanced. If the position is not blank, it saves the pointer for a length calculation.

After the first non-blank character is found, the compiler executes a translate and test instruction. The table for this instruction is set so the instruction stops on any special character (including blanks) except $. The translate and test instruction inserts the address at which it stopped in general register 1, and the non-zero byte in the table which caused it to stop in general register 2. The address in general register 1 is used to calculate the length of the symbol, and initialize GETWD the next time it is entered. The byte in general register 2 is the adjective code for the delimiter and it is also used to index the branch table in ARITH Part 2.

Subroutine GETWD has two return points. The normal return is used if the length of the symbol just scanned is greater than zero. This implies that the symbol scanned is a name, constant, or data set reference number. The second return is the zero return which is used if the symbol has length of zero (i.e., the translate and test instruction stopped at the same position at which it began). A delimiter is at the position that the translate and test instruction began and ended its scan.

If an end mark is encountered as a delimiter, subroutine GETWD calls subroutine GET to read another card. The read area is double buffered (i.e., it can process a card in one buffer, while a card is being read into the second buffer). If an end mark is encountered in buffer 1, GETWD calls subroutine GET to read a card into buffer 1, and prepare the pointers to process the card in buffer 2. At the same time the card is being read into buffer 1, the card in buffer 2 is printed. If this card was a comments card, subroutine GETWD calls subroutine GET to read a card into buffer 2 when the card reader is available.

While the card that is about to be processed is being printed, it is scanned four bytes at a time for the first significant (non-blank) character from column 73 toward column 1. The end mark is placed in the column immediately to the right of that significant character.

Subroutine GETWD checks the card being processed. If it is a continuation card, subroutine GETWD sets the pointers and registers so the calling subroutines never know a continuation card has been read. Register 2, which receives the function bytes of the translate and test instruction, is set to blank and the pointer that is stored after each translate and test instruction is set to point to column 6 of the card. The card is then processed from the point at which GETWD was entered.

EXIT: Subroutine GETWD exits to the subroutine that called it.

SUBROUTINES CALLED: During execution subroutine GETWD references subroutines:

1. GET to read a card.
2. PRINT to print a card image.

Subroutines SKPBLK, SKTEM: Chart CD

Subroutine SKPBLK

When a subroutine expects to find a delimiter, it calls subroutine SKPBLK to skip blanks until it finds another delimiter. If a name, constant, or data set reference number is encountered before a delimiter, an error message is entered in the intermediate text.

ENTRANCE: The utility subroutine SKPBLK is referenced by subroutines ARITH Part 1, SUBS, EQUIVALENCE, DO, SUBIF, READ/WRITE, FUNCTION/SUBRTN, DIMENSION.

EXIT: Subroutine SKPBLK exits to:

1. The subroutine that called it.
2. Subroutine ERROR if a symbol that is not a delimiter is encountered.

SUBROUTINES CALLED: During execution subroutine SKPBLK references subroutine GETWD.

Subroutine SKTEM

When a subroutine expects to find an end mark, it calls subroutine SKTEM, which skips the remaining symbols of the card until it finds an end mark. An error has already been noted when this subroutine is called.

ENTRANCE: The utility subroutine SKTEM is referenced by subroutines END MARK CHECK, ERROR, and CONTINUE/RETURN.

EXIT: Subroutine SKTEM exits to the subroutine that called it.

SUBROUTINES CALLED: During execution subroutine SKTEM references subroutines GETWD.

Subroutine SYMTLU: Chart CE

Subroutine SYMTLU determines if the symbol has been entered on a chain in the dictionary. If the symbol has not been entered, subroutine SYMTLU enters it in the dictionary and returns the address of the entry to the calling subroutine (CSORN). If the symbol has been entered, it returns the address of the entry to the calling subroutine.

ENTRANCE: The utility subroutine SYMTLU is referenced by subroutine CSORN.

OPERATION: If the symbol is a name, the name length determines the proper chain. The length of the symbol is determined by subroutine GETWD and used by subroutine SYMTLU to find the proper address in the thumb index, so that the correct dictionary chain may be searched. The symbol is entered in the chain with a chain address, mode, type, possibly an address, and possibly an array size. The usage field is set by the subroutine that referenced CSORN.

If the symbol is not a name, it is entered on one of the chains for real constants, integer constants, double precision constants, or data set reference numbers. If it is a constant, its mode is determined by subroutine LITCON. Phase 10 distinguishes a data set reference number from a constant by the context in which the number is used.

If the symbol has already been entered, SYMTLU makes no changes in the entry. The subroutine which has called SYMTLU adjusts the mode, type and usage fields of the entry if necessary.

RESTRICTION: Subroutine SYMTLU will reject any attempt made to enter any name greater than six characters. The chains for lengths 7 through 11 are reserved strictly for FORTRAN key words. No user name can be entered in these chains.

EXIT: Subroutine SYMTLU exits to subroutines:

1. CSORN the subroutine that called it.
2. ERROR if an error is detected.

Subroutines LABLU, PAKNUM, LABTLU: Chart CF

Subroutine LABLU

Subroutine LABLU is entered only if the calling subroutine expects the symbol it receives from subroutine GETWD to be a statement number. It calls other subroutines to pack the statement number and enter it into the overflow table. Subroutine LABLU selects the correct chain for the statement number to be entered in the overflow table.

ENTRANCE: Subroutine LABLU is referenced by subroutines CLASSIFICATION, GOTO, and DO.

OPERATION: Subroutine LABLU sets a switch indicating to other subroutines that the symbol they are processing is a statement number. The switch is reset by LABLU before control returns to the subroutine which called LABLU.

EXIT: The utility subroutine LABLU exits to the subroutine that called it.

SUBROUTINES CALLED: During execution subroutine LABLU references subroutines:

1. PAKNUM to pack the statement number.
2. LABTLU to process the overflow table.

Subroutine PAKNUM

Subroutine PAKNUM either packs a statement number prior to the search of the overflow table or packs the number used to identify a PAUSE or a STOP. PAKNUM also checks for errors.

ENTRANCE: The subroutine PAKNUM is referenced by subroutines LABLU and PAUSE/STOP.

RESTRICTIONS: Any statement number or halt number is illegal if it is greater than five characters or contains any alphabetic characters. Subroutine PAKNUM checks both of these conditions.

EXIT: The utility subroutine PAKNUM exits to:

1. The subroutine that called it.
2. Subroutine ERROR if an error is detected.

Subroutine LABTLU

Subroutine LABTLU enters all information into the overflow table. It searches for and enters, if necessary, all statement numbers, subscript information, and dimension information.

ENTRANCE: The utility subroutine LABTLU is referenced by subroutines SUBS, DIMSUB, and LABLU.

CONSIDERATION: A switch is set in subroutine LABLU to indicate a statement number to LABTLU.

OPERATION: Subroutine LABTLU first gets the correct address for the beginning of a chain in the overflow table. Then it searches the contents of each entry in the overflow table, comparing the assembled entry against each entry in the chain for that type of entry until it finds the entry for that symbol or the chain ends.

If an entry is not found, it attaches the entry to the end of the chain. The switch indicating a statement number is tested so that the correct compare instructions can be used while LABTLU is searching the table. It is also tested in order that the correct move instructions are executed in moving the entry into the overflow table.

EXIT: Subroutine LABTLU exits to:

1. The subroutine that called subroutine LABTLU.
2. Subroutine ERROR if an error is detected.

Subroutines CSORN, INTCON: Chart CG

Subroutine CSORN

The functions of subroutine CSORN are:

1. To determine if the symbol is a name, constant or a data set reference number and to call the proper subroutines to process the symbol.
2. To determine how to enter the parameter in the intermediate text if a constant is a DO parameter.

ENTRANCE: Subroutine CSORN receives control from subroutines CLASSIFICATION, ARITH Part 1, BKSP/REWIND/END/ENDFILE, STOP/PAUSE, GOTO, ARITH Part 3, SUBS, EQUIVALENCE, DO, ASF, READ/WRITE, CALL, FUNCTION/SUBRTN, DIMENSION.

OPERATION: Subroutine CSORN first determines if the symbol is a name or a constant by checking the first character. If the symbol is an integer constant, subroutine CSORN checks a switch for the context in which the symbol is used. It may be a data set reference number.

By checking another switch CSORN determines if an integer constant is a DO parameter and determines the magnitude of the constant. If the constant is less then 4096, it can be carried in the displacement field of an instruction and directly in the pointer field of an intermediate text entry. It will not be entered on a chain in the dictionary. A constant greater than 4096 cannot be entered in the intermediate text or the displacement field of an instruction, and must be entered on a chain in the dictionary.

CONSIDERATIONS: DO parameters less than 4096 are entered in the object program in the displacement field of a load address instruction. Otherwise, storage has to be allocated for the constant.

EXIT: Control is passed from subroutine CSORN to the subroutine that referenced it.

SUBROUTINES CALLED: During execution, subroutine CSORN references the following subroutines:

1. LITCON to convert EBCDIC numbers to a format that can be used internally.
2. SYMTLU to make entries in the dictionary.

Subroutine INTCON

Subroutine INTCON calls a subroutine to convert integers in subscript expressions to binary numbers.

ENTRANCE: Subroutine INTCON receives control from subroutines SUBS, DIM90.

OPERATION: INTCON checks whether integer constants are properly located within the subscript expression and calls LITCON to convert a decimal number to a binary constant.

EXIT: Control is passed from subroutine INTCON to:

1. The subroutine that referenced it.
2. ERROR if an error is detected.

SUBROUTINES CALLED: During execution, subroutine INTCON references subroutine LITCON to convert numbers to an internal format.

Subroutine LITCON: Charts CH, CI, CJ

The functions of subroutine LITCON are:

1. To convert any numeric constants to a format that can be used internally.
2. To convert double precision and real constants to double-precision floating-point numbers.
3. To convert integer constants to binary full word numbers.

ENTRANCE: Subroutine LITCON receives control from CSORN, INTCON

OPERATION: Subroutine LITCON is divided into three parts, each with its own functions and objectives. Each part is discussed separately.

EXIT: Control is passed from subroutine LITCON to the subroutine that referenced it.

Subroutine LITCON Part 1

Subroutine LITCON Part 1 is entered only if the first character of a symbol is a number or a decimal point. LITCON Part 1 scans the constant, examining each character. If the character is numeric, it is added to a binary accumulator to form the number. If the character is a delimiter, control is passed to LITCON Part 2 and appropriate action is taken.

ENTRANCE: Subroutine LITCON Part 1 receives control from CSORN, INTCON, LITCON Part 2.

CONSIDERATION: Subroutine GETWD maintains two pointers. The first points to the first character of the symbol. The second points to the delimiter which stopped the translate and test instruction.

For example,

```
123.456        236E7
↑  ↑           ↑    ↑

123.456E+3     236E7+
↑  ↑           ↑    ↑
```

If the number is an integer, the pointers refer to the first digit of the constant and the delimiter which defines the end of the constant. For example:

```
123456
↑     ↑

14589+
↑    ↑
```

OPERATION: When subroutine LITCON Part 1 is entered, a register is cleared. It is used to build a binary constant. The first pointer furnished by subroutine GETWD is used to scan the constant. This pointer will be incremented by 1 each time LITCON Part 1 must examine another character. If the character is not a digit, control is given to LITCON Part 2 to process that character.

If the character is a digit the contents of the register are multiplied by 10 and the digit is added to the register. If a decimal point is encountered in the scan, control is given to LITCON Part 2 which sets a decimal indicator on and returns to Part 1. Using this indicator as a program switch, a count is maintained to indicate the number of decimal places to the right of the decimal point. This number is used in LITCON Part 3 to normalize the constant.

If an E or D is encountered in the scan, the register is saved and cleared by LITCON

Part 2. The same register is used by LITCON Part 1 to build the exponent.

EXIT: Control is passed from subroutines LITCON Part 1 to LITCON Part 2.

Subroutine LITCON Part 2

Subroutine LITCON Part 2 processes any character not a digit, that is encountered by LITCON Part 1 while it is scanning the symbol.

ENTRANCE: Subroutine LITCON Part 2 receives control from LITCON Part 1 and LITCON Part 3.

OPERATION: LITCON Part 2 sets indicators used as program switches for the three parts of LITCON. It sets one indicator if a decimal point is encountered in the scan of a symbol. Another indicator is set if the characters D or E are encountered in the scan. Either of these characters indicates that this constant is exponentiated.

When a D or E is recognized, LITCON Part 2 stores the binary number that was in the register used by Part 1, and clears the register so the LITCON Part 1 may accumulate the exponent.

Subroutine LITCON exits through Part 2 when the entire number is converted to a fixed- or floating-point number. The pointers used by GETWD to scan the remainder of the statement must be updated so that both are fixed on the character immediately following the last character of the constant.

EXIT: Control is passed from subroutine LITCON Part 2 to subroutines:

1. LITCON Part 1.
2. LITCON Part 3.
3. The subroutine that referenced subroutine LITCON.

Subroutine LITCON Part 3

Subroutine LITCON Part 3 is entered only if the constant is a real or double precision number. It converts the mantissa and characteristic generated from Parts 1 and 2 to an internal double precision number.

ENTRANCE: Subroutine LITCON Part 3 receives control from subroutine LITCON Part 2.

CONSIDERATION: All real or double-precision constants are converted into double-precision, floating-point numbers. When LITCON Part 3 was entered, one part of LITCON converted the mantissa to a binary number and stored it in main storage. The second part of LITCON converted the exponent to a binary number and placed it in a register. A count of the number of decimal places had also been kept.

OPERATION: LITCON Part 3 uses these three binary numbers to form a double-precision normalized floating-point constant. For the constant, 23.456+06, Parts 1 and 2 would place the binary expression of the decimal integer 23456 in a storage location, the binary integer 6 in a register, and a count 3 in another location.

The mantissa must be handled as an integer because the computer cannct place a physical decimal point in a field. The number 23.456 takes the form of 23456 with a count of 3 decimal places.

The hexadecimal equivalent of 23456 is 5BA0.

If the mantissa is treated as an integer, some adjustment must be made to the exponent. The decimal count is subtracted from the original exponent. In our example the exponent is changed to 3 and the decimal count is cleared.

$23.456*10^{6}=23456*10^{3}$

23.456E+06=23456E+03

The exponent is changed from +6 to +3. If the result of the subtraction is negative, a switch is set to indicate a negative result, and the result is set to its absolute value.

Up to this point, then, we have a hexadecimal mantissa 5BA0 and a decimal exponent, 3. A double word is then established in storage; the first byte contains the hexadecimal number 4E, and the rest of the bytes contain hexadecimal zeros. That double word is:

4E00000000000000

The sign bit is set to zero. The hexadecimal mantissa is then "ored" to the second word of this double word. Our double word then becomes

$4E00000000005BA0)_{16}$

In hexadecimal notation this means

$.00000000005BA0)_{16} * 16^{14}$

Actually, the double word is an unnormalized System/360 floating-point constant.

Characteristic 4E, is the exponent 14 in excess 64 notation. This is explained in the System Reference Library publication, IBM System/360 Principles of Operation, Appendix C.

To normalize the constant, the double word is added to a double floating-point register, which contains floating-point zero. The result of this operation is:

445BA00000000000

or

.5BA0 * 16^4

Finally, the decimal exponent must be used to adjust this double word. A switch is set to indicate a positive or negative exponent, and the exponent is set to its absolute value. If the exponent is negative, the double word is divided by the value 10 ** exponent. If it was positive, the double word is multiplied by the value, 10 ** exponent.

LITCON Part 3 uses a table with floating-point values for various exponents of 10. In our example the exponent 3 indicates a value of 1000 in the decimal number system. Using the table, Part 3 finds that 1000 is 3E8 in the hexadecimal system. This hexadecimal number is converted to a floating-point constant and used to multiply the number that is currently in the double floating-point register, 445BA00000000000. The result of this multiplication would be:

47165E90000000000

or

.165E900 * 16^7

Then,

23.456E+06=47165E90000000000,

and the floating-point number is converted to an internal machine constant.

EXIT: Control is passed from subroutine LITCON Part 3 to:

1. The subroutine that called subroutine LITCON if an error is detected.
2. Subroutine LITCON Part 2 to exit from subroutine LITCON.

Subroutine SUBS: Chart CI

Subroutine SUBS processes all subscripted variables in arithmetic expressions, making the necessary entries to the dictionary, overflow table, and the intermediate text.

ENTRANCE: Subroutine SUBS is referenced by subroutines ARITH Part 1 and ARITH Part 2.

OPERATION: Subroutine SUBS processes variables with one, two, or three subscripts. The subscripts must conform to the general FORTRAN subscript expression:

C1*V1+J1

where C1 and J1 are unsigned integer constants, and V1 is an integer variable.

The overflow table entry for a subscripted variable is assembled in the overflow buffer. Just before control is returned to the calling routine, the contents of the overflow buffer are inserted in the overflow table by use of subroutine LABLTU.

Constants of the form J1 are stored in main storage until control is about to be given to the calling subroutine. The offset is then computed using these constants and the dimension information entered for this array in the overflow table, when the array was defined by a DIMENSION statement. The offset is then entered in the intermediate text.

EXIT: The utility subroutine SUBS exits to:

1. The subroutine which has called it.
2. Subroutine ERROR if an error is detected.

SUBROUTINES CALLED: During execution subroutine SUBS references subroutines:

1. GETWD to access symbols and delimiters.
2. INTCON to process integer constants.
3. SKPBLK to access delimiters.
4. CSORN to process dictionary entries.
5. LABTLU to process overflow table entries.

Subroutines DIMSUB, DIM90: Chart CM

Subroutine DIMSUB

Subroutine DIMSUB scans the subscript portion of a statement that is dimensioning an array. It also inserts the dimension information into the overflow table entry and the size of the array in the dictionary.

ENTRANCE: The utility subroutine DIMSUB is referenced by subroutine DIMENSION.

OPERATION: The type code is set to represent an array because Phase 10 has now determined that this statement defines an array.

Subroutine DIMSUB uses another subroutine, DIM90, to actually compute the constants entered in the overflow table.

EXIT: Subroutine DIMSUB exits to:

1. The subroutine that called it.
2. ERROR if an error has occurred.

SUBROUTINES CALLED: During execution subroutine DIMSUB references subroutines:

1. DIMGO to compute constant.
2. WARNING/ERRET if an error is detected.

Subroutine DIM90

Subroutine DIM90 computes the constants, D1*L and D1*D2*L, which are inserted in the overflow table, and the size of the array (D3*D2*D1*L) which is inserted in the size field of the dictionary where the general form for the array is:

ARRAY (D1,D2,D3)

ENTRANCE: The utility subroutine DIM90 is referenced by subroutine DIMSUB.

OPERATION: Subroutine DIM90 uses GETWD and INTCON to get the integer and convert it. Then for the first subscript it computes the product D1*L and saves the result. If the array has only one dimension, this product is the size of the array. If there is more than one dimension for this array, the product D2*D1*L is computed. If the array has only two dimensions, that is the size of the array. If there is another dimension for the array, the product D3*D2*D1*L is computed. This product is the size in bytes of a 3-dimensional array. For information concerning the format of these entries in the overflow table, see the introduction to Phase 10.

EXIT: Subroutine DIM90 exits to:

1. The subroutine that called it.
2. Subroutine ERROR if an error is detected.

SUBROUTINES CALLED: During execution subroutine DIM90 references subroutines:

1. GETWD to access symbols and delimiters.
2. SKPBLK to access delimiter.
3. INTCON to process integers.

Subroutine END MARK CHECK: Chart CN

Subroutine END MARK CHECK calls subroutine PUTX to write the end mark entry for the majority of FORTRAN statements. It also tests for the card reader end of data set or if the END card has been read. Part of END MARK CHECK is used to find redundant commas.

ENTRANCE: Subroutine END MARK CHECK receives control from subroutines EXTERNAL, CONTINUE/RETURN,STOP/PAUSE, GOTO, ARITH Part 3, EQUIVALENCE, DO, ASF, CALL, FUNCTION/SUBRTN, DIMENSION.

OPERATION: Subroutine END MARK CHECK is composed of two distinct sections. One section with entry points RCOMA, RCOMA1, RCOMA2, and RCOMA3 is entered if no intermediate text is written for this statement. It is entered by subroutines processing COMMON, EQUIVALENCE, DIMENSION, REAL, INTEGER, DOUBLE PRECISION, and EXTERNAL statements. It enters the second portion from the first portion of subroutine END MARK CHECK, only if a warning message must be issued.

A portion of the first section issues warning messages when used as a subroutine to skip redundant commas. The return to the subroutine that called it returns to a program step above the call instruction. The compiler then stays in a loop until all redundant commas have been skipped.

The first section of END MARK CHECK sets off all type switches used to direct Phase 10 through explicit specification statements.

The second portion of subroutine END MARK CHECK has entry points EOSR, EOSR1, EOSR2, EOSR2A, and EOSR3. This section is entered if intermediate text is written for this statement.

This portion of subroutine END MARK CHECK calls subroutine ASF if the arithmetic statement function switch is set. Subroutine ASF then resets dictionary entries and defaces those made to define the statement function arguments.

EXIT: Subroutine END MARK CHECK exits to:

1. Subroutine CLASSIFICATION to process another source card entry.
2. The subroutine that called it if entry was made at block RCOMA.
3. Subroutine ASF to finish processing an

arithmetic statement function.
4. FORTRAN System Director to load Phase 12.
5. Subroutine BKSP/REWIND/END/ENDFILE to simulate an end card being read.

SUBROUTINES CALLED: During execution subroutine END MARK CHECK references subroutines:

1. GETWD to access symbols and delimiters.
2. SKTEM to skip to the end mark.
3. PUTX to make entries to the intermediate text.
4. WARNING/ERRET if a warning is detected.

Subroutine PUTX, PUTBUF, PUTRET: Chart CO

Subroutine PUTX makes entires to the intermediate text buffer area, consisting of an adjective code, type code, and an address pointing to an entry either in the dictionary or the overflow table. If a buffer area is full, PUTX gives control to the FORTRAN System Director in order to write a tape record and free the buffer.

Subroutines PUTBUF and PUTRET are parts of PUTX which are used for specific functions in some Phase 10 subroutines. PUTBUF is called by subroutine END MARK CHECK to output the buffers at the end of Phase 10 execution.

PUTRET is called by subroutines not making standard text entries (e.g., FORMAT statements) to the intermediate text buffers to check if a buffer area is full.

ENTR NCE: The utility subroutine PUTX is referenced by
ARITH Part 1,
BKSP/REWIND/END/END FILE, GOTO, ARITH Part 3, DO, SUBIF,
READ/WRITE, CALL, FUNCTION/SUBRTN,
CONTINUE/RETURN,STOP/PAUSE, FORMAT, END MARK CHECK

OPERATION: When the translate and test instruction senses the first delimiter in the statement under consideration, that delimiter is placed in DELIM. PUTX then moves the contents of ADJ to the intermediate text buffer area, and then moves the contents of DELIM to ADJ. Subroutine ARITH Part 1, subroutine ASF, or a keyword subroutine uses a special adjective code for the first intermediate text entry for that statement. These codes are moved directly to ADJ.

Subroutine PUTX is entered at different points depending on the information the calling subroutine has to enter in the intermediate text. One entry point exists in PUTX for a subroutine that has a statement number to be entered in text, a separate entry point exists for a subroutine that has an adjective code to be entered, etc.

Subroutines SYMTLU and LABTLU place the address of the dictionary or overflow table entry in a general register. PUTX moves that pointer from the general register to the intermediate text.

The type code is moved directly to the intermediate text from the mode/type field in the dictionary of overflow table.

EXIT: Subroutine PUTX exits to the subroutine that called it.

SUBROUTINES CALLED: The utility subroutine PUTX will reference the FORTRAN System Director to write the output buffers on tape.

Subroutines ERROR, WARNING/ERRET: Chart CP

Subroutine ERROR

Subroutine ERROR creates the intermediate text entry for an error message. Errors are not printed in Phase 10. Entries which indicate to Phase 30 that an error message should be printed are made to the intermediate text. The remainder of the statement in which the error occurred is not processed. An indicator is set in the Communications area so that the other phases of the compiler know an error has occurred.

ENTRANCE: The utility subroutine ERROR is referenced by subroutines EXTERNAL, INTEGER/REAL/DOUBLE, CLASSIFICATION, ARITH Part 1, ARITH Part 2, ARITH Part 3, CONTINUE/RETURN, STOP/PAUSE, BKSP/REWIND/END/ENDFILE, GOTO, SUBS, SYMTLU, EQUIVALENCE Part 1, EQUIVALENCE Part 2, DO, PAKNUM, LABTLU, ASF, SUBIF, INTCON, READ/WRITE, FUNCTION/SUBRTN, COMMON, DIMENSION, DIMSUB, DIM90, END MARK CHECK.

OPERATION: The intermediate text entry for an error message is the error adjective code, an error number which is inserted in the position normally occupied by a mode/type code, and the internal statement number of the statement in which the error was detected.

The error number is retrieved in an unusual manner. When an error condition is found in any of the statements processed by

Phase 10, a branch is taken to an instruction in a table of branch instructions. Each of these branch instructions represents a particular error message. All of the instructions are branch and link instructions to the subroutine ERROR. Subroutine ERROR makes use of an address constant which is the address of the beginning of the branch table.

The branch table list in Phase 10 appears as follows:

```
ERR1   BAL 13,ERROR
ERR2   BAL 13,ERROR
ERR3   BAL 13,ERROR
  .      .      .
  .      .      .
  .      .      .
ERR27  BAL 13,ERROR
  .      .      .
  .      .      .
  .      .      .
```

The subroutine which branches to a point in the branch table determines the nature of the error, and which error message is to be generated. If the calling subroutine has determined that this is error #27, it will issue this instruction:

```
BC    15,ERR27
```

When the computer executes the instruction located at ERR27 it branches to subroutine ERROR and saves the address from which it branched to ERROR in register 13. Each instruction in the branch table places its address in register 13, and each branch instruction has a particular error message associated with it.

If the beginning address of the branch table (the address of the instruction labeled ERR1) is loaded as an address constant in subroutine ERROR, the error message number can be computed by subtracting the beginning address from the address loaded into the register by the branch and link instruction, and then dividing by 4. The length of a branch and link instruction is one word or 4 bytes.

A program switch is set any time subroutine ERROR is entered. If an error has occurred in statements of the program written by the user, the compiler knows that it cannot compile the program properly. It does not generate the machine language coding necessary to run the object program. Instead, if the GOGO option is not on, after Phase 20 is completed, it enters Phase 30 which will use the error entries in the intermediate text to print error message.

EXIT: Subroutine ERROR exits to subroutine END MARK CHECK.

SUBROUTINES CALLED: During execution subroutine ERROR references subroutine PUTRET to see if the intermediate text output buffers are full.

Subroutine WARNING/ERRET

Subroutine WARNING/ERRET enters a warning or an error message in the intermediate text. Subroutine WARNING/ERRET attempts so recover and continue processing the statement, whereas subroutine ERROR goes directly to subroutine END MARK CHECK and aborts the rest of the statement from the compilation.

ENTRANCE: Subroutine WARNING/ERRET is entered to generate a warning message by subroutines CLASSIFICATION, CONTINUE/RETURN BKSP/REWIND/END/ENDFILE, GOTO ARITH Part 3, EQUIVALENCE Part 1, DIMENSION, DIMSUB, END MARK CHECK. Subroutine WARNING/ERRET is entered to generate an error message by subroutine DIMSUB.

OPERATION: A warning does not force the compilation to be ended at Phase 20 as an error does. The compiler generates the object coding for the FORTRAN source program, and then calls Phase 30 to process the warning messages that were entered in the intermediate text during Phase 10. A warning would occur if a statement such as:

```
DIMENSION, A(20),B(2,2,2)
```

were processed by Phase 10. The comma between the names DIMENSION and A is redundant.

The same problem for error and warning messages processed by subroutine WARNING/ERRET does not develop as it did in subroutine ERROR. The error or warning message number is inserted in a register. The contents of the register are then stored in the intermediate text entry for that error or warning message.

Every time subroutine WARNING/ERRET is entered when a warning has occurred, a bit is set on in the Communications area to indicate that at least one of the source statements has a condition which merits a warning message. If this switch is on, Phase 30 is called after Phase 25 has been completed to process any warning messages placed in the intermediate text.

If subroutine WARNING/ERRET is called because an error has occurred, the same switch set by subroutine ERROR is set. This switch indicates to the compiler not to call Phase 25 to assemble the machine language instructions. Instead, Phase 30

is called at the end of Phase 20 to process the errors.

When an attempt is made to re-enter the subroutine that called WARNING/ERRET, the intermediate text messages normally entered for errors must be saved until the statement has been completely processed. A bit is set in the communications area to indicate that entries for warnings and errors must be made to the text. When the end of statement is reached, subroutine END MARK CHECK tests this bit and enters the entries for warnings and errors to the intermediate text after the end statement entry has been made to the intermediate text.

EXIT: Subroutine WARNING/ERRET exits to the subroutine that called it.

Subroutine PRINT: Chart CQ

Subroutine PRINT assembles a line to be printed and calls the FORTRAN System Director to print the line.

ENTRANCE: The utility subroutine PRINT is referenced by subroutine GETWD.

OPERATION: Subroutine PRINT always prints the card image of the card, and the internal statement number that has been assigned to this statement by subroutine CLASSIFICATION. The internal statement number has been assigned to this statement before it is processed.

Subroutine PRINT through use of a Supervisor Call instruction calls the FORTRAN System Director to print a line. The FSD will then call the I/O routine that commands the printer.

EXIT: Subroutine PRINT exits to the subroutine that called it.

SUBROUTINES CALLED: During execution subroutine PRINT references the FORTRAN System Director to print a source card.

Subroutine GET: Chart CR

Subroutine GET reads a new card image and switches the buffers in the double buffering scheme.

ENTRANCE: Utility subroutine GET is referenced by subroutine GETWD.

OPERATION: Subroutine GET calls the FORTRAN System Director to read cards. Control may be returned to subroutine GET by two exits. The first is for normal processing. The second exit is used if the last card has been read from the card reader. A switch is turned on signifying the end of file for the card reader.

EXIT: Subroutine GET exits to the subroutine that call subroutine GET.

SUBROUTINES CALLED: During execution subroutine GET references the FORTRAN System Director to print a source card. Text is written for this statement.

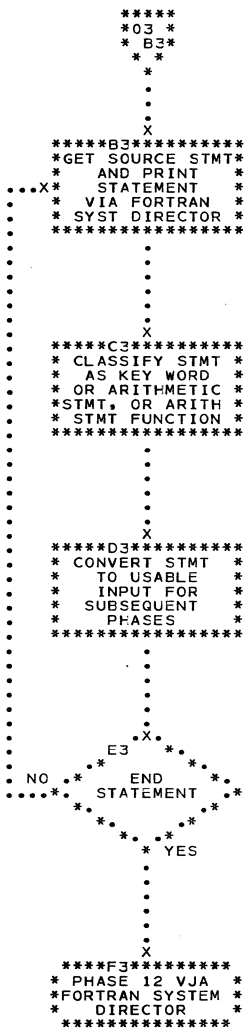


Chart 03. Phase 10 Overall Logic Diagram

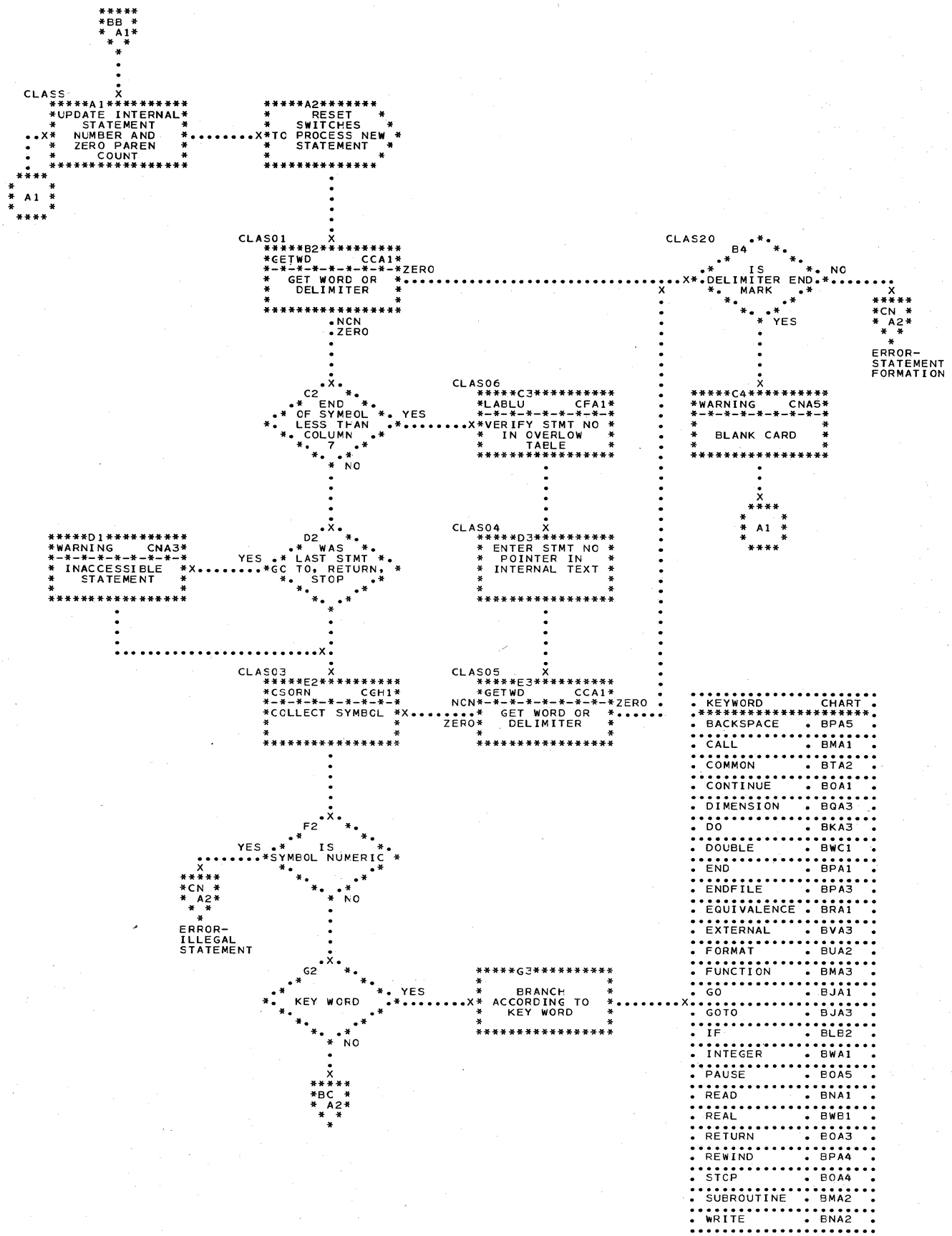


Chart BB. Subroutine CLASSIFICATION

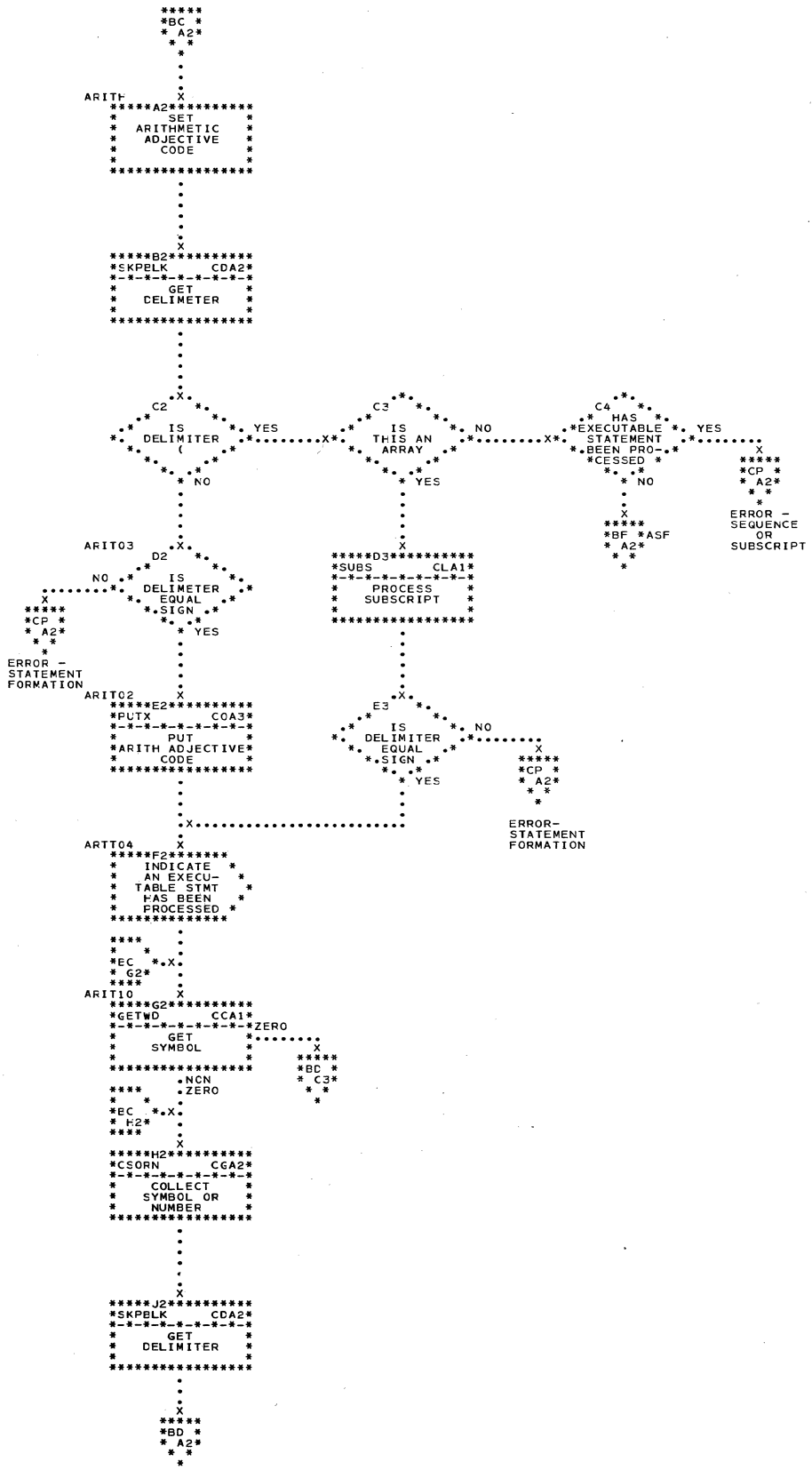


Chart BC. Subroutine ARITH Part 1

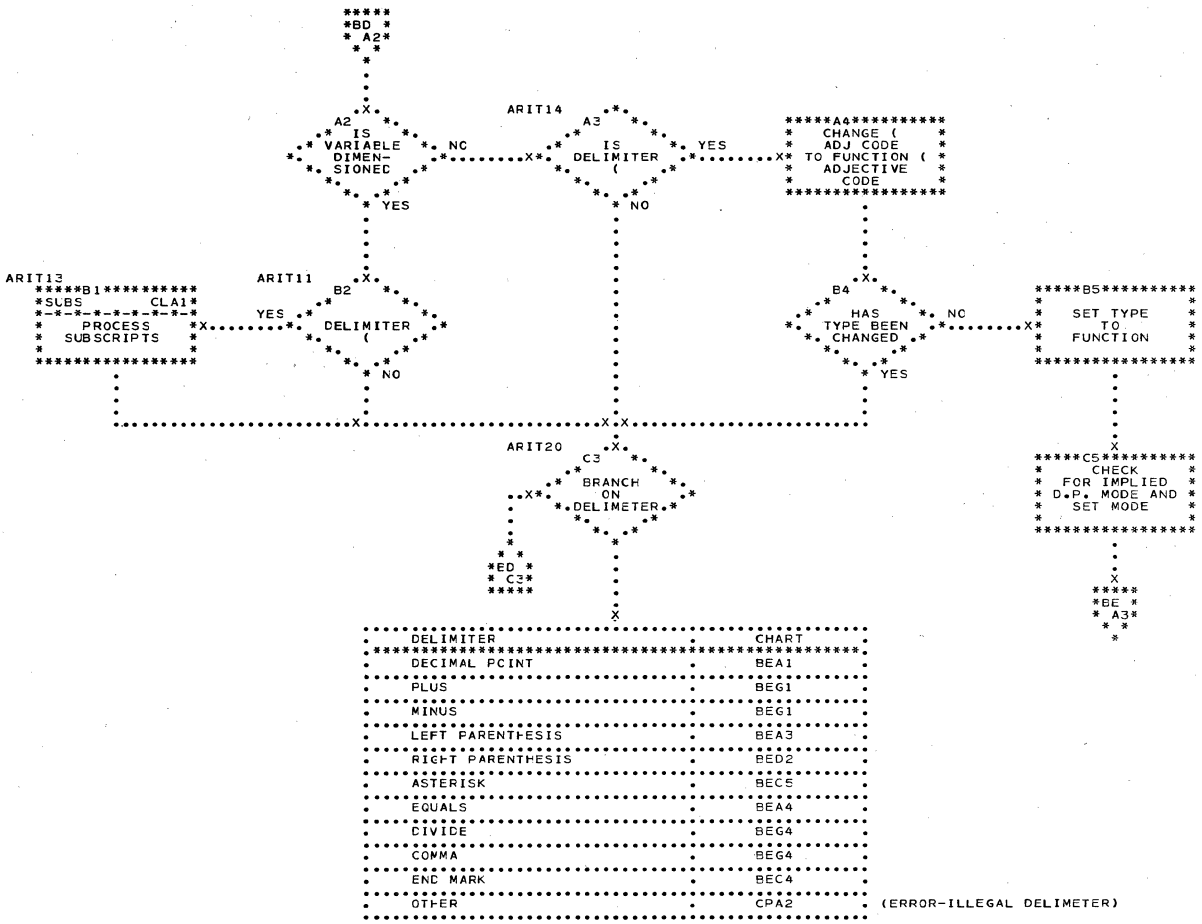


Chart BD. Subroutine ARITH Part 2

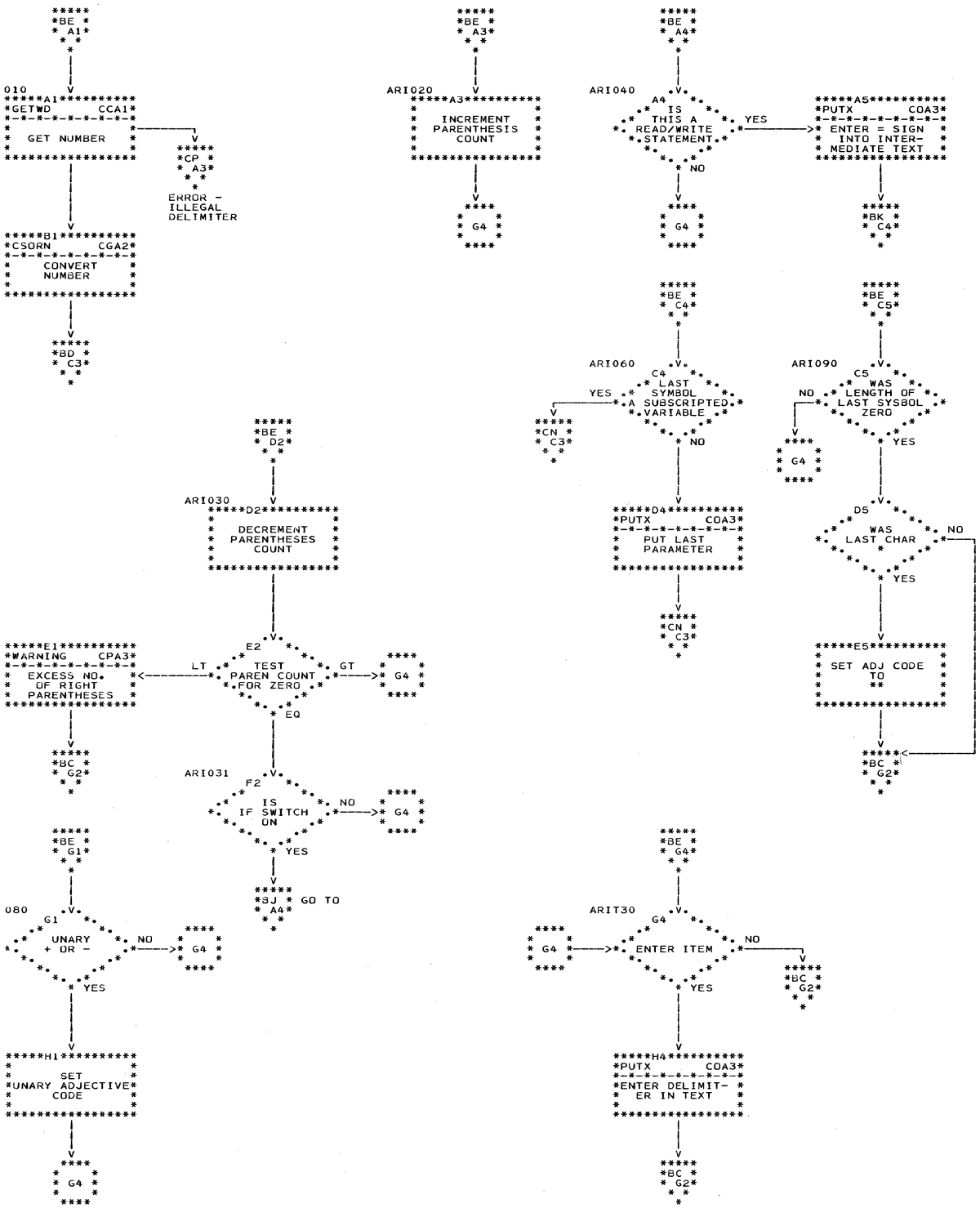


Chart BE. ARITH Part 3

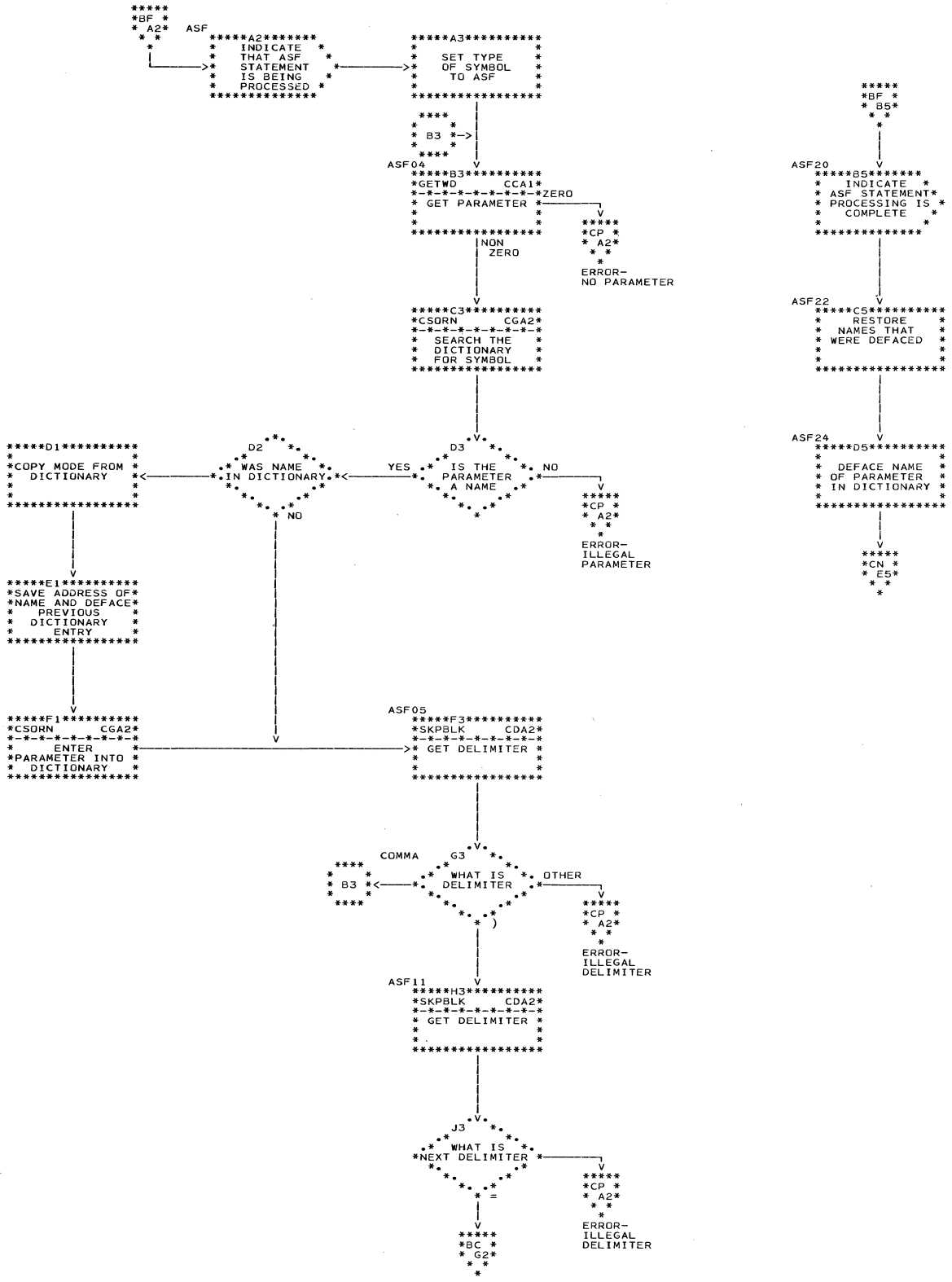


Chart BF. Subroutine ASF

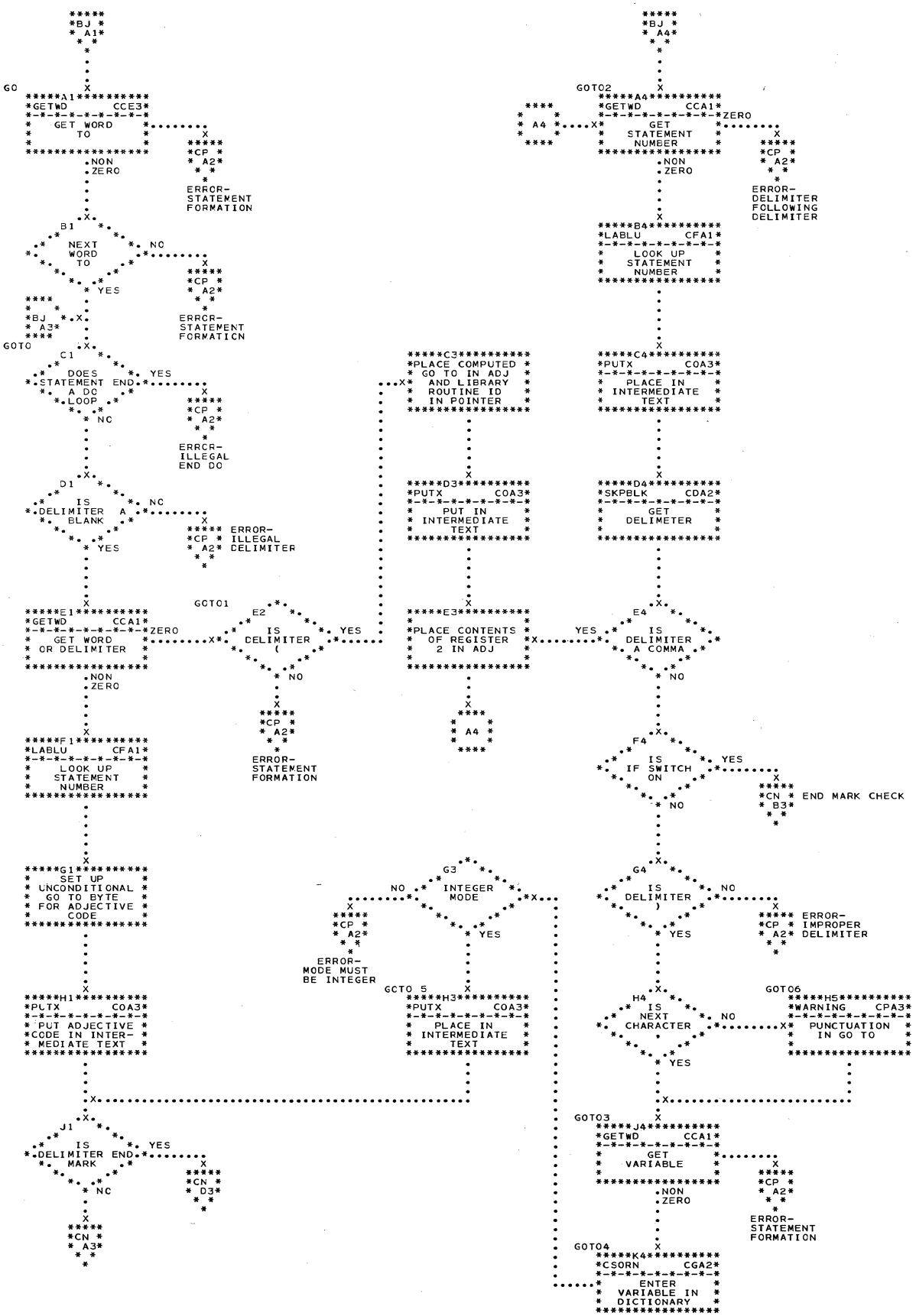


Chart BJ. Subroutine GOTO

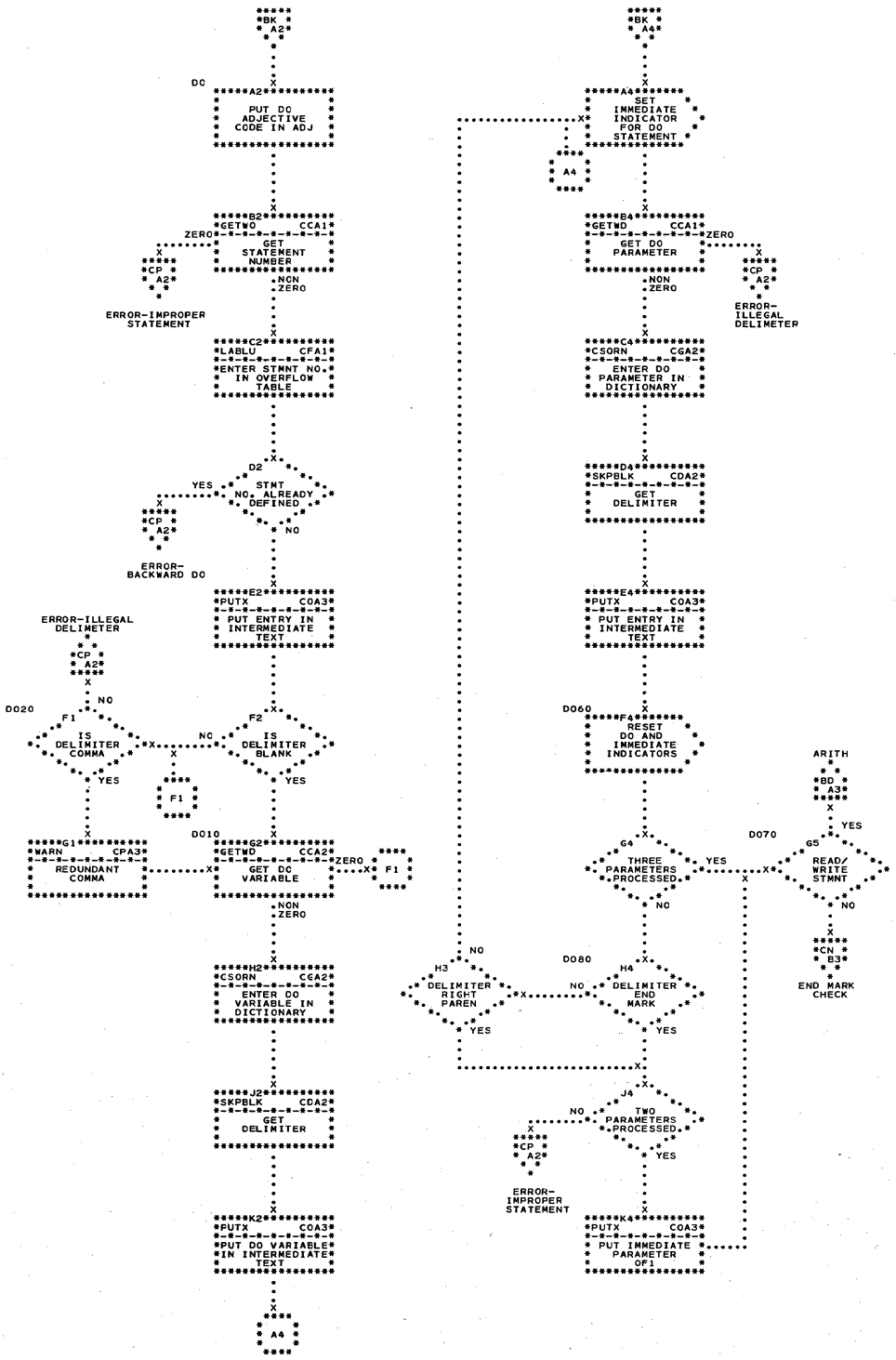


Chart BK. Subroutine DO

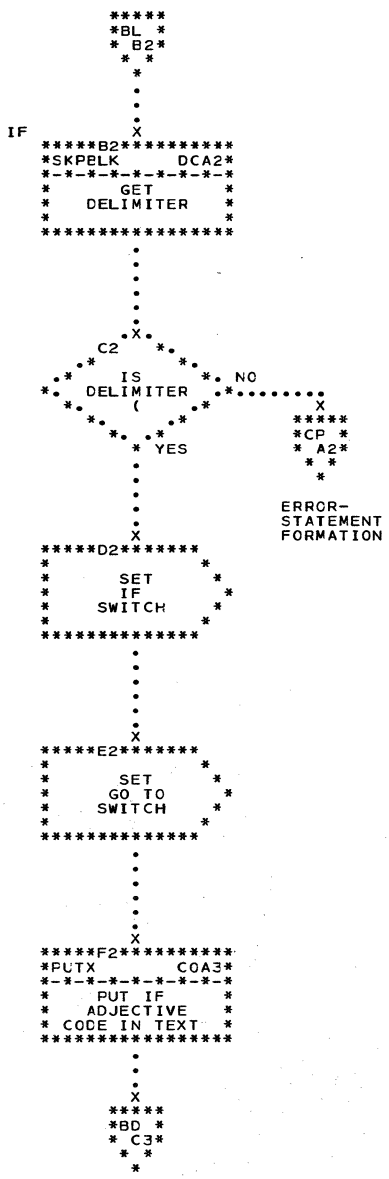


Chart BL. Subroutine SUBIF

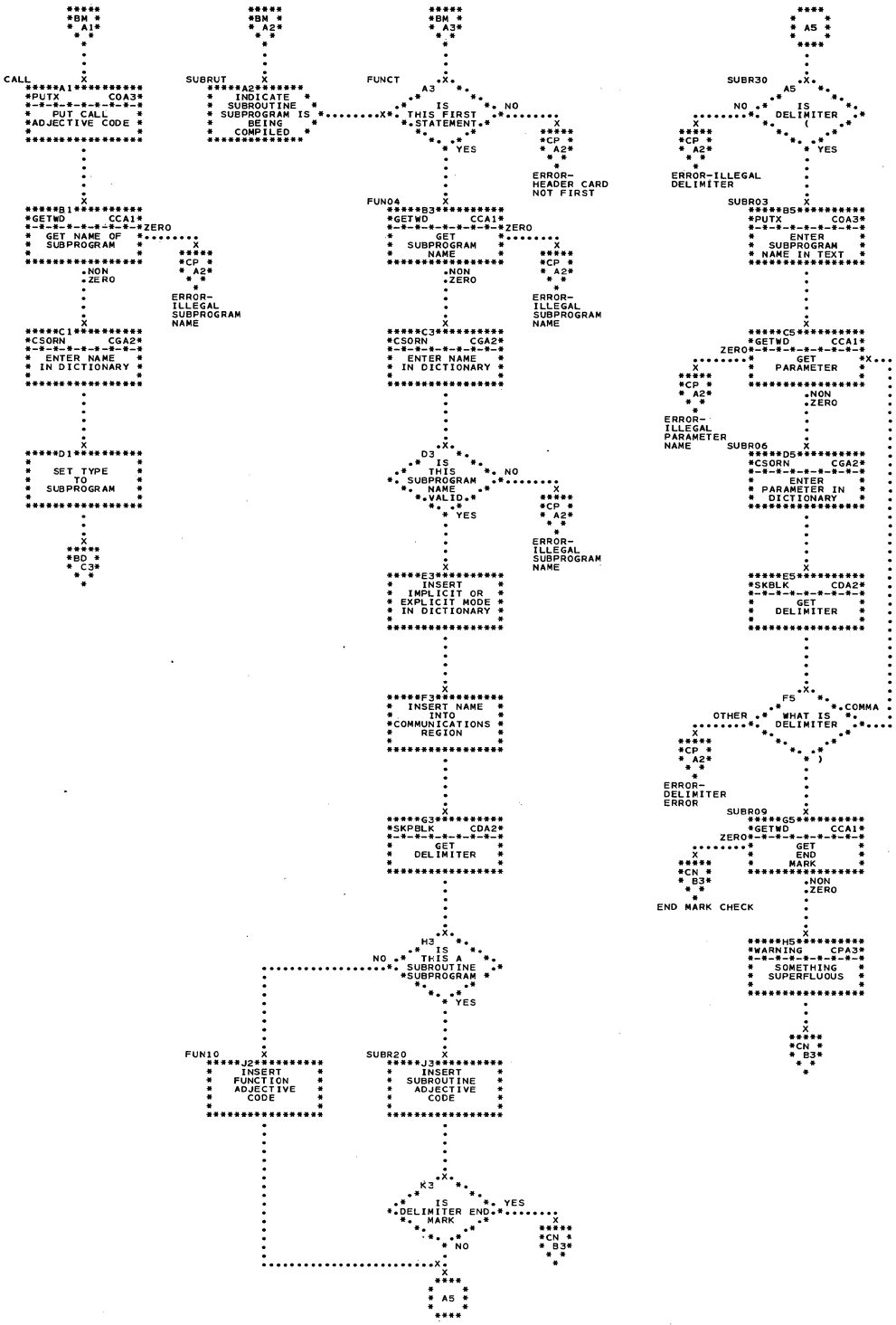


Chart BM. Subroutines CALL, FUNCTION/SUBRTN

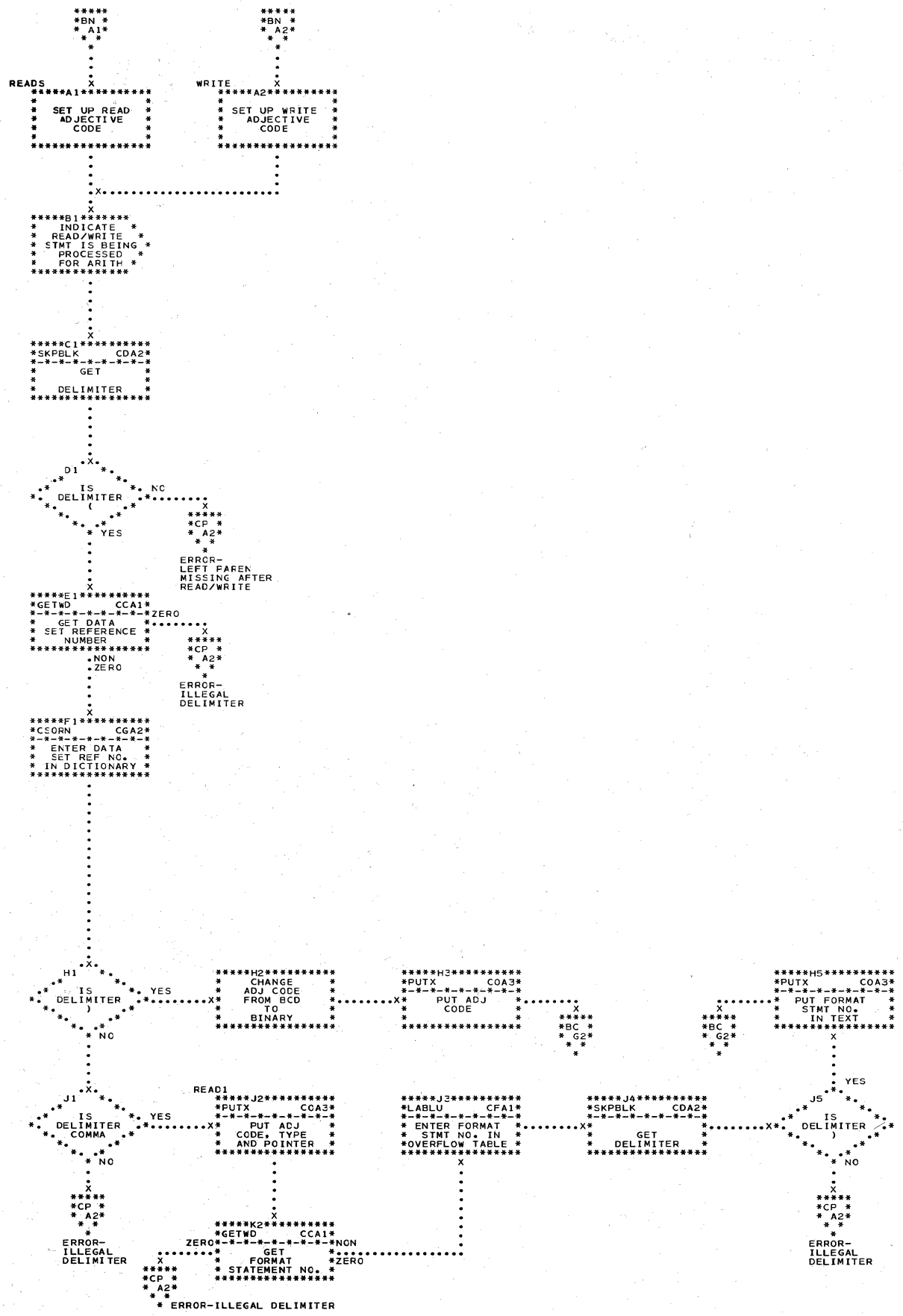


Chart BN. Subroutine Phase 10 READ/WRITE

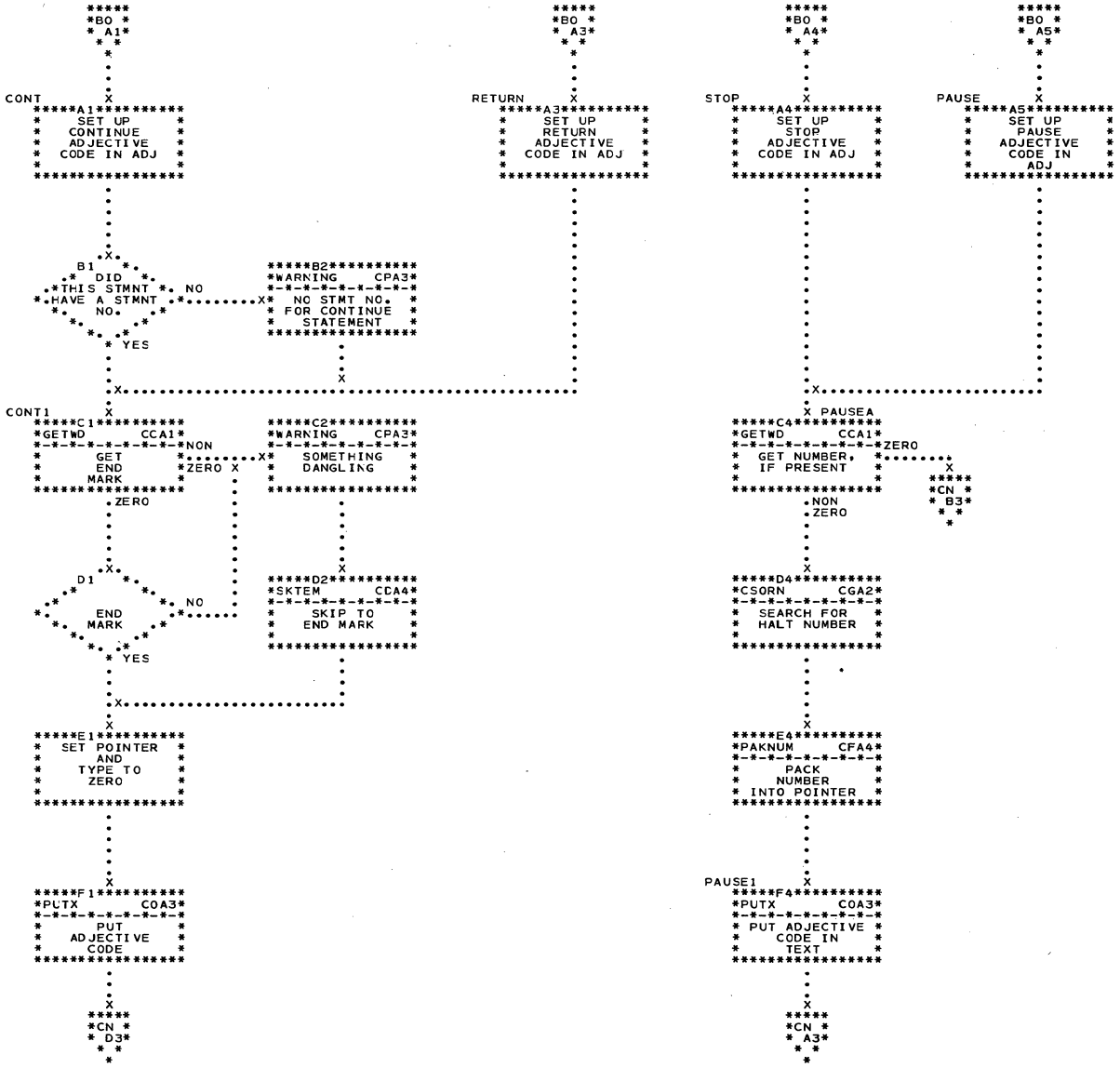


Chart BO. Subroutines CONTINUE/RETURN, STOP/PAUSE

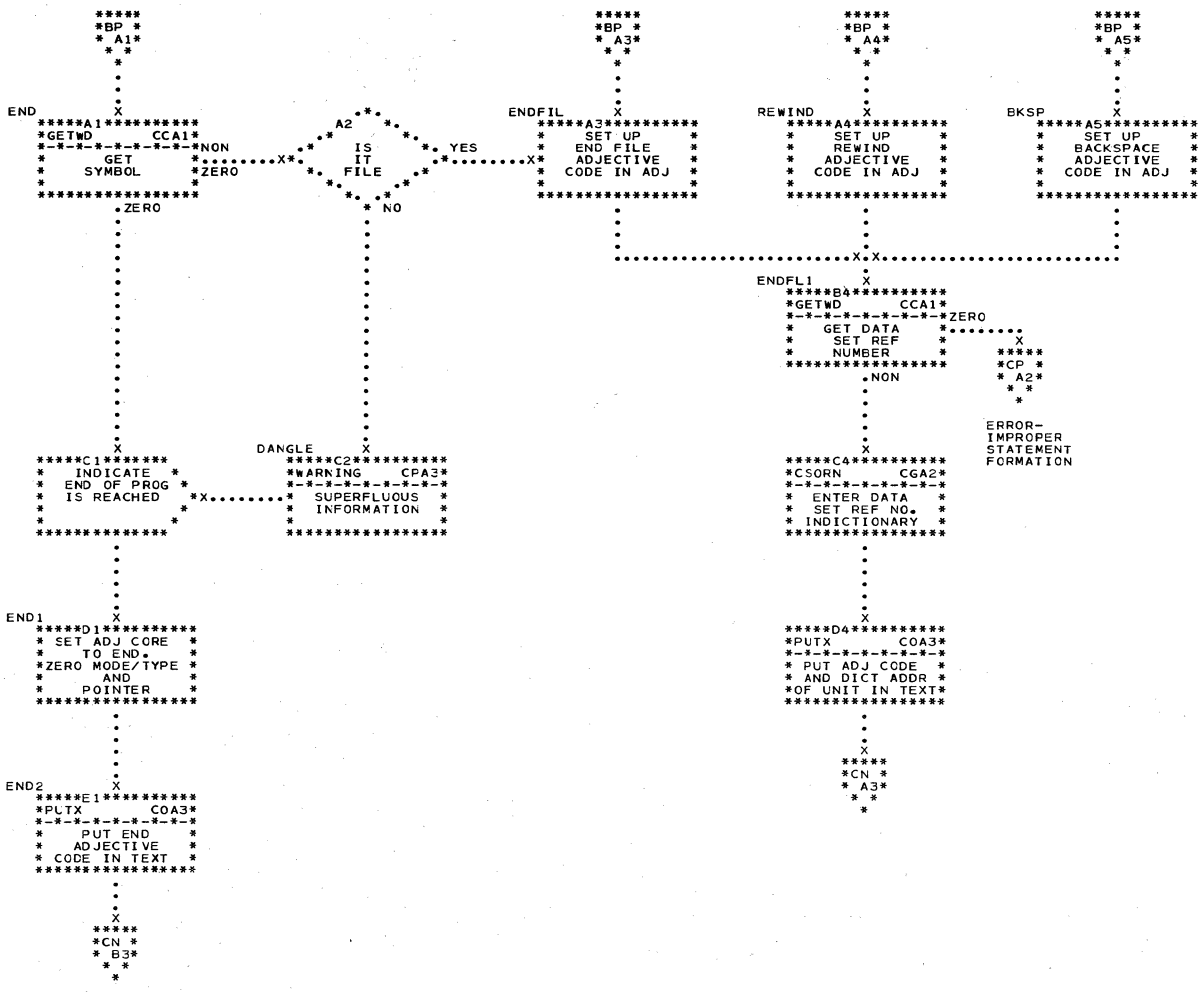


Chart BP. Subroutine BKSP/REWIND/END/ENDFILE

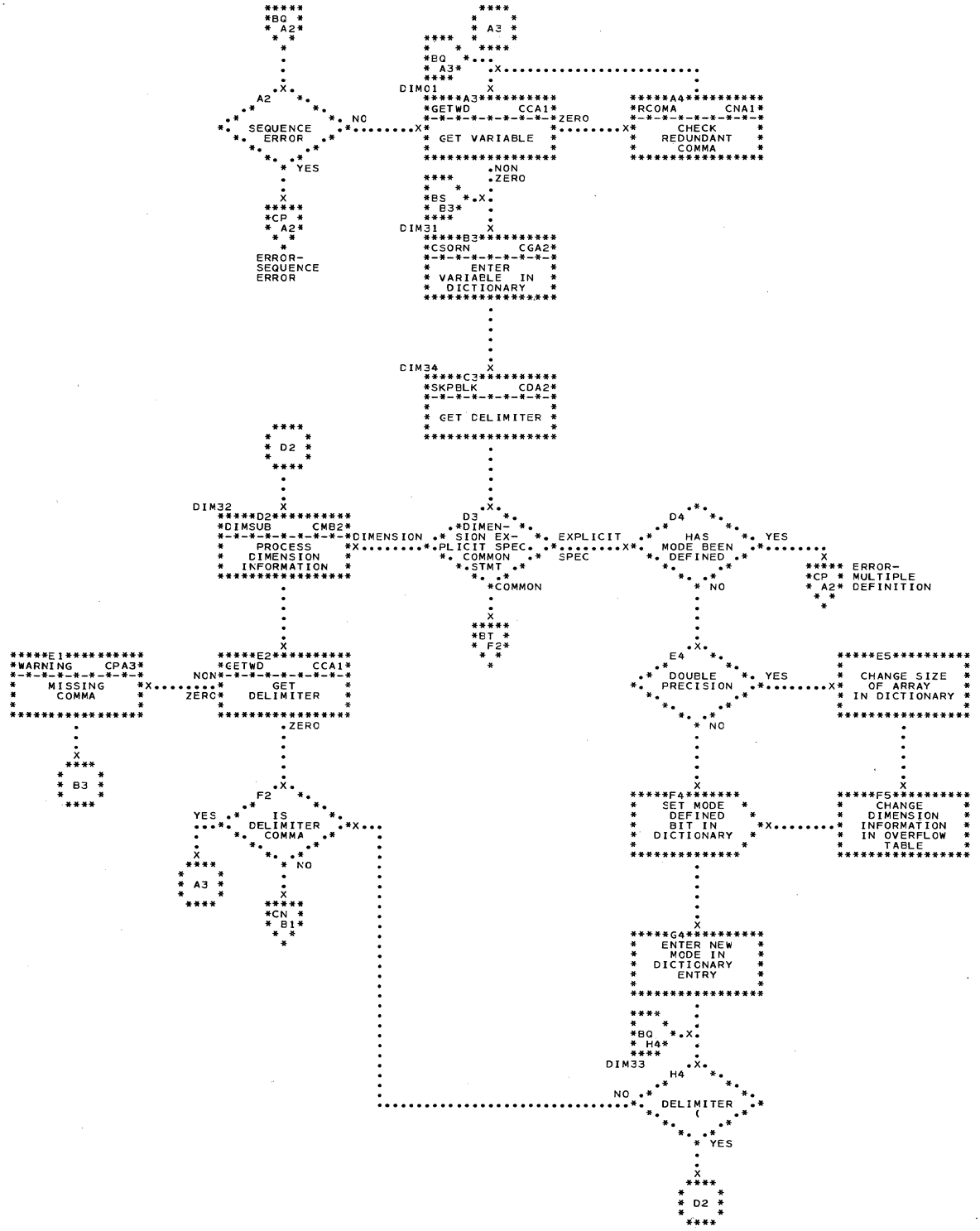


Chart BQ. Subroutine DIMENSION

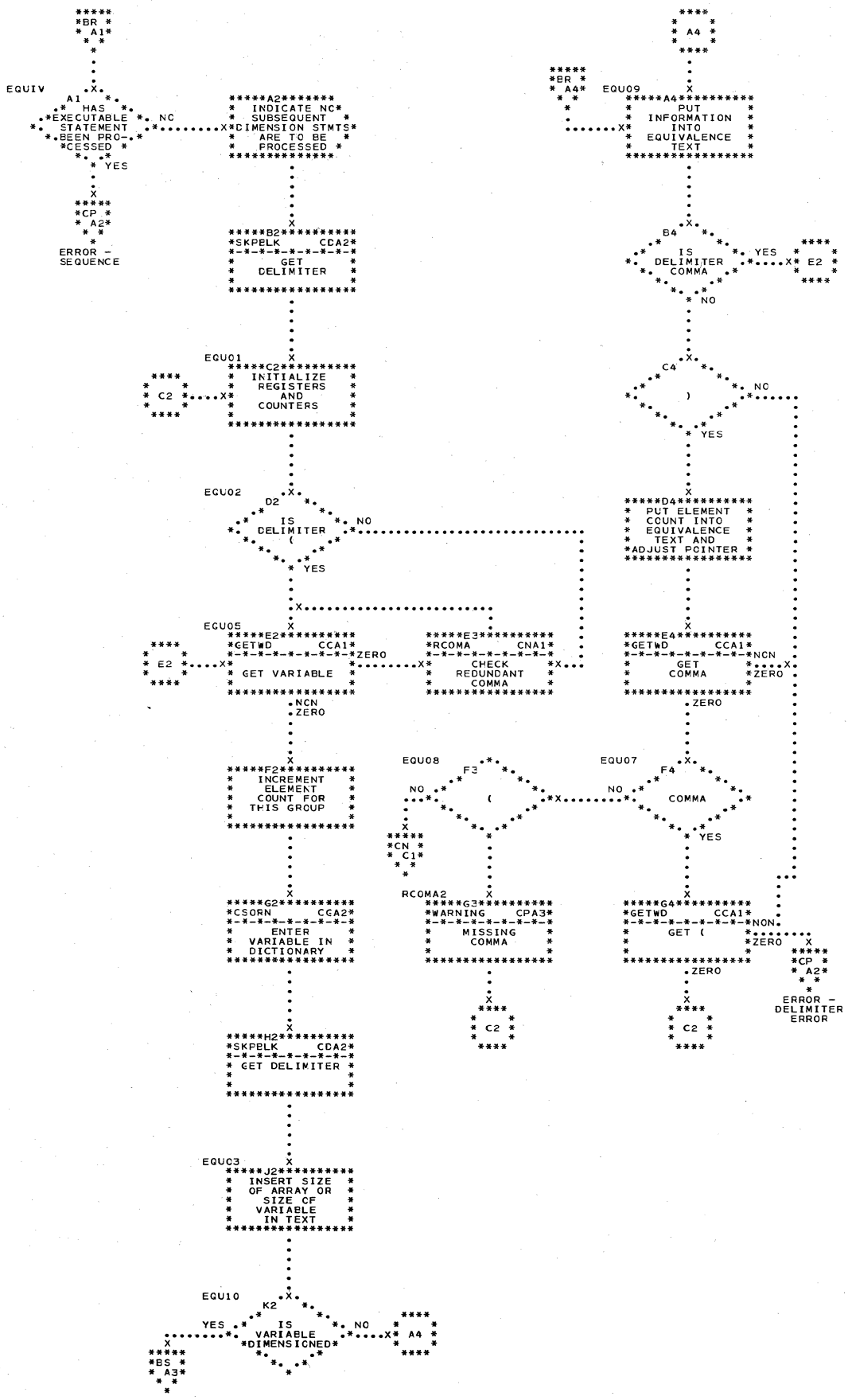


Chart BR. Subroutine EQUIVALENCE Part 1

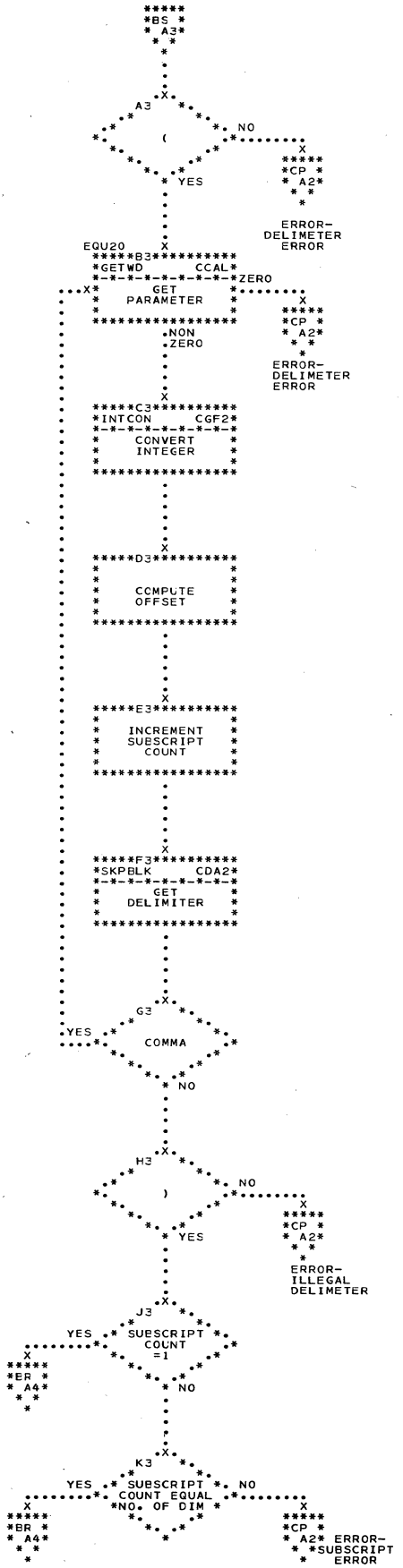


Chart BS. Subroutine EQUIVALENCE Part 2

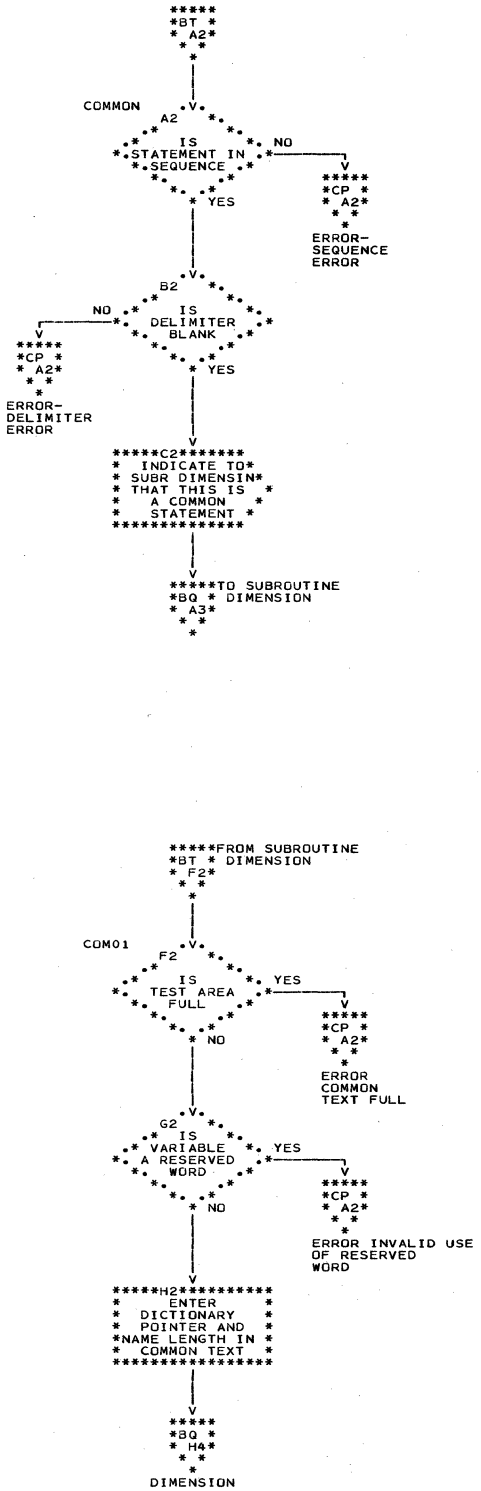


Chart BT. COMMON Routine

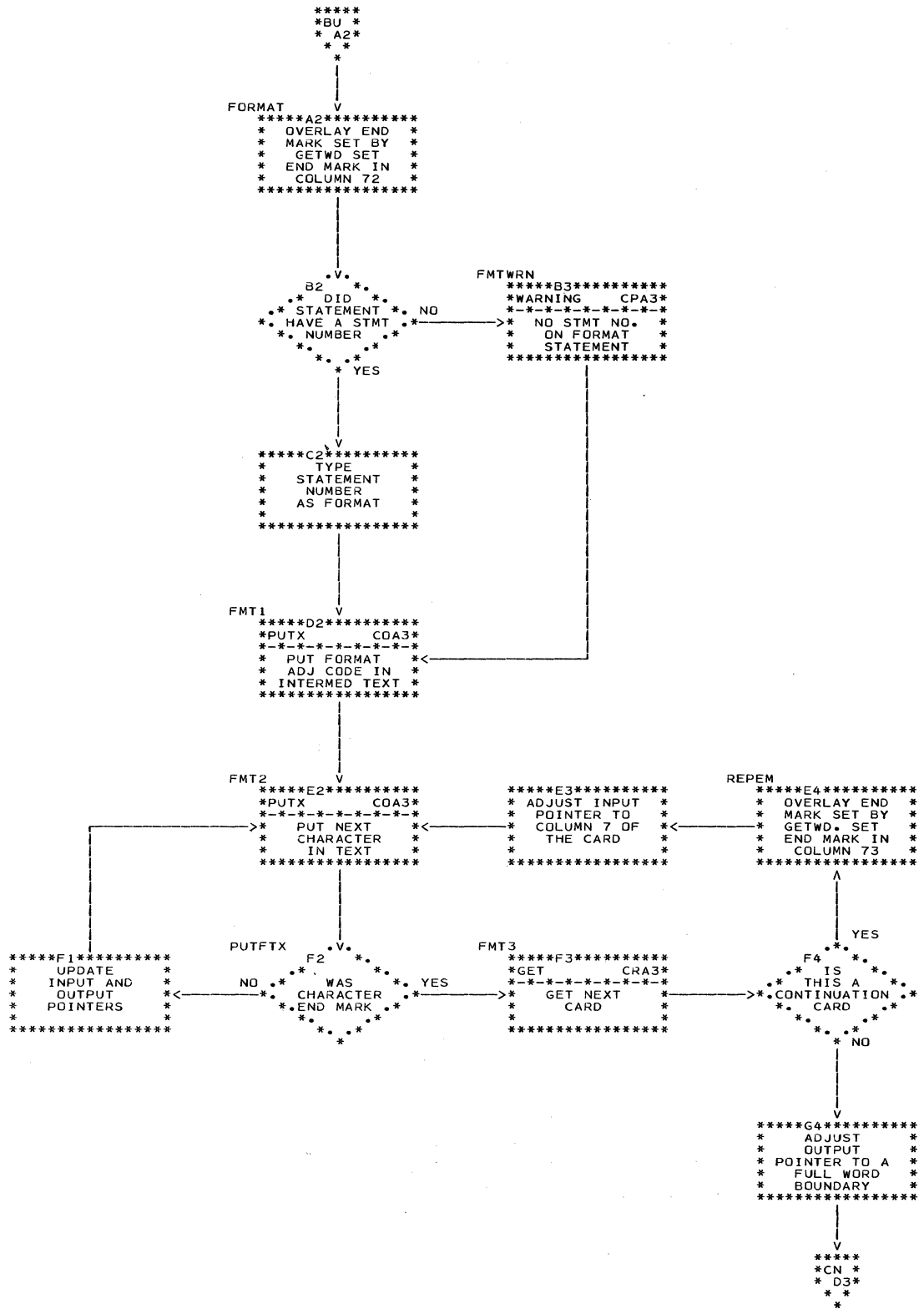


Chart BU. Subroutine FORMAT

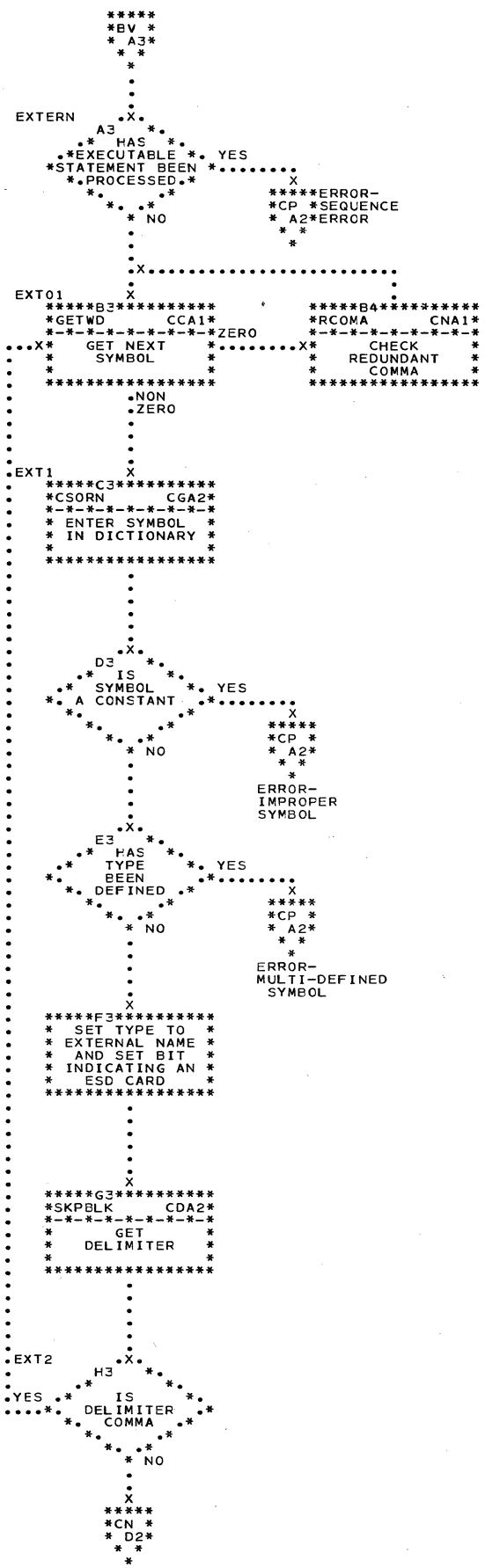


Chart BV. Subroutine EXTERNAL

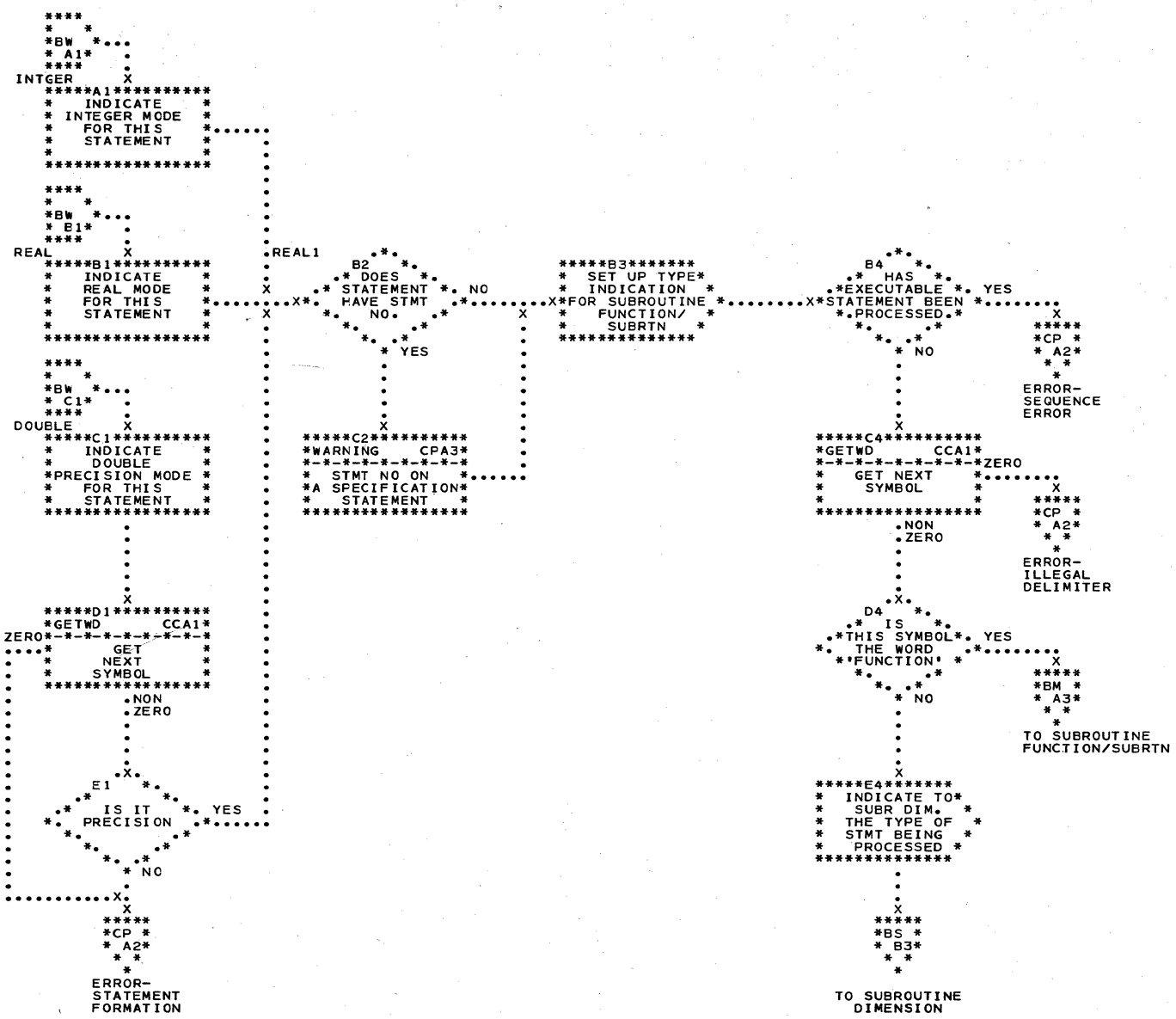


Chart BW. Subroutine INTEGER/REAL/DOUBLE

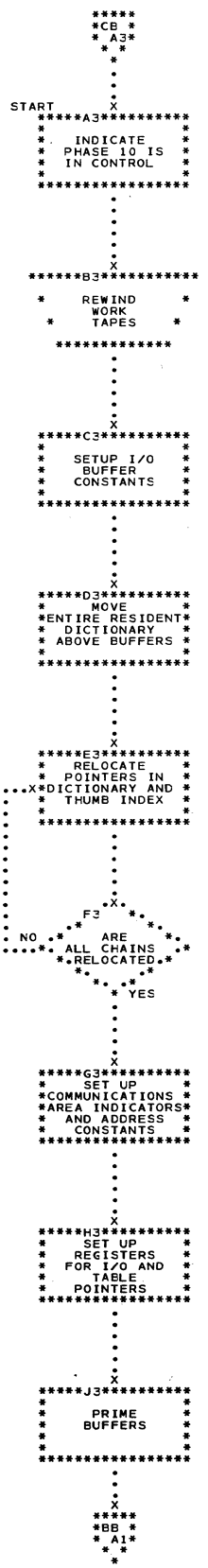


Chart CB. Subroutine Phase 10 HOUSEKEEPING

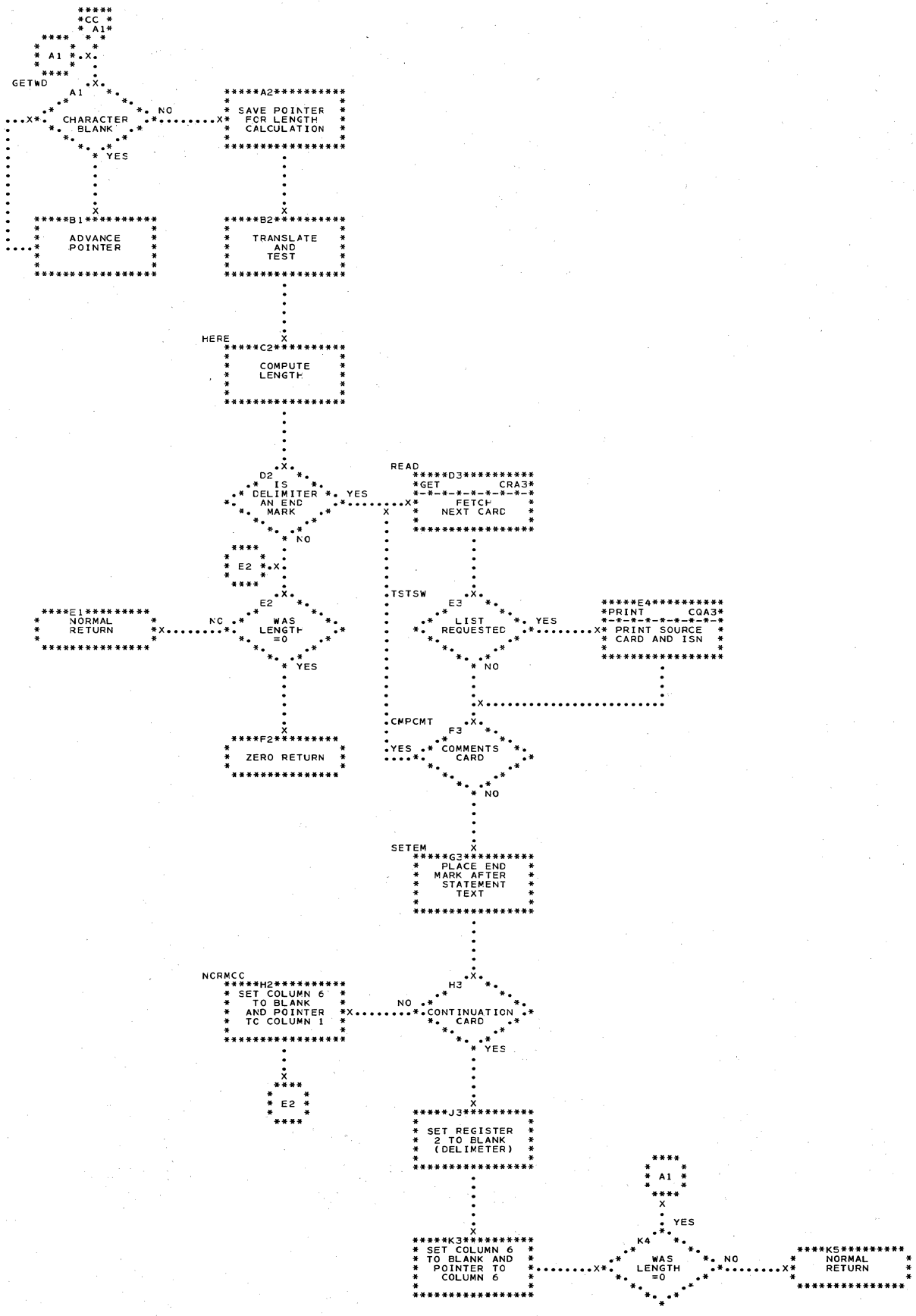


Chart CC. Subroutine GETWD

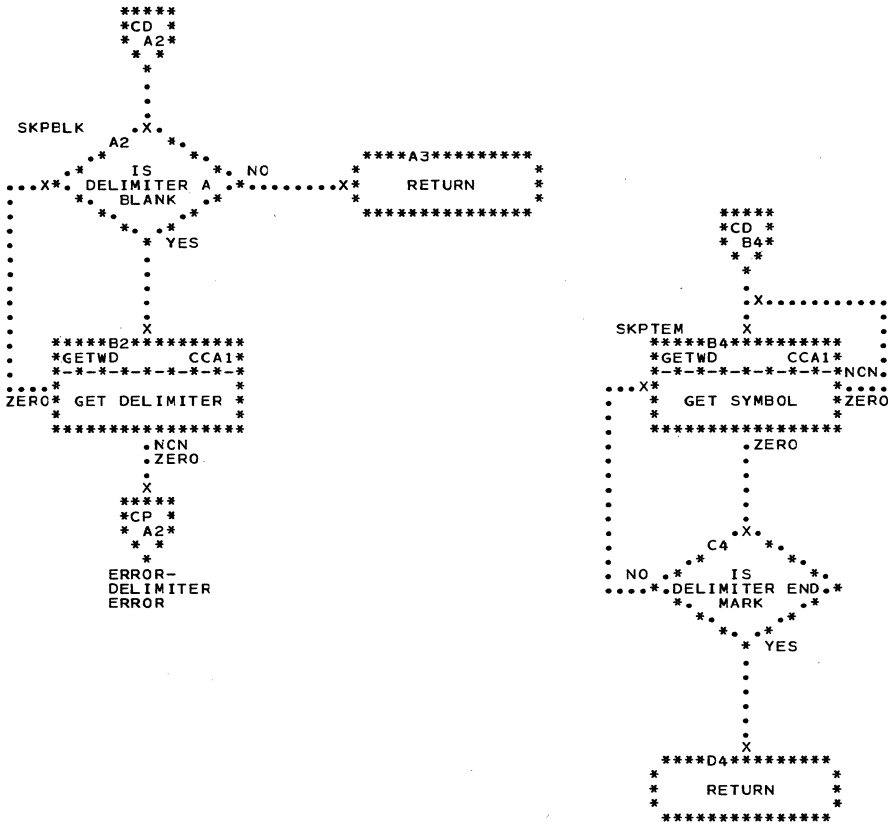


Chart CD. Subroutines SKPBLK, SKTEM

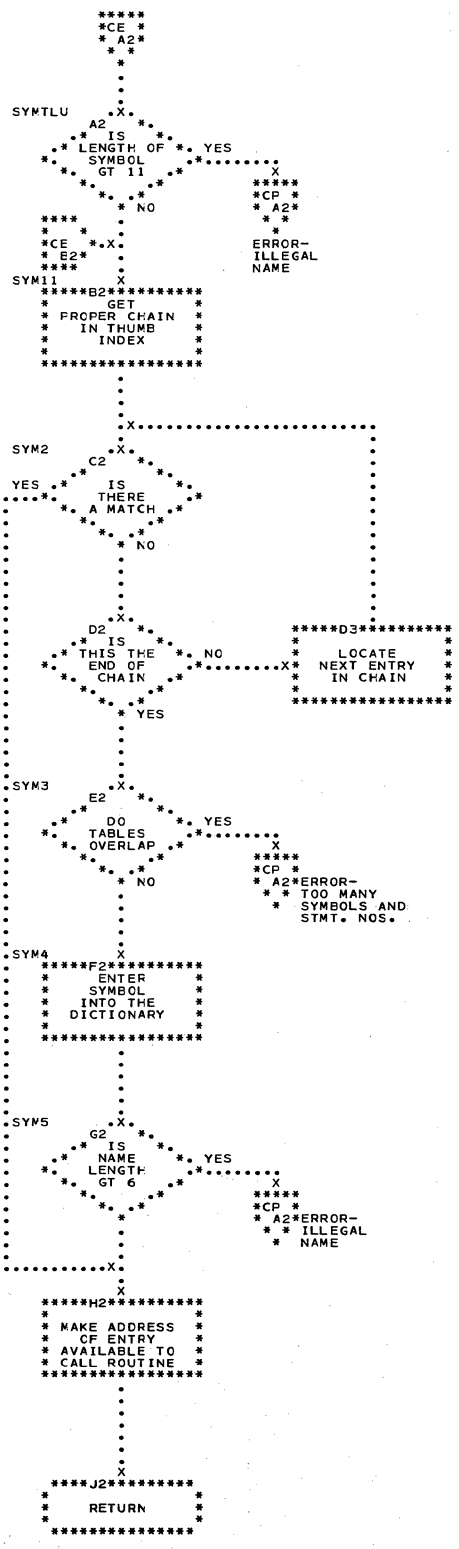


Chart CE. Subroutine SYMTLU

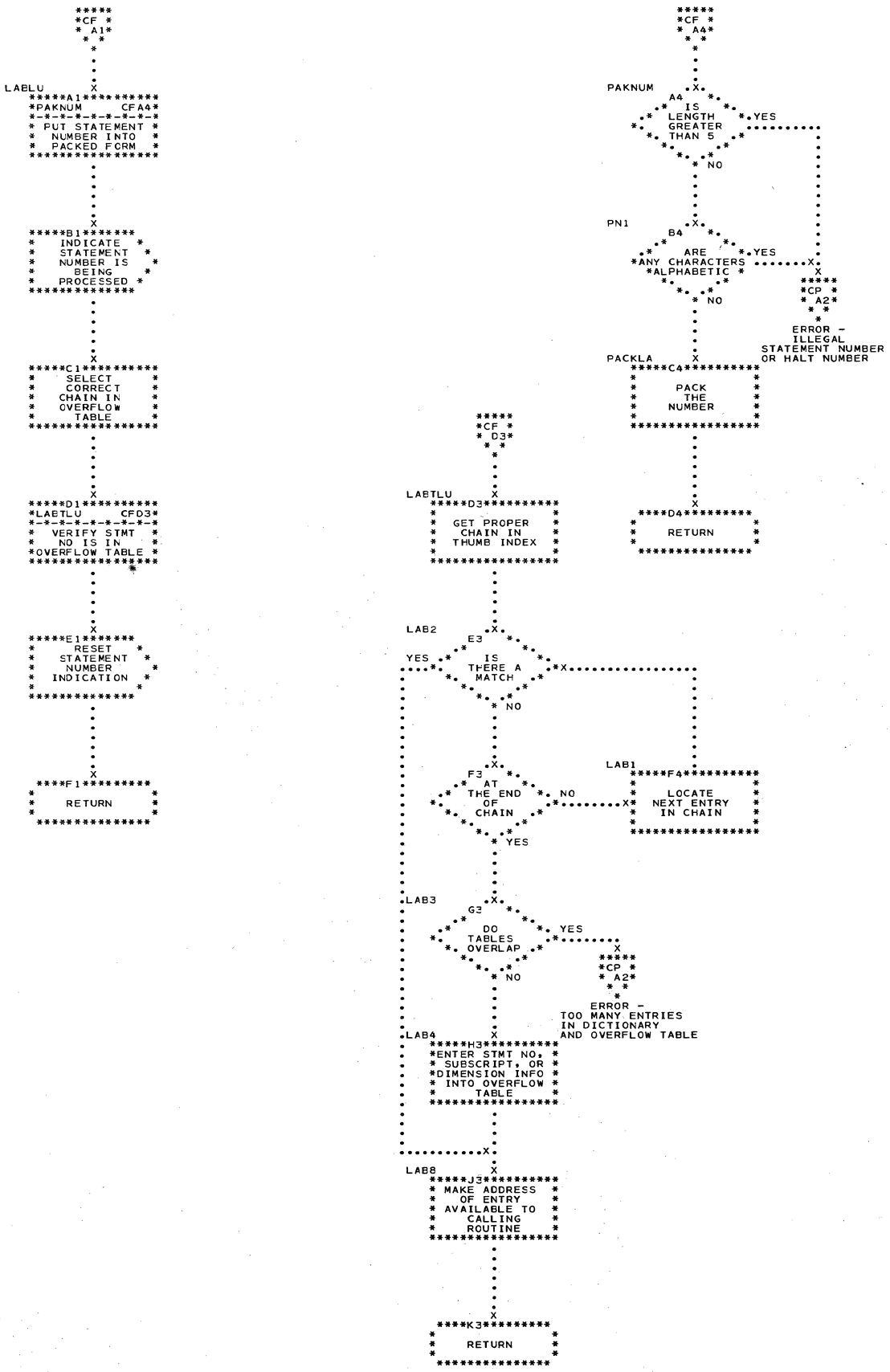


Chart CF. Subroutines LABLU, PAKNUM, LABTLU

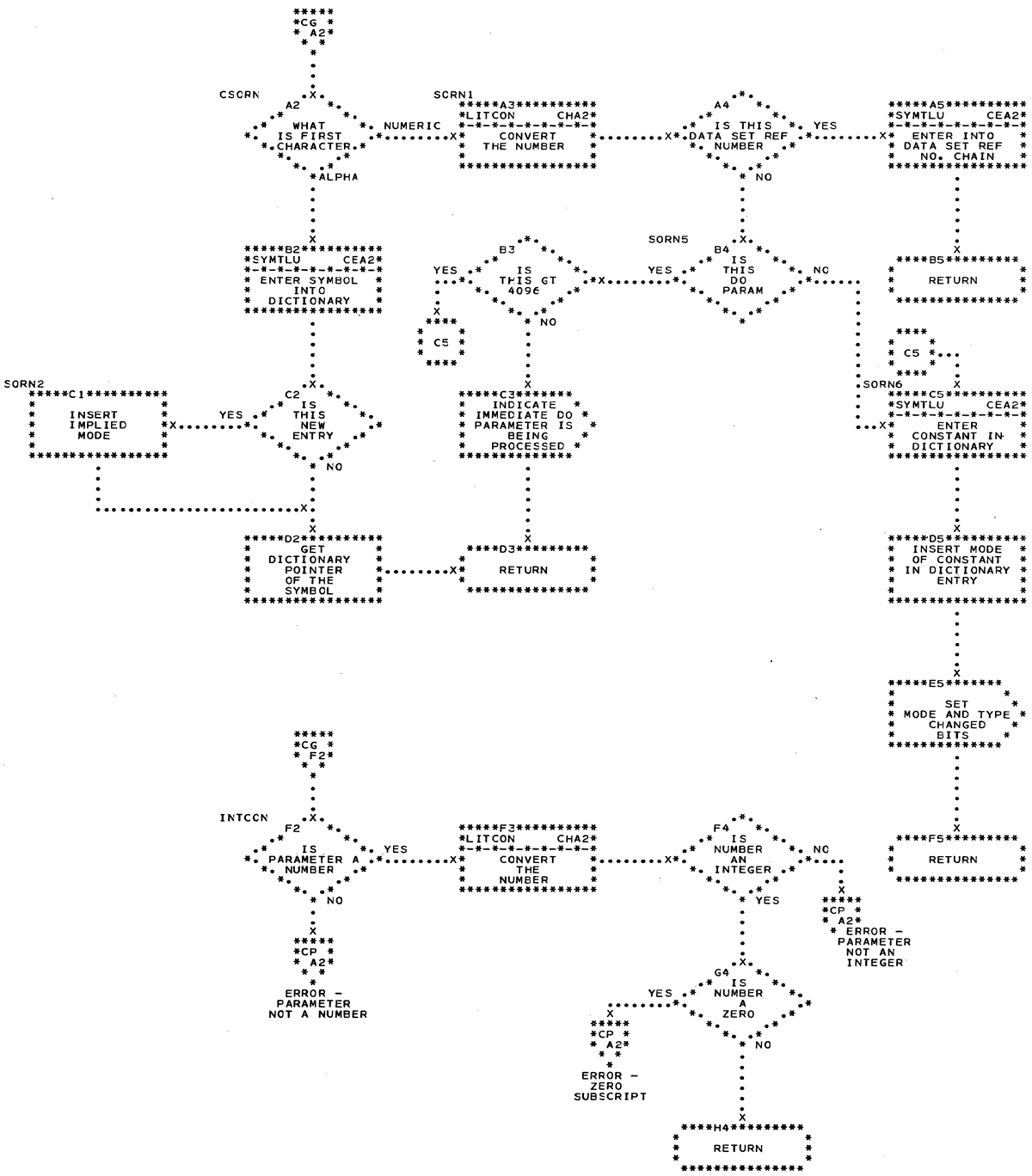


Chart CG. Subroutines CSORN, INTCON

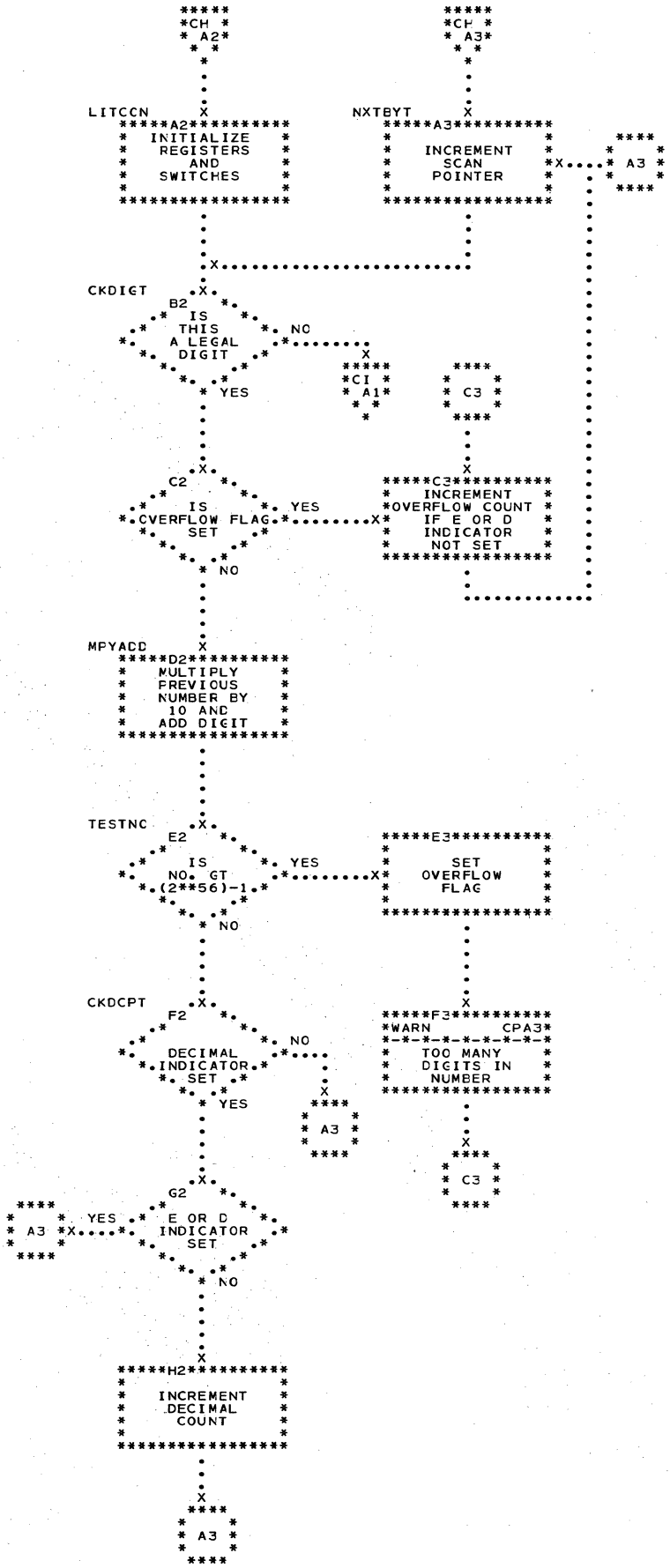


Chart CH. Subroutine LITCON Part 1

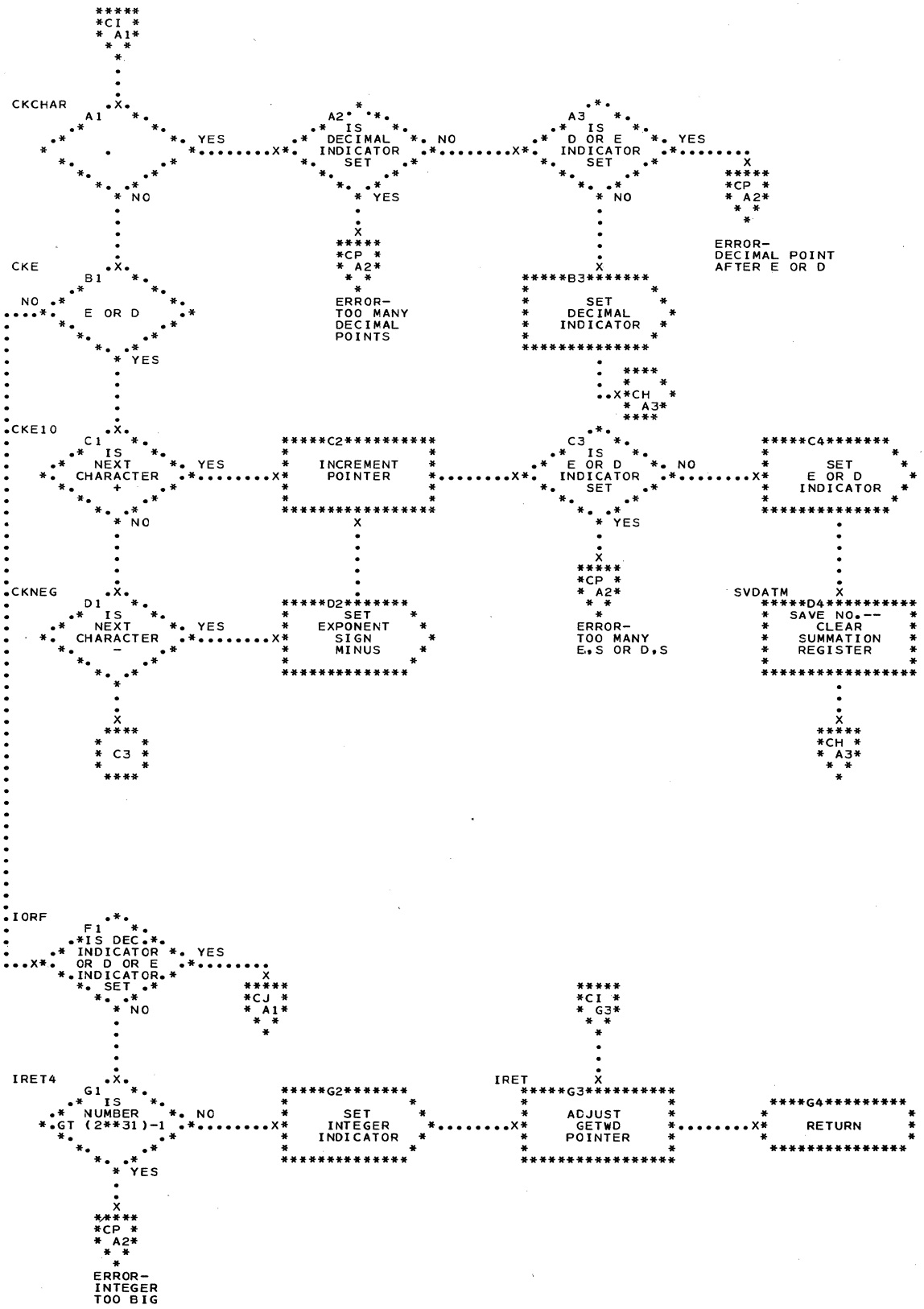


Chart CI. Subroutine LITCON Part 2

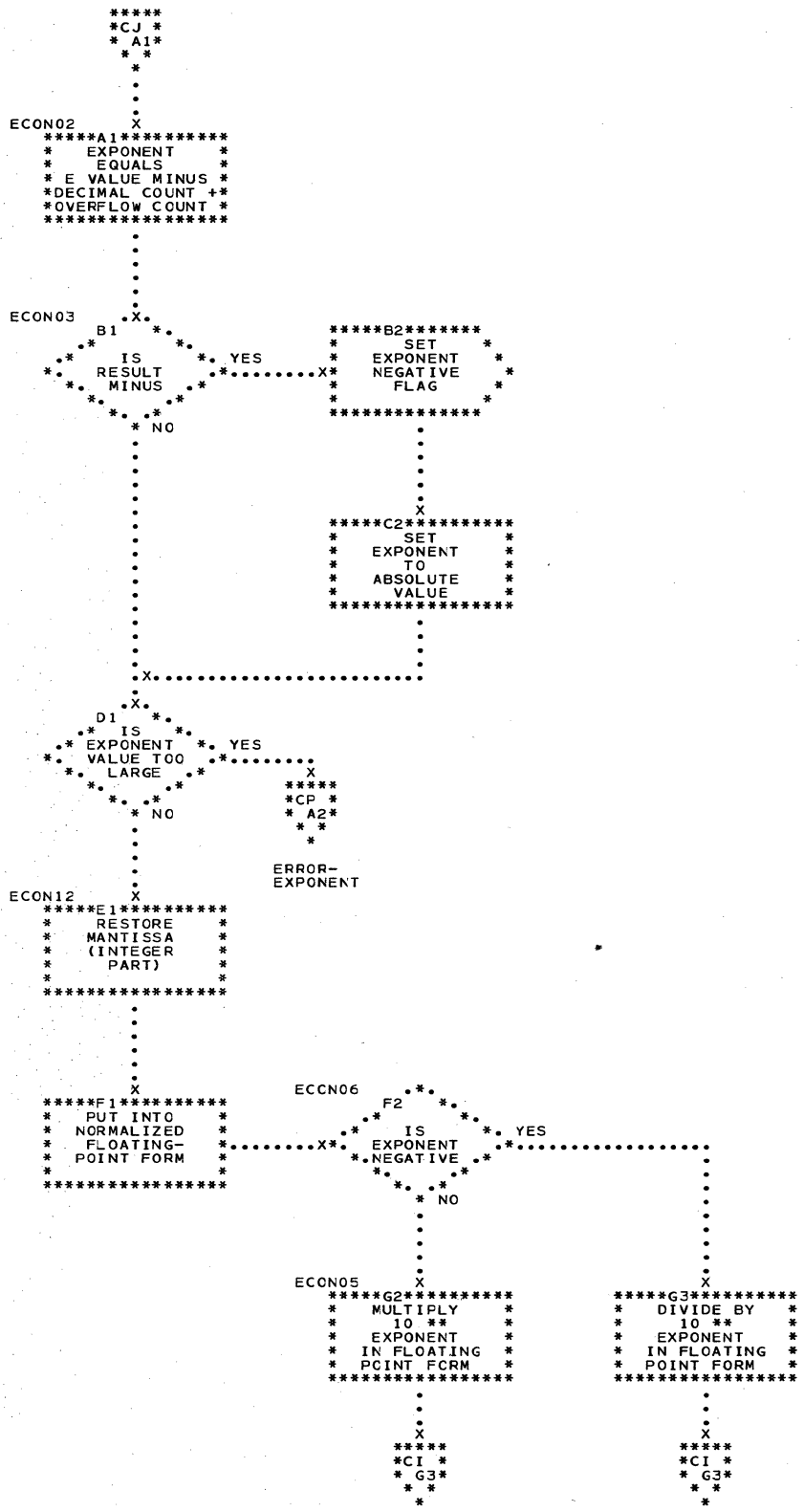


Chart CJ. Subroutine LITCON Part 3

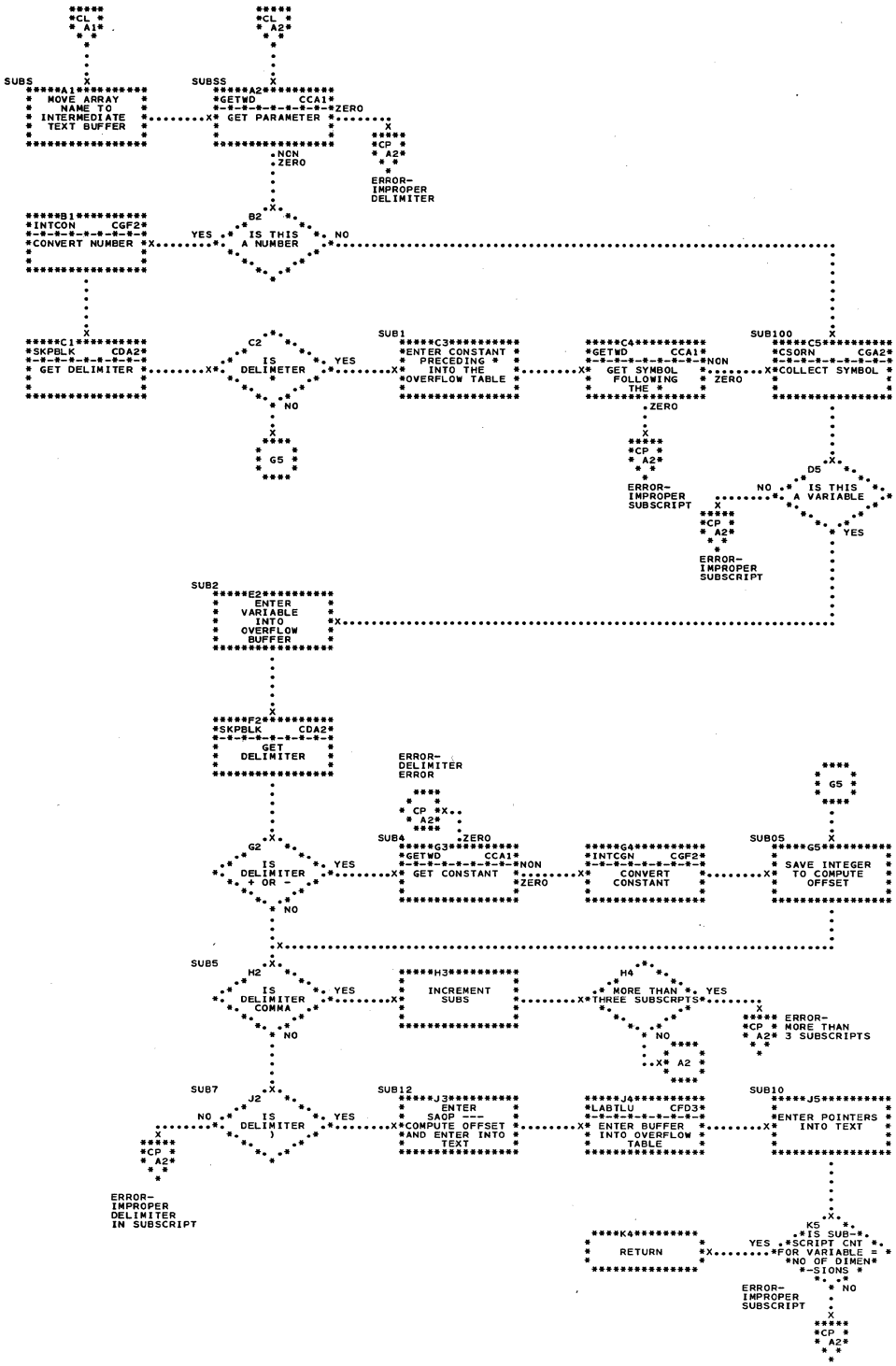


Chart CL. Subroutine SUBS

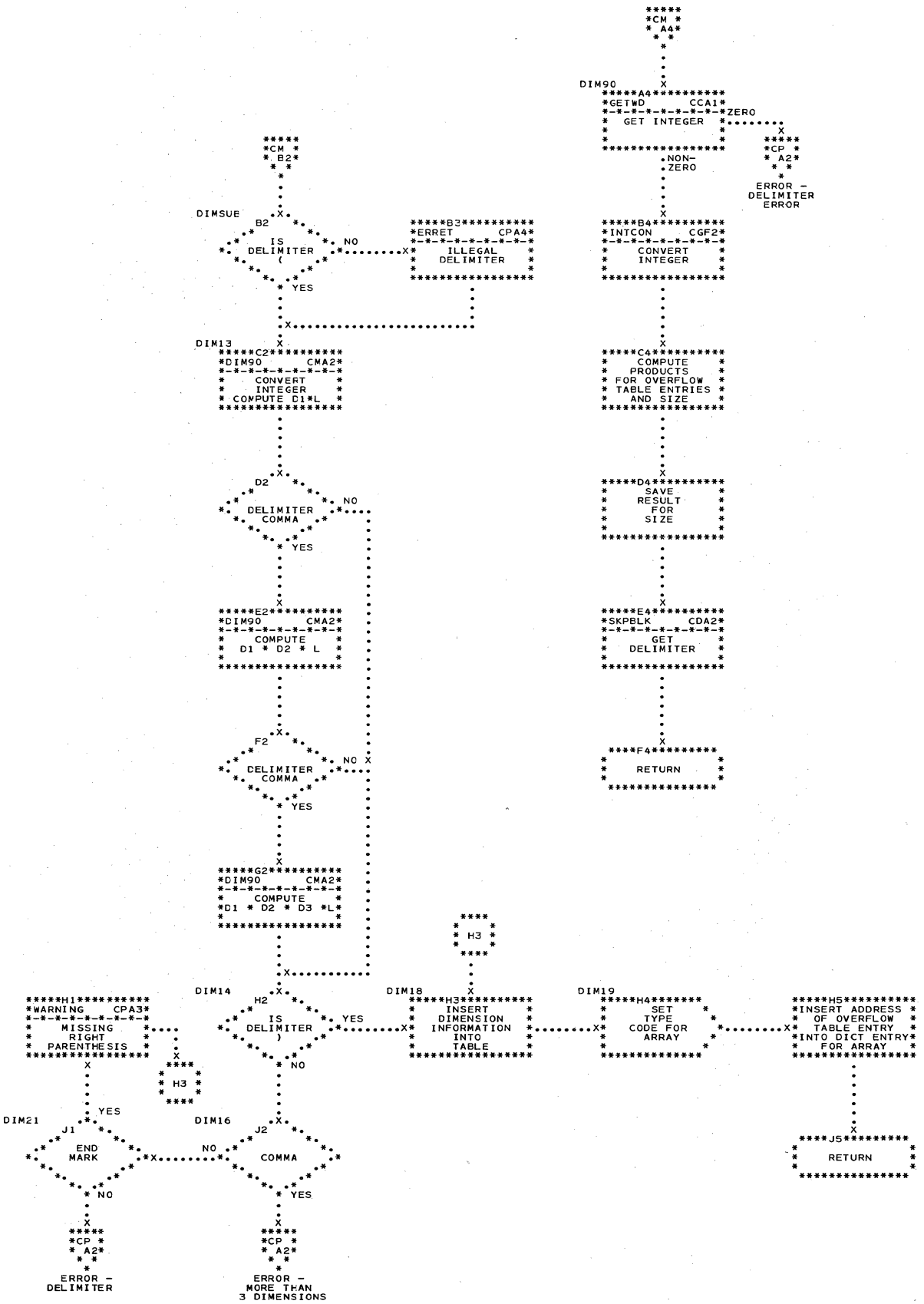


Chart CM. Subroutines DIMSUB, DIM90

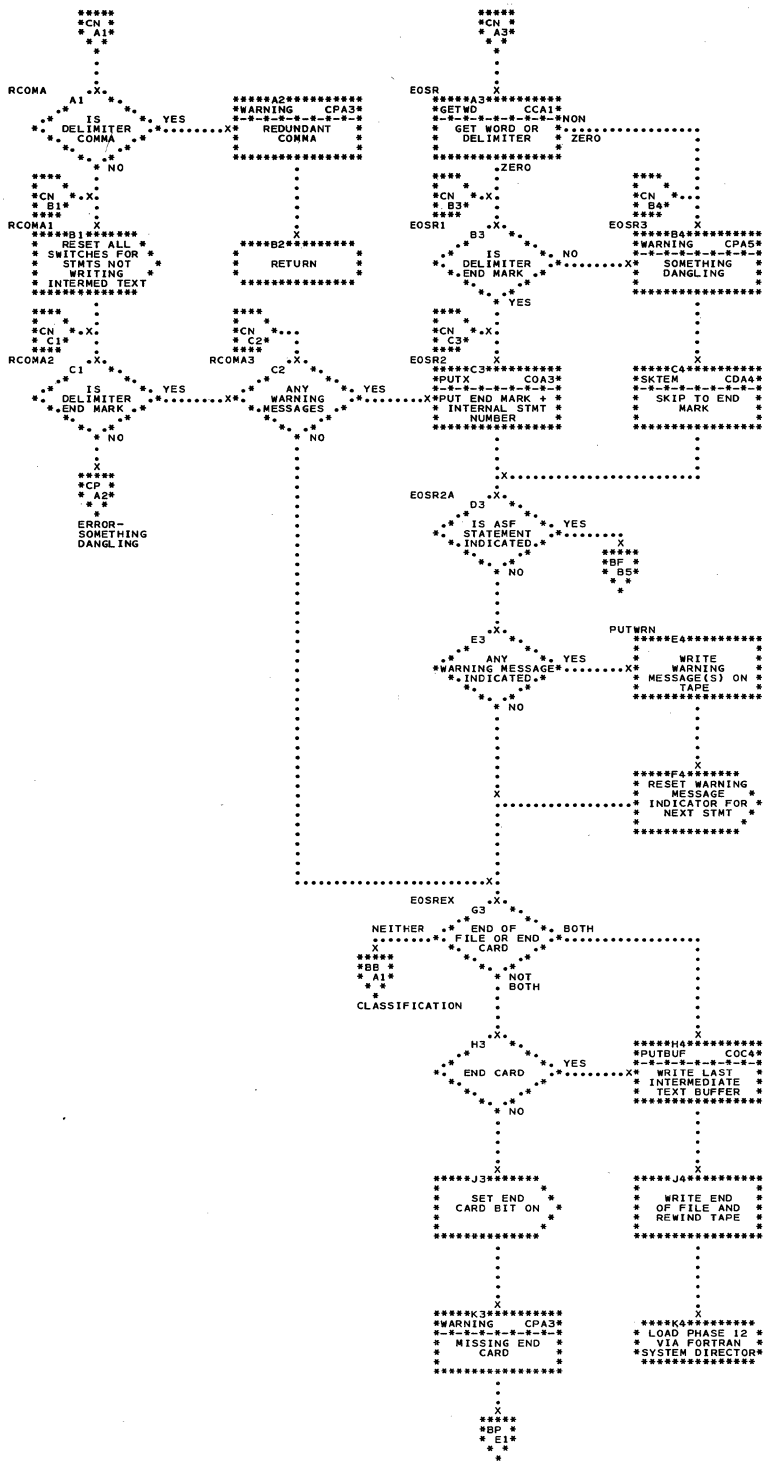


Chart CN. Subroutine END MARK CHECK

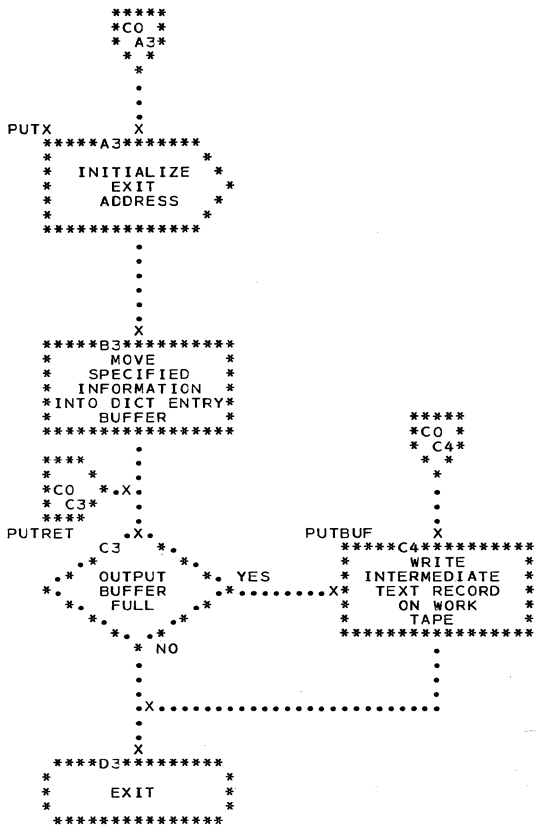


Chart CO. Subroutine PUTX

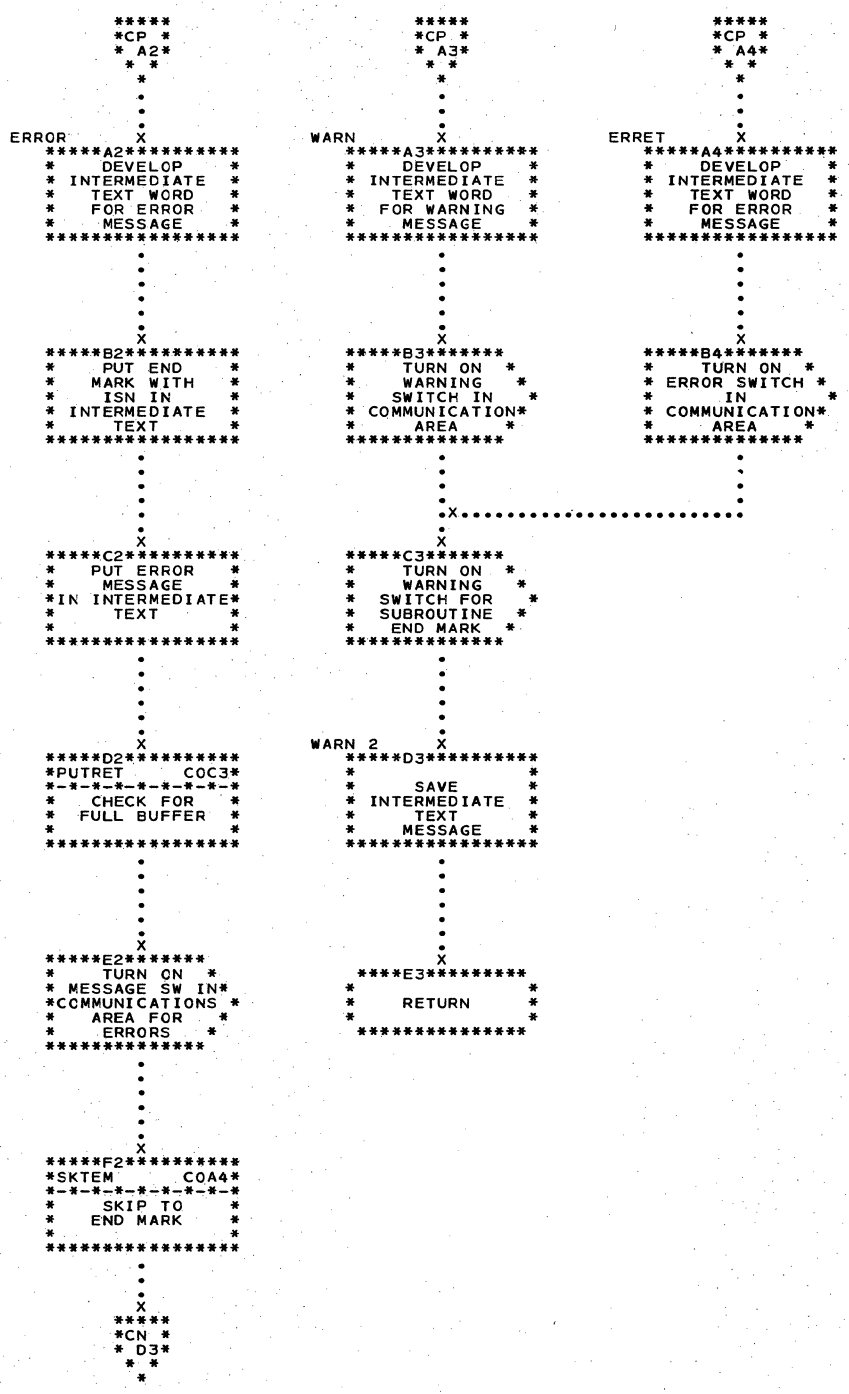


Chart CP. Subroutines ERROR, WARNING/ERRET

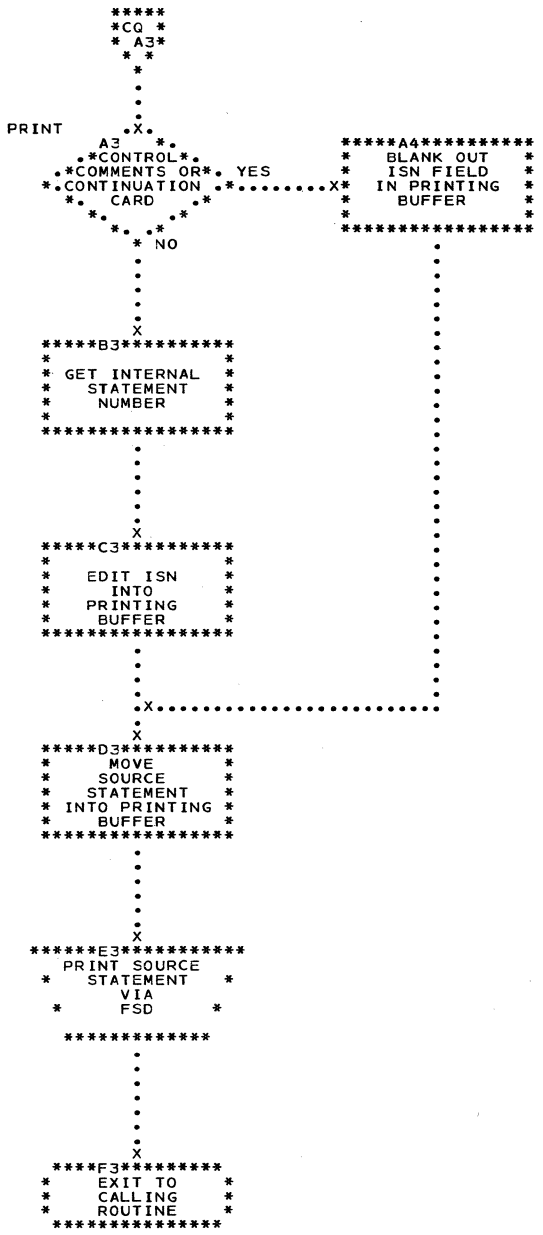


Chart CQ. Subroutine PRINT

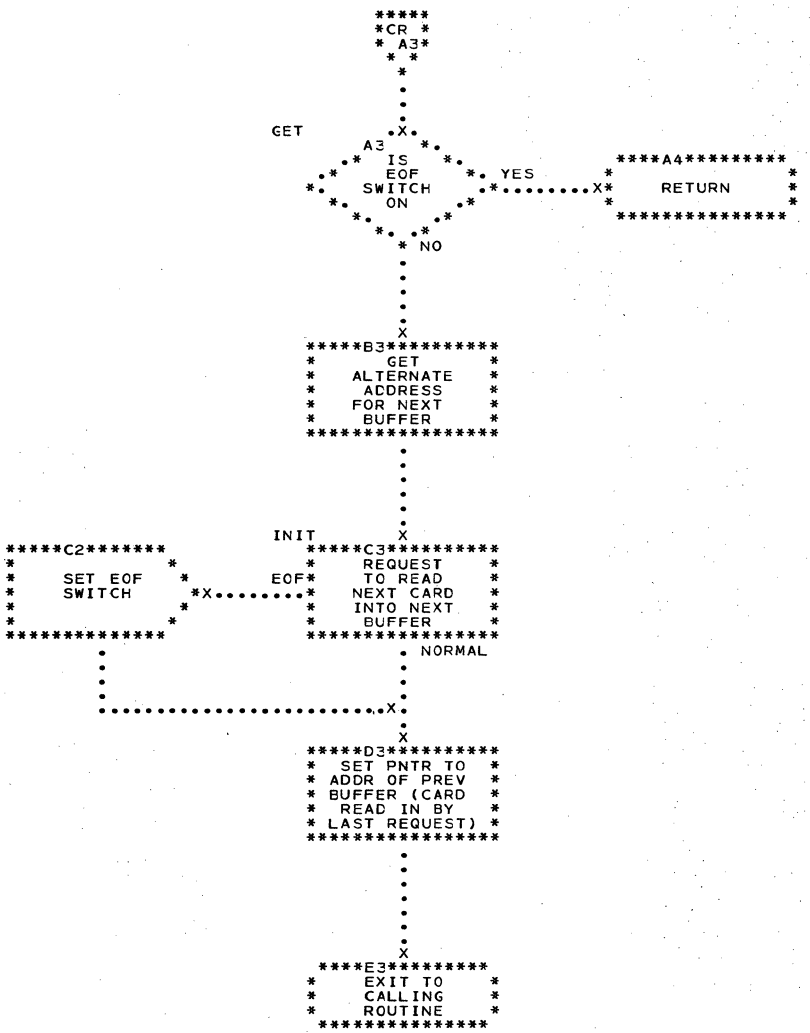


Chart CR. Subroutine GET

The primary function of Phase 12 is the allocation of storage to symbols entered in the dictionary, overflow table, COMMON text, and EQUIVALENCE text. Variables, constants, external references, and arrays are assigned object program addresses. The statement numbers referenced by control statements are assigned relative locations in a branch list table that is used by the object program to control branching.

Several secondary functions are performed. If the DECK option is specified, Phase 12 punches ESD and RLD cards for the object program and text cards for all constants used by the program. If the Compile and Go option is specified, these ESD, RLD, and text cards are written on the GO tape. As Phase 12 assigns addresses, the symbol, its address, and possibly some indicators are entered in the storage map and printed if the MAP option is specified.

Chart 04, the Phase 12 Overall Logic Diagram, indicates the entrance to and exit from Phase 12 and is a guide to the overall functions of the phase.

After the FORTRAN System Director has loaded Phase 12, all variables and arrays in COMMON are assigned addresses and removed from their dictionary chains. Phase 12 next processes the EQUIVALENCE text and creates an EQUIVALENCE table used later in this phase to assign addresses to equated variables.

Addresses are assigned to all names in the dictionary; first to double precision variables and arrays and then to real and integer variables and arrays. No distinction is made between real and integer variables and arrays. They are intermixed in the object program. Using the EQUIVALENCE table, equated variables are removed from their dictionary chains and assigned addresses.

Addresses are then assigned to address constants, and the ESD and RLD cards are punched or written on the GO data set. Address constants are locations at which the loader places the address assigned to external functions or symbols. The ESD cards contain names of external functions for the program being compiled. The RLD cards contain the addresses of address constants for the external functions. In-line functions are processed, and the dummy variables and arrays are assigned addresses.

All referenced statement numbers, other than those for FORMAT or specification statements, are assigned relative positions in a branch table. The subscript chains in the overflow table are scanned, and the dictionary pointers for variables in subscript expressions are replaced by the address assigned to the variable in the object program.

Addresses are assigned to constants in the following order: integer, real, and double-precision. If the DECK option is specified, text cards are punched for the constants; if the Compile and GO option is specified, the constants are written on the GO data set. The FORTRAN System Director is then called to read Phase 14.

ADDRESS ASSIGNMENT

Variables, arrays, constants, and address constants are assigned addresses in Phase 12. A base-displacement address is assigned through the use of a location counter, and the variable assigned an address in Phase 10 is removed from its dictionary chain.

Base Displacement Addresses

The base-displacement address assigned by Phase 12 is a 2-byte address. The first hexadecimal number (four bits) represents a general register used as a base register; the three remaining hexadecimal numbers (12 bits) represent the displacement in a machine language instruction. This address is accessed and inserted in machine language instructions in subsequent phases. An effective address in IBM System/360 is the address in the base register plus the displacement inserted in the instruction. All symbols in a FORTRAN object program are referenced by their base displacement address. An address assigned to an array refers to the first element of the array.

Location Counter

The base-displacement address is assigned through the use of a location counter which is initialized and then

incremented as addresses are assigned. The location counter is incremented by the number of bytes needed in main storage to contain the object program field assigned to the variable, array, constant, address constant, or EQUIVALENCE group.

Variables in COMMON are first assigned addresses in main storage. The location counter is set to the base-displacement address of 4000. The number 4 represents the first base register used for storage allocation, and 000 represents the displacement for the first variable entered in COMMON. The statements:

```
DOUBLE PRECISION A
COMMON A, B
```

set indicators in the dictionary and COMMON text to indicate that the variable A is double precision and that A and B are entered in COMMON. If A is the first variable entered in COMMON, it is assigned the object program address 4000. The location counter is incremented by 8 because A is a double precision variable. The next variable in COMMON, B, is assigned the address 4008 indicating the base register 4 and a displacement of 8. The location counter is then incremented by 4; B is not double precision.

This process of contiguously assigning addresses using the location counter may vary for the following conditions.

1. No other variables can be assigned a base register that has been assigned to variables in COMMON. After the last variable has been assigned an address in COMMON, the next available register is assigned to the next variable.

   For example, if the last variable in COMMON is assigned the address 42A4, the next variable should have an address of 42A8. Because the next variable is not in COMMON, it must be assigned a new base register. The location counter is set to 5000 (the base register 5 with a displacement of 000).
2. Integer and real constants are assigned addresses before double precision constants. After addresses are assigned to integer and real constants, the location counter may have to be adjusted to a double-word boundary to accommodate the double precision constants. If this happens, there is a full-word gap between the last real constant and the first double precision constant.

The location counter is incremented by the size of the variable, array, constant, address constant, or EQUIVALENCE group assigned. If more than 4096 bytes are assigned, adding to the location counter automatically assigns a new base register. For example, if the location counter is set at 4FFC and an integer variable is assigned to this address, the location counter is incremented by 4 to 5000.

If a real array A(5) is assigned to the address 4FFC, the location counter is incremented by 20 bytes (the hexadecimal number 14). The location counter is incremented to 5010 representing the base register 5 plus a displacement of 010.

Removing Entries From Chains

When a variable is assigned an address, that address must be placed in the chain address field of the dictionary. This requires switching three addresses in the dictionary chain. In Phase 10, the chain field points to the next entry in the chain (see Figure 37).

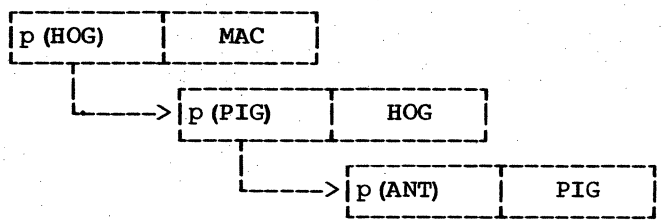


Figure 37. Dictionary Chain Entries

If HOG is the variable to be assigned an address, this address is placed in the chain field. The pointer to HOG in the entry for MAC must be replaced by the pointer to PIG to keep the chain for unassigned variables intact (see Figure 38).

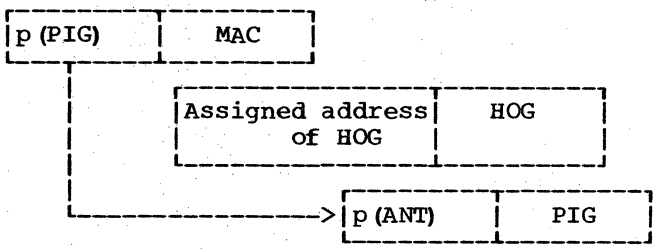


Figure 38. Removing a Symbol From a Dictionary Chain

EQUIVALENCE PROCESSING

An EQUIVALENCE group consists of the names between a left and right parenthesis in an EQUIVALENCE statement. For example, in the statement:

```
EQUIVALENCE (A,B,C,D,) , (W,X,Y,Z)
```

the variables A,B,C, and D form one EQUIVALENCE group, and the variables W,X,Y, and Z form another EQUIVALENCE group. In the statements:

```
DIMENSION TOAD(20)
EQUIVALENCE (TOAD(3) ,FLUB,SHARK)
```

the array TOAD and the variables FLUB and SHARK form an EQUIVALENCE group.

An EQUIVALENCE class is a number of EQUIVALENCE groups linked together by names that are common to two or more groups. In the statement:

```
EQUIVALENCE (A,B,C) , (C,X)
```

the two groups form a class because they are linked together by the variable C. In the statement:

```
EQUIVALENCE (A,B,C) , (C,D) , (D,Y,Z)
```

all three groups form an EQUIVALENCE class because they are linked together by the variables C and D. In the statement:

```
EQUIVALENCE (X,Y,Z) , (X1,G,H)
```

an EQUIVALENCE class is not formed.

The root of an EQUIVALENCE group or class is the member of a group or class to which all other variables are equated. The root is assigned an address and all other variables and arrays are assigned addresses relative to the root. If a member of a group or class is in COMMON, it is automatically determined to be the root. If none of the variables are in COMMON, the root cannot be determined until the displacement is calculated.

The displacement, the distance in bytes between a variable and its root, is calculated by subtracting the offsets entered in the EQUIVALENCE table in Phase 10. For non-subscripted variables, the offset is always zero. To determine the root and displacement of variables and arrays, the first name in the EQUIVALENCE group is established as a temporary root. The offset for the other arrays and variables is subtracted from the offset of the temporary root. For example, in the statements:

```
DIMENSION A(5) ,XMAS(4)
EQUIVALENCE (A(2) ,B,XMAS(1) )
```

the first name in the group is established as a temporary root. The offset of A(2) is 4. The offset of both B and XMAS(1) is 0. The root for an EQUIVALENCE group is changed only when the result of the displacement calculation is negative. By calculating the displacement, the relative position for the elements of the variables and arrays is determined. When the offset of B is subtracted from the offset of A(2) the result is 4, and when the offset of XMAS(1) is subtracted from the offset of A(2) the result is 4. The relative positions of the members in this EQUIVALENCE group are shown in Figure 39.

A(1)	A(2) B XMAS(1)	A(3) XMAS(2)	A(4) XMAS(3)	A(5) XMAS(4)

Figure 39. EQUIVALENCE Group Without Root Switching

The symbols A(2), B, and XMAS(1) all refer to the same field. In the statements:

```
DIMENSION MAC(4) ,HERBIE(4)
EQUIVALENCE (MAC(1) ,WINDY,HERBIE(2) )
```

when the EQUIVALENCE text is processed, the array MAC is temporarily established as the root of the EQUIVALENCE group. The offset for both MAC(1) and WINDY is 0, and the displacement is 0. The offset for HERBIE(2) is subtracted from the offset for MAC(1); the result is -4. Because the result is negative, the array HERBIE must be established as the new root. The displacement for WINDY and MAC must be changed to 4.

The relative positions of the members in this EQUIVALENCE group are shown in Figure 40.

The symbols MAC(1), WINDY, and HERBIE(2) all refer to the same field.

	MAC(1) WINDY	MAC(2)	MAC(3)	MAC(4)
HERBIE(1)	HERBIE(2)	HERBIE(3)	HERBIE(4)	

Figure 40. EQUIVALENCE Group With Root Switching

The size of an EQUIVALENCE group or class is the size in bytes necessary to contain the entire EQUIVALENCE group or class. For example, if the elements shown in Figure 40 are all real, the size of that EQUIVALENCE group is 20 bytes. There are five 4-byte fields in the EQUIVALENCE group.

Phase 12 constructs an EQUIVALENCE table used by the subroutines that assign addresses in Phase 12. The format of the EQUIVALENCE table is shown in Figure 41.

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

Figure 41. EQUIVALENCE Table Format

Each field in the table is two bytes long. 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 in COMMON are assigned before this table is constructed. The fourth field is the size in bytes of the EQUIVALENCE group or class.

BRANCH TABLE

An object program uses a branch table to control branching. Each referenced statement number for an executable statement is assigned a position in the branch table. Phase 25 places an address in this position to denote where the instructions begin for the statement defined by the statement number.

Phase 12 allocates storage for the branch table by assigning a relative number in the table to each statement number not in a FORMAT or specification statement but referenced in a control statement. Statement numbers assigned relative numbers are removed from their chain in the overflow table and the relative number is placed in the chain address field.

The relative number is directly related to the position the statement number occupies in the branch list generated for the object program. The statement number chains in the overflow table are scanned sequentially. Each time a statement number is referenced and is not the statement number for a FORMAT or specification statement, a counter is incremented by 4. The contents of the counter are then placed in the chain field in the overflow table. This counter is initialized at 0; therefore, the first statement number in the first chain is assigned the relative number 0. The second statement number in the first chain is assigned the relative number 4. The third statement number in the first chain is assigned the relative number 8, and so on.

COMMUNICATIONS AREA

At the end of execution for Phase 12, the following items have been entered in the communications area:

1. If an error has been detected, a switch is set denoting at least one error in this program.
2. A switch is set to indicate whether the program is large or small.
3. The address of the first available location for the object program.
4. A page number that has been updated while the storage map was being printed, provided the MAP option was requested by the user.
5. The number of bytes required to contain COMMON.
6. The number of bytes required to store the branch table.
7. The last external symbol identification number that was used.
8. The card sequence number for the last card that was punched.
9. The address of the beginning of the subscript entries in the overflow table.
10. The assigned address of the branch table.
11. The assigned address of the first value in each base register.

STORAGE MAP

The storage map for Phase 12 is shown in Figure 42.

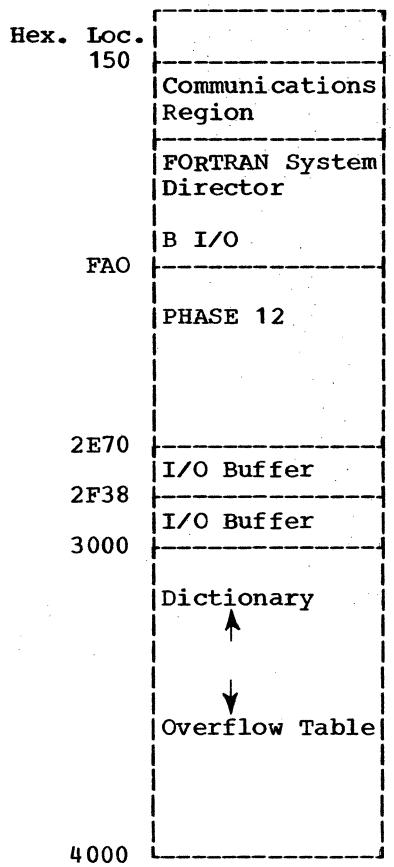


Figure 42. Storage Map for Phase 12

SUBROUTINES

Two types of subroutines (mainline and utility) are necessary to perform address assignment.

The mainline subroutines assign addresses to symbols entered in COMMON; build the EQUIVALENCE table; assign addresses to variables, arrays, and address constants; assign relative numbers to statement numbers; assign addresses to constants; and insert object program addresses into overflow table entries for subscripts. (See charts DA through DL.)

The utility subroutines are called by mainline subroutines to perform perfunctory functions such as scanning the EQUIVALENCE table, entering items in the EQUIVALENCE table, scanning the chains in the dictionary, calling input/output devices, and putting the ESD, RLD, text cards, and storage map into a format. (See charts DM through DU.)

Subroutines STARTA, COMAL: Chart DA

Subroutine STARTA

Subroutine STARTA initializes Phase 12.

ENTRANCE: Subroutine STARTA is entered from the FORTRAN System Director after Phase 12 is loaded into main storage.

OPERATION: Subroutine STARTA initializes the base register and adjusts the pointers for the input and output buffers.

EXIT: Subroutine STARTA exits to subroutine COMAL after the initialization of Phase 12 is completed.

Subroutine COMAL

Subroutine COMAL assigns addresses for variables or arrays to be placed in the COMMON area and removes those variables or arrays from the dictionary chains.

ENTRANCE: Subroutine COMAL is entered from subroutine STARTA after Phase 12 is initialized.

OPERATION: When subroutine COMAL is entered, it gets the beginning and ending addresses of the COMMON text left in the communications area by Phase 10. The location counter in the communications area also tells subroutine COMAL where to begin assigning addresses for variables in the COMMON area in the object program.

Subroutine COMAL checks for entries in the COMMON area by comparing the beginning and ending addresses of the COMMON text. If they differ, entries are made in COMMON. If they are the same, there is no COMMON text to process.

If there are variables or arrays to be entered in COMMON, subroutine COMAL accesses the character length of the name in the COMMON text, goes to the correct chain in the dictionary, and searches for the entry. Subroutine COMAL finds the entry by comparing the address entered in the COMMON text with the chain addresses entered in the dictionary. When the comparision is equal, the next entry in the dictionary chain is the entry in COMMON.

The address in the location counter is then placed in what was originally the chain address field of the dictionary entry.

Dummy variables or arrays, used to define the operations performed on parameters passed in CALL statements, may not be entered in COMMON. The location of these dummy variables is passed from the main program in a calling sequence.

If the mode of any variables or arrays entered in the COMMON area is double precision, these variables must be on a double-word boundary. COMAL can not adjust the variable or array to a double-word boundary because no gaps may exist in COMMON. The beginning address of the COMMON area starts on a double-word boundary. If the double precision variables or arrays are assigned first to COMMON, an error can not occur.

EXIT: Subroutine COMAL exits to subroutine EQUIVALENCE Part 1 after all COMMON text is processed.

SUBROUTINES CALLED: During execution subroutine COMAL references the following subroutines:

1. ALERET to process any errors that are detected.
2. SORSYM to print storage map if requested.

Subroutine EQUIVALENCE: Charts DB, DC, DD

The EQUIVALENCE subroutines use the EQUIVALENCE text constructed in Phase 10 to build an EQUIVALENCE table. The table is used when addresses are assigned to variables later in Phase 12.

ENTRANCE: Subroutine EQUIVALENCE is entered from subroutine COMAL after addresses are assigned to the variables placed in the COMMON area.

OPERATION: Each flowchart is discussed separately.

EXIT: After the EQUIVALENCE text is processed to form the EQUIVALENCE table, control is passed to subroutine EXTCOM.

SUBROUTINES CALLED: During execution subroutine EQUIVALENCE references subroutines SWROOT, EQSRCH, RENTER/ENTER, and ALERET.

Subroutine EQUIVALENCE Part 1

Subroutine EQUIVALENCE Part 1 initializes the registers to scan the EQUIVALENCE text for each EQUIVALENCE group and processes the first name entered in an EQUIVALENCE group.

ENTRANCE: Subroutine EQUIVALENCE Part 1 is entered from subroutine COMAL after the entries in COMMON are assigned a COMMON address, and from EQUIVALENCE Part 2 to get the first entry in the next EQUIVALENCE group after an entire EQUIVALENCE group is processed.

OPERATION: Subroutine EQUIVALENCE Part 1 is initialized with the beginning address of the EQUIVALENCE text and the EQUIVALENCE table. These pointers are incremented as the text is processed.

Subroutine EQUIVALENCE Part 1 checks for additional entries in the EQUIVALENCE text by testing for the EQUIVALENCE adjective code. If there are no entries, control is passed to subroutine EXTCOM. (The adjective code and the number of entries in the EQUIVALENCE group are entered in Phase 10 for a header entry in the EQUIVALENCE text.)

The EQUIVALENCE table is then searched to check for a previous entry of the variable in another EQUIVALENCE group. This condition forces the EQUIVALENCE groups to be combined, forming an EQUIVALENCE class. For example, the statement:

```
EQUIVALENCE (A,B) , (A,C)
```

indicates that A,B, and C are all members of one EQUIVALENCE class.

If the variable is not entered in another EQUIVALENCE group or class, it is entered as a root in the EQUIVALENCE table. This status may change as other variables in the EQUIVALENCE group or class are processed.

If the variable was previously entered in the table, subroutine EQUIVALENCE Part 1 gets the root for that variable in its previous entry and checks for the root in COMMON. If the root is not in COMMON, the displacement is inserted in the EQUIVALENCE table; if the root is in COMMON, the COMMON address is used.

EXIT: Subroutine EQUIVALENCE Part 1 passes control to EQUIVALENCE Part 2 after the first member of an EQUIVALENCE group is processed.

SUBROUTINES CALLED: During execution subroutine EQUIVALENCE Part 1 references the following subroutines:

1. EQSRCH to see if a variable was entered in the EQUIVALENCE table previously.
2. RENTER/ENTER to enter a variable as a root.

Subroutine EQUIVALENCE Part 2

Subroutine EQUIVALENCE Part 2 processes all members following the first one in an EQUIVALENCE group. If EQUIVALENCE Part 2 determines that a member should be the root for the EQUIVALENCE group, it changes the root for the EQUIVALENCE group.

ENTRANCE: Subroutine EQUIVALENCE Part 2 is entered from EQUIVALENCE Part 1 after the first member of an EQUIVALENCE group is entered in the EQUIVALENCE table. Subroutines SWROOT and EQUIVALENCE Part 3 enter EQUIVALENCE Part 2 to check for additional members in this EQUIVALENCE group. Subroutine EQUIVALENCE Part 3 also enters EQUIVALENCE Part 2 either to enter a variable in the EQUIVALENCE table or prepare the subroutines for switching roots.

OPERATION: When subroutine EQUIVALENCE Part 2 is entered from EQUIVALENCE Part 1, the EQUIVALENCE text pointer is incremented by EQUIVALENCE Part 1 so the next entry in the EQUIVALENCE text may be processed. The address of the root is entered in a register. When control is passed to EQUIVALENCE Part 2, a root for this group is determined either by setting the first member in the EQUIVALENCE group as a root or switching the root in EQUIVALENCE Part 2.

Any time EQUIVALENCE Part 2 determines that a member or its root is in COMMON, control is passed to EQUIVALENCE Part 3.

If the mode of the variable is double precision, a bit in the usage field of the dictionary entry for the root is set to 1. Other bits in the usage field are set in the EQUIVALENCE subroutines to indicate that the variable is a root or equated to a root. These two bits are not set to 1 at the same time. The first bit indicates a root. The second bit is set only for variables that are equated to some root.

The displacement of an equated variable is its distance in bytes from the root. That is, the displacement is the offset of the equated variable subtracted from the offset of the root. The offsets are entered in the Phase 10 EQUIVALENCE text. For example, if two arrays are dimensioned as A(3,3,3) and B(2,2,2) and the EQUIVALENCE statement:

```
EQUIVALENCE (A(2,1,1),B(1,1,1))
```

is read in Phase 10; the offset for the first member of the EQUIVALENCE group A(2,1,1) is 4. The offset for the second member of the EQUIVALENCE group B(1,1,1), is 0. Then the displacement is:

```
4-0=4
```

This means that the array B begins 4 bytes away from the first element of the array A, as follows:

```
A(1,1,1)  A(2,1,1)  A(3,1,1)  A(1,2,1)  ...
          B(1,1,1)  B(2,1,1)  B(1,2,1)  ...
```

In subroutine EQUIVALENCE Part 1 the first member of the EQUIVALENCE group is temporarily entered as a root. When the displacement in the preceding example is calculated, the result is positive because the first member of the EQUIVALENCE group is entered as a root. If the position in the statement of the members of the group is changed, for example:

```
EQUIVALENCE (B(1,1,1), A(2,1,1))
```

the result of the displacement calculation is different. In subroutine EQUIVALENCE Part 1 for this example, the first member B(1,1,1) is entered as a root. The displacement is now:

```
0-4=-4
```

which means that the root must be changed to indicate that A, not B, is the root for this EQUIVALENCE group.

If the result of the displacement calculation is positive, a check is made to see if this variable has already been entered in the EQUIVALENCE table. If it has not been entered, it is entered in the EQUIVALENCE table, and a check is made for an additional variable in this EQUIVALENCE group. The following statement would have entered the variable previously:

```
EQUIVALENCE (A(2,1,1),C), (B(1,1,1),C)
```

This statement forms an EQUIVALENCE class. At this point, both A and B are temporarily roots. Because they are equated to C, one must be selected as the root for this class. The displacement is computed for the two roots, and the offset of B(1,1,1) is subtracted from the offset of A(2,1,1), yielding:

```
4-0=4
```

The root for this EQUIVALENCE class is A, and B is then equated to A.

If the result of the first displacement calculation is negative, resulting from the statement:

```
EQUIVALENCE (B(1,1,1),C), (A(2,1,1),C)
```

subroutine EQUIVALENCE Part 2 again switches roots.

EXITS: Subroutine EQUIVALENCE Part 2 passes control to subroutine SWROOT if the root for an EQUIVALENCE group must be changed. Subroutine EQUIVALENCE Part 1 is entered from EQUIVALENCE Part 2 if there are no other members in the EQUIVALENCE group. Subroutine EQUIVALENCE Part 3 is entered if subroutine EQUIVALENCE Part 2 determines that either the variable being processed or its root is in COMMON.

SUBROUTINE CALLED: During execution subroutine EQUIVALENCE Part 2 references the following subroutines:

1. EQSRCH to search the EQUIVALENCE table.
2. RENTER/ENTER to enter a variable in the EQUIVALENCE table.

Subroutine EQUIVALENCE Part 3

Subroutine EQUIVALENCE Part 3 handles special processing for all equated variables or arrays entered in COMMON. If the variable or its root is in COMMON, subroutine EQUIVALENCE Part 3 processes that variable.

ENTRANCE: Subroutine EQUIVALENCE Part 2 enters subroutine EQUIVALENCE Part 3 if either a variable or its root entered in COMMON is encountered.

OPERATION: There are three entries used to enter subroutine EQUIVALENCE Part 3.

Entry DDA1 is used if subroutine EQUIVALENCE Part 2 determines that the variable has been entered in COMMON. It then checks for the root in COMMON. If the root is entered in COMMON, it has been assigned an address by subroutine COMAL, which also assigned an address to the variable. Using the offsets entered in the EQUIVALENCE text, EQUIVALENCE Part 3 computes an address for the variable relative to its root. If the address assigned to the variable by COMAL and the address computed by EQUIVALENCE Part 3 are not equal, an error is noted. In the example:

```
COMMON A(3),B(3)
EQUIVALENCE (A(2),B(1))
```

the COMMON statement places the arrays A and B in COMMON as follows:

```
A(1)  A(2)  A(3)  B(1)  B(2)  B(3)
```

The EQUIVALENCE statement then attempts to place the arrays A and B in COMMON as follows:

```
A(1)  A(2)  A(3)
      B(1)  B(2)  B(3)
```

If these two statements are processed, two different addresses are assigned to B(1). Subroutine COMAL assigns B(1) an address that places it adjacent to A(3). Subroutine EQUIVALENCE Part 3 attempts to assign to B(1) the address that subroutine COMAL assigned to A(2).

If the root is not assigned to COMMON and the variable is assigned to COMMON, the roots must be changed. If an EQUIVALENCE group contains a variable entered in COMMON, that variable must be used as the root. Subroutine EQUIVALENCE Part 3 then enters this variable as a root.

Entry DDA4 is used if the root, but not the variable, is entered in COMMON. The COMMON address relative to the root is computed, and the variable is entered in the EQUIVALENCE table.

If the variable was previously entered in the table, it was assigned a root when the entry was made. A check is made for the root in COMMON. If the previous root is in COMMON, an additional check is made to see if the two roots assign conflicting addresses to the variable. If the previous root was not entered in COMMON, it must be equated to the current root entered in COMMON.

Entry DDE3 is used when the variable or current root is not entered in COMMON. Previously the variable was entered in the EQUIVALENCE table, and its root was in COMMON. These statements generate the following condition:

```
COMMON A
EQUIVALENCE (A,C), (B,C)
```

The current root, B, must be equated to the previous root, A. All variables equated to B must also be equated to A.

EXITS: Subroutine EQUIVALENCE Part 3 exits to subroutines SWROOT if the root for a group must be changed. Subroutine EQUIVALENCE Part 3 also exits to EQUIVALENCE Part 2 if an error is detected or to set the mechanism for changing roots.

SUBROUTINES CALLED: During execution EQUIVALENCE Part 3 calls subroutines ALERET and RENTER/ENTER.

Subroutine EXTCOM: Chart DE

Subroutine EXTCOM finds and enters the size of COMMON in the communications area. It then adjusts the location counter to the next base and increments the location counter by the number of bytes necessary to contain the initialization routine for the object program.

ENTRANCE: Subroutine EXTCOM is entered from subroutine EQUIVALENCE Part 1 after the EQUIVALENCE table is constructed.

OPERATION: The size of COMMON is determined by subtracting the location counter used to allocate COMMON from the beginning address of COMMON. The difference is entered in the communications area.

The location counter is adjusted to insure that the next variables assigned addresses have a different base register than the variables previously placed in COMMON.

Addresses are assigned using the base-displacement addressing scheme in IBM System/360. Variables are assigned a 2-byte address. The first hexadecimal number in the address is the base register. If the address of a variable assigned to COMMON is 40C8, the variable is assigned the base register 4 with a displacement of 0C8. After assigning addresses to all variables in COMMON, the register number is incremented by 1. If all variables in COMMON are referenced by base register 4, the first item not in COMMON is assigned an address using base register 5.

Indicators in the communications area are set in Phase 10 to indicate that the program being compiled is a FUNCTION or SUBROUTINE subprogram or a main program. A bit is also set if the program being compiled calls subprograms. Phase 12 must reserve storage for a calling sequence depending on these different conditions. In Phase 12, the size of the area needed to contain the initialization routine for the object program is computed, and the location counter is adjusted accordingly.

A check is made to determine if more than three base registers are used to allocate variables in COMMON. If so, a warning message is issued.

EXIT: Subroutine EXTCOM exits to subroutine DPALOC where storage is allocated for double-precision variables.

Subroutine DPALOC: Chart DF

Subroutine DPALOC assigns addresses to all double precision variables or arrays entered in the dictionary.

ENTRANCE: Subroutine DPALOC is entered from subroutine EXTCOM to allocate storage for all double precision variables and arrays. Entry is also made from subroutine INTDCT after it retrieves a name from the dictionary.

OPERATION: Subroutine DPALOC references subroutine INTDCT to search the dictionary. INDCT returns to several subroutines. A switch is set when subroutine DPALOC is entered to force subroutine INTDCT to return to subroutine DPALOC. Subroutine INTDCT only retrieves entries in the dictionary; it does not check mode or type. Subroutine DPALOC only processes the dictionary entry under the following conditions:

1. It is not in COMMON, equated, or a keyword.
2. It is double precision.
3. It is a variable or an array.

The location counter is adjusted to a double-word boundary and the variable is assigned an address. The dictionary entry is removed from the chain and the variable is inserted into the storage map.

If the variable is the root of an EQUIVALENCE group or class, its entry is accessed in the EQUIVALENCE table to allow the location counter to be increased by the size of the class. If the variable is an array, the dictionary entry is accessed to allow the location counter to be increased by the size of the array. If the variable is neither an array nor the root of an EQUIVALENCE class, the location counter is increased by 8, the size in bytes of a double precision variable.

EXITS: Subroutine DPALOC exits to the following subroutines:

1. INTDCT to retrieve another variable or array.
2. SALO to assign addresses to real and integer variables after all double precision variables have been allocated.

SUBROUTINES CALLED: During execution subroutine DPALOC references the following subroutines:

1. INTDCT to retrieve entries in the dictionary.
2. SORSYM to enter variables in the storage map and print them.

3. EQSRCH to search the entry for a root in the EQUIVALENCE table.

Subroutine SALO: Chart DG

Subroutine SALO retrieves real and integer variables and arrays from the dictionary by use of subroutine INTDCT and assigns an address to them.

ENTRANCE: Subroutine SALO is entered from subroutine DPALOC after DPALOC has assigned addresses to all double precision variables and arrays. SALO is also entered from subroutine INTDCT after INTDCT has returned a name.

OPERATION: When subroutine SALO is entered, a switch is set to indicate to subroutine INTDCT that the entry was from SALO. Control is passed to INTDCT to retrieve the first entry in the dictionary.

A name removed from a dictionary chain has an address assigned to it by subroutine SALO if the following conditions are fulfilled:

1. The variable is not equated to a root.
2. The symbol is a name of a variable or an array.
3. The name is not a keyword or in-line function.

Both integer and real variables are processed by subroutine SALO.

If the variable is the root of an EQUIVALENCE group that contains a double precision variable, it is adjusted to a double-word boundary, even through the variable is single precision. The variable is removed from a dictionary chain, assigned an address, entered into the storage map and printed if the MAP option is specified.

If the variable is the root of an EQUIVALENCE group, its entry in the EQUIVALENCE table is accessed. The EQUIVALENCE group size is used to increment the location counter. If the name is an array, the size of the array is accessed in the dictionary and used to increment the location counter. If the name is not an array, the location counter is incremented by 4, the size in bytes of a real or integer variable.

EXITS: Subroutine SALO exits to subroutines:

1. INTDCT to access another dictionary entry.
2. ALOC after all real and integer variables are assigned addresses.

SUBROUTINE CALLED: During execution subroutine SALO references subroutines:

1. EQSRCH to find the entry for the root of an EQUIVALENCE group in the EQUIVALENCE table.
2. SORSYM to enter a variable in the storage map.

Subroutine ALOC: Chart DH

Subroutine ALOC assigns addresses to all equated variables. The address of the root of the variable is assigned in subroutines DPALOC, SALO, and COMAL.

ENTRANCES: Subroutine ALOC is entered by subroutines SALO after SALO has assigned addresses for the real and integer variables. ALOC is also entered from INTDCT when INTDCT retrieves a name from a dictionary chain.

OPERATION: The first time control is passed to subroutine ALOC, it sets the multiple switch used in subroutine INTDCT to return to subroutine ALOC. Control is passed to INTDCT to retrieve the first name in the dictionary.

Subroutine ALOC assigns addresses only to those entries in the dictionary that are equated. If a returned entry is not equated, ALOC returns to INTDCT to retrieve another entry. If an equated name that is neither a variable nor an array is returned to ALOC, an error condition is noted.

If the name is equated and it is the name of a variable or an array, subroutine ALOC removes it from the dictionary chain and finds the variable in the EQUIVALENCE table. If the root of the variable is not in COMMON, the address is computed using the displacement (computed in the Phase 12 EQUIVALENCE subroutines) and the address assigned to its root by either DPALOC or SALO. If the variable is in COMMON, the address assigned in the EQUIVALENCE subroutines is retrieved from the EQUIVALENCE table. The assigned address of the variable is moved into the dictionary entry. The symbol is then entered and printed in the storage map if the MAP option is specified by the user.

EXITS: Subroutine ALOC exits to the following subroutines:

1. LDCN after the name chains in the dictionary are processed.
2. INTDCT to retrieve another dictionary entry.

SUBROUTINES CALLED: During execution subroutine ALOC references the following subroutines:

1. ALERET if an error is detected.
2. EQSRCH to find the variables in the EQUIVALENCE table.

Subroutine LDCN: Chart DI

Subroutine LDCN may punch and/or write ESD section definition cards on the GO tape for the program and the COMMON area. LDCN processes all dictionary entries that are external functions, in-line functions, external references, or arguments for a function definition.

ENTRANCE: Initially, subroutine LDCN is entered from subroutine ALOC after ALOC has assigned addresses for all equated variables. Subsequently, LDCN is entered from subroutine INTDCT each time INTDCT has retrieved another name from the dictionary.

OPERATION: The section definition card for the program contains the program name. The section definition card for COMMON contains the word COMMON and the number of bytes necessary to contain the variables and arrays in COMMON. An entry ESD card contains the name of the program entry point and its displacement from the beginning of the program. LDCN calls a subroutine to punch and/or write these cards on the GO tape.

Subroutine LDCN removes all name entries from the dictionary, except keywords, by destroying the chain address entry. Either the assigned address for the variable or an in-line function code is placed in the dictionary field that contained the chain address in Phase 10. After subroutines DPALOC, SALO, and ALOC have completed processing, only the in-line functions, dummy variables for functions, external functions, and keywords remain in the dictionary name chains.

When subroutine LDCN is first entered, the multiple switch is set to allow subroutine INTDCT to return to LDCN. Subroutine LDCN references subroutine INTCON to retrieve names from the dictionary. When subroutine INTDCT returns the name, LDCN checks the ESD bit in the dictionary entry. If the ESD bit is not on, subroutine LDCN assumes the name is a keyword and references subroutine INTDCT to obtain another name from the dictionary. If the ESD bit is on, the name is either a dummy name used to define parameters for a FUNCTION or SUBROUTINE subprogram, an external symbol, or an in-line function.

If the name is a dummy variable, an address is assigned and the variable is removed from the dictionary chains. If the name is an in-line function, the number code inserted in the address field of the dictionary entry is moved to the field that was originally the chain field for that dictionary entry. If the name is an external function, an address constant address is assigned and ESD and RLD cards may be punched and/or written on the GO tape.

EXITS: Subroutine LDCN exits to the following subroutines:

1. INTDCT to retrieve names from the dictionary.
2. ASGNBL to begin assignment of the branch list.

SUBROUTINES CALLED: During execution subroutine LDCN references the following subroutines:

1. ESD to punch and/or write ESD and RLD cards.
2. GO FILE to punch and/or write the control section cards for the program and the COMMON area.

Subroutine ASGNBL: Chart DJ

Subroutine ASGNBL scans the statement number chains in the overflow table, and allocates a branch list position for each statement number referenced by a branch statement.

ENTRANCE: Subroutine ASGNBL is entered from subroutine LDCN after external and in-line function names and external references are processed.

OPERATION: The first statement number chain address is accessed in the thumb index of the overflow table, and ASGNBL begins to scan the statement number chain. A relative number is assigned to each referenced statement number excluding those assigned to a FORMAT or specification statement. The first statement number in the first chain is assigned the relative number 0 if it is to be entered in the branch list. The relative number is inserted in the chain field for that statement number, and the statement number is removed from the overflow table chain. The relative number is then incremented by 4 and the next statement number is assigned a position in the branch list.

When the end of a chain is reached, ASGNBL accesses the thumb index for the beginning of the next chain; the next statement number chain is then processed.

When the last chain is processed, ASGNBL increments the location counter by the size of the branch list and exits to subroutine SSCK.

EXIT: After the statement numbers are assigned relative numbers for the branch list, subroutine ASGNBL exits to subroutine SSCK.

Subroutine SSCK: Chart DK

Subroutine SSCK searches the three subscript variable chains in the overflow table, and replaces the dictionary pointer to variables with the address assigned the variable by Phase 12.

ENTRANCE: Subroutine SSCK is entered from subroutine ASGNBL after ASGNBL has assigned branch list positions for statement numbers.

OPERATION: Subroutine SSCK obtains the address of the first subscript chain from the thumb index of the overflow table and then searches the chain for variables in a subscript expression. In Phase 10, if a subscript expression contained no variables, it was not entered in the overflow table. For example, there was no entry in the overflow table for the subscripted variable A(1,5,2). If only one subscript parameter contained a variable, however, the subscript expression was entered in the overflow table. For those subscripts within the same expression that contained no variables, the space allotted for variables was filled with zeros. In the subscripted variable A(1,5,2*I), space had to be allotted for variables in the first and second subscripts, even though this space was filled with zeros.

In Phase 12, a check is made in SSCK to determine if zeros are in a field where there normally would be a dictionary pointer. If there is a zero, this field is ignored and the next field expected to contain a variable is checked. If the next field contains a pointer to a dictionary entry for a variable, the dictionary entry is accessed and the address assigned to that variable is placed in the subscript variable chain. (The address for the variable in the subscript expression has been inserted in the dictionary entry for the variable by subroutine SALO.)

When SSCK reaches the end of one of the subscript chains, it checks to see if this is the last chain to be processed. If another chain remains to be processed, SSCK goes to the overflow table thumb index and obtains the address for the start of the next chain.

EXIT: After all subscripts in the overflow table are processed, subroutine SSCK passes control to subroutine SORLIT.

Subroutine SORLIT: Chart DL

Subroutine SORLIT assigns addresses and calls subroutine TXT to punch and/or write text cards for all literals entered in the dictionary. SORLIT then calls the FORTRAN System Director to load Phase 14.

ENTRANCE: Subroutine SORLIT is entered from subroutine SSCK after SSCK has processed the overflow table subscript entries.

OPERATION: Subroutine SORLIT accesses the beginning address for the integer constant chain in the dictionary, assigns an address for each constant in the chain, and calls subroutine TXT to put the constants into a text card buffer. Text cards may be punched and/or written on the GO data set.

When the end of the integer constant chain is reached, the beginning address of the real constant chain is accessed in the thumb index of the dictionary. The real constants are then assigned an address and placed in a text card buffer.

When the end of the real constant chain is reached, the address for the chain containing double precision constants is accessed, and the location counter is set to a double-word boundary. All double precision constants are assigned an address; the constants are then put into a text card buffer. When all double precision constants are processed and put in the text card format, any buffers used to punch the ESD and RLD cards are closed, buffers used for text cards are closed, and the FORTRAN System Director is called to read Phase 14.

EXIT: After all constants have been processed, subroutine SORLIT exits to the FORTRAN System Director to read Phase 14 from the system tape.

SUBROUTINES CALLED: During execution subroutine SORLIT references the following subroutines:

1. TXT to put the constant and its address out to a text card and to close the text card output buffers.
2. ESD to close the ESD and RLD card output buffers.
3. SORSYM to enter the constant into the storage map.

Subroutines EQSRCH, RENTER/ENTER: Chart DM

Subroutine EQSRCH

Subroutine EQRSCH searches the EQUIVALENCE table for a previously entered variable. EQSRCH is also used by subroutine SWROOT to find all variables that have been equated to a root.

ENTRANCES: Subroutine EQSRCH is entered from subroutines EQUIVALENCE Part 1, Part 2, and Part 3 to determine if a variable was entered previously in the EQUIVALENCE table. EQSRCH is also entered from subroutine SWROOT to find all variables equated to one root that must be equated to another root.

OPERATION: Subroutine EQSRCH is entered at two points. At the first point of entry, registers are initialized to search the EQUIVALENCE table for a previously entered variable. The registers are initialized to compare the dictionary pointers to the variable now being processed. The pointer of the variable currently being processed is in the EQUIVALENCE text.

At the second point of entry, the registers have been initialized by the calling subroutine to compare the pointer to the roots in the EQUIVALENCE table. EQSRCH checks the EQUIVALENCE table to find all variables previously equated to a root.

EQSRCH searches the EQUIVALENCE table by checking each entry for the correct pointer until it finds the entry or reaches the end of the EQUIVALENCE table. If EQSRCH reaches the end of the table without finding the entry for the variable, it takes the not found return; if the variable is found, EQSRCH takes the found return.

EXIT: Subroutine EQSRCH exits to the calling subroutine.

Subroutine RENTER/ENTER

Subroutine RENTER/ENTER enters variables in the EQUIVALENCE table either as a root or an equated variable.

ENTRANCE: Subroutine RENTER/ENTER is entered at two points. The first point of entry (RENTER) is entered by subroutines EQUIVALENCE Part 1, Part 2, and Part 3, when a variable is to be entered in the EQUIVALENCE table as a root. The second point of entry (ENTER) is entered by EQUIVALENCE Part 2 when a variable is to be entered as an equated variable.

OPERATION: If the variable is entered as a root, the bit that indicates a root is set in the usage field of its dictionary entry. If the variable is in COMMON, the address assigned to the variable by subroutine COMAL is accessed from its dictionary entry. If the variable is not entered as a root, the processing becomes the same for both roots and equated variables.

If the variable is double precision and is entered in COMMON, it must be assigned a COMMON address that is on a double-word boundary. If the address assigned to it is not on a double-word boundary, an error is detected.

The pointer to the dictionary entry for the variable, a pointer to the root, and the displacement are entered in the EQUIVALENCE table. (If the variable is entered as a root, the two pointers are the same.)

If the variable is entered as a root and is being entered for the first time, the size entry for the variable in the EQUIVALENCE text is accessed and inserted in the size field in the EQUIVALENCE table. If the variable is equated to a root, a size computation is necessary.

After the size is entered, a check is made to determine if the size of the COMMON area has been extended by equating arrays to variables previously entered in COMMON. For example, the statement:

```
COMMON A
```

places the real variable A in COMMON. The size of the COMMON area is now four bytes.

The statement:

```
DIMENSION B(5)
```

defines a real array whose size is 20 bytes. If the statement:

```
EQUIVALENCE (A,B(1))
```

is processed, the array B must be placed in COMMON. Since the size of the array is 20 bytes, the size of COMMON must be increased to 20 bytes, and the location counter adjusted accordingly.

A check is made to determine if the COMMON area is extended backward, e.g., if the preceding EQUIVALENCE statement is changed to:

```
EQUIVALENCE (A,B(2))
```

Since A is in COMMON, an attempt must be made to include B in COMMON. The second element in B is equated to A, therefore

B(1) should be the first element in COMMON. An error is noted because an attempt was made to "extend COMMON backward."

The count of the number of entries entered in the EQUIVALENCE table is incremented by 1 for each entry.

EXIT: The utility subroutine RENTER/ENTER returns control to the calling subroutine.

SUBROUTINE CALLED: During execution subroutine RENTER/ENTER calls subroutine ALERET if an error is detected.

Subroutine SWROOT: Chart DN

The utility subroutine SWROOT is referenced by subroutine EQUIVALENCE Part 1 or Part 2 when EQUIVALENCE Part 1 or Part 2 determines that the root entered previously for this EQUIVALENCE group or class must be changed, and a new root inserted.

ENTRANCE: Subroutine SWROOT is entered by subroutine EQUIVALENCE Part 1 and 2.

OPERATION: Subroutine SWROOT searches the EQUIVALENCE table for the entry for the new root and saves its address so the size of the group can be computed. SWROOT then compares the new root and the old root. If the roots are the same, SWROOT computes the displacement between them. If the displacement is a number other than zero, an error is detected. If the displacement is zero, SWROOT returns to EQUIVALENCE Part 2.

If the old root and new root differ, the equated bit in the usage field of the dictionary for the old root is set to indicate that the old root has been equated to another root.

If the old root is double precision, a bit is set in the usage field to indicate that the new root must be placed on a double-word boundary. If the old root is in COMMON, a check is made to determine if the new root is on a double-word boundary; if not, an error is noted.

A bit is set in the usage field of the dictionary entry for the new root. This bit indicates that this variable is a root of an EQUIVALENCE group or class. Using the address of the old root, the EQUIVALENCE table is searched for all variables that were equated to the old root. The EQUIVALENCE table entries for these variables must be changed to indicate the new root, and the displacement and size fields in the table must also change. If the root is in COMMON, and COMMON has been extended forward, the location counter must be adjusted. If COMMON was extended backward, an error condition is noted.

EXIT: Subroutine SWROOT exits to EQUIVALENCE Part 2 to determine if there are additional variables in this EQUIVALENCE group.

SUBROUTINES CALLED: During execution subroutine SWROOT references the following subroutines:

1. ALERET if an error has been detected.
2. EQSRCH to search the EQUIVALENCE table for the entry made for the new root and for variables equated to the old root.

Subroutine INTDCT: Chart DO

Subroutine INTDCT is used by subroutines DPALOC, ALOC, SALO, and LDCN to retrieve entries from the dictionary.

ENTRANCES: Subroutine INTDCT is referenced by subroutines LDCN, ALOC, SALO, and DPALOC to search the name chains in the dictionary.

OPERATION: Subroutine INTDCT initializes registers and pointers for a search, and checks for names entered in the dictionary. If there is another name, its entry is fetched and INTDCT returns to the subroutine that called it.

If subroutine INTDCT determines that it is at the end of a dictionary chain, it goes to the thumb index, gets the starting address of the next chain, and begins retrieving variables from that chain. A switch is set by the subroutine that called subroutine INTDCT. By testing this switch, subroutine INTDCT returns to the calling subroutine.

EXIT: Subroutine INTDCT returns to the calling subroutine after retrieving an entry from the dictionary. If subroutine INTDCT has returned the last variable in the dictionary chain, INTDCT returns to another point in the calling subroutine.

Subroutine SORSYM: Chart DP

Subroutine SORSYM puts into format and prints the storage map for all arrays, constants, and external references assigned addresses in Phase 12.

ENTRANCE: Subroutine SORSYM is referenced by subroutines SORLIT, COMAL, DPALOC, SALO, ALOC, and ESD.

OPERATION: Variables and arrays, external symbols, and constants are printed for the storage map. If a heading is not printed for a category, SORSYM prints the heading.

The symbol and the address assigned to it are moved to the print buffer; if the buffer is full, a line is printed. There are four symbols with associated addresses inserted in a print line.

EXIT: Subroutine SORSYM exits to the calling subroutine.

SUBROUTINES CALLED: During its execution subroutine SORSYM references the FORTRAN System Director to initialize printing.

Subroutine ESD: Chart DQ

Subroutine ESD enters external symbols into the ESD card format and initializes the action for punching ESD cards.

ENTRANCE: Subroutine ESD is entered by subroutine LDCN every time an external symbol should be entered in an ESD card. Subroutine SORLIT also calls a portion of subroutine ESD to close the ESD data set.

CONSIDERATIONS: An ESD card contains one external symbol identification number, one to three names used as external symbols, and the number of significant bytes in the ESD card. This number on the ESD card includes the number of card columns used to contain the external symbol identification number, the external symbols, and the number itself. Each external symbol in the external symbol dictionary is assigned an external symbol identification number. However, in the ESD card, only the symbol identification number for the first symbol is included.

Subroutine ESD and Phase 10 intermediate text use a similar double-buffer system. There are two card output areas. One buffer is punched as the other is filled.

OPERATION: Subroutine ESD removes the entry for an external symbol from the dictionary chain. The address that contains either the beginning address of the external function or the address of the external symbol is inserted in the chain field.

If neither the DECK nor Compile and Go option is taken, subroutine ESD returns to the calling subroutine. If either the DECK or Compile and Go option is taken, ESD moves the external symbol from the dictionary entry to a buffer and increments the external symbol identification number by 1.

Subroutine ESD determines how many external symbols are entered in the buffer. If this is the first entry for this card, the external symbol identification number is moved to the buffer for the first external symbol entered in the card. The number of significant bytes entered in the buffer for this external symbol is then updated.

If this is the second entry for this card, the number of significant bytes for the second entry is added to the register containing the number of bytes for this card. If this is the third entry to the buffer for this card, the number of significant bytes for the third entry is added to the significant byte accumulator. After the third entry is made to the buffer, the card may be punched and/or written on the GO tape.

ESD is entered to close the buffers after subroutine SORLIT has processed the last constant chain entered in the dictionary. ESD closes the buffer. A switch is set in ESD to indicate to subroutine RLD that the output buffers must be closed. If any external symbols in the buffer are not punched, ESD initializes the punching for the last ESD card.

EXIT: Subroutine ESD exits to subroutine RLD to enter an address in an RLD card and put out the last RLD card if subroutine ESD was entered to close the ESD and RLD buffers.

SUBROUTINE CALLED: Subroutine ESD references subroutine GOFILE to initialize the output for an ESD card.

Subroutine RLD: Chart DR

Subroutine RLD enters the addresses assigned to external symbols in an RLD card and initializes the action for RLD card output.

ENTRANCE: Subroutine RLD is entered by subroutine ESD each time ESD makes an entry to an ESD buffer and when ESD closes the output buffers for RLD and ESD cards.

CONSIDERATIONS: An RLD card contains up to six address constant addresses. Each address and its external symbol identification number may be punched in the RLD card and/or written on the GO tape. An RLD address and ESD name are linked by the same external symbol identification number.

Subroutine RLD uses a double buffer system similar to the one used for ESD cards. There are two card output areas.

An RLD card is punched from one buffer, while the next RLD card is put into format in the other buffer.

OPERATION: Subroutine ESD sets an end switch to indicate to subroutine RLD that the RLD and ESD buffers are to be closed. If this end switch is on, RLD checks for any entries in the current buffer that have not been put into an RLD card format. If there are current entries, RLD calls subroutine GOFILE to initialize card output.

If the end switch is not on, RLD must make an entry to an RLD card buffer. Subroutine RLD inserts the address constant address in the RLD buffer along with the external symbol identification number and updates the location counter so that the correct address constant address is is assigned to the next external symbol. If the RLD card buffer does not contain six items, the RLD buffer pointer is incremented by the length of the entry just made in the buffer pointer so that the next load constant address can be inserted in the proper position in the buffer. The external symbol identification number is incremented in subroutine ESD. If an RLD card buffer is full, subroutine GOFILE is called to put out the RLD cards.

EXIT: Subroutine RLD exits to the subroutine that originally referenced subroutine ESD.

SUBROUTINES CALLED: Subroutine RLD references subroutine GOFILE to put out the RLD card.

Subroutine TXT: Chart DS

Subroutine TXT enters constants along with a beginning address in a text card and calls a subroutine to allow text card output.

ENTRANCE: Subroutine TXT is referenced by subroutine SORLIT when SORLIT processes the constant chains in the dictionary and closes the text card buffers.

CONSIDERATIONS: A text card generated by Phase 12 contains an address and a string of integer, real, or double precision constants. The address entered in the card indicates to the object program loader where to put the first constant in the text card. The remaining constants follow the first constant.

Subroutine TXT uses a double-buffer system similar to subroutines ESD and RLD.

OPERATION: Subroutine TXT initializes itself in Phase 12 by moving the pointer to the dictionary entry for the constant being processed from subroutine SORLIT. The constant is then moved from the dictionary to a text card buffer, and the number of significant bytes for this card. If this is the first entry for this card, the address in the location counter is inserted in the card.

The location counter is adjusted by the length of the constant (8 for double precision and 4 for real or integer). Subroutine TXT then checks for a DECK or GO option. If neither option is on, TXT returns to the subroutine that called it and does not put out a text card.

If either option is on, a check is made to see if the buffer is full. If the buffer is not full, the buffer pointer is updated by the size of the entry just made, and subroutine TXT returns to the calling subroutine. If the buffer is full, subroutine GOFILE is called to initialize card output.

Subroutine TXT is entered by subroutine SORLIT after SORLIT has processed the last constant chain. If either text buffer contains entries, subroutine GOFILE is called to put out the entries.

EXIT: Subroutines TXT returns to the calling subroutine.

SUBROUTINE CALLED: During execution subroutine TXT references subroutine GOFILE to punch a card and/or write a record on the GO tape.

Subroutine GOFILE: Chart DT

Subroutine GOFILE updates the card sequence number and inserts it and the program identification into the card buffer. GOFILE calls the FORTRAN System Director to punch a card or write the card image on the GO tape.

ENTRANCES: Subroutine GOFILE is entered from subroutines ESD, RLD, or TXT if the user has specified that either an object deck be punched under the DECK option or the object program loaded on tape under the Compile and Go option.

CONSIDERATION: The calling subroutine assembles the card in the correct format for the FORTRAN System Loader. The card sequence number and the card identification for the program are inserted in the card by GOFILE.

OPERATION: Subroutine GOFILE increments the card sequence number by 1, and enters it in the ESD, RLD, or text cards. GO FILE places the program identification in the card and tests the Compile and Go option. If the Compile and Go option was exercised, GOFILE references the FORTRAN System Director to write the card to the tape used for the GOFILE output tape. If the DECK option is taken, GOFILE calls the FORTRAN System Director to punch the card for the object deck.

EXIT: Subroutine GOFILE returns control to the calling subroutine.

SUBROUTINE CALLED: During execution, subroutine GOFILE references the FORTRAN System Director to write the card image to the GO tape or to punch a text card.

Subroutine ALOWRN/ALERET: Chart DU

Subroutine ALOWRN/ALERET processes errors and warnings detected during Phase 12.

ENTRANCES: Subroutine ALOWRN is entered from subroutine EXTCOM if the COMMON area is too large. Subroutine ALERET is entered from subroutines EQUIVALENCE Part 2 and Part 3, RENTER/ENTER, SWROOT, and COMAL when an error is detected.

OPERATION: Subroutine ALOWRN/ALERET is entered at two points. The first entry point, ALOWRN, is used if a warning is issued. The second, ALERET, is used if an error is issued. The difference between errors and warnings is discussed in the Phase 10 subroutine ERROR, ERRET/WARNING.

The only intermediate text entries made during Phase 12 are the entries for errors and warnings detected during execution of the Phase 12. The adjective code is set when ALOWRN/ALERET is entered, depending whether the subroutine is entered for an error or warning. The error or warning number is inserted with the adjective code in the intermediate text output buffers. The internal statement number for the statement which caused the error or warning is not entered in Phase 12. There is no way to access the internal statement number and in most cases the condition that caused the error or warning cannot be pinpointed to a specific statement.

The output buffer pointer is updated, and a test is made to check if the output buffer is full. If the buffer is full, subroutine ALOWRN/ALERET exits to subroutine SORLIT to abort the rest of Phase 12 and read in Phase 14.

EXIT: Subroutine ALOWRN/ALERET exits to the subroutine that called it or to subroutine SORLIT.

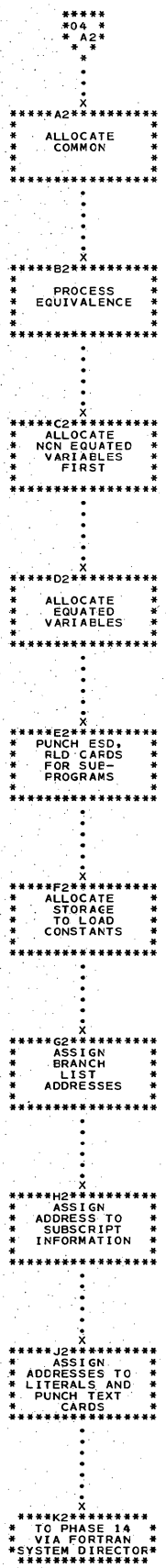


Chart 04. Phase 12 Overall Logic Diagram

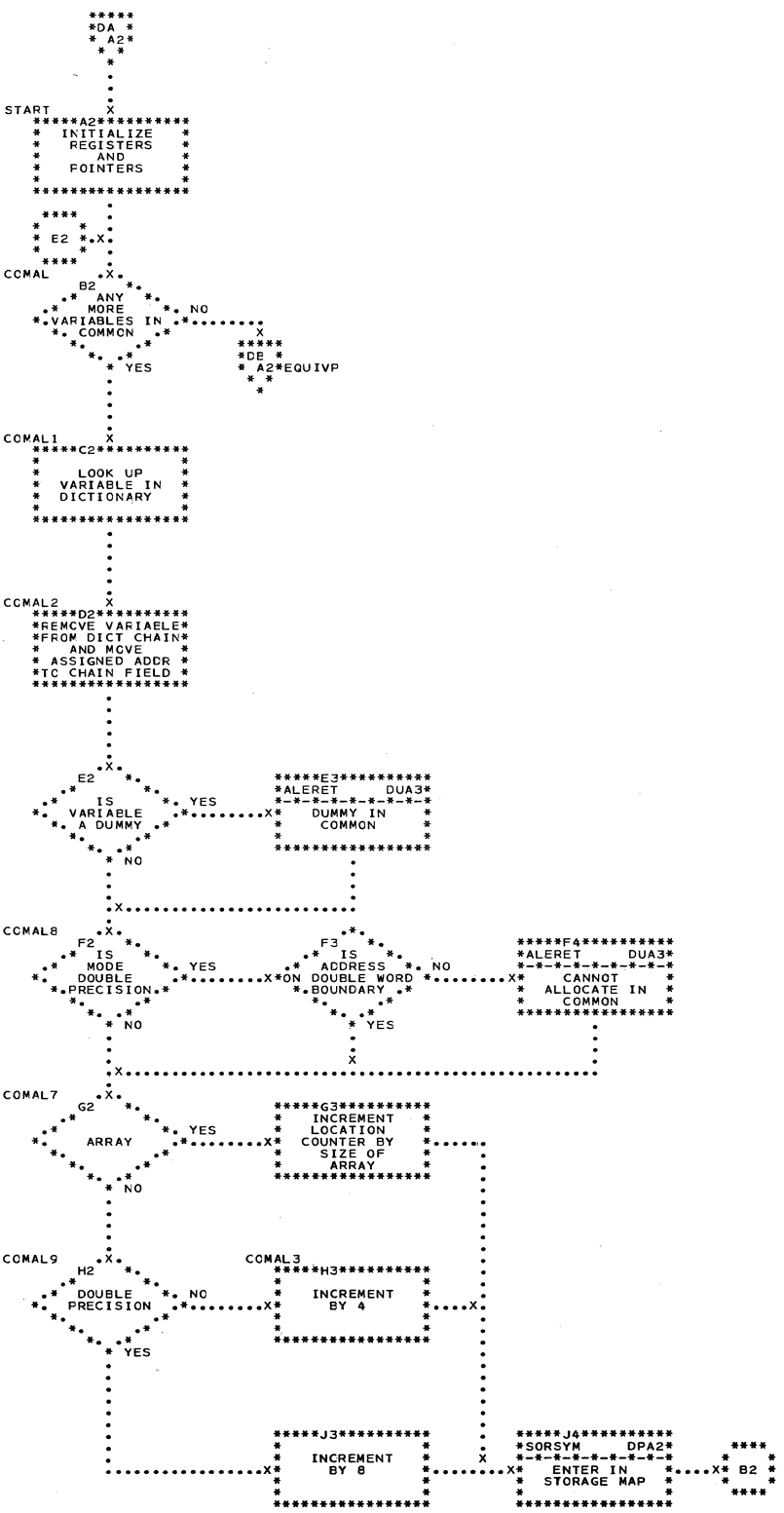


Chart DA. Subroutine COMAL

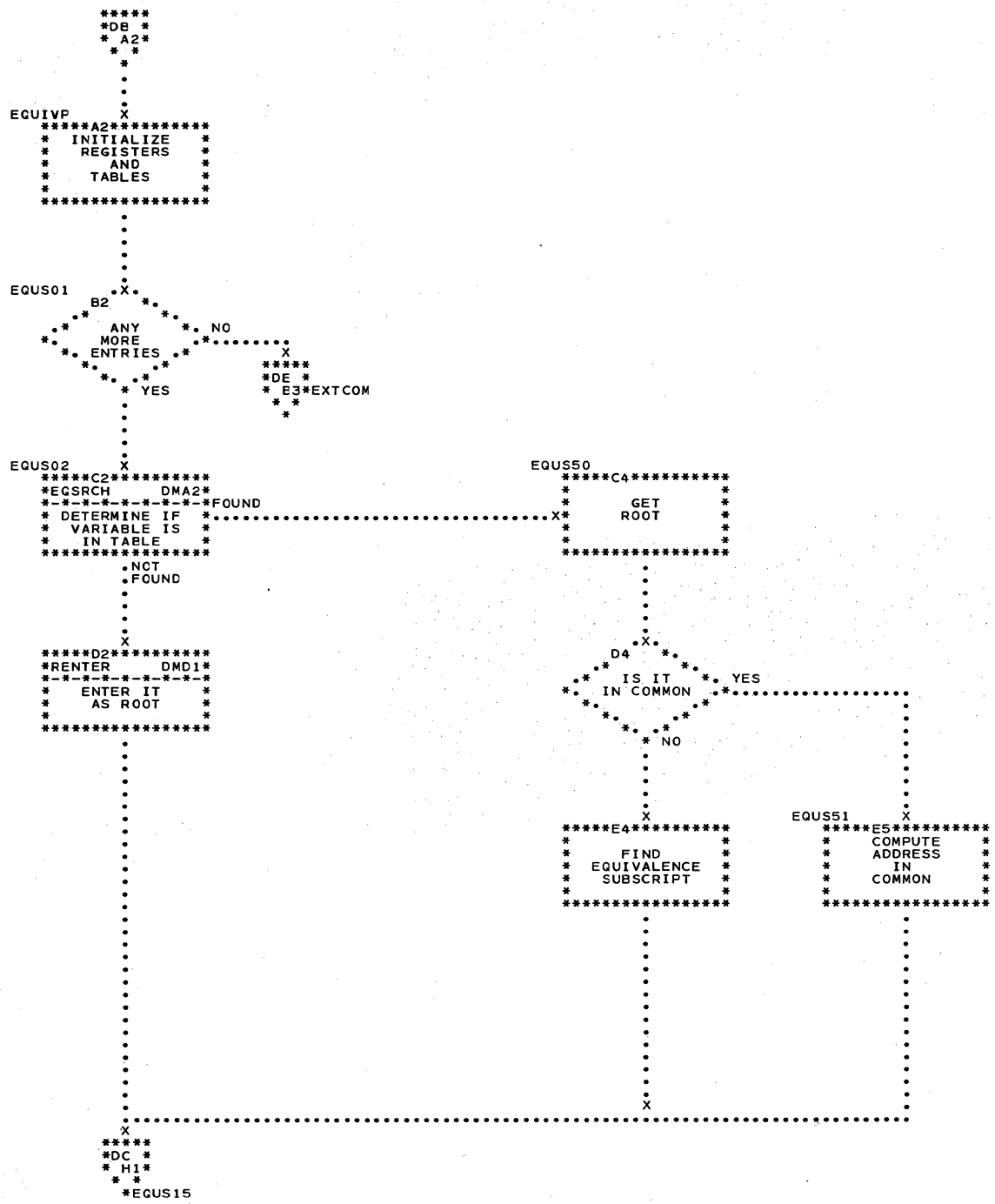


Chart DB. Subroutine EQUIVALENCE Part 1

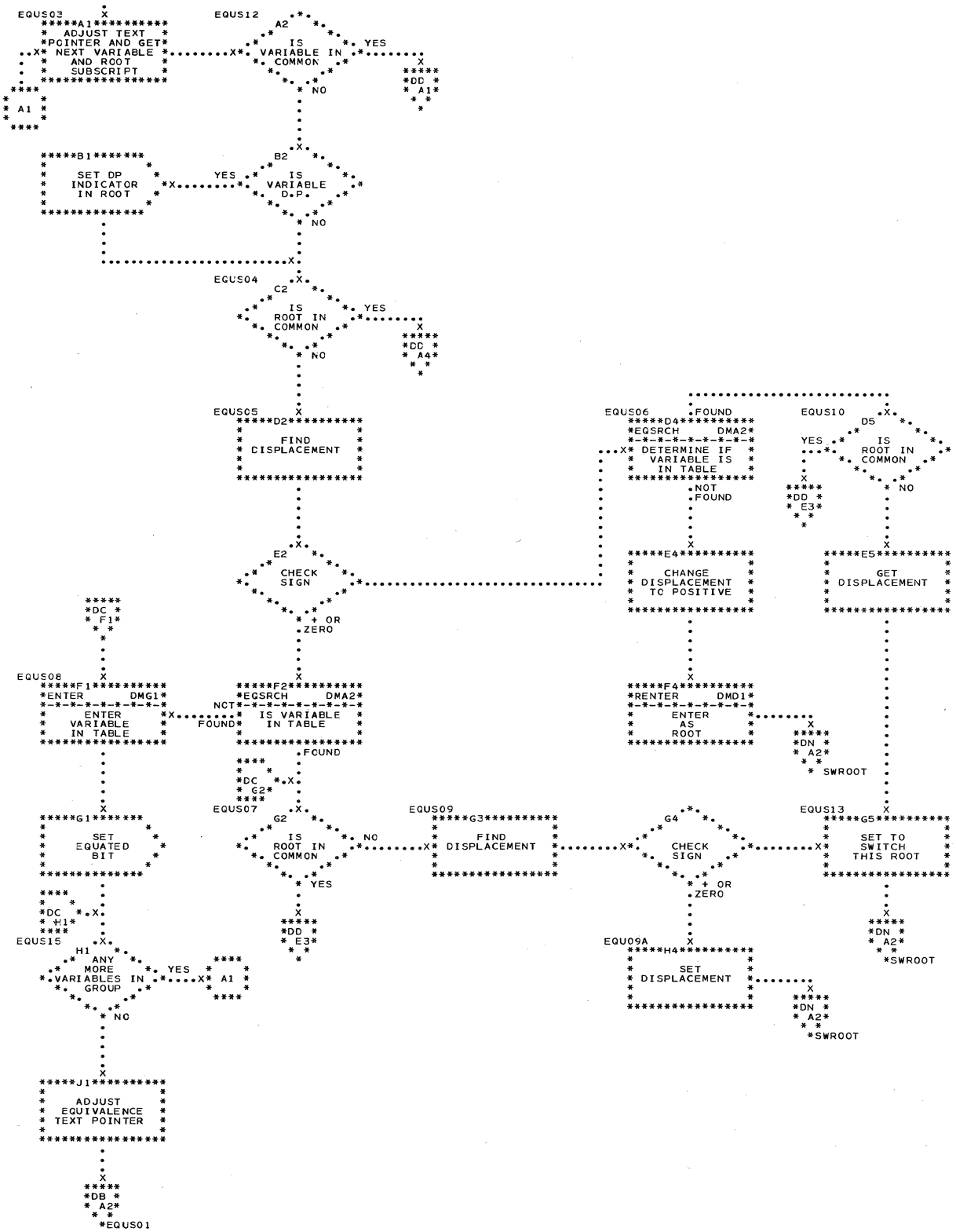


Chart DC. Subroutine EQUIVALENCE Part 2

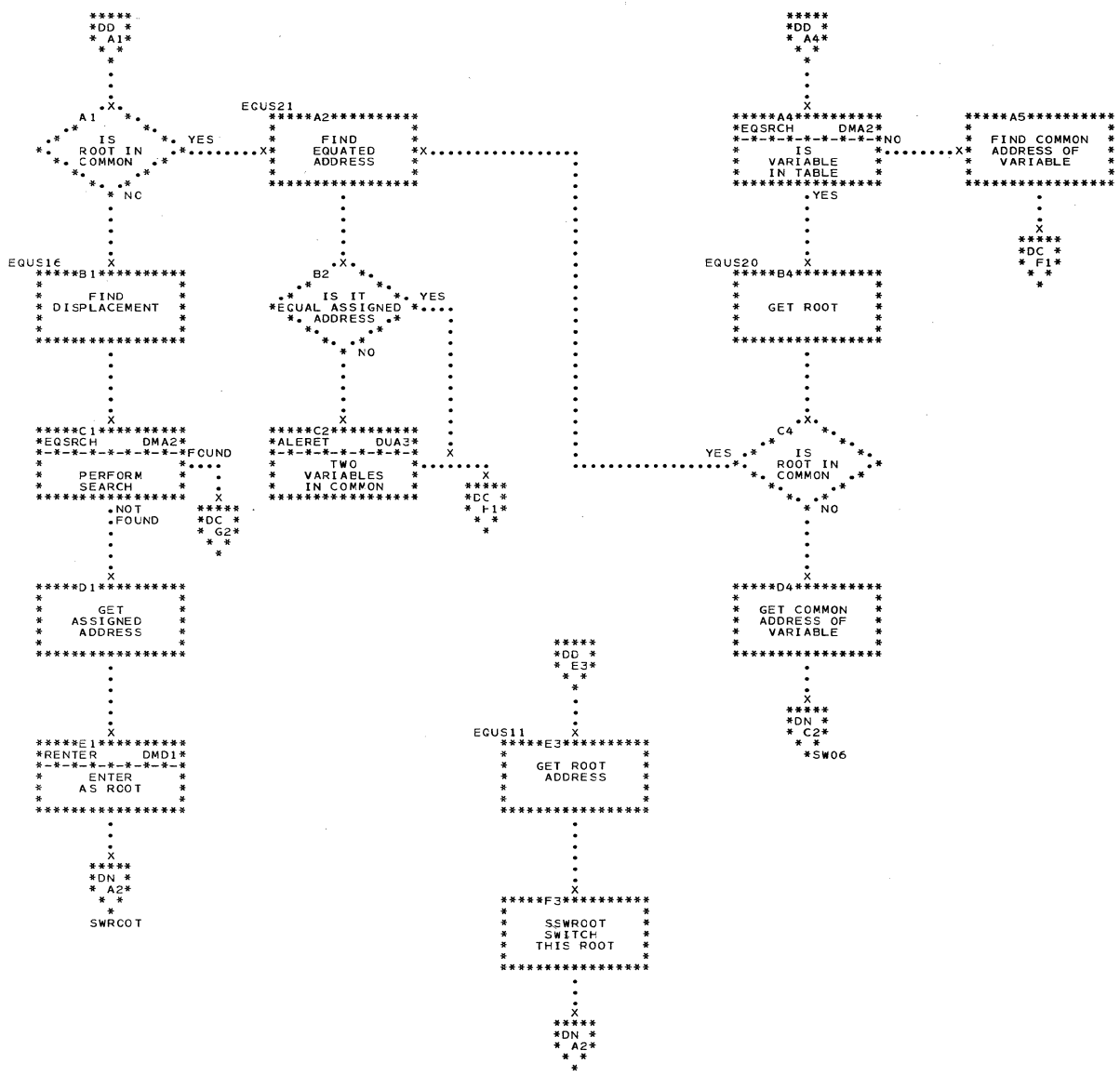


Chart DD. Subroutine EQUIVALENCE Part 3

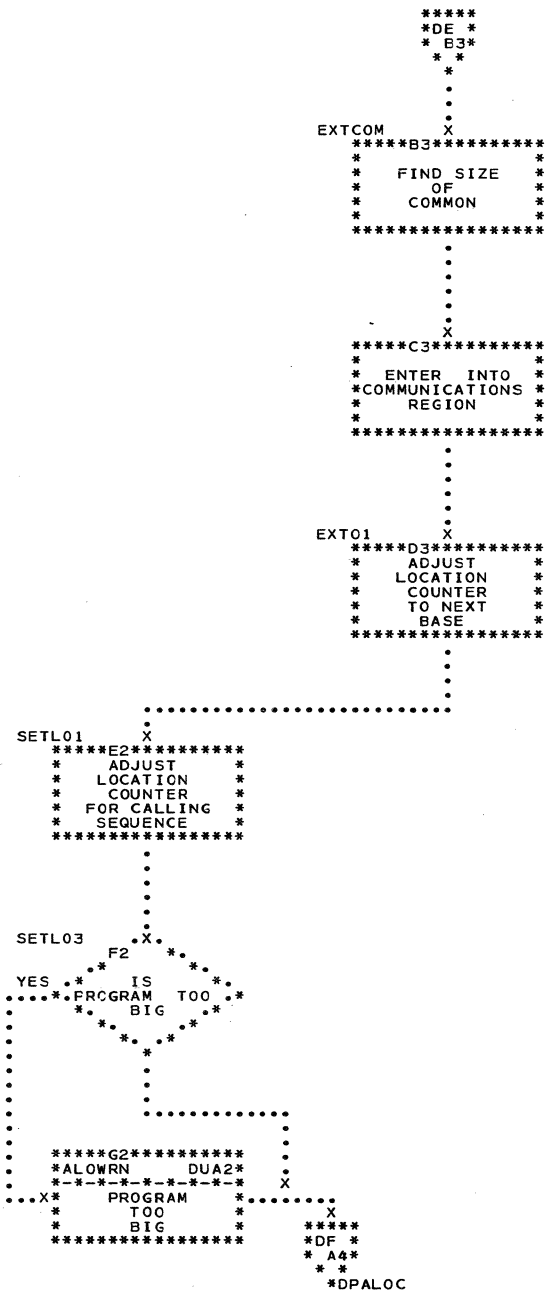


Chart DE. Subroutine EXTCOM

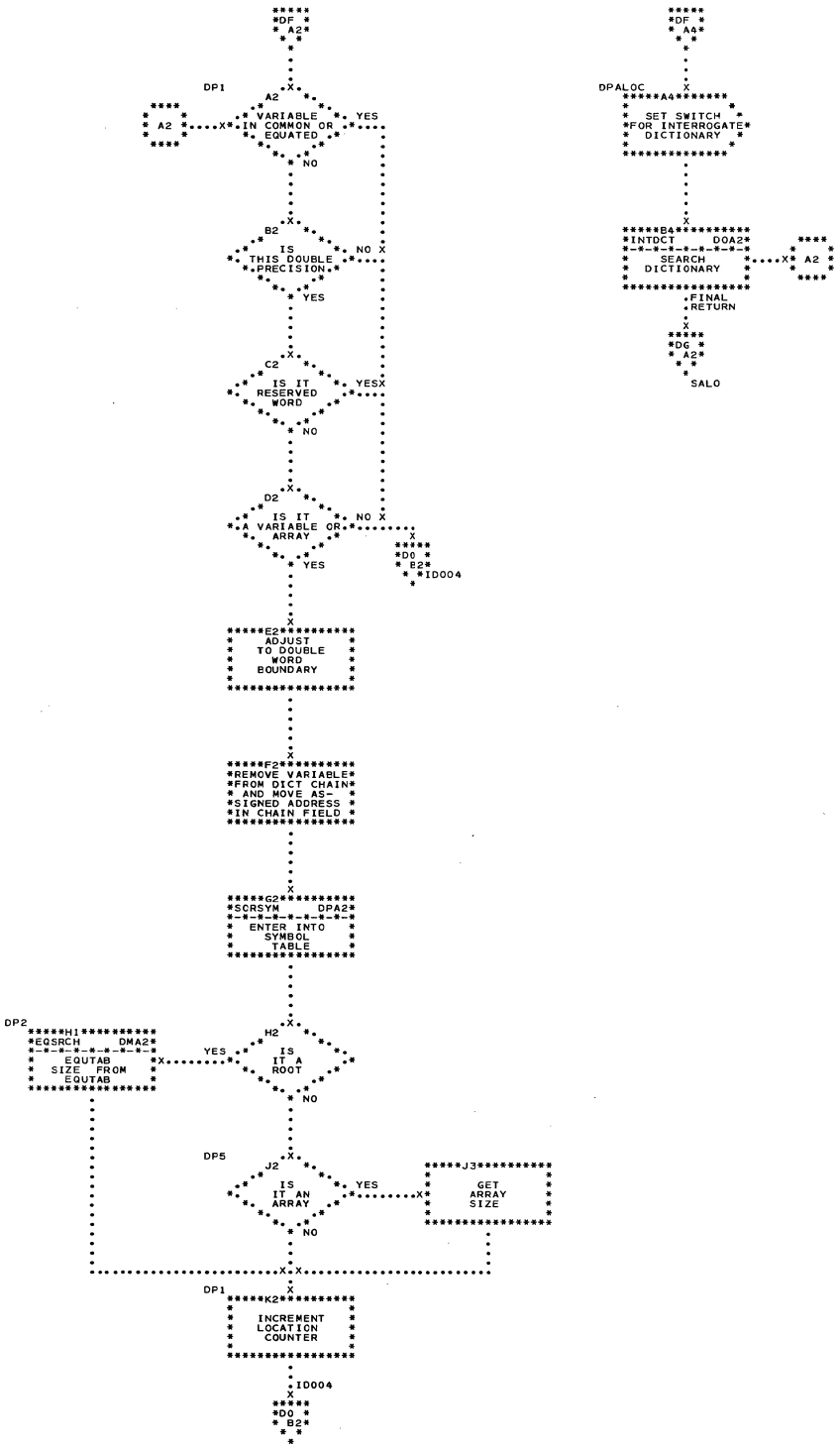


Chart DF. Subroutine DPALOC

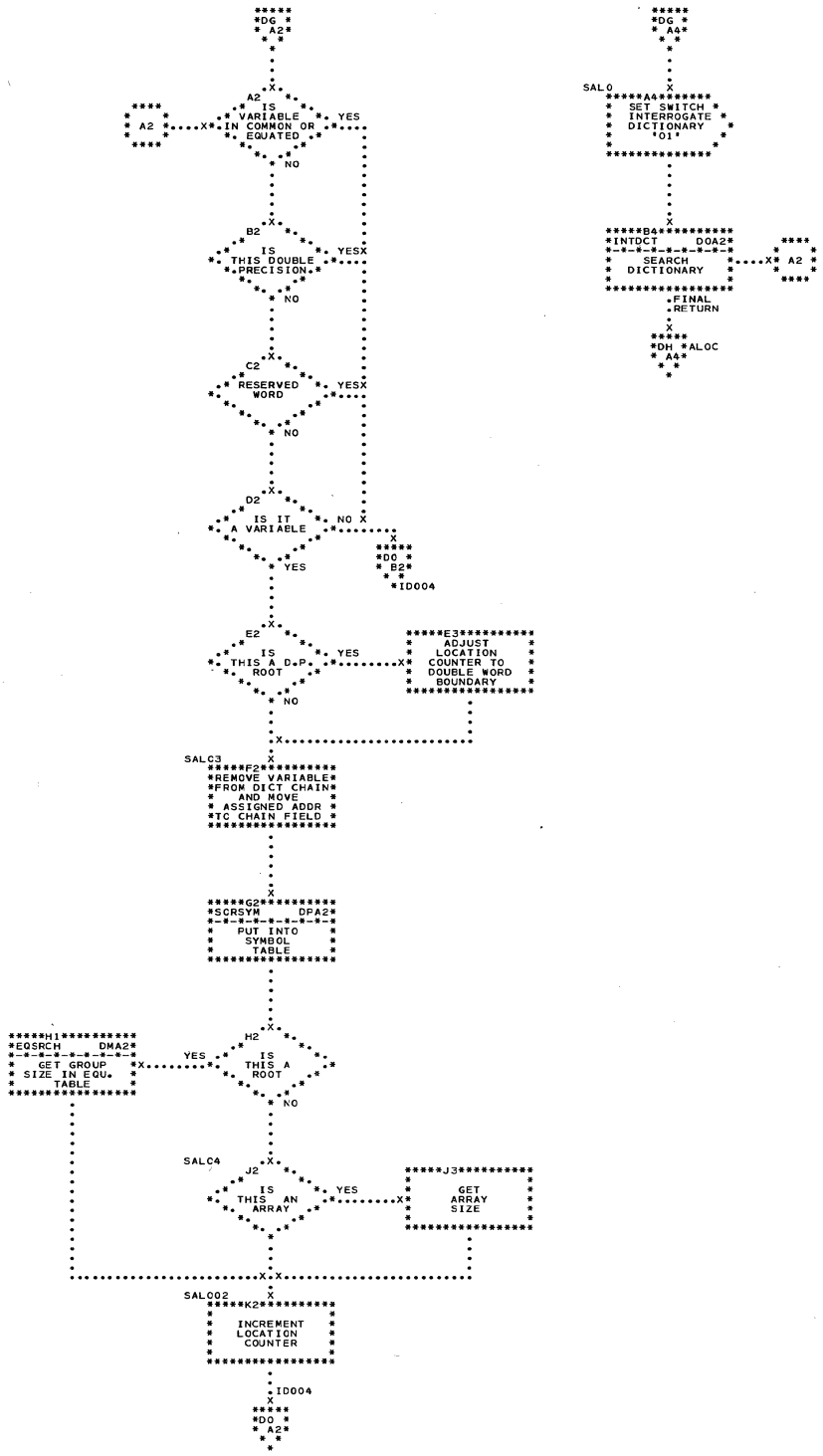


Chart DG. Subroutine SALO

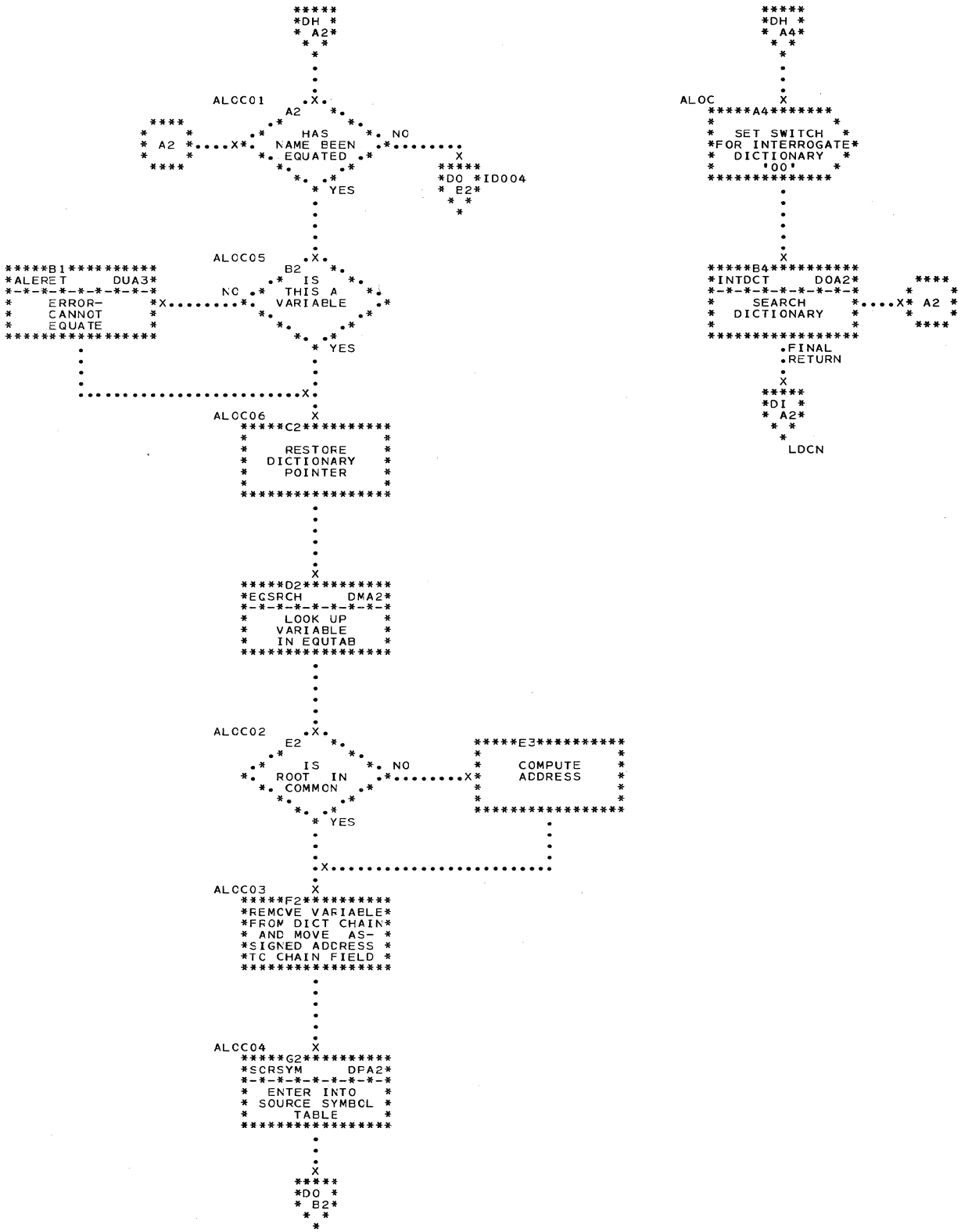


Chart DH. Subroutine ALOC

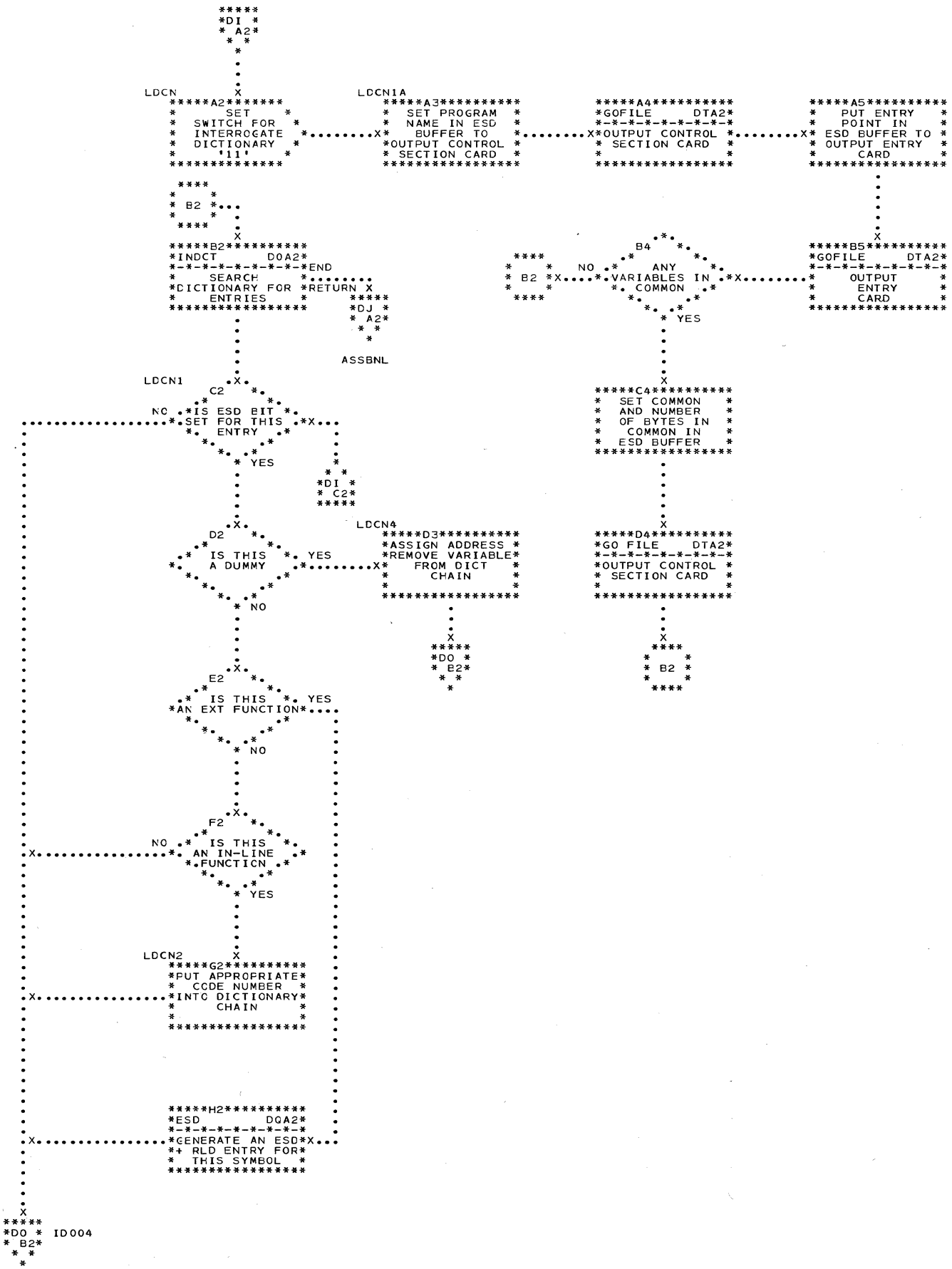


Chart DI. Subroutine LDCN

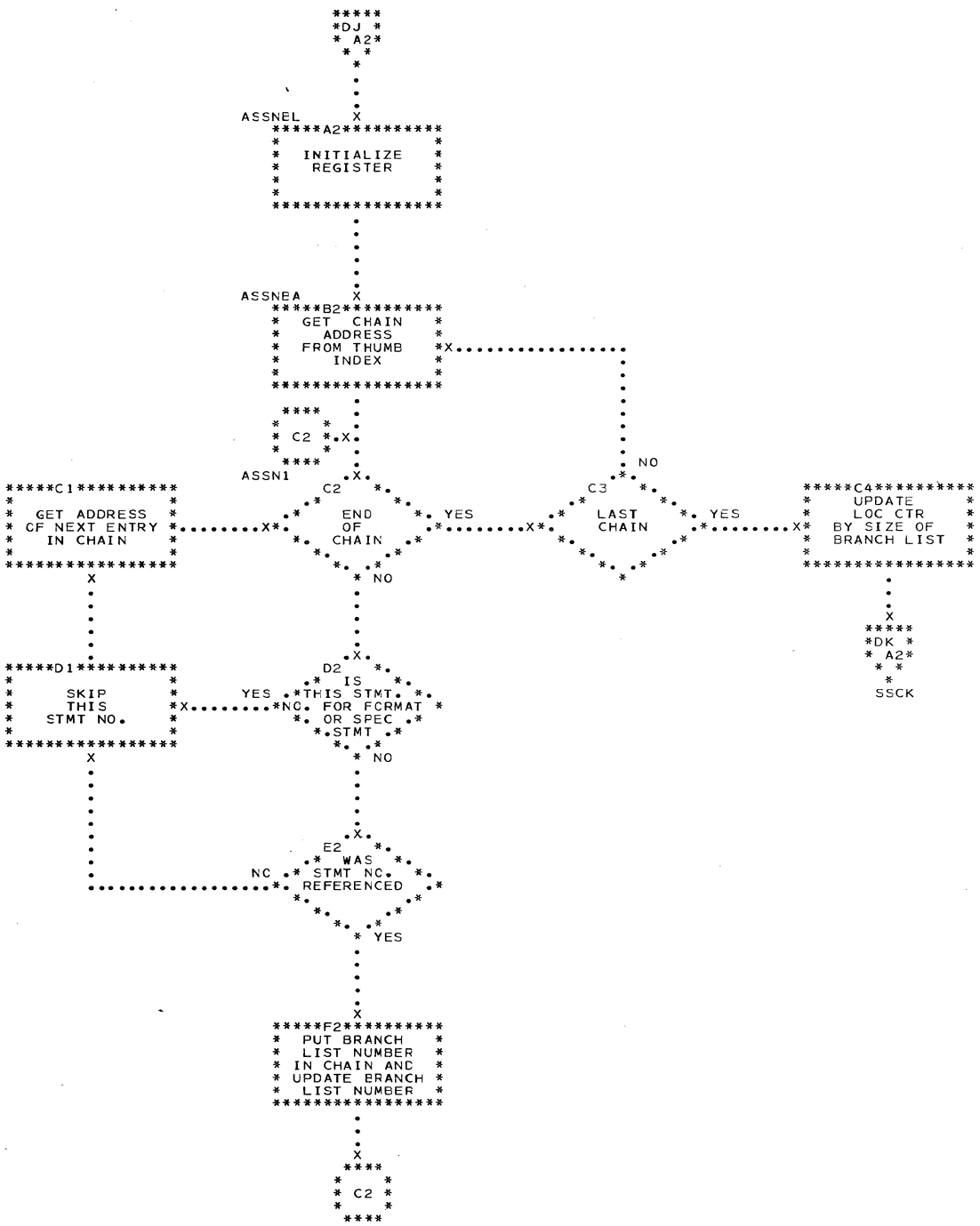


Chart DJ. Subroutine ASSNBL

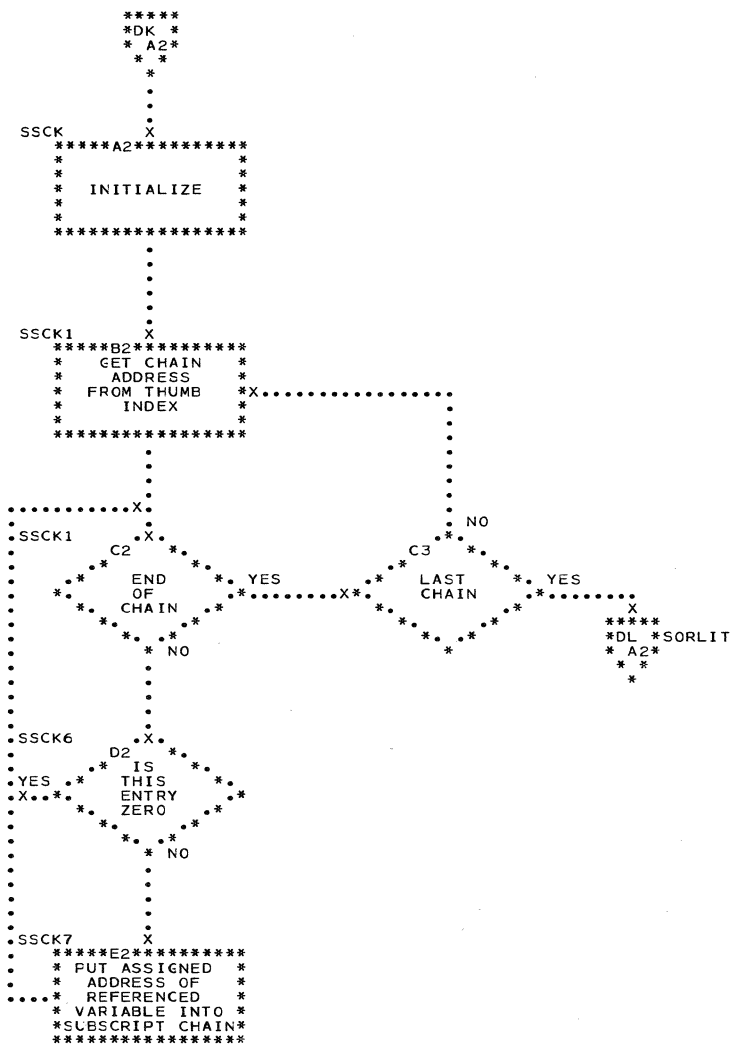


Chart DK. Subroutine SSCK

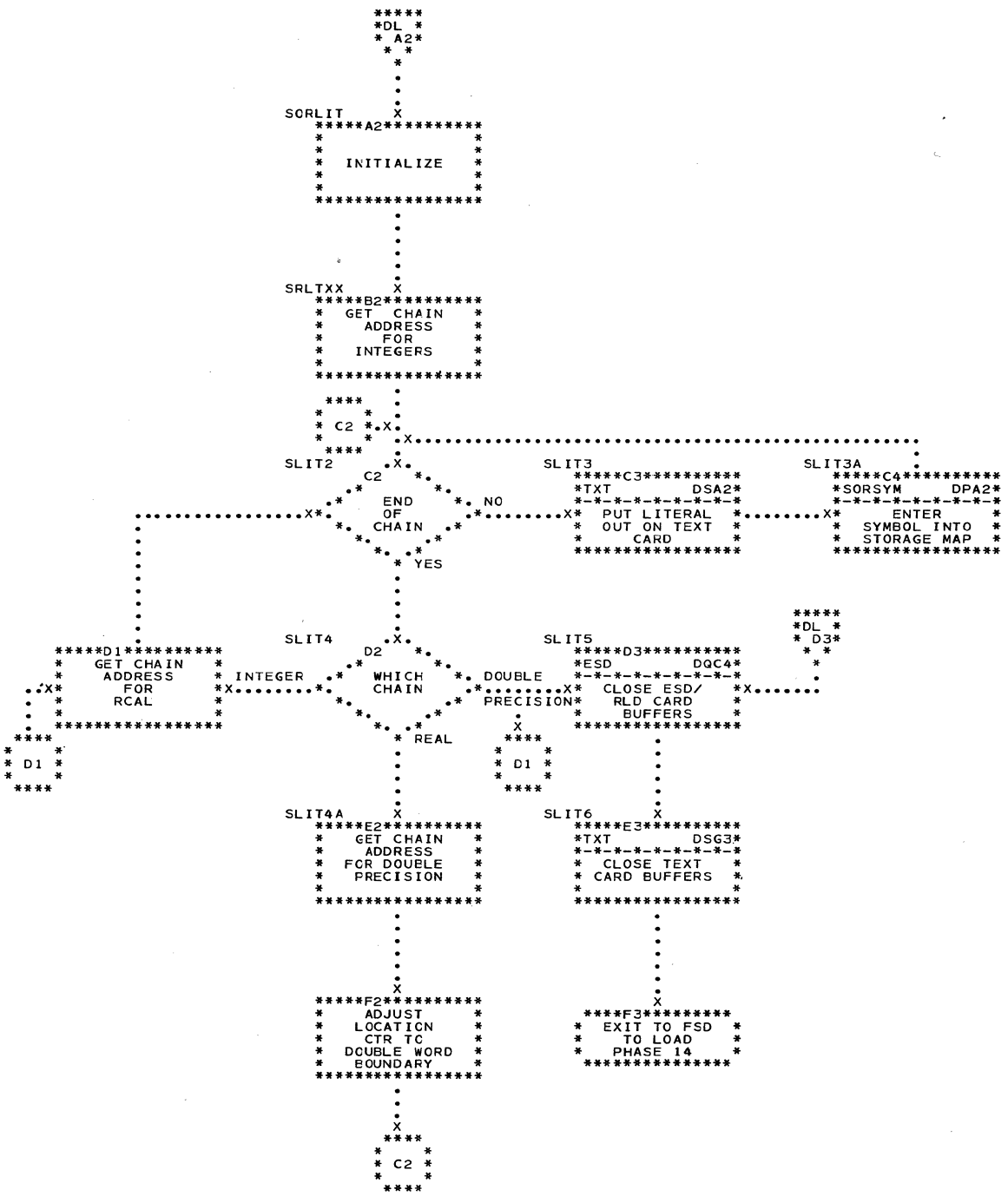


Chart DL. Subroutine SORLIT

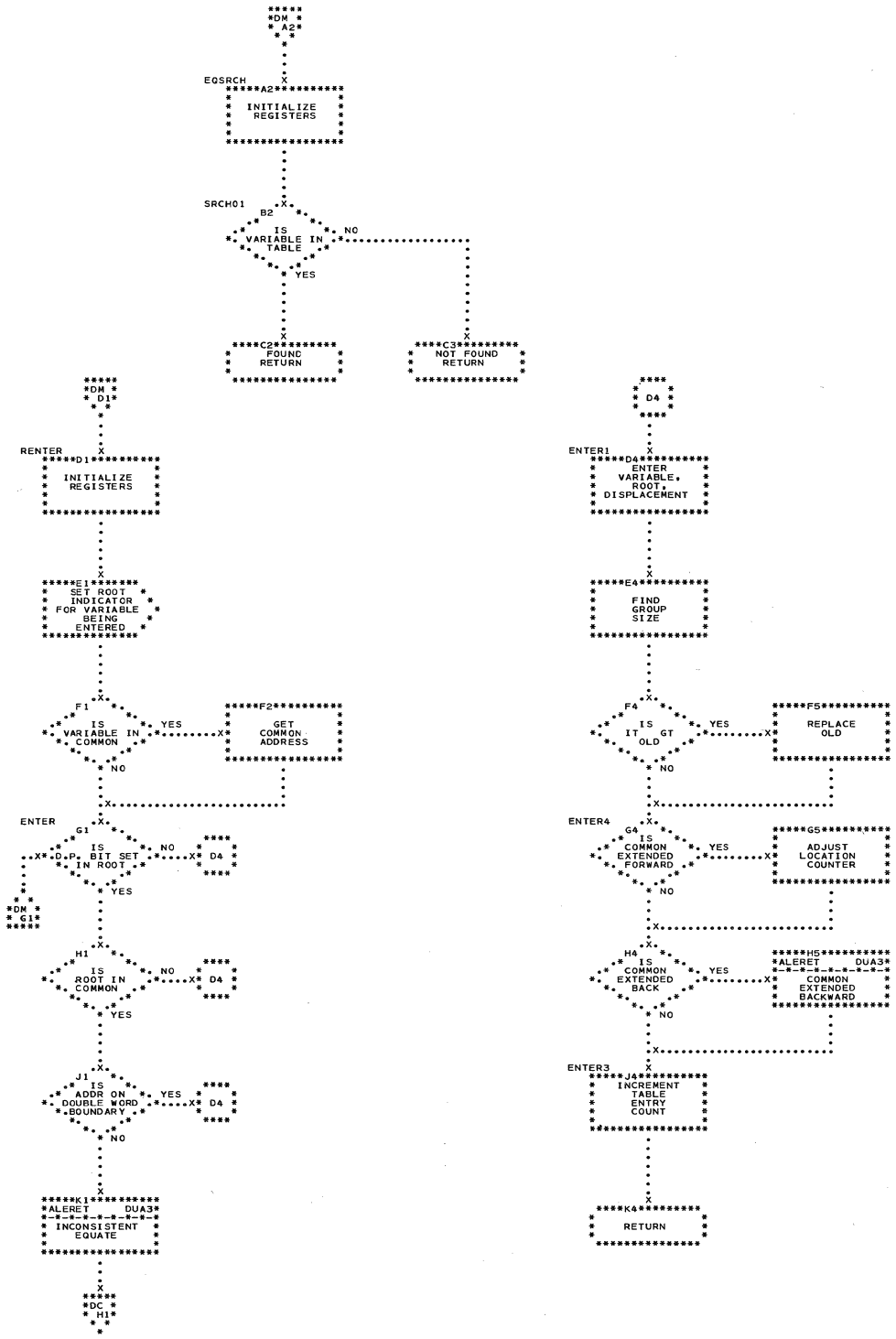


Chart DM. Subroutines EQSRCH, RENTER/ENTER

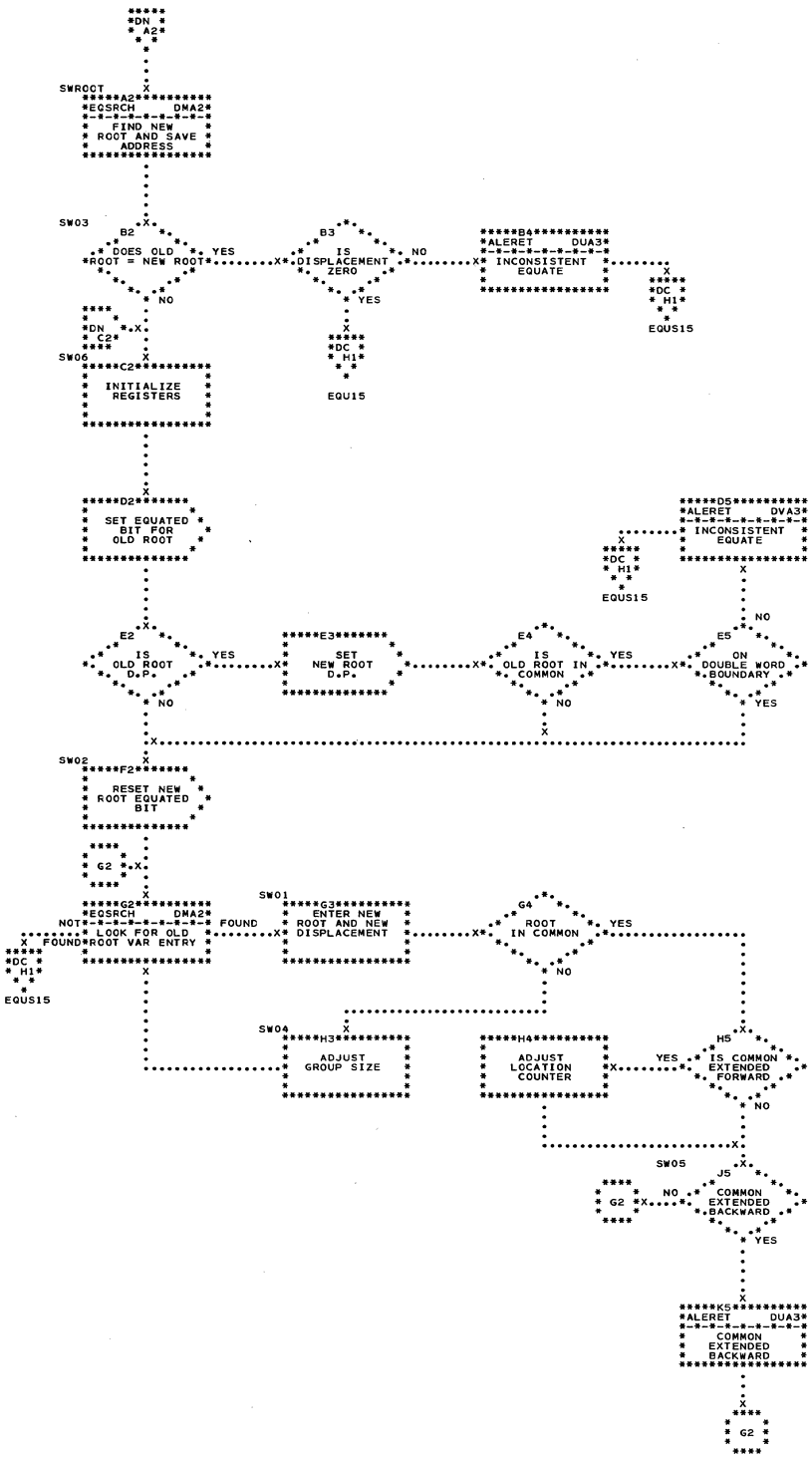
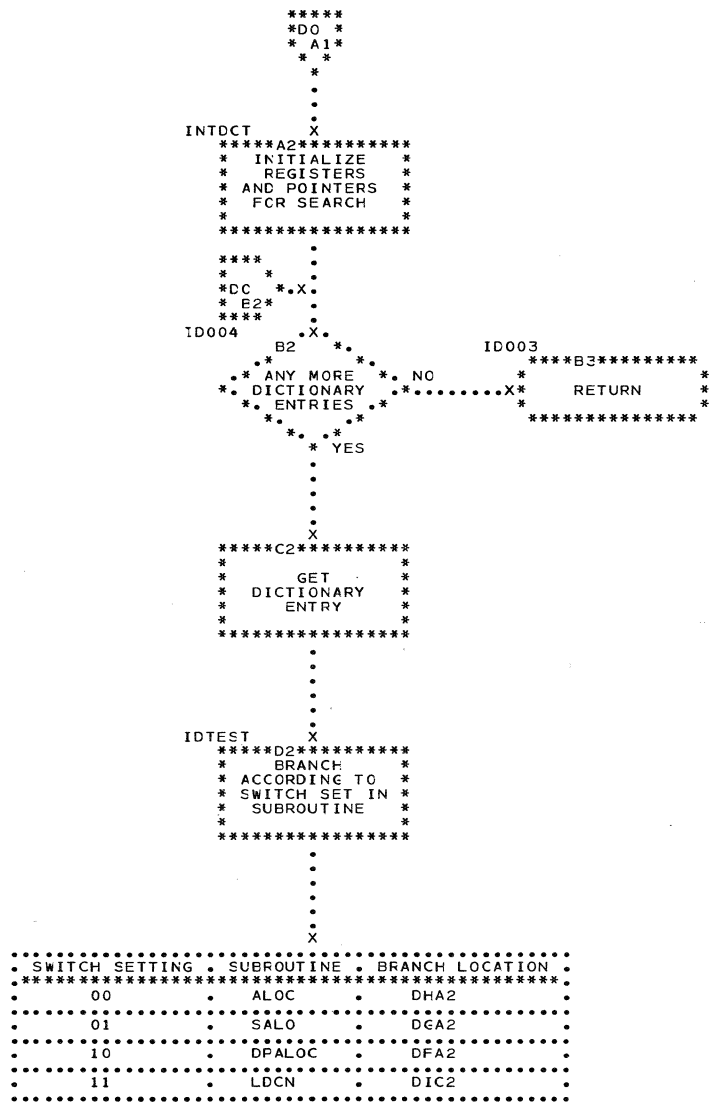


Chart DN. Subroutine SWROOT



SWITCH SETTING	SUBROUTINE	BRANCH LOCATION
00	ALOC	DHA2
01	SALO	DGA2
10	DPALOC	DFA2
11	LDCN	DIC2

Chart DO. Subroutine INTDCT

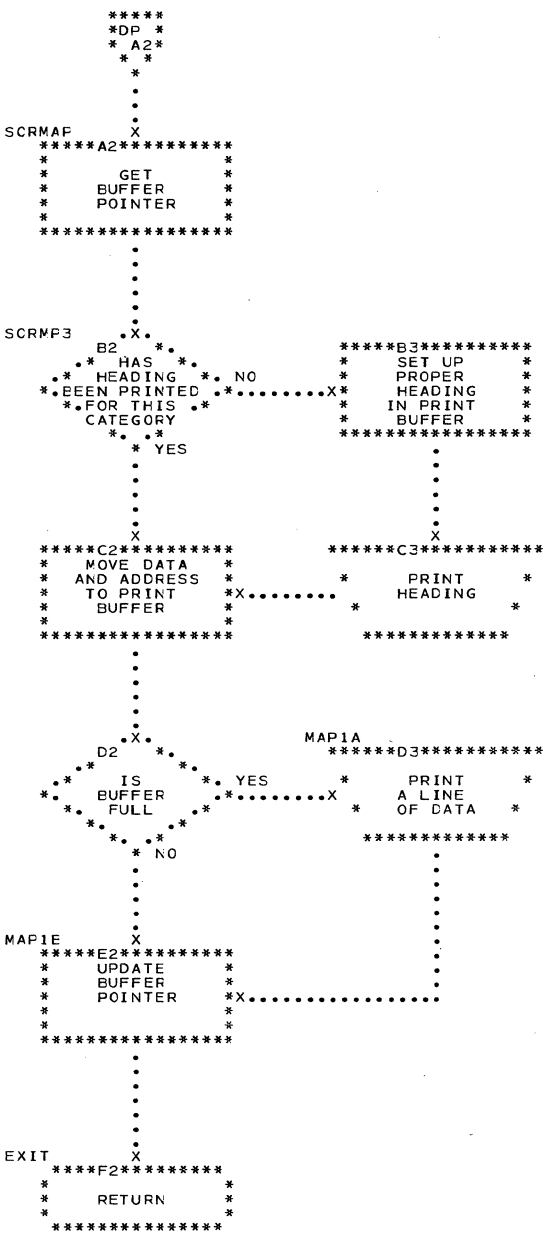


Chart DP. Subroutine SORSYM

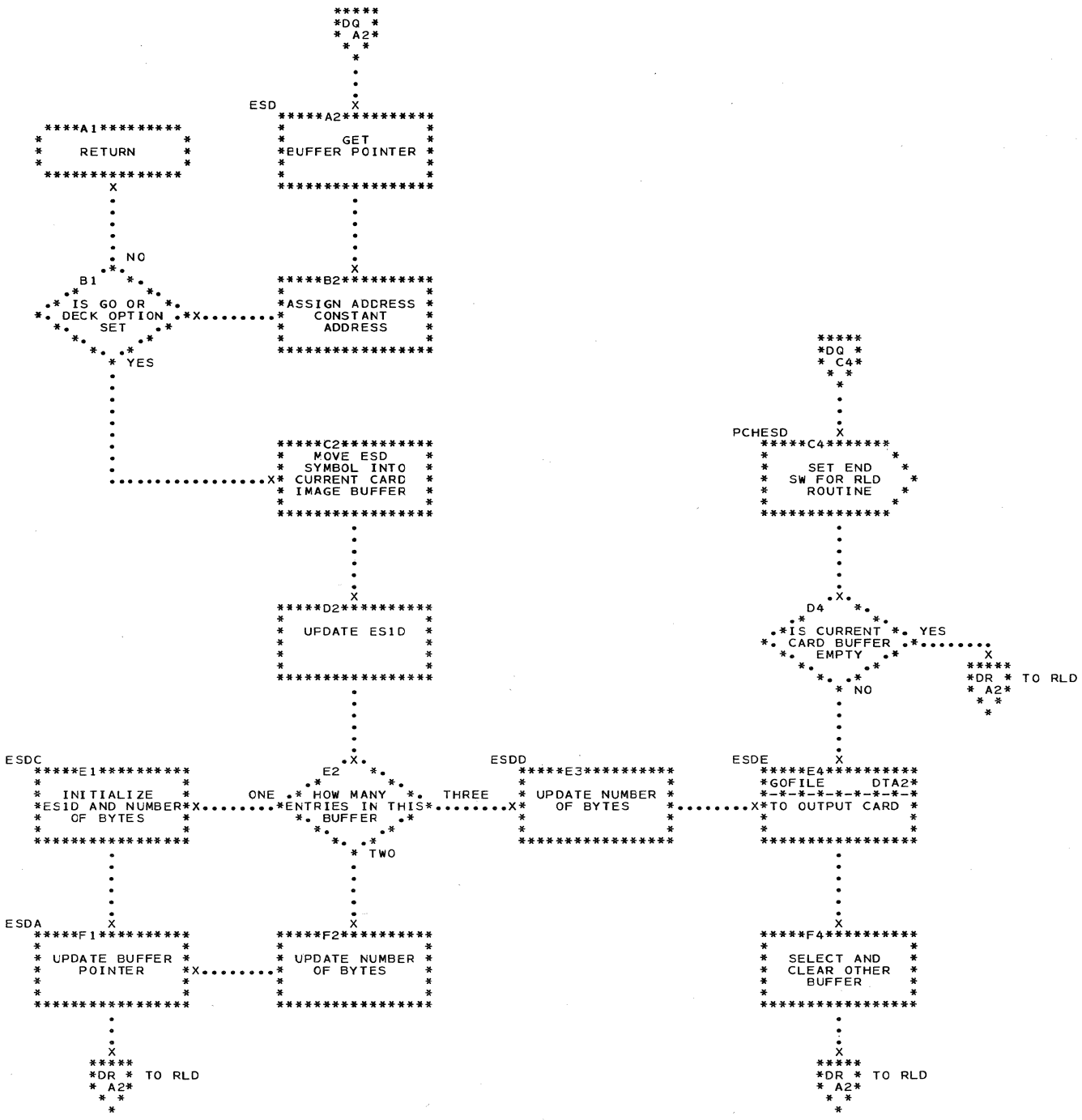


Chart DQ. Subroutine ESD

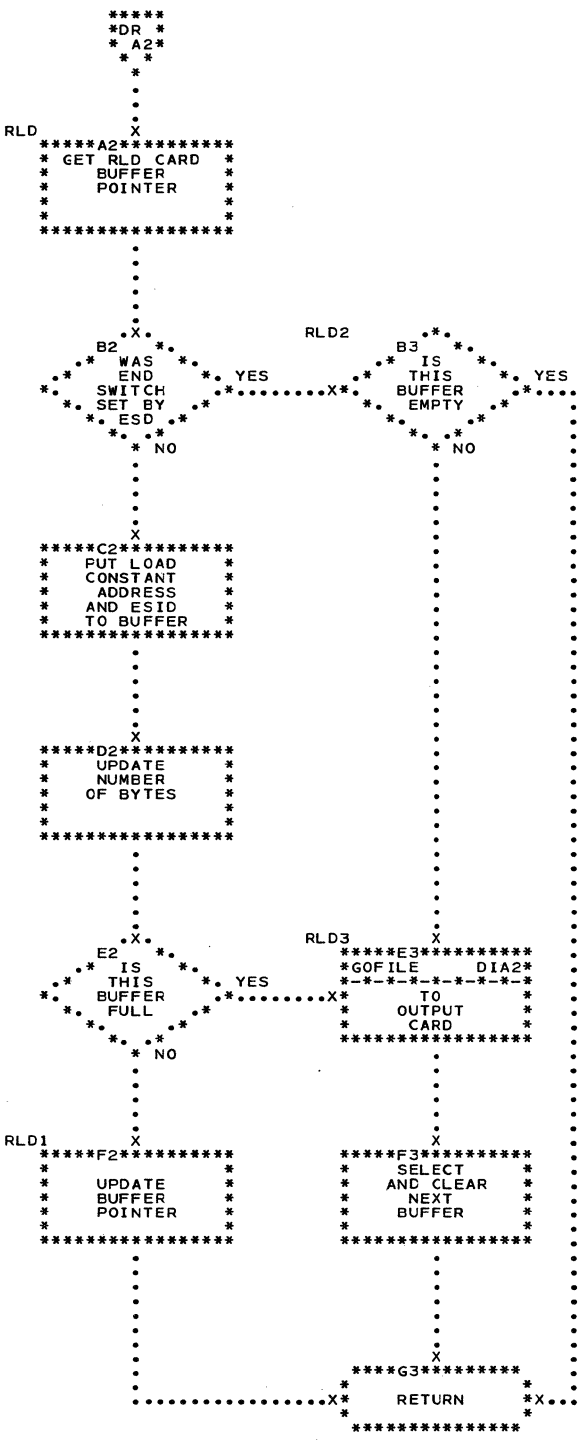


Chart DR. Subroutine RLD

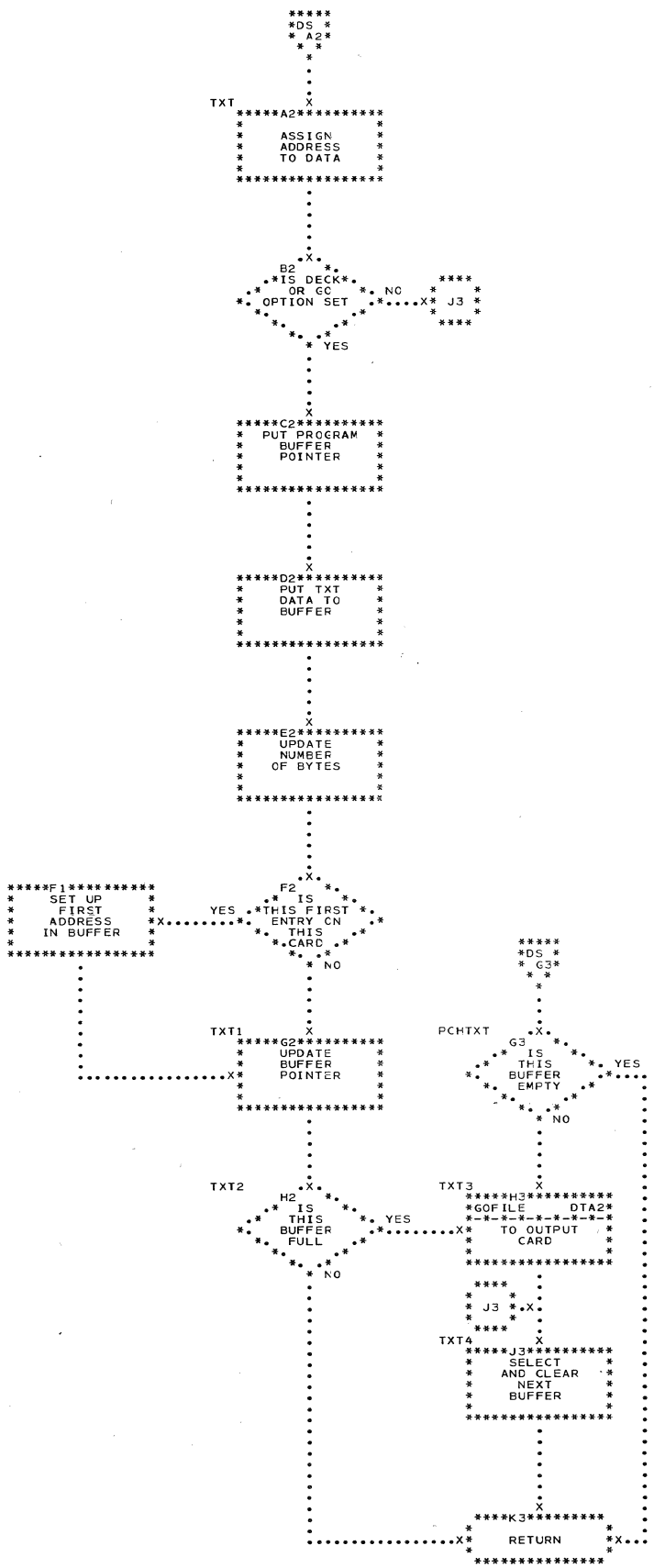


Chart DS. Subroutine TXT

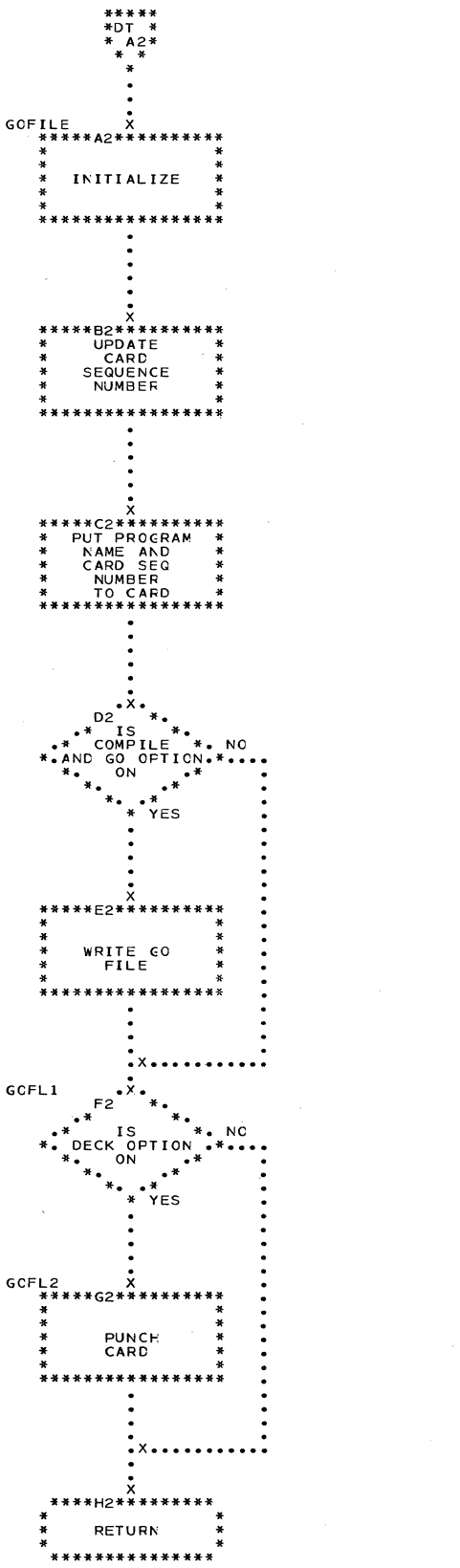


Chart DT. Subroutine GOFILE

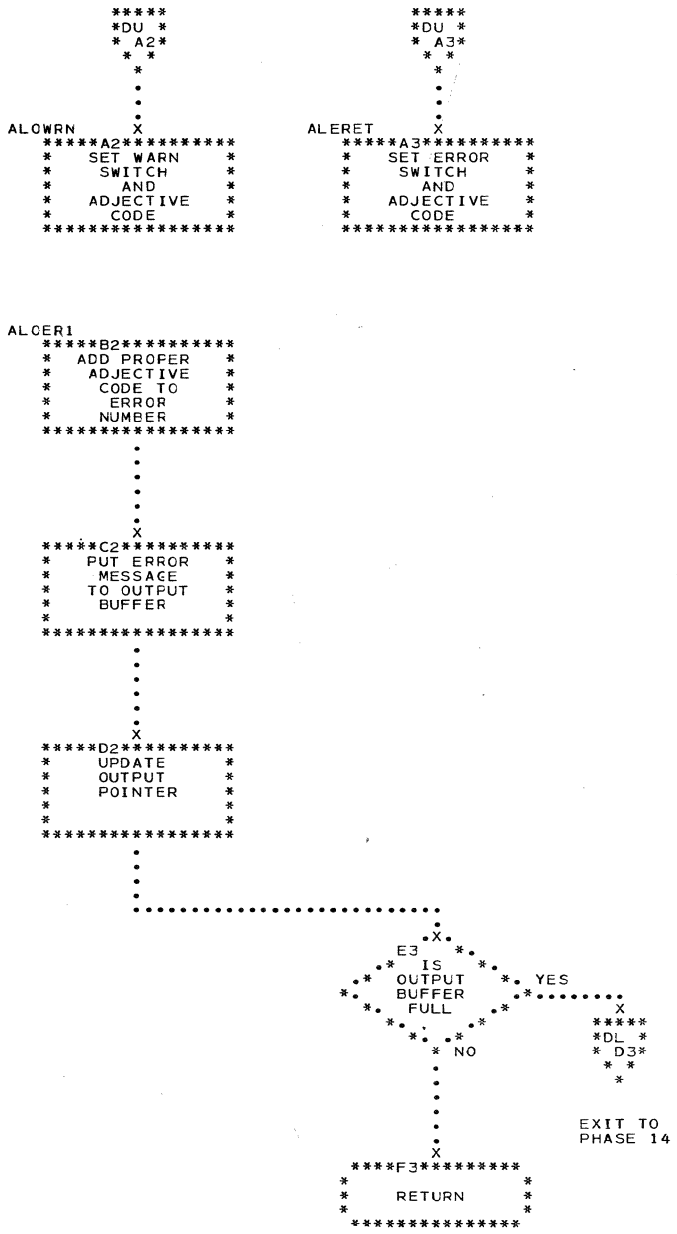


Chart DU. Subroutine ALERT/ALOWRN

Phase 14 reads the intermediate text created by Phase 10 and replaces any dictionary pointers with information accessed from the dictionary. Phase 14 converts intermediate text for FORMAT statements to an internal code that is used by IBCOM to place input and output records into a format specified for the object program.

Chart 05, the Phase 14 Overall Logic Diagram, indicates the entrance to and exit from Phase 14 and is a guide to the overall functions of the phase.

The adjective code of the first intermediate text entry for a statement indicates the type of statement and which subroutine is to process intermediate text entries for the particular statement.

All statements except END, FORMAT, READ/WRITE, and arithmetic statement function definition statements are processed similarly by Phase 14.

When Phase 14 reaches the intermediate text entry for the END statement, it puts that entry on the intermediate text output tape and then continues to access text words, putting out any error entries following the END statement.

Upon encountering intermediate text for a FORMAT statement, Phase 14 converts that text to an internal code used by IBCOM at object time. If requested, the code is written on the GO tape and/or punched on a text card. The code consists of a 1-byte FORMAT adjective code and a 1-byte field containing a binary number associated with the FORMAT adjective code.

When Phase 14 processes READ/WRITE statements, it checks for any implied DOs, inserts implied DO adjective codes in the intermediate text, and puts the text entries for the implied DOs into a new format.

When an arithmetic statement function definition is encountered in Phase 14, a unique number is assigned to the arithmetic statement function. This number is used to identify the statement function in later phases.

In the intermediate text entries for the FORTRAN statements, other than the END and FORMAT statement, dictionary pointers are replaced. If the pointer refers to an entry for a variable, constant, array, or external function, the address assigned to it and placed in the chain field by Phase 12 is inserted in the intermediate text entry. If the pointer refers to a data set reference number, that number is inserted in the intermediate text entry in place of the pointer. If the pointer refers to an arithmetic statement function, the arithmetic statement function number replaces the dictionary pointer.

As the entries are updated, they are written on the intermediate text output tape for Phase 14.

READ/WRITE STATEMENTS

The READ/WRITE text entries are scanned for 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 entries 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 entries for READ/WRITE statements are rearranged so later phases of the compiler may process them.

The implied DO variable and parameters are placed ahead of the subscripted variable. The entries for the offset, dimension pointer and the subscript pointer are placed ahead of the entry for the variable itself. If an implied DO:

```
(A(I),I=1,10)
```

is processed by Phase 10, the output has the format shown in Figure 43.

Adjective Code	Mode/Type	Pointer Field
(	real subscripted variable	p(A)
SAOP	00	Offset
p(dimension)		p(subscript)
,	integer variable	p(I)
=	immediate DO parameter	1
,	immediate DO parameter	10
)	00	0000

Figure 43. Implied DO Text Input to Phase 14

After Phase 14 has processed the implied DO, the intermediate text output takes the format shown in Figure 44.

Adjective Code	Mode/Type	Pointer Field
implied DO	00	0000
,	integer variable	address (I)
=	immediate DO parameter	1
,	immediate DO parameter	10
,	immediate DO parameter	1
begin I/O	00	0000
SAOP	00	Offset
p(dimension)		p(subscript)
(	real subscripted variable	address (A)
end DO	00	0000

Figure 44. Implied DO Text Output from Phase 14

ARITHMETIC STATEMENT FUNCTION DEFINITIONS

Each arithmetic statement function (ASF) must be defined before it can be used in another statement. When Phase 14 encounters an ASF definition, a number is assigned to that function. A counter, reserved in the communications area, is initialized at zero and incremented by 1 each time an ASF number is assigned. The ASF number is placed in the chain field for the ASF dictionary entry.

When a statement that references the ASF is encountered, the dictionary entry for the ASF is accessed. The ASF number is placed in the pointer field of the intermediate text entry for the statement that references the ASF.

FORMAT STATEMENTS

FORMAT statements are converted to an internal code which is accessed and used by IBCOM to place input and output records into the format specified for the object program. The internal code is written on the GO tape and/or text cards, depending on the user's options.

STRUCTURE OF A FORMAT STATEMENT

A FORMAT statement is composed of a series of format specifications, establishing the format of the input and output records for a FORTRAN object program.

The field length for a FORMAT specification is the number of bytes reserved in the input or output record for the variable in the record. The field length is the number located immediately to the right of the format code. In the specification I6, the number 6 represents the field length. Six positions are reserved in the record for any variable using this specification.

The decimal length is the number of bytes reserved for decimal places within the field. In a format specification using a decimal length, the decimal length is the number following the decimal point. The point is located immediately to the right of the field length. In the specification F10.2, the number 2 represents the decimal length and the number 10 represents the field length. The field length must be large enough to contain the number of decimal places, a decimal point, a sign, and an exponent if there is one for the specification.

The field count represents the number of times a conversion is to be repeated for an I/O list. In a format specification using a field count, the field count is the number located immediately to the left of the format code. In the specification 5F10.2, the number 5 represents the field count. This means that the conversion F10.2 will be repeated 5 times in an input/output record.

Format specifications may be repeated by enclosing them in parentheses. The format specifications within the parentheses is called a group. In the FORMAT statement,

```
10 FORMAT(I6,F12.5,(F10.2,I4))
```

(F10.2,I4) is a group. After the first two fields are processed, the variables following in the I/O list for the input statements alternate with the specifications F10.2 and I4.

A group may be repeated any number of times by placing a number immediately preceding the left parenthesis that defines a group. This number is called the group count. In the FORMAT statement,

```
10 FORMAT(I6,F12.5,7(F10.2,I4),D9.3)
```

the number 7 represents the group count. The group (F10.2,I4) is repeated 7 times for the I/O list referencing this FORMAT statement.

The length of a record is determined by information in a FORMAT statement. In the statement,

```
20 FORMAT(F7.3,5X,4F15.7,'HOG')
```

75 bytes constitute the record length. In the FORMAT statement,

```
30 FORMAT(F20.4,4I10/6(I2,4X).5HSTEEL)
```

two records are represented. The two records are separated by the character /. The first record is 60 bytes long, and the second is 41 bytes long.

A user has the option of specifying a length (referred to as a specified length) for records through a control card. The specified length is compared to a record length that is computed by Phase 14 as it processes the FORMAT statement. If the record length exceeds the specified length, a warning message is issued.

FORMAT TEXT CARD

FORMAT statements are translated into an internal representation used by IBCOM to place input/output records into a format specified for the object program. The internal representation is a series of 1-byte hexadecimal numbers (two hexadecimal digits for each byte). A byte may contain a FORMAT adjective code, which indicates either the format conversion (H, I, F, P, X, etc.) to IBCOM, or a group or field count, or the end of a FORMAT statement. The byte may also contain a number that represents a field count, field length, group count, or decimal length. There are four ways bytes may be entered into a text card.

Adjective Code and Number

An adjective code followed by a number is entered for format specification P, I, T, A, and X, and for entries made to indicate a field or group count. The number entered along with the correct adjective code for each specification is as follows:

1. P specification: the scale factor is entered as the number.
2. I and A specifications: the field length is entered as the number.
3. T specification: the record position is entered as the number.
4. X specification: the number of blanks to be inserted on output or the number of characters to be skipped on input is entered as the number.
5. Field count: the actual field count is entered as the number.
6. Group count: the actual group count is entered as the number.

An example that illustrates the preceding text card entries is the group 5(I2,I10). The text card entry for that group is 04051002100A. The entry is comprised of three parts: 0405 for the group count; 1002 for I2; and 100A for I10. In each case, the first two hexadecimal digits represent the adjective code, while the last two represent the associated number.

Adjective Code

Entries consisting of only the adjective code are made 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

An adjective code followed by a field length and a decimal length are entered in the format text cards for D, E, or F specifications. For example, the specification F9.3 is entered in a text card as 0A0903 where 0A is the adjective code, 09 is the field length, and 03 is the decimal length.

If the format specification +2PF9.3 is entered in a FORMAT statement, the code is entered in the text card as 08020A0903. The scale factor is denoted by 0802.

Adjective Code, Field Length, and Literal

An adjective code followed by a field length and a literal is entered in a text card for H and quote specifications. The specification 5HTIMER is entered as 1A05TIMER in the text card. The same entry is generated for the specification 'TIMER'.

SUBROUTINES

A section of Phase 14 is devoted to processing FORMAT statements, exclusively. The other subroutines in Phase 14 process all statements, except FORMAT statements.

Subroutine PRESCN gets the first intermediate text word for a statement and passes control to the Phase 14 subroutine for that statement (Chart EA). The adjective code subroutines (Chart EB) process all statements except READ/WRITE, DO, and FORMAT. Subroutine PINOUT, INOUT, MSG/MSGMEM, CEM/RDPOTA (Chart EC) are utility subroutines used by Phase 14.

Subroutines ERROR/WARNING, UNITCK/UNIT1 (Chart ED) are closed subroutines used by Phase 14 subroutines. Subroutine ERROR/WARNING processes error or warning conditions; subroutine UNITCK/UNIT1 processes symbols, used to represent data set reference numbers.

Subroutines PUTFTX, ININ/GET, GOFILE (Chart EE) are used by all Phase 14 subroutines to initiate I/O operations for the phase. Subroutine DO, CKENDO (Chart EF) are used to process DO statements, and to determine if a statement has invalidly ended a DO loop. Subroutine READ/WRITE (Chart EG) processes READ and WRITE statements in Phase 14.

The subroutines in charts EH through EO are used exclusively to process FORMAT statements. Chart 21 illustrates the overall logic of FORMAT processing. Subroutine FORMAT (Chart EH) initializes processing for each FORMAT statement and the format specifications within a statement. Subroutine D/E/F/I/A (Chart EI) processes D, E, F, I, and A specifications. Subroutines QUOTE/H,X (Chart EJ) process format specifications for literal, H, X specifications. Subroutines +/-/P, BLANKZ, FILLEG, FCOMMA (Chart EK) process P specifications and blanks, commas, and illegal delimiters found in a FORMAT statement. Subroutines LPAREN, RPAREN (Chart EL) process any parentheses found in a FORMAT statement. Subroutines T, FSLASH process the T specifications and any slash found in the statement. Subroutines LINETH, LINECK FLDCNT, NOFDCT (Chart EN) perform operations concerning the record length and field counts for a FORMAT statement. Subroutines GETWD-A, INTCON (Chart EO) are used to scan FORMAT statements and convert any integers in the statement to binary number.

Subroutine PRESCN: Chart EA

Subroutine PRESCAN performs initialization for Phase 14. PRESCN checks the adjective code of the first intermediate text entry for each statement and calls the correct Phase 14 subroutine to process intermediate text for that statement.

ENTRANCES: Subroutine PRESCAN is entered from:

1. The FORTRAN System Director, after Phase 14 has been read into main storage, for initialization.
2. Subroutine MSG/MSGMEM, after entries are moved to the intermediate text output buffers.
3. Subroutine LABEL DEF, after a statement number has been entered in the intermediate text output buffers.
4. Subroutine LPAREN, when the closing left parenthesis for a FORMAT statement is recognized.

OPERATION: When PRESCN is entered from the FORTRAN System Director, PRESCN begins to process entries by setting pointers to the beginning and end of the intermediate text input and output buffers.

Subroutine PRESCN is entered only for the first entry in the intermediate text for a FORTRAN source statement. The adjective code of this first entry indicates the type of source statement that caused Phase 10 to generate the following series of intermediate text entries. With the adjec-

tive code, PRESCN can index to the correct position in a branch table. The address in the branch table directs the processing of Phase 14 to the proper subroutine.

The adjective code indicates if the statement is a keyword statement (and what kind of keyword), an arithmetic statement, or the definition of an arithmetic statement function. It also indicates if there is an error or warning entry in the intermediate text.

EXIT: Subroutine PRESCN exits to the subroutine indicated by the adjective code.

Adjective Code Subroutines: Chart EB

The adjective code subroutines process all intermediate text entries in Phase 14 except the READ/WRITE, DO, and FORMAT statement entries. Several subroutines process intermediate text in the adjective code subroutines. Subroutine PRESCN determines the type of intermediate text entry to be processed and gives control to the correct adjective code subroutine.

ENTRANCE: The adjective code subroutines are entered from subroutine PRESCN after it determines the current adjective code. (The first adjective code entered for a statement in the intermediate text indicates the type of statement to be processed.)

OPERATION: A series of text entries are made for each source statement. Separate adjective code subroutines control the processing of source statements and intermediate text error entries. The various subroutines and their operation are discussed in the following paragraphs according to the statements they process.

SUBROUTINE/FUNCTION Statements: Subroutine SUBFUN controls the processing for FUNCTION and SUBROUTINE statement entries in intermediate text. SUBFUN replaces any dictionary pointers for arguments passed to the subprogram with the address assigned to those arguments in Phase 12.

In Phase 10, when the FUNCTION or SUBROUTINE statement was translated into intermediate text, the mode/type code for the arguments was not entered in the intermediate text. Mode/type for the arguments was established as the statements in the subprogram were processed by Phase 10. In Phase 14, the dictionary entry for each parameter is accessed and the mode/type code is inserted into the intermediate text.

Computed GO TO, GO TO, CALL, IF, Arithmetic Statements: Subroutine PASSON processes intermediate text entries for these statements. It scans the intermediate text until it reaches an end mark entry, and replaces each dictionary pointer encountered during the scan, with the assigned address for that symbol.

BACKSPACE, REWIND, END FILE Statements: Subroutine BSPREF processes the intermediate text entries for the BACKSPACE, REWIND, and END FILE statements and checks if a valid symbol identifies the data set reference number. Subroutine BSPREF then passes control to subroutine PASSON to search for the end mark entry.

Arithmetic Statement Function Definitions: Subroutine ASF processes intermediate text entries for arithmetic statement function definitions. A unique number, in sequence from 01, is assigned to each arithmetic statement function in the program.

When subroutine ASF is entered, the arithmetic statement function count is incremented by 1, and the count is moved to the dictionary entry for the name of the function. The intermediate text dictionary pointer is also replaced by the count. Subroutine ASF then passes control to subroutine PASSON to replace all pointers with actual addresses and search for the end mark entry.

Error/Warning Intermediate Text Entries: Subroutine ERWNEM processes intermediate text entries for errors, warnings, and end marks. These entries are not changed by Phase 14, but are moved from the intermediate text input buffer directly to the intermediate text output buffer.

CONTINUE Statements: Subroutine SKIP processes CONTINUE statements. The subroutine does not change the intermediate text entry; it moves the entry from the input to the output buffer.

RETURN Statements: Subroutine RETURN processes RETURN statements. RETURN calls subroutine CKENDO to determine if a RETURN statement invalidly ends a DO loop. If it does not, a check is made to determine if the RETURN statement appears in a main program. If so, the adjective code is changed to STOP. The intermediate text entry is moved to the output buffer.

PAUSE Statements: Subroutine PAUSE processes the PAUSE statements. Subroutine PAUSE calls subroutine CKENDO to test for an illegal end DO.

There are three text entries made for an errorless PAUSE statement:

1. An adjective code for PAUSE.
2. The number identifying the PAUSE.
3. The end mark entry.

The first entry is written on the intermediate text output tape. Subroutine SKIP places the second entry on that tape; subroutine MSG/MSGMEM, the end mark entry.

STOP Statements: Subroutine STOP controls the processing of STOP statements. Subroutine STOP calls subroutine CKENDO to test for an illegal end DO. There are three text entries made for the STOP statement:

1. An adjective code for STOP with zero entries for the mode/type and dictionary pointer fields.
2. Zero entries in the adjective and mode/type fields and the number identifying the STOP in the pointer field.
3. The end mark entry.

The first entry is moved unchanged from the input to the output buffers. Subroutine SKIP moves the second entry unchanged to the output buffers; subroutine MSG/MSGMEM, the end mark entry.

Statement Number Definition Entries: Statement number definitions are processed by subroutine LABEL DEF. A statement number entry, other than for a FORMAT statement number, is moved unchanged from the input buffer to the output buffer. A warning is issued for a FORMAT statement number which is not referenced. If the FORMAT statement ends a DO loop, an error is detected. If neither an error nor warning is noted, the contents of the location counter are inserted in the chain address field for the FORMAT statement number. (The location counter is incremented when the FORMAT statement is processed, because a FORMAT statement follows the statement number entry in the intermediate test.) The statement number entry is moved unchanged from the input to the output buffer.

END Statement: Subroutine END processes the intermediate text entries for the END statement by moving the unchanged intermediate text word from the input to the output buffer. If other text entries follow the END statement entry, those text words are also moved to the output buffer. Subroutine END then writes an end of data set on the intermediate text output tape and rewinds the input and output tapes.

Subroutine END calls the FORTRAN System Director to read Phase 15 into main storage.

EXITS: The adjective code subroutine exit to:

1. Subroutine MSG/MSGMEM to process the end mark entry for the adjective code subroutines.
2. Subroutine PRESCAN to process the series of intermediate text entries for the next statement.
3. FORTRAN System Director to read Phase 15 after the entry for the END statement has been processed.

SUBROUTINES CALLED: During execution the adjective code subroutines reference the following subroutines:

1. RDPOTA to replace a dictionary pointer by the assigned address.
2. CKENDO to test for an illegal end DO.
3. UNITCK/UNIT1 to check for the validity of a symbol referencing a data set.
4. PINOUT to move an input intermediate text word to an output buffer.
5. ERROR/WARNING if an error or warning condition is encountered.

Subroutines PINOUT, INOUT, MSG/MSGMEM, CEM/RDPOTA: Chart EC

Subroutine PINOUT

Subroutine PINOUT moves an intermediate text word from an input buffer to an output buffer. It increments both the input and output buffer pointers for the next intermediate text word.

ENTRANCE: Subroutine PINOUT is entered from either the adjective code subroutines, subroutine READ/WRITE, or subroutine DO after a text word is processed, or subroutine FORMAT.

OPERATION: Intermediate text words are processed by Phase 14 while they are in the intermediate text input buffers. Subroutine PINOUT moves the entire text word to an intermediate text output buffer after the word has been processed. PINOUT then calls subroutines ININ and INOUT to increment the input and output buffer pointers.

EXIT: Subroutine PINOUT exits to the calling subroutine.

SUBROUTINES CALLED: During execution subroutine PINOUT references the following subroutines:

1. ININ to increment the input buffer pointer.
2. INOUT to increment the output buffer pointer.

Subroutine INOUT

Subroutine INOUT increments the output buffer pointer to the next available position so Phase 14 may insert the next intermediate text word.

ENTRANCE: Subroutine INOUT is referenced by subroutine PINOUT to increment the output buffer pointer.

OPERATION: Subroutine INOUT increments the intermediate text output buffer pointer by four bytes each time it is referenced. INOUT then checks the buffer. If it is full, the FORTRAN System Director is called to write that input buffer on the output tape. The next buffer in the double-buffer system is selected so the next series of intermediate text words can be read as the first buffer is being written on the output tape.

EXIT: Subroutine INOUT exits to the subroutine that called it.

SUBROUTINES REFERENCED: Subroutine INOUT references the FORTRAN System Director to initiate writing a buffer on the output tape.

Subroutine MSG/MSGMEM

Subroutine MSG/MSGMEM searches for an end mark entry in the intermediate text. This entry, when found, is moved to the intermediate text output buffer. If Phase 14 has noted further error or warning conditions for that statement, subroutine MSG/MSGMEM inserts those error/warning entry numbers in the intermediate text output buffer.

ENTRANCE: Subroutine MSG/MSGMEM is entered by the adjective code subroutines and the FORMAT subroutines when an end mark entry should be the next intermediate text entry. Subroutine MSG/MSGMEM is entered by subroutines CEM and READ/WRITE when an end mark entry is detected. Subroutine MSG/MSGMEM is entered by the FORMAT subroutines where an error is detected.

OPERATION: Subroutine MSG/MSGMEM is entered at two points. The first entry point, MSG, is used when a subroutine has found an end mark entry. The second entry point, MSGMEM, is entered when a subroutine has processed intermediate text entries for a FORTRAN statement and expects to find an end mark entry next in the intermediate text buffer. MSGMEM is given control to search the intermediate text buffer until it finds the end mark entry, ignoring all other text entries.

Subroutine MSG/MSGMEM puts the intermediate text word for the end mark entry into an output buffer, and checks an indicator to determine if Phase 14 has found any error/warning conditions in that series of intermediate text entries. If any were detected, MSG/MSGMEM puts the corresponding message numbers in the output buffer.

EXITS: Subroutine MSG/MSGMEM exits to subroutine PRESCN to begin processing the intermediate text entries for the next statement.

SUBROUTINES CALLED: During execution subroutine MSG/MSGMEM references subroutines:

1. PINOUT to put out the intermediate text entry for the end mark.
2. INOUT to put out error/warning entries.

Subroutine CEM/RDPOTA

Subroutine CEM/RDPOTA is used to process intermediate text entries for arithmetic, BACKSPACE, REWIND, END FILE, computed GO TO, GO TO, DO, CALL, IF statements, and arithmetic statement function definitions. A portion of CEM/RDPOTA is used to insert the arithmetic statement function number in the intermediate text and replace the PAUSE library function number with its assigned address constant.

ENTRANCE: Subroutine CEM/RDPOTA is entered from subroutines PASSON, PAUSE, and ASF to replace dictionary pointers.

OPERATION: Subroutine CEM/RDPOTA is entered at two points. The first, CEM, checks the intermediate text word being processed. If it is an end mark entry, control is passed to subroutine MSG/MSGMEM to put out the entry for the end mark. If not, subroutine CEM/RDPOTA checks the mode/type code in the intermediate text entry to determine the item referenced in the pointer field.

If the pointer field points to a data set reference number dictionary entry, the pointer is replaced by the reference number. If it does not, the adjective code in the entry is checked for a subscript expression (adjective code of SAOP). If the code is SAOP, this entry and the next entry are moved from the input to the output buffer. (These two entries contain the offset for the subscript expression and pointers to dimension and subscript information.) Control is then returned to the subroutine that called subroutine CEM/RDPOTA.

If the adjective code was not SAOP and the mode/type code is an immediate DO parameter, the entry is moved from the input to the output buffer. The subroutine exits to the subroutine that called it.

If the mode/type code is not an immediate DO parameter, the mode/type code is checked for a dictionary pointer of any kind. Any dictionary pointer is replaced by the contents of the chain field. (The chain field at this time contains either an address assigned by Phase 12 or an arithmetic statement function number assigned earlier by Phase 14.) The word is then moved from the input to the output buffer.

This last portion of coding in subroutine CEM/RDPOTA may be entered independently by other Phase 14 subroutines, if a dictionary pointer is to be replaced by the contents of the chain field. The portion of coding is called RDPOTA.

EXIT: Subroutine CEM/RDPOTA exits either to subroutine MSG/MSGMEM if it detects an end mark entry, or to the subroutine that called it.

SUBROUTINES CALLED: During execution subroutine CEM/RDPOTA references subroutine:

1. UNITCK/UNIT1 to replace a dictionary pointer with a data set reference number.
2. PINOUT to move an intermediate text word from input to output buffers.

Subroutines ERROR/WARNING, UNITCK/UNIT1: Chart ED

Subroutine ERROR/WARNING

Subroutine ERROR/WARNING generates intermediate text words for errors and warnings detected by Phase 14.

ENTRANCE: Subroutine ERROR/WARNING can be entered from every Phase 14 subroutine.

OPERATION: Subroutine ERROR/WARNING has two entry points. One entry point, WARNING, is used by a Phase 14 subroutine when it detects a warning condition. (The difference between an error and a warning condition was discussed in subroutine ERROR/WARNING of Phase 10.) When subroutine ERROR/WARNING is entered for a warning condition, the warning number is placed by the calling subroutine in a general register. Subroutine ERROR/WARNING accesses the warning number and composes an intermediate text word for the warning.

A second entry point is used by a Phase 14 subroutine when it detects an error condition. The calling sequence for errors is the same as the calling sequence used to enter subroutine ERROR/WARNING in Phase 10. A branch table is used to calculate the error number; the intermediate text word for an error is composed.

Both error and warning text words are saved in main storage after they are composed. When the end mark entry for the current statement is encountered, the text words (maximum of four) are accessed and inserted in the intermediate text output buffer by subroutine MSG/MSGMEM. If more than four errors and/or warning text words are detected by Phase 14, the remaining intermediate text entries for the statement are ignored. Subroutine ERROR/WARNING passes control to subroutine MSG/MSGMEM to find the end mark entry for the statement and put out the error/warning text words.

EXIT: Subroutine ERROR/WARNING exits to the subroutine that called it or to subroutine MSG/MSGMEM when more than four error or warning text words are detected by Phase 14 for a single FORTRAN statement.

Subroutine UNITCK/UNIT1

Subroutine UNITCK/UNIT1 checks the validity of a symbol used to reference a data set reference number. UNITCK/UNIT1 replaces the dictionary pointer with the address assigned to the data set reference number.

ENTRANCES: Subroutine UNITCK/UNIT1 is entered by subroutines READ/WRITE and BSPREF to determine if the symbol used to reference a data set reference number is valid. The subroutine is also entered by subroutine CEM/RDPOTA to replace a dictionary pointer by a data set reference number.

OPERATION: Subroutine UNITCK/UNIT1 is entered at two points. One entry point is used for checking the data set representa-

tion in BACKSPACE, REWIND, END FILE, READ, or WRITE statements. A data set can be represented by a data set reference number, an integer variable, or a dummy integer variable. If it is represented by a data set reference number, the dictionary pointer in the intermediate text word is replaced by the data set reference number. If it is represented by an integer variable or a dummy integer variable, the dictionary pointer is replaced by the address assigned to the variable in Phase 12. Any other representation of a data set is illegal and an error condition is noted.

A second entry point is used by subroutine CEM/RDPOTA if it determines from the mode/type entry that the pointer refers to a data set reference number. Subroutine UNITCK/UNIT1 replaces the dictionary pointer in text with the data set reference number in the dictionary entry.

The intermediate text word is moved from the input to the output buffer.

EXITS: Subroutine UNITCK/UNIT1 returns control to the subroutine that called it.

SUBROUTINES CALLED: During execution subroutine UNITCK/UNIT1 references subroutine PINOUT to output an intermediate text word, and subroutine RDPOTA to insert the assigned address of an integer variable or dummy integer variable in an intermediate text word.

Subroutines PUTFTX, ININ/GET, GOFILE: Chart EE

Subroutine PUTFTX

Subroutine PUTFTX enters the FORMAT adjective codes and FORMAT numbers for FORMAT specifications in a text card. If the DECK and/or the Compile and Go options are set and a buffer is filled, the card image is either punched in a card, and/or written on the GO tape.

ENTRANCE: Subroutine PUTFTX is entered by subroutines D/E/F/I/A, QUOTE/H, X, +/-/P, LPAREN, RPAREN, T, FSLASH, and FLDCNT.

OPERATION: The operation of subroutine PUTFTX is similar to the operation of subroutine TXT, Chart DS, in Phase 12.

EXIT: Subroutine PUTFTX exits to the subroutine that called it.

SUBROUTINES CALLED: During execution subroutine PUTFTX references subroutine GOFILE if a buffer is full.

Subroutine ININ/GET

Subroutine ININ/GET increments the input buffer pointer so the intermediate text entries may be processed. It also initiates the reading of the intermediate text input tape if the end of the buffer is reached, or a subroutine requests another record be read.

ENTRANCE: Subroutine ININ/GET is entered by subroutines PINOUT and MSG/MSGMEM to increment the input buffer pointer, and by subroutine GETWDA to read another record.

OPERATION: Subroutine ININ/GET calls the FSD to read an intermediate text record into a buffer. The buffers in the double-buffer system are switched to allow the second buffer to be filled with an intermediate text buffer.

EXIT: Subroutine ININ/GET exits to the subroutine that called it.

Subroutine GOFILE

Subroutine GOFILE punches a text card and/or writes on the GO tape, the card image which has been assembled by subroutine PUTFTX, if the DECK and/or Compile and Go options are specified.

ENTRANCE: Subroutine GOFILE is entered by subroutine PUTFTX if an output buffer is full.

OPERATION: The operation of subroutine GOFILE is similar to the operation of subroutine GOFILE, Chart DT, in Phase 12.

EXITS: Subroutine GOFILE exits to subroutine PUTFTX.

SUBROUTINE CALLED: During execution subroutine GOFILE references the FORTRAN System Director to punch text cards and/or write the card image on the GO tape.

Subroutines DO, CKENDO: Chart EF

Subroutine DO

Subroutine DO processes DO statements in Phase 14. It performs a diagnostic check on the DO variable and the DO parameter.

ENTRANCE: Subroutine DO is entered from subroutine PRESCN when PRESCN encounters a DO adjective code in the intermediate text.

OPERATION: Subroutine DO determines if a DO statement is used to end another DO. If it is, an error condition is noted. If an invalid end DO error does not occur, the DO adjective code and the statement number are entered in the intermediate text output buffers.

The mode/type field in the intermediate text for the next entry is checked to determine if the next entry is for an integer variable. If it is, the dictionary pointer in the text entry is replaced by the address assigned to the variable in Phase 12. The entry is then entered in one of the intermediate text output buffers. If the mode/type field does not indicate an integer variable, an error condition is noted.

If an error does not occur, the next text entry is checked. If the adjective code is not an equal sign, an error condition is noted. If the code is for an equal sign, the mode/type field is checked.

If the symbol referenced by the pointer field is an integer variable, the dictionary pointer for the entry is replaced by the address assigned the variable in Phase 12, and the text word is entered in the intermediate text output buffer. If the symbol is not an integer variable, but is an immediate DO parameter, the intermediate text word is moved to the output buffer. If the symbol is not an immediate DO parameter, an error condition is noted.

If the adjective code for the intermediate text array is a comma, subroutine DO repeats the same series of tests for the next DO parameter. For any other adjective code, however, subroutine DO passes control to subroutine MSG/MSGMEM.

EXIT: Subroutine DO exits to subroutine MSG/MSGMEM, when the DO statement is processed or an error is detected.

SUBROUTINES CALLED: During execution subroutine DO calls the following subroutines:

1. CKENDO to determine if a DO statement illegally ended another DO.
2. PINOUT to move intermediate text entries from an input to an output buffer.
3. RDPOTA to replace a dictionary pointer with an assigned address and move the entry to an output buffer.
4. ERROR/WARNING if an error is detected.

Subroutine CKENDO

Subroutine CKENDO determines if a statement has invalidly ended a DO loop.

ENTRANCE: Subroutine CKENDO is referenced by subroutines processing statements not permitted to end a DO loop. These are subroutines PAUSE/STOP, RETURN, FORMAT, and DO.

OPERATION: If a statement has a statement number, subroutine CKENDO checks the usage field of the statement number entry in the overflow table to determine if the statement ended a DO loop. If the statement ended a DO loop, an error condition is noted. If it does not, subroutine CKENDO returns control to the subroutine that called it.

EXIT: Subroutine CKENDO exits to the subroutine that called it.

SUBROUTINES CALLED: If an error is detected, subroutine CKENDO references subroutine ERROR/WARNING.

Subroutine READ/WRITE: Chart EG

Subroutine READ/WRITE processes intermediate text entries made in Phase 10 for READ and WRITE source statements.

ENTRANCE: Subroutine READ/WRITE is entered by subroutine PRESCAN if the adjective code for a statement represents either a READ or WRITE statement.

OPERATION: Subroutine READ/WRITE is divided into three sections for discussion purposes:

1. Section 1 (blocks EF01 through EF09): processes the FORMAT statement number and data set reference number in a READ/WRITE statement.
2. Section 2 (blocks EF09 through EF19): processes any implied DO in the I/O list of a READ/WRITE statement.
3. Section 3 (blocks EF20 through EF34): processes variables in the I/O list, excluding those variables used as subscript parameters.

Blocks EF01 through EF09: The first intermediate text word containing the READ or WRITE adjective code is moved to the intermediate text output buffer.

The symbol used to represent the data set reference number is checked for validity and moved to an output buffer. The FORMAT statement number, if present, is moved to the output buffer.

Absence of a FORMAT statement number signifies a binary operation and the next intermediate text word is moved to the output buffer. An end mark entry is then generated and moved to the output buffer. The end mark entry is made for Phase 20 to denote the end of the first text entries for the READ/WRITE statement. The I/O list, which is now treated as a separate statement in text, is moved to a save area. After processing the data set reference number and the FORMAT statement number, the statement pointer used to process the statement is at the beginning of the I/O list.

Blocks EF09 through EF19: Subroutine READ/WRITE determines if the symbol pointed at is a left parenthesis, indicating an implied DO. The recognition encounter of any symbol other than a left parenthesis causes control to be passed to Section 3 of this subroutine. However, if a left parenthesis is encountered, the value of the statement pointer is saved for subsequent processing and an implied DO adjective code is immediately inserted in the intermediate text output buffer. For reference purposes, the pointer is now referred to as the implied DO pointer. This pointer is used to scan the text entries between the outermost set of parentheses for any implied DOs.

A parentheses count is used to determine whether an implied DO is nested within another implied DO. Each recognition of a left parenthesis causes this count to be incremented by 1; each recognition of a right parenthesis causes the count to be decremented by 1.

If the parenthesis count is non-zero after a right parenthesis has caused it to be decremented, an implied DO is nested within another implied DO, and the implied DO pointer must be used to continue scanning the list until the parentheses count is zero. Using this technique, the implied DO which contains other implied DOs can be processed first. For example, in the statement:

```
READ (2,15) ((C (I,J) ,D (I,J) ,J=1,5) ,I=1,4)
```

the DO implied by I=1,4 is processed first.

When the parentheses count reaches zero, the implied DO pointer is decremented by four text words (16 bytes) to allow the DO variable to be moved to the output buffer. The DO parameters are then moved to the intermediate text output buffer. A SKIP adjective code is inserted in a save area to indicate that these text words have already been placed in the intermediate text output buffer. The processing for the outermost implied DO is now complete.

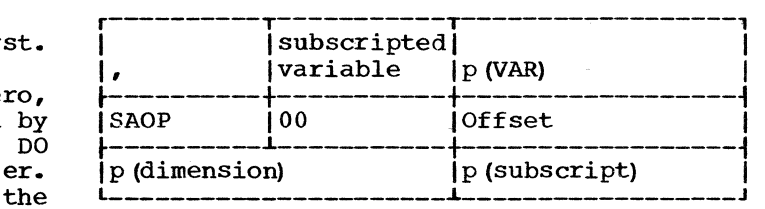
To resume I/O list processing, the pointer is restored to the value that was saved during the first encounter of the left parenthesis (indicating an implied DO). If there is a nested implied DO, the pointer again points at a left parenthesis. The process, just described for an implied DO, is repeated.

When the last nested implied DO is processed, the restored pointer points at the first symbol in the I/O list. To process the symbols in the list, control is passed to Section 3 of this subroutine.

Blocks EF20 through EF34: To indicate the beginning of an I/O list to later phases, a BEGIN I/O adjective code is placed in the intermediate text output buffer.

The dictionary pointer in the intermediate text word for the symbol is replaced by the address assigned to the symbol in Phase 12. The mode/type entry is checked for a valid entry in an I/O list. Variables or array names are the only legal entries in an I/O list. If the symbol is illegal, an error condition is noted, and the remainder of the I/O list is aborted from the compilation.

If the symbol is legal, READ/WRITE determines if the adjective code following the variable or array entry is a SAOP, denoting a subscripted variable. If it is SAOP, the intermediate text words containing the offset and the pointers to dimension and subscript information are moved to the intermediate text output buffers. Then the entry for the variable is inserted in the intermediate text. The order for subscripted variable entries in an I/O list is changed somewhat. For example, the entries made for a subscripted variable in Phase 10 follow this format:



In Phase 14 the format for subscripted variable entries is changed to:

SAOP	00	Offset
p (dimension)		p (subscript)
	subscripted variable	address (VAR)

If the next adjective code was not an SAOP, the intermediate text entry for the variable is inserted in the intermediate text output buffer.

A series of checks are made to determine what action subroutine READ/WRITE should take. If the adjective code denotes a comma and the mode/type field is not blank for the next entry, the entry is a variable (subscripted or non-subscripted) and subroutine READ/WRITE takes action to enter the variable in the output buffer. If the adjective code for the next entry is a skip (inserted in section 2 of this subroutine) the next four entries in the save area are the integer variable and parameters for an implied DO. (They were entered in the output buffer in section 2 of the READ/WRITE subroutine.) READ/WRITE then skips four text words and performs the same series of checks on the current text entry.

If the adjective code denotes neither a variable nor a skip, READ/WRITE checks if the adjective code is an end mark. If it is, the end I/O adjective code is inserted in the output buffer, because all entries in the I/O list have been processed. If the adjective code is not an end mark, READ/WRITE determines if it denotes an implied DO. If it does, control is given to section 2 to process the implied DO. If the adjective code is neither an end mark, variable, skip, nor an implied DO, an error condition is noted and the remainder of the I/O list is aborted.

EXIT: Subroutine READ/WRITE exits to the subroutine MSG/MSGMEM if:

1. An error condition is detected.
2. The end mark entry is encountered in the save area.

SUBROUTINES CALLED: During execution subroutine READ/WRITE references subroutines:

1. PINOUT to make entries to the output buffer.
2. UNITCK to check the validity of a data set reference number.
3. ERROR/WARNING if an error is detected.

Examples: The statement:

READ (2,10) A,B,C

generates the following intermediate text in Phase 10:

READ	0000	0000
(	data set reference number	2
,	statement number	10
)	real variable	p (A)
,	real variable	p (B)
,	real variable	p (C)
end mark	00	internal statement number

From these entries Phase 14 generates these intermediate text entries:

READ	00	0000
(	data set reference number	2
,	statement number	p (10)
end mark	00	0000
begin I/O	00	0000
,	real variable	address (A)
,	real variable	address (B)
,	real variable	address (C)
end I/O	00	0000
end mark	00	internal statement number

The statement:

```
WRITE (N,2) ((A (I,J) ,J=1,10) ,I=1,15) ,B
```

generates the following intermediate text in Phase 10:

WRITE	00	0000
(	integer variable	p (N)
,	statement number	p (2)
)	00	0000
(	00	0000
(	real subscripted variable	p (A)
SAOP	00	Offset
p (dimension)		p (subscript)
,	integer variable	p (J)
=	immediate DO parameter	1
,	immediate DO parameter	10
,	immediate DO parameter	1
)	00	0000
,	integer variable	p (I)
=	immediate DO parameter	1
,	immediate DO parameter	15
,	immediate DO parameter	1
)	00	0000
,	real variable	p (B)
end mark	00	internal statement number

From these entries Phase 14 generates these text entries:

WRITE	00	0000
(	integer variable	address (N)
,	statement number	p (2)
end mark	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 (dimension)		p (subscript)
(	real subscripted variable	address (A)
end DO	00	0000
end DO	00	0000
begin I/O	00	0000
,	real variable	address (B)
end I/O	00	0000
end mark	00	internal statement number

Phase 14 Format Overall Logic, Chart 21

Several subroutines (Charts EH through EO) are used to process FORMAT statements. The overall logic for this processing is shown in Chart 21.

The FORMAT subroutine is entered from subroutine PRESCN after PRESCN recognizes an adjective code for a FORMAT statement. Using the translate and test table for FORMAT statements, subroutine FORMAT retrieves the first FORMAT code for the statement and passes control to a specific subroutine. The functions of the specific subroutines are generally the same.

If a FORMAT code is a + or -, the scale factor and the P are retrieved and entered in a text card, and the next format code is fetched.

While retrieving the FORMAT code, subroutine FORMAT may find a field count for this code. If it does, the field count adjective code is entered in the text card along with the field count, itself; then, the adjective code for the format code is entered. A left parenthesis, encountered within the outside parentheses for the FORMAT statement, indicates a FORMAT group. If a group is indicated within a FORMAT statement, the series of FORMAT subroutines retrieves the FORMAT code for the next FORMAT specification and begins processing that specification.

If a group is not indicated, the FORMAT code is checked to see if it is defining literals (H or quote specifications). If the specification defines a literal, the literal is inserted in the text card, and the FORMAT subroutines then branch to compute the record length.

If a literal is not being defined, the FORMAT code is then checked to see whether this specification contains a field length. If it does not, the FORMAT subroutines branch to compute the record length. If a field length is part of this specification, the translate and test table is used to get the field length and the following delimiter. The field length is then entered in the text card.

The FORMAT code is checked again to see if the specification contains a decimal length. If it does not, the FORMAT subroutines branch to compute the decimal length. If the specification contains a decimal length, the translate and test table is used again to get the length and the following delimiter. The decimal length is entered in the text card.

Record length accumulators are then updated. There are three accumulators used to calculate record length: the record length, the leading length, and the tail length accumulators.

The record length accumulator is used exclusively if there is no attempt made in the FORMAT statement to define a group of format specifications. After each specification is processed, the field length is multiplied by the field count for this specification and added to the record length accumulator.

The leading length accumulator is used to accumulate the length of a group. If there are no slash or T FORMAT specifications, this accumulator is used to accumulate the sum of the field lengths in the group. After each specification in a group is processed, the field length is multiplied by the field count for this specification and added to the leading length accumulator. When the closing right parenthesis is found, the leading length accumulator is multiplied by the group count and added to the record length accumulator.

If a slash or T specification is found within a group, the record length accumulator is set equal to the number entered in the field length position. If the FORMAT code is a slash, the tail length accumulator is set to zero. If the FORMAT code is a T, the tail length accumulator is set to zero. After each specification following the T or slash in the group is processed, the field length is multiplied by the count for this specification and the result is added to the tail length accumulator. After the end of the group is sensed, the record length accumulator is set equal to the tail length accumulator.

After FORMAT specifications are checked, and their field lengths added to the accumulators, the contents of the record length accumulator are compared to the specified length entered by the user in a control card. If the record length is greater than the specified length, a warning is issued.

If the right parenthesis ending the FORMAT statement is sensed, control is passed to subroutine PRESCN to process another statement. If the ending right parenthesis is not sensed, the FORMAT subroutines return to the beginning of subrou-

tine FORMAT to begin processing another format specification.

Subroutine FORMAT: Chart EH

Subroutine FORMAT initializes the FORMAT subroutines to process a FORMAT statement. Subroutine FORMAT is entered by other FORMAT subroutines to process a FORMAT specification. If there is a field count for a FORMAT specification, subroutine FORMAT processes the field count. Using a branch table and the delimiter accessed by the FORMAT statement scanning subroutine, GETWDA, it passes control to the proper subroutine to process the specification.

ENTRANCE: Subroutine FORMAT is entered by subroutine PRESCAN when PRESCAN finds a FORMAT adjective code. Subroutine FORMAT is entered by the other FORMAT subroutines, using certain entry points under given conditions, as follows:

1. FMTONE when subroutine FORMAT must get another delimiter.
2. FMTTWO when a FORMAT subroutine already has a delimiter, and the delimiter must be processed by subroutine FORMAT.
3. FMTGCE when a FORMAT subroutine has already checked the delimiter and determined that the delimiter was a comma, left parenthesis, or a P.
4. FMTBRN when the previous delimiter was a blank, and the delimiter fetched by subroutine BLANKZ must be processed.

OPERATION: To initialize the FORMAT subroutines to process a FORMAT statement, subroutine FORMAT sets:

1. The left parenthesis switch off.
2. The record length, leading length, and tail length accumulators to zero.
3. The group count to zero.
4. The comma switch on.

Subroutine FORMAT then gets the first delimiter. If the first delimiter is a left parenthesis, the parentheses count is set to 1, and the adjective code for a FORMAT statement is inserted in the text card. If not, an error condition is recognized.

Subroutine FORMAT converts the field count, if there was one, to a binary integer. If there was no field count for this FORMAT specification, the field count is set to 1. The delimiter returned from subroutine GETWDA is then used to access the branch table, and control is passed to the subroutine which processes the format specification.

EXIT: Subroutine FORMAT exits to subroutine MSG/MSGMEM, if subroutine FORMAT has detected an error, or to a FORMAT subroutine to process the delimiter accessed by subroutine GETWDA.

SUBROUTINES CALLED: During execution subroutine FORMAT references the following subroutines:

1. GETWDA to get another delimiter.
2. INTCON to convert an EBCDIC integer to a binary integer.
3. ERROR/WARNING if an error has been detected.

Subroutine D/E/F/I/A: Chart EI

Subroutine D/E/F/I/A processes the FORMAT specifications for D, E, F, I, and A conversions.

ENTRANCE: Subroutine D/E/F/I/A is entered from subroutine FORMAT, if the FORMAT code is a D, E, F, I, or A.

OPERATION: Subroutine D/E/F/I/A enters in a text card the field count and the field count adjective code, if there is a field count for the specification; the adjective code for the FORMAT code; the field length; and the decimal length, if there is a decimal length for this specification.

EXITS: Subroutine D/E/F/I/A exits to subroutines:

1. FORMAT to process the next FORMAT specification.
2. MSG/MSGMEM if an error is detected.

SUBROUTINES CALLED: During execution subroutine D/E/F/I/A references subroutine:

1. FLDCNT to check this specification for a field count and enter it in a text card, if there is a count.
2. PUTFTX to make entries in the text card.
3. GETWDA to get the delimiter, and number ahead of the delimiter.
4. INTCON to convert an EBCDIC integer to a binary constant.
5. LINETH to add the field length to the record length accumulator.
6. ERROR/WARNING if an error or warning is detected.

Subroutines QUOTE/H, X: Chart EJ

Subroutine QUOTE/H

Subroutine QUOTE/H processes the FORMAT specifications for H or quote literals.

ENTRANCE: Subroutine QUOTE/H is entered from subroutine FORMAT if the FORMAT code is a quote mark or H.

OPERATION: Subroutine QUOTE/H enters the adjective code for the FORMAT specification, the field count, and the literal in a text card.

EXIT: Subroutine QUOTE/H exits to subroutine:

1. FORMAT to process the next FORMAT specification.
2. MSG/MSGMEM if an error has been detected.

SUBROUTINES CALLED: During execution subroutines QUOTE/H references subroutine:

1. NOFDCT to insure that there was no count for the quote FORMAT codes.
2. PUTFTX to enter adjective codes, field count and the literal in text cards.
3. LINETH to add the field length to the record length accumulator.
4. ERROR/WARNING if an error is detected.

Subroutine X

Subroutine X processes FORMAT specifications for X FORMAT codes. Subroutine X enters the X adjective code and the field count in a text card.

ENTRANCE: Subroutine X is entered from subroutine FORMAT if the FORMAT code is an X.

EXIT: Subroutine X exits to subroutine FORMAT to process the next FORMAT specification.

SUBROUTINES CALLED: During execution subroutine X references subroutine:

1. PUTFTX to make entries in the text card.
2. LINETH to add the field length to the record length accumulator.
3. ERROR/WARNING if a warning is detected.

Subroutines +/-/P, BLANKZ, FILLEG, FCOMMA: Chart EK

Subroutine +/-/P

Subroutine +/-/P processes the format specification codes for scale factors.

ENTRANCE: Subroutine +/-/P is entered by subroutine FORMAT if the format code is a +, -, or P.

CONSIDERATION: If there is a count for a format specification with a scale factor, it should be placed between the P and the FORMAT code for the field and decimal lengths. For example, if the specification +3PF10.2 is to be repeated five times, the specification is written +3P5F10.2.

OPERATION: If the sign preceding the scale factor is negative, the scale factor is converted to a 1-byte binary integer, and a 1 bit is placed in the high-order bit for the byte. A scale factor of -3 is represented by the hexadecimal number 83, while the scale factor of +3 is represented by the hexadecimal number 03.

Subroutine +/-/P enters the scale factor adjective code and the scale factor in the text card.

EXIT: Subroutine +/-/P exits to subroutine:

1. FORMAT to process the remainder of the format specification, following the P.
2. MSG/MSGMEM if an error is detected.

SUBROUTINES CALLED: During execution subroutine +/-/P references subroutine:

1. GETWDA to get the scale factor and the following delimiter.
2. INTCON to convert the scale factor to a binary number.
3. PUTFTX to make entries to text cards.
4. ERROR/WARNING if an error is detected.

Subroutine BLANKZ

Subroutine BLANKZ processes any blanks encountered in scanning a FORMAT statement.

ENTRANCE: Subroutine BLANKZ is entered from subroutine FORMAT when a blank delimiter is encountered.

OPERATION: Subroutine BLANKZ references subroutine GETWDA. If the non-zero return is used by GETWDA, an error condition is noted. If the zero return is taken, sub-

routine BLANKZ passes control to subroutine FORMAT.

EXIT: Subroutine BLANKZ exits to the following subroutines:

1. FORMAT to process the delimiter it fetched.
2. MSG/MSGMEM if an error is detected.

SUBROUTINE CALLED: During execution subroutine BLANKZ references subroutine GETWDA to get the next delimiter.

Subroutine FILLEG

Subroutine FILLEG processes any invalid delimiters found in a FORMAT statement.

ENTRANCE: Subroutine FILLEG is entered from subroutine FORMAT when an illegal delimiter is found.

OPERATION: An error text word is constructed for each invalid delimiter.

EXIT: Subroutine FILLEG exits to subroutine MSG/MSGMEM.

SUBROUTINE CALLED: During execution subroutine FILLEG references subroutine ERROR/WARNING.

Subroutine FCOMMA

Subroutine FCOMMA processes any commas found in a FORMAT statement.

ENTRANCE: Subroutine FCOMMA is entered from subroutine FORMAT when a comma is found in a FORMAT statement.

OPERATION: Subroutine FCOMMA checks an indicator that denotes whether or not the current comma is valid. If the indicator is off and entry is made into this subroutine, the current comma is valid. Subroutine FCOMMA skips over the comma, but sets the comma indicator on. Until this indicator is set off, any subsequent encounter of a comma is considered an error. If the indicator is on when subroutine FCOMMA is entered, the error condition is noted.

EXIT: Subroutine FCOMMA exits to subroutine:

1. FORMAT to get another delimiter.
2. MSG/MSGMEM if an error is detected.

SUBROUTINE CALLED: If an error is detected, subroutine FCOMMA references subroutine ERROR/WARNING.

Subroutines LPAREN, RPAREN: Chart EL

Subroutine LPAREN

Subroutine LPAREN processes any left parentheses, other than the opening left parenthesis, in the FORMAT statement.

ENTRANCE: Subroutine LPAREN is entered from subroutine FORMAT when a left parenthesis is sensed.

OPERATION: A left parenthesis, other than the first parenthesis in a FORMAT statement, indicates the beginning of a group. A group indicator is set on by subroutine LPAREN any time a left parenthesis is encountered. If the group indicator is on when subroutine LPAREN is entered, an error is noted because BPS FORTRAN does not permit nesting groups.

If a field count preceded the left parenthesis, the group count is set equal to the field count, and the group count and the group adjective code are inserted in a text card. The leading and tail length accumulators are set to zero. The comma indicator is set on, so a comma between the left parenthesis and the first format specification for the group is an error. The group indicator is set on to indicate that a group is being processed. A leading length indicator is set on so the length for the group elements is accumulated in the leading length accumulator. The parenthesis count is then incremented by 1.

EXIT: Subroutine LPAREN exits to subroutine:

1. FORMAT to process the first FORMAT specification in the group.
2. MSG/MSGMEM if an error is detected.

SUBROUTINE CALLED: During execution subroutine LPAREN references subroutine:

1. PUTFTX to make entries to text cards.
2. ERROR/WARNING if an error is detected.

Subroutine RPAREN

Subroutine RPAREN processes any right parenthesis, in a FORMAT statement, to end a group or a FORMAT statement.

ENTRANCE: Subroutine RPAREN is entered by subroutine FORMAT when a right parenthesis is encountered.

OPERATION: Subroutine RPAREN insures that no field count preceded the right parenthesis, and decrements the parentheses count. If the parentheses count is less than zero, an error is noted. If the parentheses count is zero, the end of the FORMAT statement is reached.

If the end of the FORMAT statement is reached, the adjective code for the end of a FORMAT statement is sent to the text card, the record length is compared to the specified length, and the input pointer is adjusted to the next full-word boundary. Control is then passed to subroutine PRESCN to process the next intermediate text entry.

If the parentheses count is positive, the end of a group has been reached. If the sum of the leading and tail length accumulators exceeds the specified length, a warning is issued. If the tail length accumulator is zero, the record length accumulator is incremented by the product of the leading length accumulator and the group count. If the tail length accumulator is not zero, the record length accumulator is set equal to the tail length accumulator.

The group indicator is set off indicating that the group has been processed, and the adjective code for end group is entered in the text card. Control is then passed to subroutine FORMAT to process the next format specification.

EXIT: Subroutine RPAREN exits to subroutine:

1. FORMAT to process the next FORMAT specification, after all the elements of a group have been processed.
2. PRESCN if the closing right parenthesis for the FORMAT statement is processed.
3. MSG/MSGMEM if an error has been detected.

SUBROUTINE CALLED: During execution subroutine RPAREN references the following subroutines:

1. NOFDCT to insure that no field count preceded the right parenthesis.
2. PUTFTX to make entries to a text card.
3. LINECK to insure the accumulated length in the record length accumulators has not exceeded the specified length.
4. ERROR/WARNING if an error or warning is detected.

Subroutines T, FSLASH: Chart EM

Subroutine T

Subroutine T processes T FORMAT specifications in the FORMAT statement and enters the adjective code and the record position in a text card.

ENTRANCE: Subroutine T is entered from subroutine FORMAT when a T FORMAT code is recognized.

OPERATION: Subroutine T insures that there was no count for this FORMAT specification. Subroutine GETWDA is referenced to get the record position for the T specification. The T adjective code and the position number are then entered in the text card. The record length accumulator is compared to the specified length.

The group indicator is tested for the T specification within a group. If T was specified, the record length accumulator is set to zero and the tail length accumulator is set equal to the position number. If the T specification is not in a group, the record length accumulator is set equal to the position number.

The leading length indicator is set off so the specifications within a group are accumulated in the tail length accumulator instead of the leading length accumulator.

EXIT: Subroutine T exits to subroutine FORMAT to process the next FORMAT specification.

SUBROUTINES CALLED: During execution subroutine T references the subroutines:

1. NOFDCT to insure that there is no count immediately preceding a T specification.
2. GETWDA to get the position number.
3. INTCON to convert the position number to a binary integer.
4. PUTFTX to enter the T adjective code and the position number in the text card.
5. LINECK to insure that the accumulated length in the record length accumulator does not exceed the specified length.
6. ERROR/WARNING if an error is detected.

Subroutine FSLASH

Subroutine FSLASH processes the slash FORMAT specifications in a FORMAT statement, and enters the slash adjective code in the text card.

ENTRANCE: Subroutine FSLASH is entered by subroutine FORMAT when a slash FORMAT specification is recognized.

OPER TION: Subroutine FSLASH enters the slash adjective code in the text card and insures there was no field count for the slash entry. The record length accumulator is compared to the user-specified length.

The record and tail length accumulators are set to zero and the leading indicator is set off so, if the slash is part of a group, the specifications following the slash are accumulated in the tail length accumulator.

EXIT: Subroutine FSLASH exits to subroutine FORMAT to process the next FORMAT specification.

SUBROUTINES CALLED: During execution subroutine FSLASH references the following subroutines:

1. PUTFTX to enter the slash adjective code in the text card.
2. NOFDCT to insure that there is no field count for a slash specification.
3. LINECK to insure the accumulated length in the record length accumulator does not exceed the specified length.

Subroutines LINETH, LINECK, FLDCNT, NOFDCT: Chart EN

Subroutine LINETH

Subroutine LINETH adds the total length for the field(s) in a FORMAT specification to the record length accumulator.

ENTRANCE: Subroutine LINETH is entered by subroutines D/E/F/I/A, QUOTE/H, and X.

OPERATION: The total length for a FORMAT specification is computed by multiplying field length by the field count. The total length is added to the record length accumulator.

If the group indicator is off, the total length is added to the record length accumulator. If the group indicator is on, the total length is added to either the leading or tail length accumulator.

If the leading indicator is on, the total length is added to the leading length accumulator. If the leading indicator is off, the total length is added to the tail length accumulator.

EXIT: Subroutine LINETH exits to the subroutine that called it.

Subroutine LINECK

Subroutine LINECK determines if the length of the record exceeds the user-specified length.

ENTRANCE: Subroutine LINECK is entered by subroutines RPAREN, T, and FSLASH.

OPERATION: If the group indicator is off, the record length accumulator is compared to the specified length. If the record length is greater, a warning is issued.

If the group indicator and the leading indicator are both on, the leading length accumulator is added to the record length accumulator and the record length accumulator is compared to the specified length. If the leading indicator is off, the record length accumulator is set equal to the tail length accumulator, and the comparison to the specified length is made.

EXIT: Subroutine LINECK exits to the subroutine that called it.

SUBROUTINES CALLED: If the record length accumulator exceeds the user-specified length, subroutine LINECK calls subroutine ERROR/WARNING to insert the warning entry in the intermediate text.

Subroutine FLDCNT

Subroutine FLDCNT checks for a field count.

ENTRANCE: Subroutine FLDCNT is entered by subroutine D/E/F/I/A.

OPERATION: If a field count, other than 1, exists, the field count adjective code and the field count are entered in a text card.

EXIT: Subroutine FLDCNT exits to the subroutine that called it.

SUBROUTINE CALLED: During execution subroutine FLDCNT references subroutine PUTFTX to put the field count adjective code and the field count in a text card.

Subroutine NOFDCT

Subroutine NOFDCT processes FORMAT specifications that cannot have a field count.

ENTRANCE: Subroutine NOFDCT is entered by subroutines QUOTE/H, RPAREN, T, and FSLASH.

OPERATION: If the current specification has a field count, an error is noted.

EXIT: Subroutine NOFDCT exits to the subroutine that called it.

SUBROUTINE CALLED: If the specification has a field count, subroutine ERROR/WARNING is referenced to issue the error text word.

Subroutines GETWDA, INTCON: Chart EO

Subroutine GETWDA

Subroutine GETWDA scans FORMAT statements, returning the delimiter that stopped the scan and the number of numeric characters preceding the delimiter. If the contents of an input buffer have been completely processed, subroutine GETWDA references another subroutine to read an intermediate text record.

ENTRANCE: Subroutine GETWDA is entered from subroutines FORMAT, D/E/F/I/A, +/-/P, BLANKZ, and T.

OPERATION: When subroutine GETWDA is entered by a calling subroutine, GETWDA increments the translate and test pointer. If the character at the translate and test pointer is blank, the pointer is incremented. If the character is not a blank, a test is made for the end of the input buffer. If the end is reached, subroutine ININ/GET is called to read another record into the buffer, and the translate and test pointer is adjusted so that it points to the beginning of the next buffer.

When the end of the buffer is not reached, a translate and test instruction is executed to get the next delimiter. The table for this instruction is set so the instruction stops on any non-numeric character. The translate and test instruction inserts the address at which it stopped in general register 1, and the non-zero byte in the table which caused the instruction to stop in general register 2. The address in general register 1 is used to calculate the length of the numeric field preceding the delimiter and to initialize GETWDA for the next time it is entered. The byte in general register 2 is the adjective code entered in the text card for FORMAT specifications and is used to index the branch table in subroutine FORMAT.

The length of the numeric field is calculated, using the address in general register 1 and the address at which the translate and test instruction began execution. The data just translated is then moved to a work area so that subroutine INTCON can access it.

A test is made to see if the scan has reached the end of either buffer. If the end has not been reached, a test is made to see if the length of the numeric data is zero. A zero length implies that there is no numeric data preceding this delimiter. For example, if a right parenthesis is immediately preceded by a slash, the length is zero and the zero return is used in returning to the calling subroutine. If the length is non-zero, the non-zero return is used.

There are two buffers used to read the intermediate text tape. There are two tests, made for the end of the buffer: one for the first buffer and a second for the remaining buffer.

The two buffers are adjacent to each other in main storage. The last storage position of the first buffer is next to the first storage position of the second buffer. When the FORMAT statement is written on the intermediate text output tape in Phase 10, it is written in a card image. There is no attempt made to translate the FORMAT statement to any form of internal code. It is possible that part of the FORMAT statement may be read into one buffer, while the remaining part of the statement is in the second buffer.

The translate and test instruction does not stop when the end of the first buffer is reached; it accesses characters in the second buffer until a delimiter is found. For example, if a FORMAT statement is set up as follows:

```
FORMAT(....,5F10.2,....)
```

and it was read in Phase 14 so the first buffer contained part of the specification 5F10.2 and the second buffer contained the remaining part, as follows:

first buffer	second buffer
....,5F1	0.2,....

the translate and test instruction does not stop because it is at the end of the buffer, splitting the 1 and 0. The instruction ceases execution when the decimal point is reached. The only action necessary in this situation is starting the read operation to refill the first buffer.

If the statement was read in Phase 14 so the second buffer contained the first part of the specification and the first buffer contained the second part as follows:

first buffer	second buffer
0.2,....	....,5F1

it is illogical for subroutine GETWDA to return merely the number and stop fetching characters when the end of the second buffer is reached. Still it cannot continue translating until it reaches a delimiter, because whatever is beyond the last position reserved for the second buffer is not part of the FORMAT statement.

A special character, which stops the translate and test instruction, is placed at the end of the second input buffer. If the translate and test instruction is stopped at this special character, subroutine GETWDA recognizes that the end of the second buffer has been reached. It calculates the length of the field just translated. In the above example, the result of the length calculation is 1. The field translated is moved to a position immediately in front of the first character in the first input buffer.

	first buffer	second buffer
1	0.2,....	....,5F1

The translate and test pointer is then adjusted so that it points to the first character in the field just moved. Then subroutine ININ/GET is referenced to start the read operation to refill the first buffer, and the translation process is repeated.

EXIT: Subroutine GETWDA exits to the subroutine that called it.

SUBROUTINE CALLED: During execution subroutine GETWDA references subroutine ININ/GET to read another intermediate text record into an input buffer.

Subroutine INTCON

Subroutine INTCON converts integer constants to binary and checks their validity.

ENTRANCE: Subroutine INTCON is entered by subroutines FORMAT, D/E/F/I/A, +/-/P, and T.

OPERATION: The data to be converted is accessed from the work area in which subroutine GETWDA stored the integer. If the data in the work area is not numeric, an error condition is noted.

The data in the work area is converted to binary. The high-order digit in the work area is accessed and inserted in a general register. If there is a second digit, the contents of the register are multiplied by 10 and the second digit is added to the product. For each succeeding digit, the contents of the register are multiplied by 10; that digit is added to the product.

If the resulting number is greater than 255, an error condition is noted. A test is made to see if this digit is to be tested for a zero value in this format specification. If it may have a zero value, control is passed from subroutine INTCON to the subroutine that called it.

If the integer constant cannot have a zero value, and the number is zero, an error condition is noted. If an error does not occur, control is passed from subroutine INTCON to the subroutine that called it.

EXIT: Subroutine INTCON exits to subroutine MSG/MSGMEM if an error is detected or to the calling subroutine.

SUBROUTINE CALLED: Subroutine INTCON references subroutine ERROR/WARNING if an error is detected.

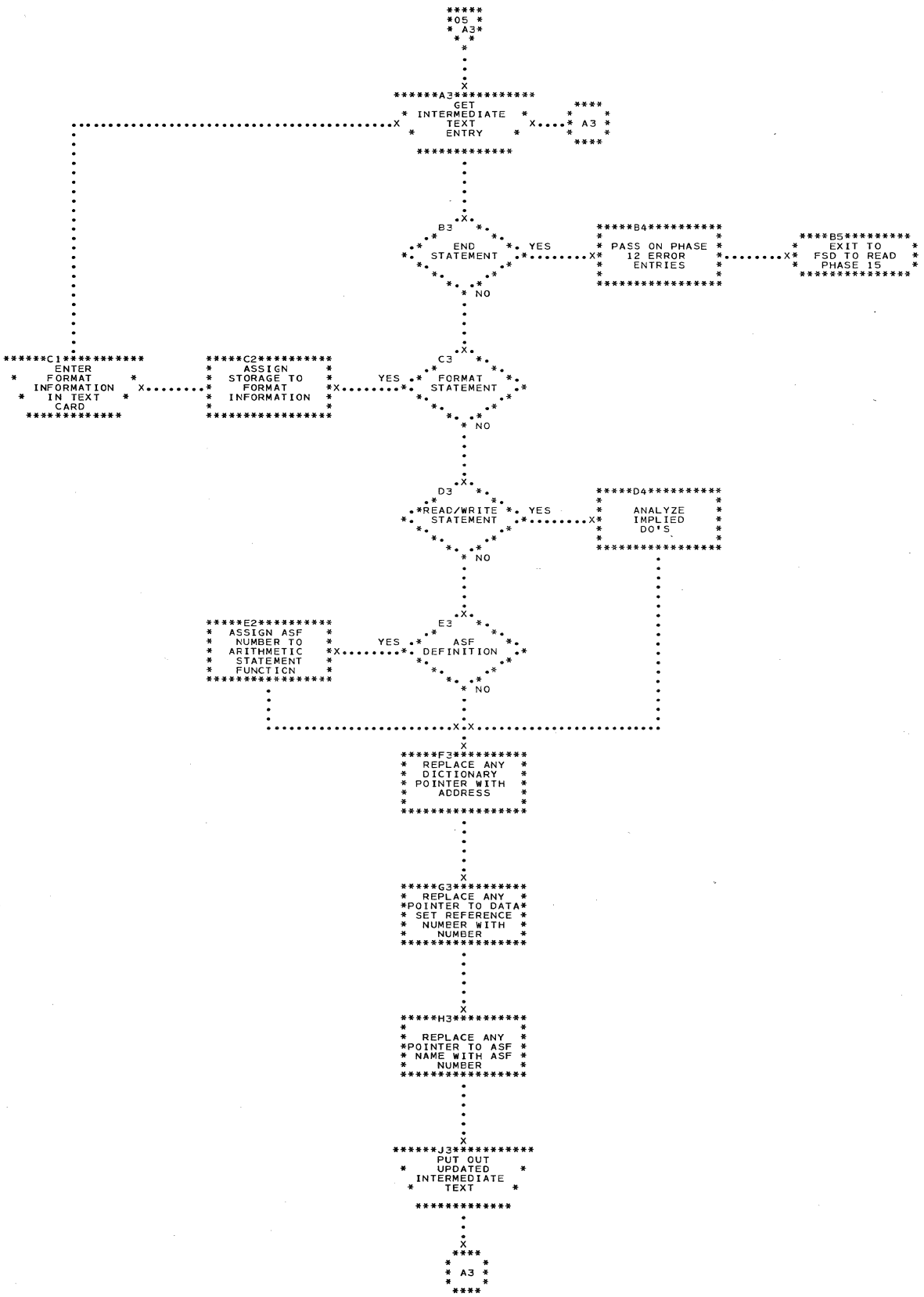


Chart 05. Phase 14 Overall Logic Diagram

Chart EA. Subroutine PRESCAN

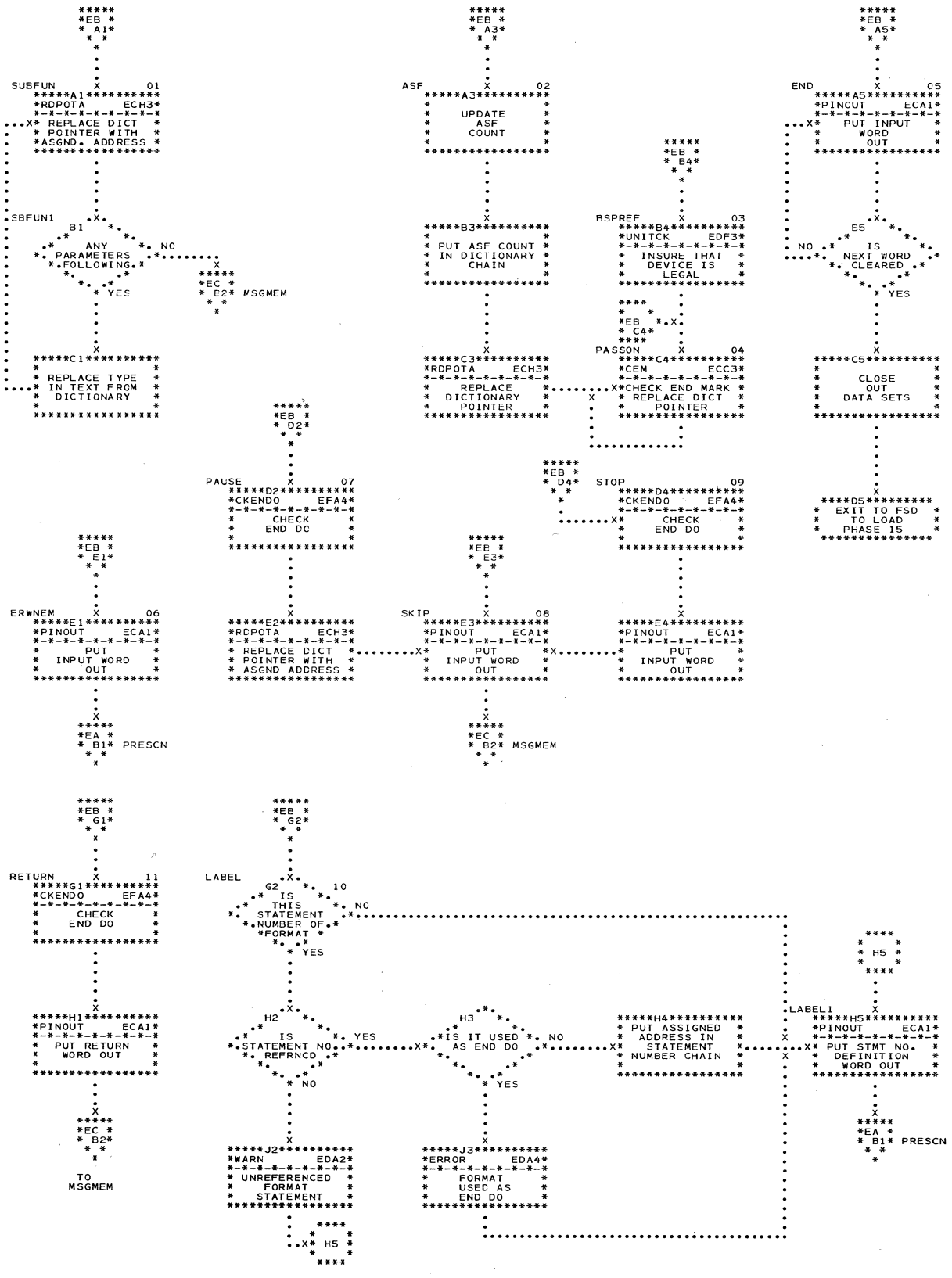


Chart EB. Subroutine Adjective Code

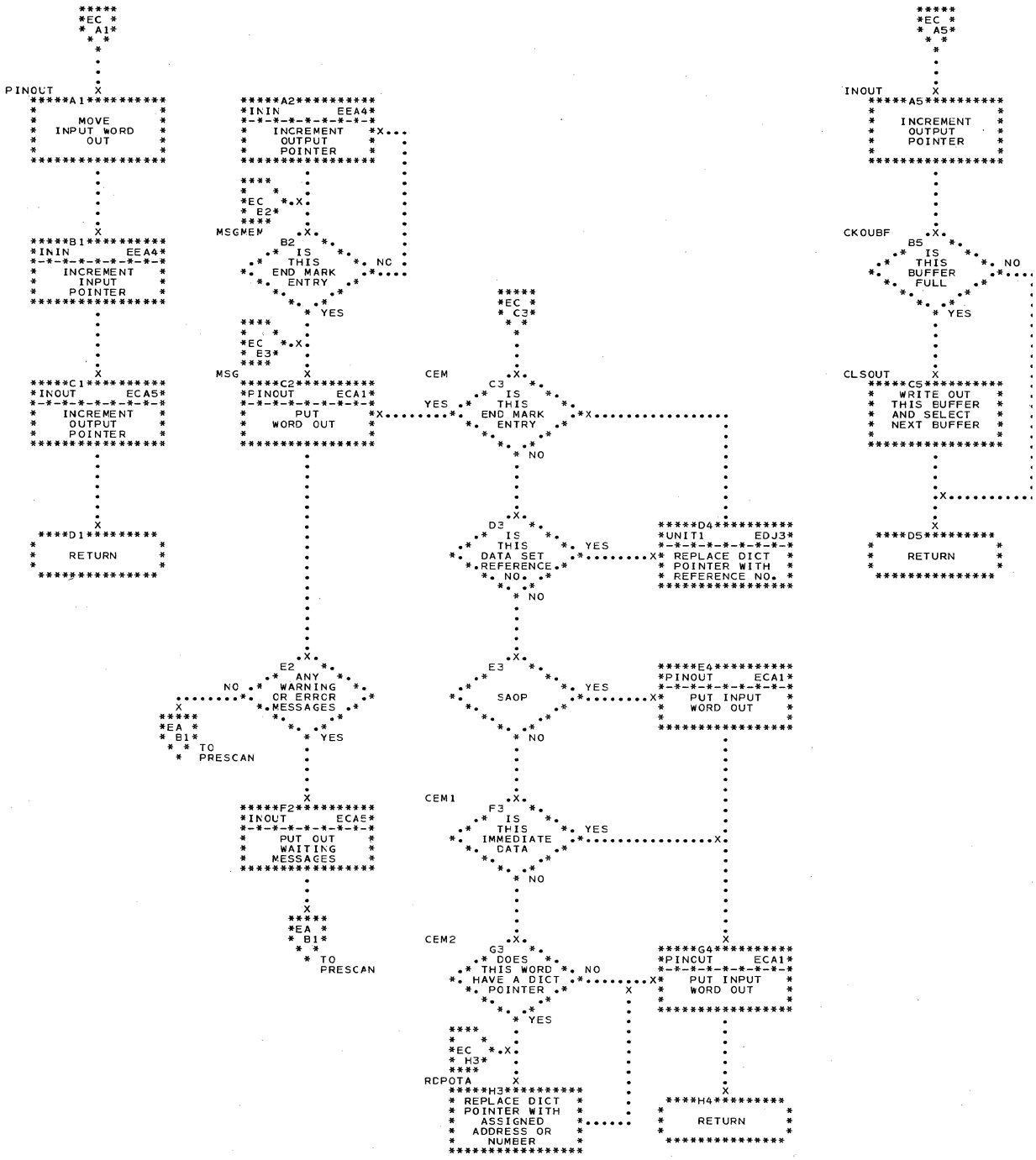


Chart EC. Subroutines PINOUT, INOUT, MSG/MSGMEM, CEM/RDPOTA

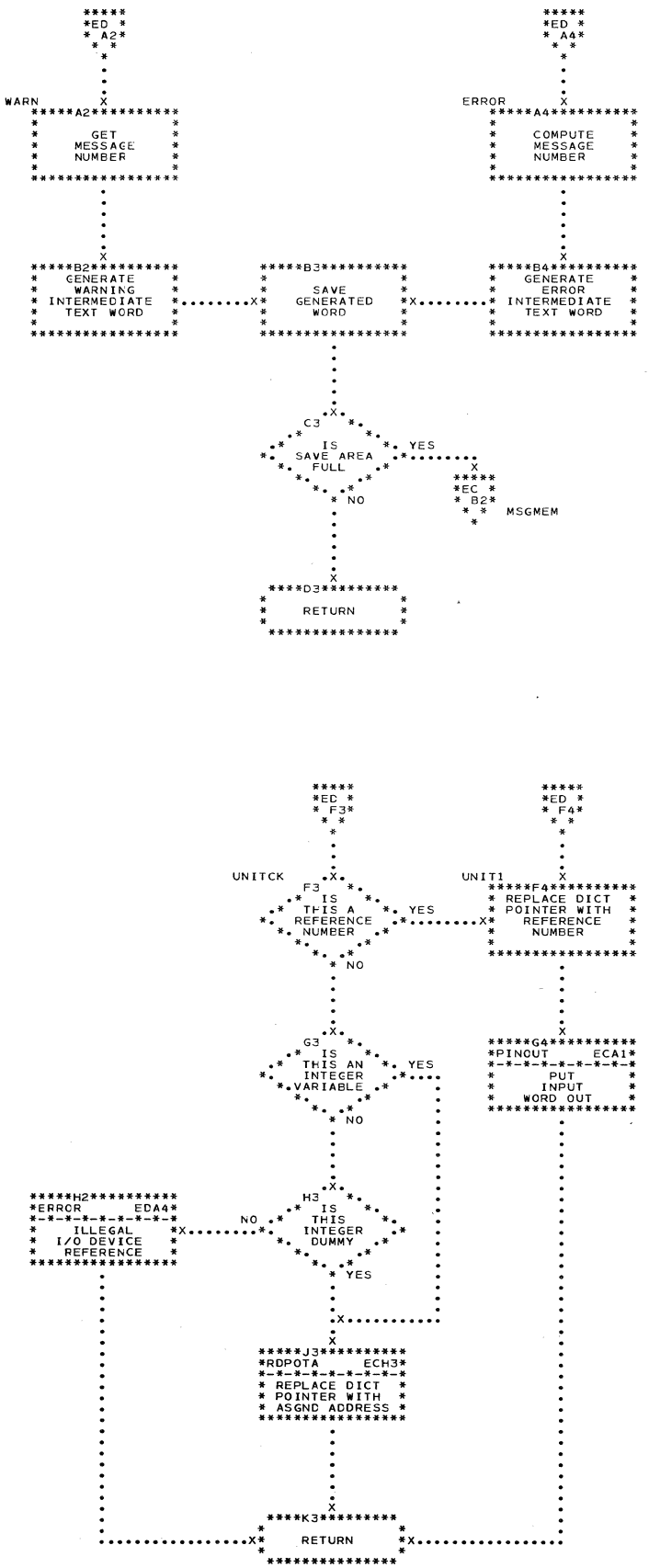


Chart ED. Subroutine ERROR/WARNING, UNITCK/UNIT1

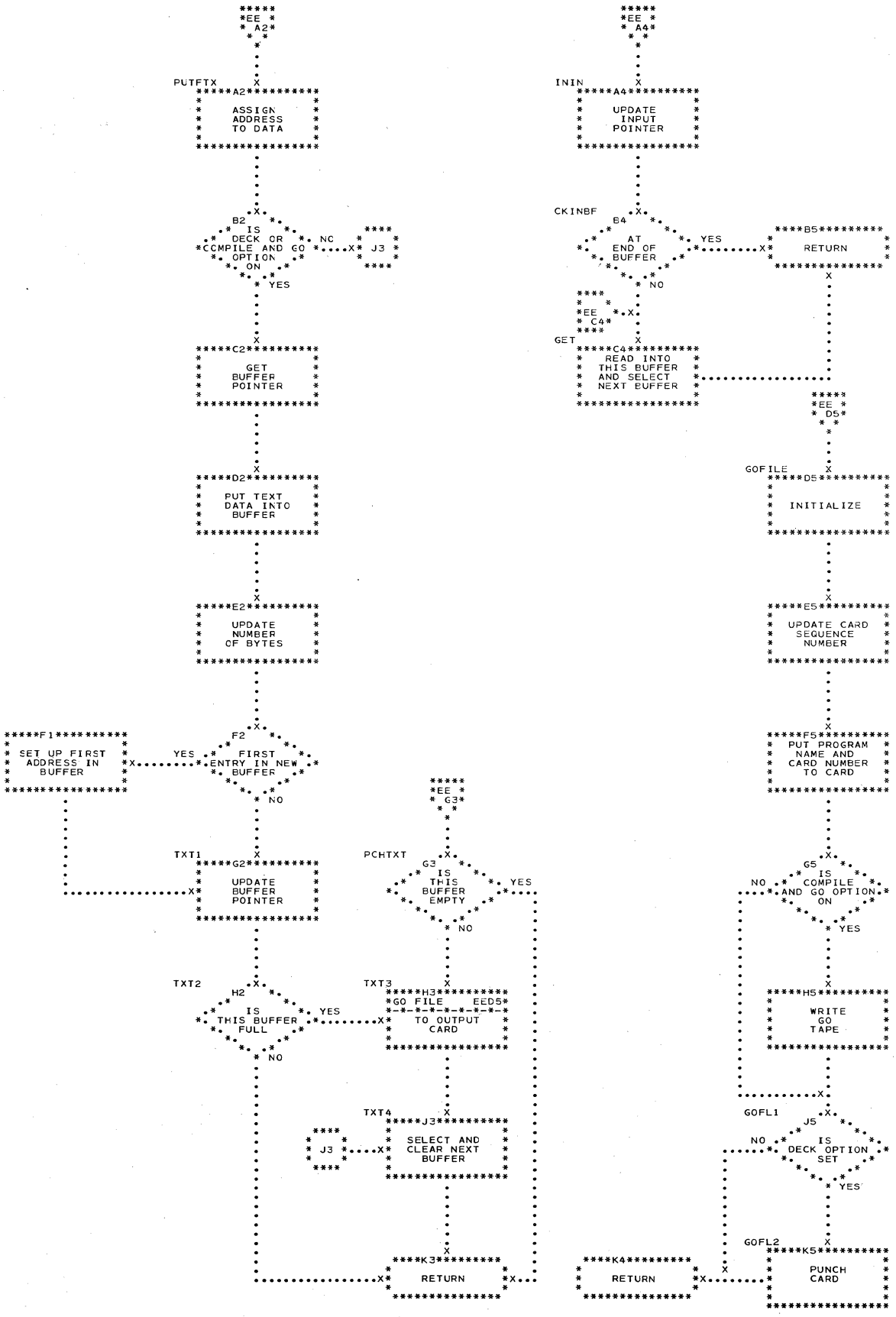


Chart EE. Subroutines PUTFTX, ININ/GET, GOFILE

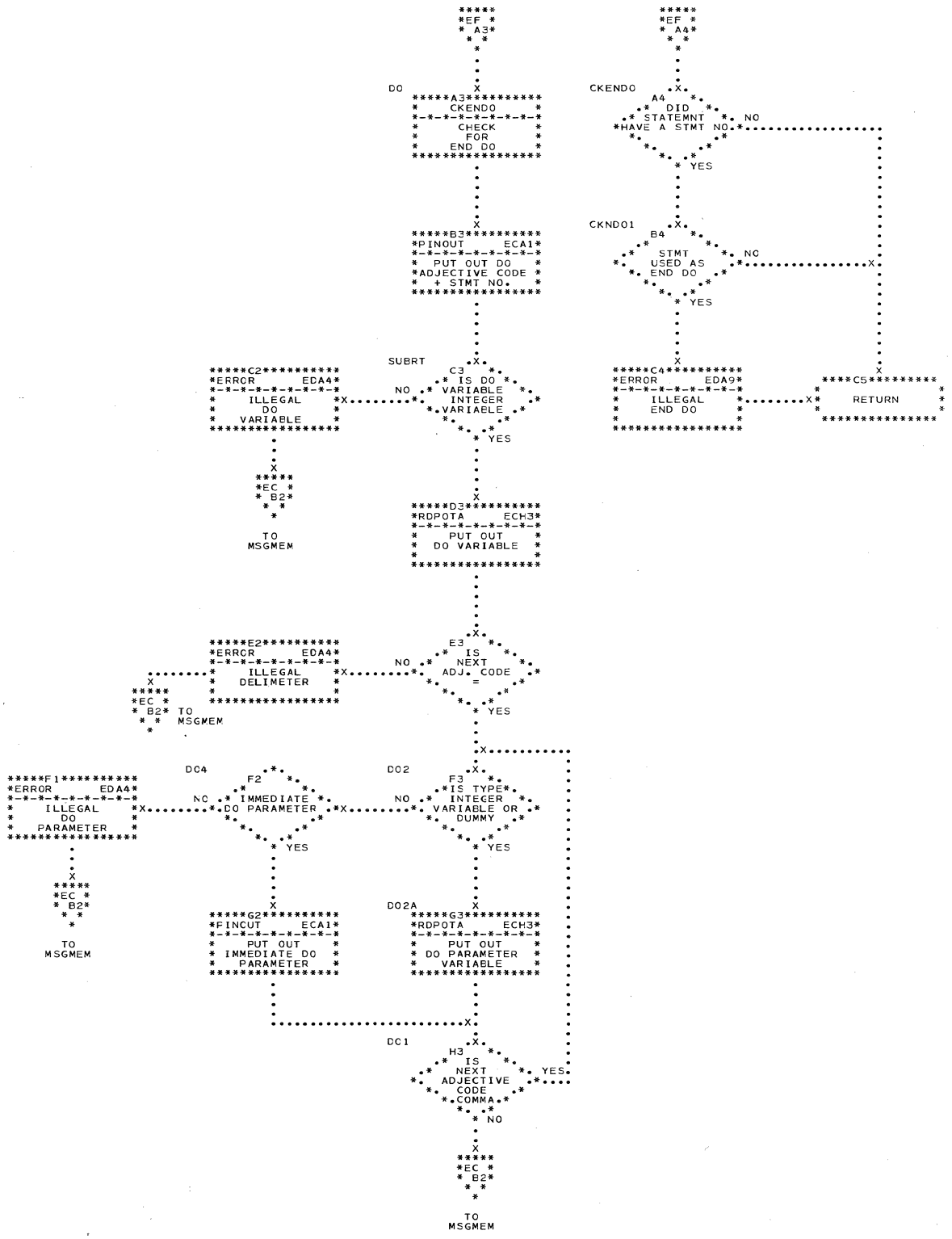


Chart EF. Subroutines DO, CKENDO

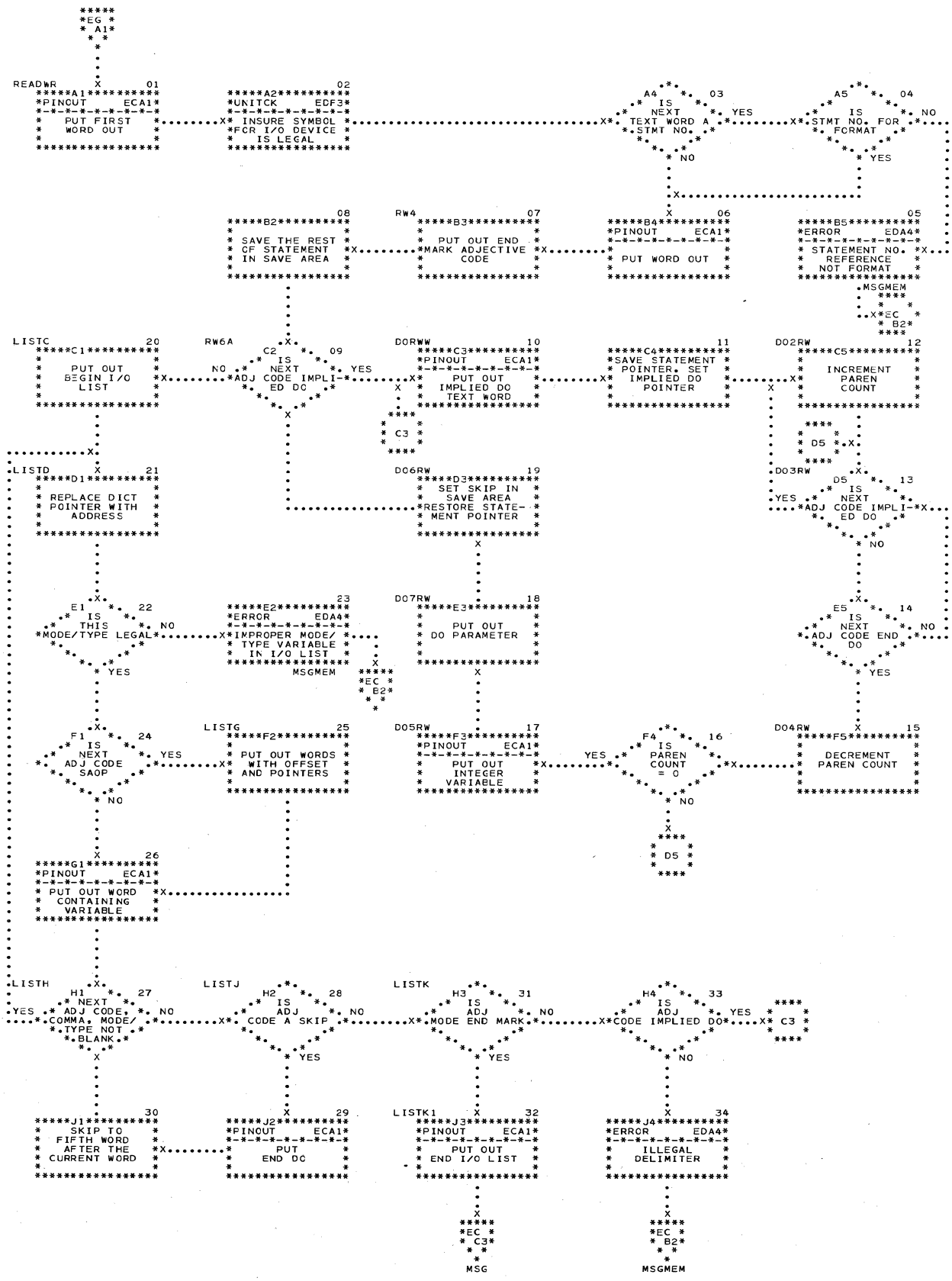


Chart EG. Subroutine READ/WRITE

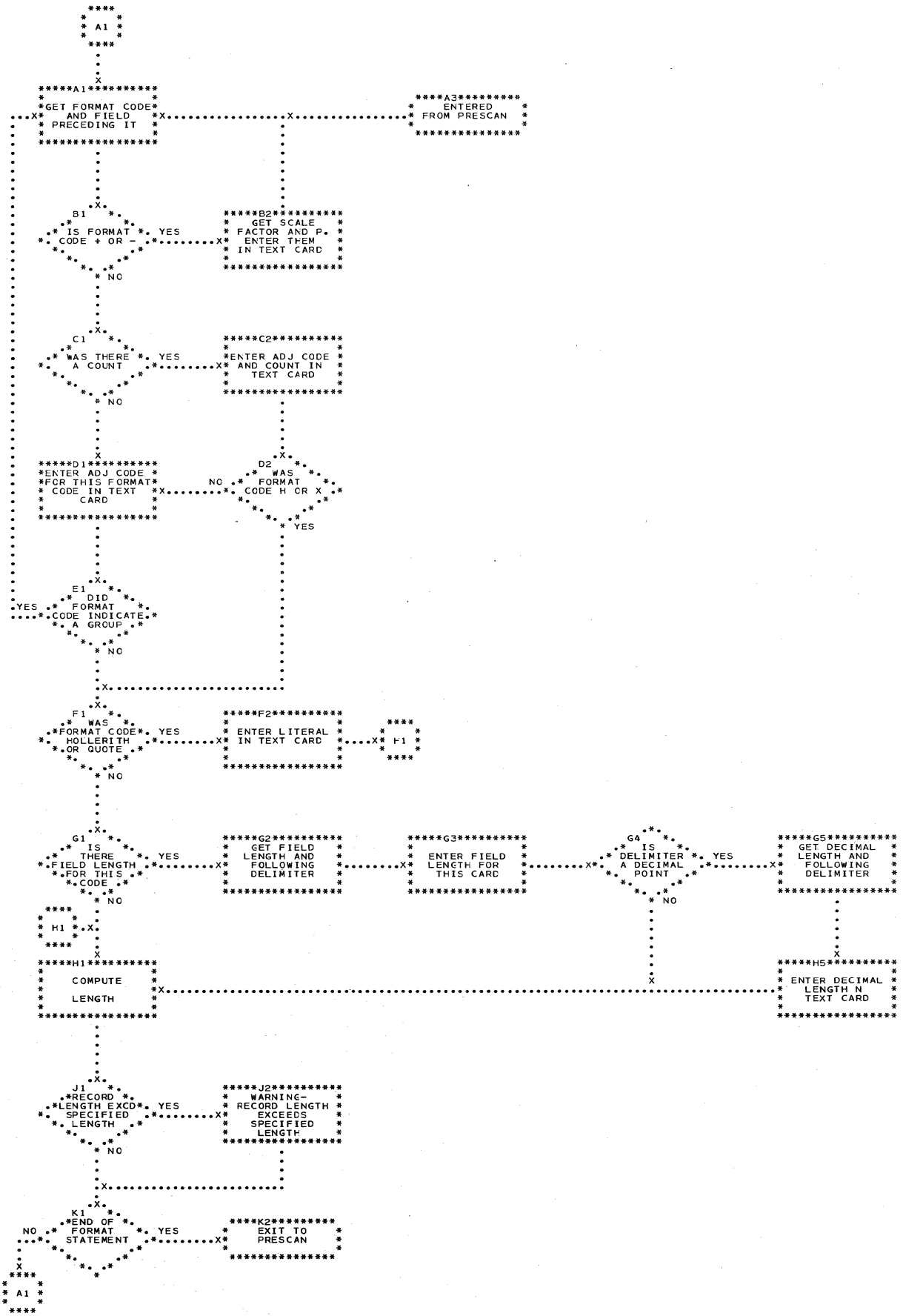


Chart 21. Phase 14 FORMAT Overall Logic Diagram

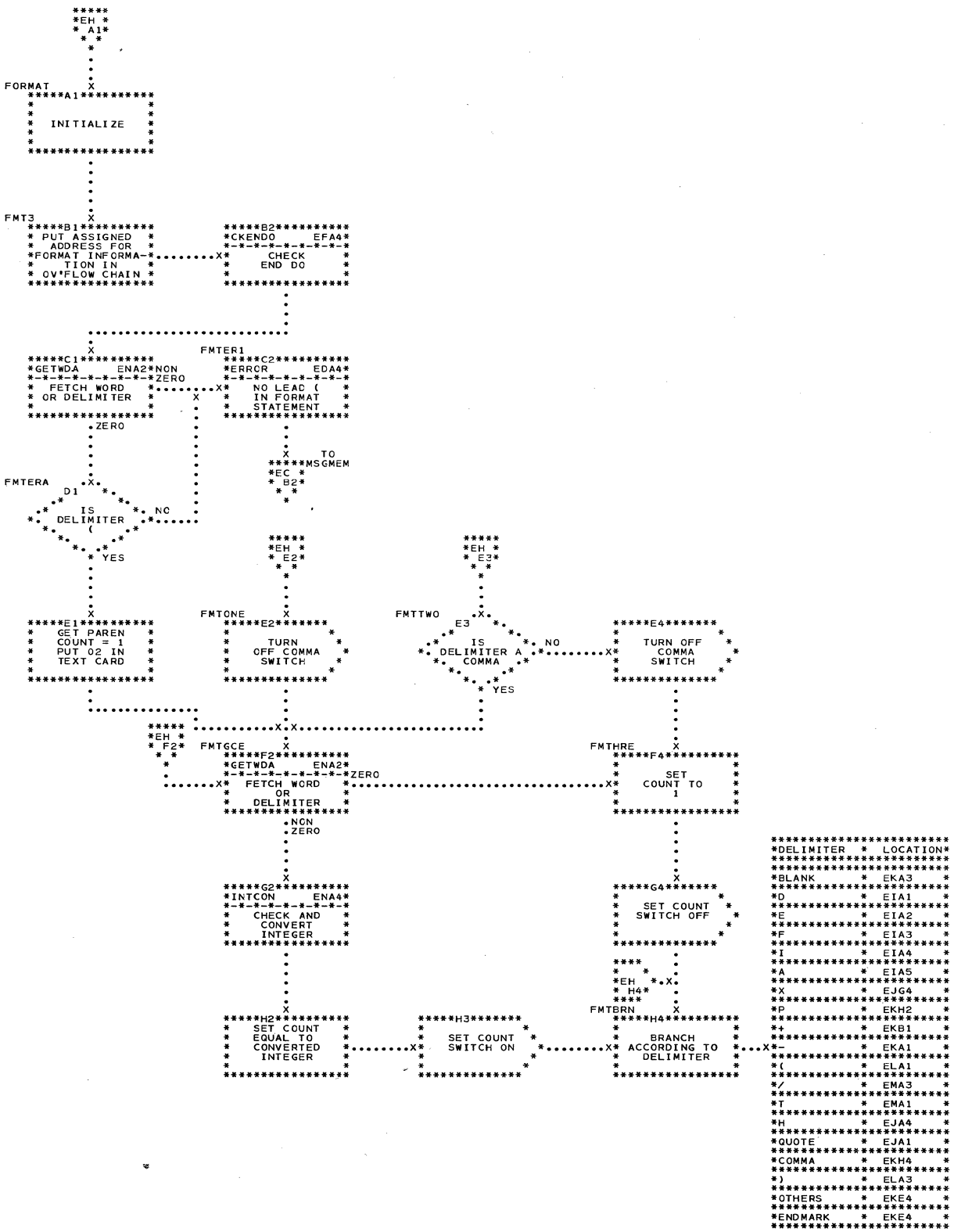


Chart EH. Subroutine FORMAT

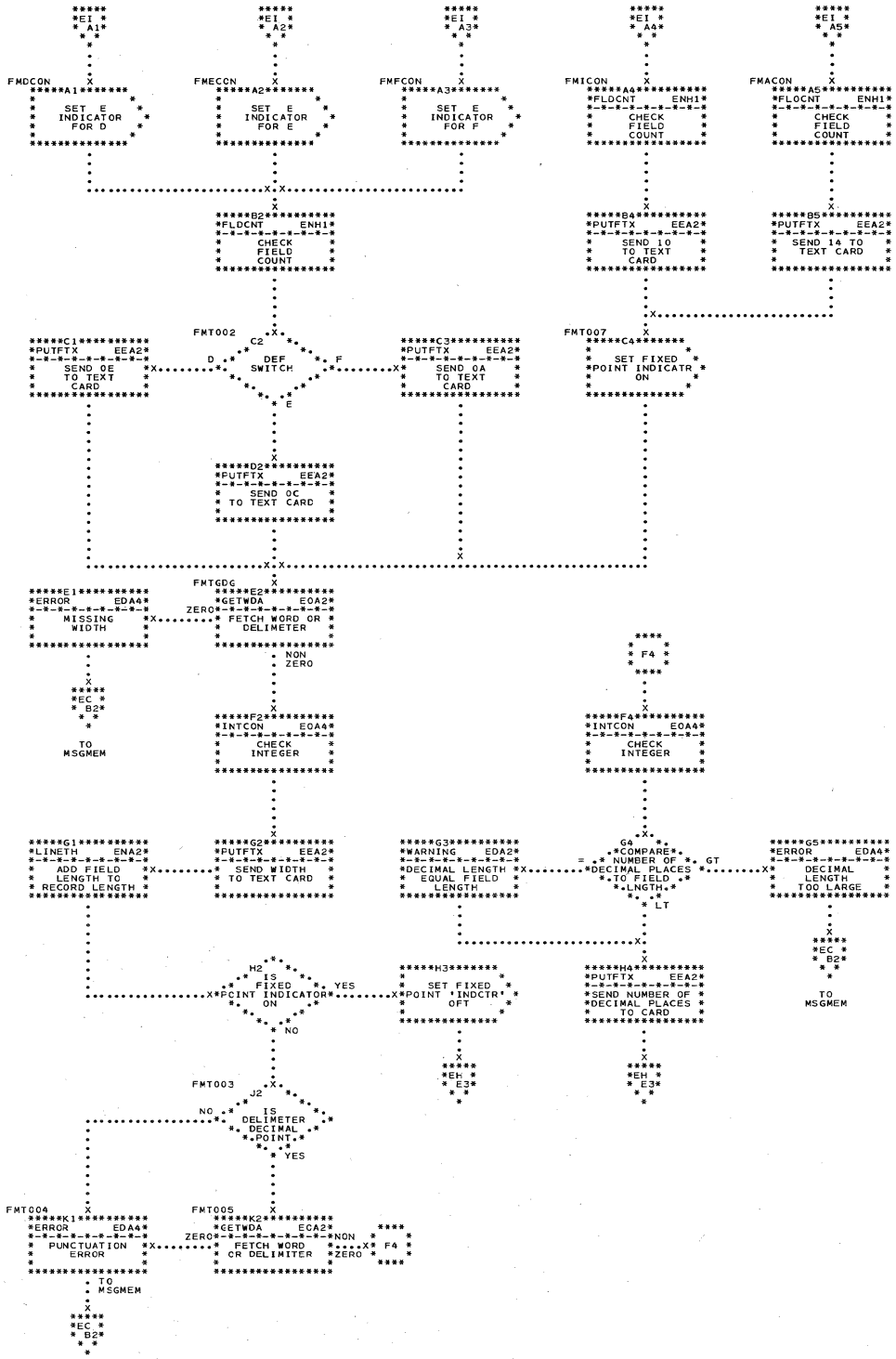


Chart EI. Subroutine D/E/F/I/A

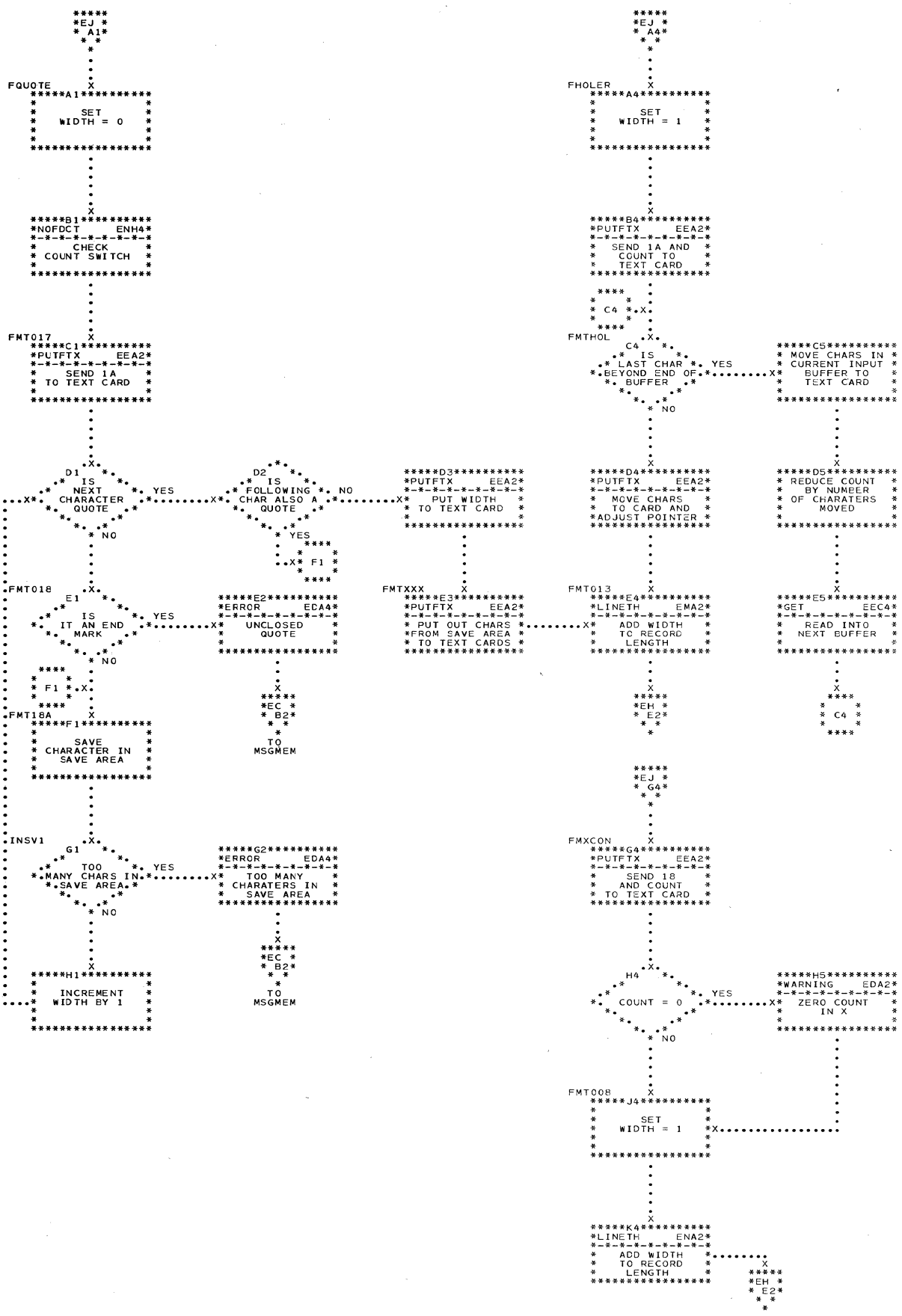


Chart EJ. Subroutine QUOTE/H,X

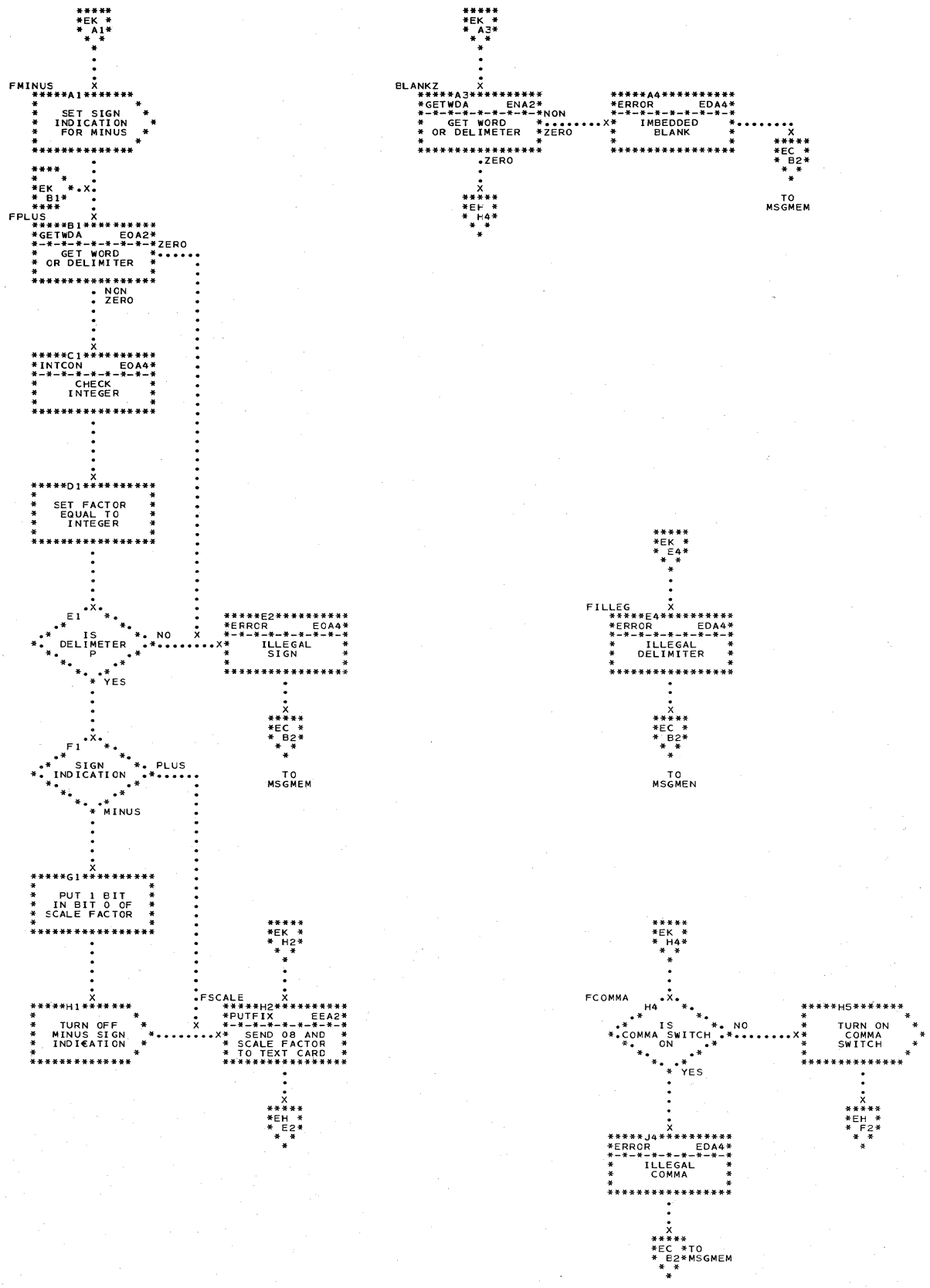


Chart EK. Subroutines +/-/P, BLANKZ, FILLEG, FCOMMA

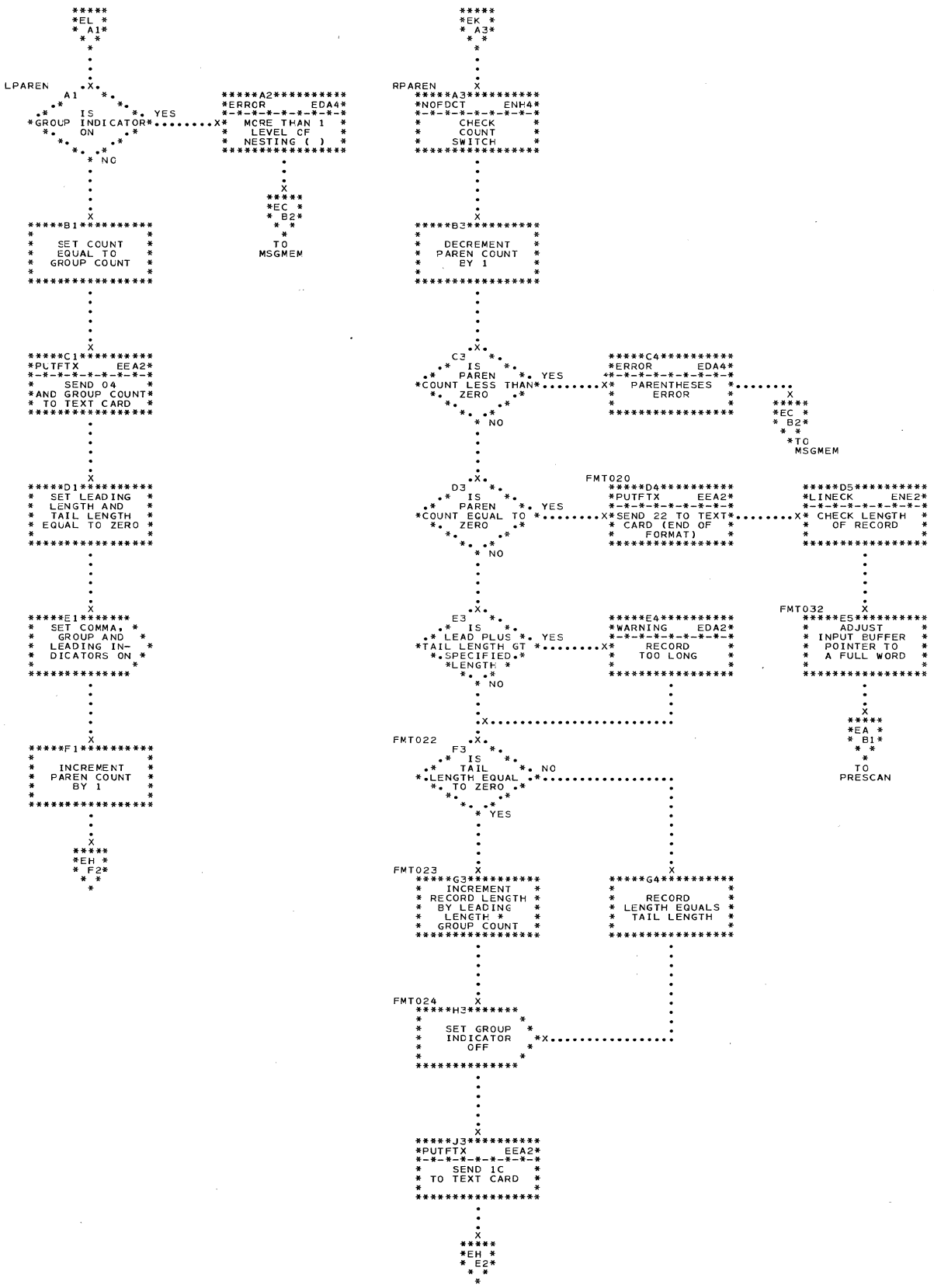


Chart EL. Subroutines LPAREN, RPAREN

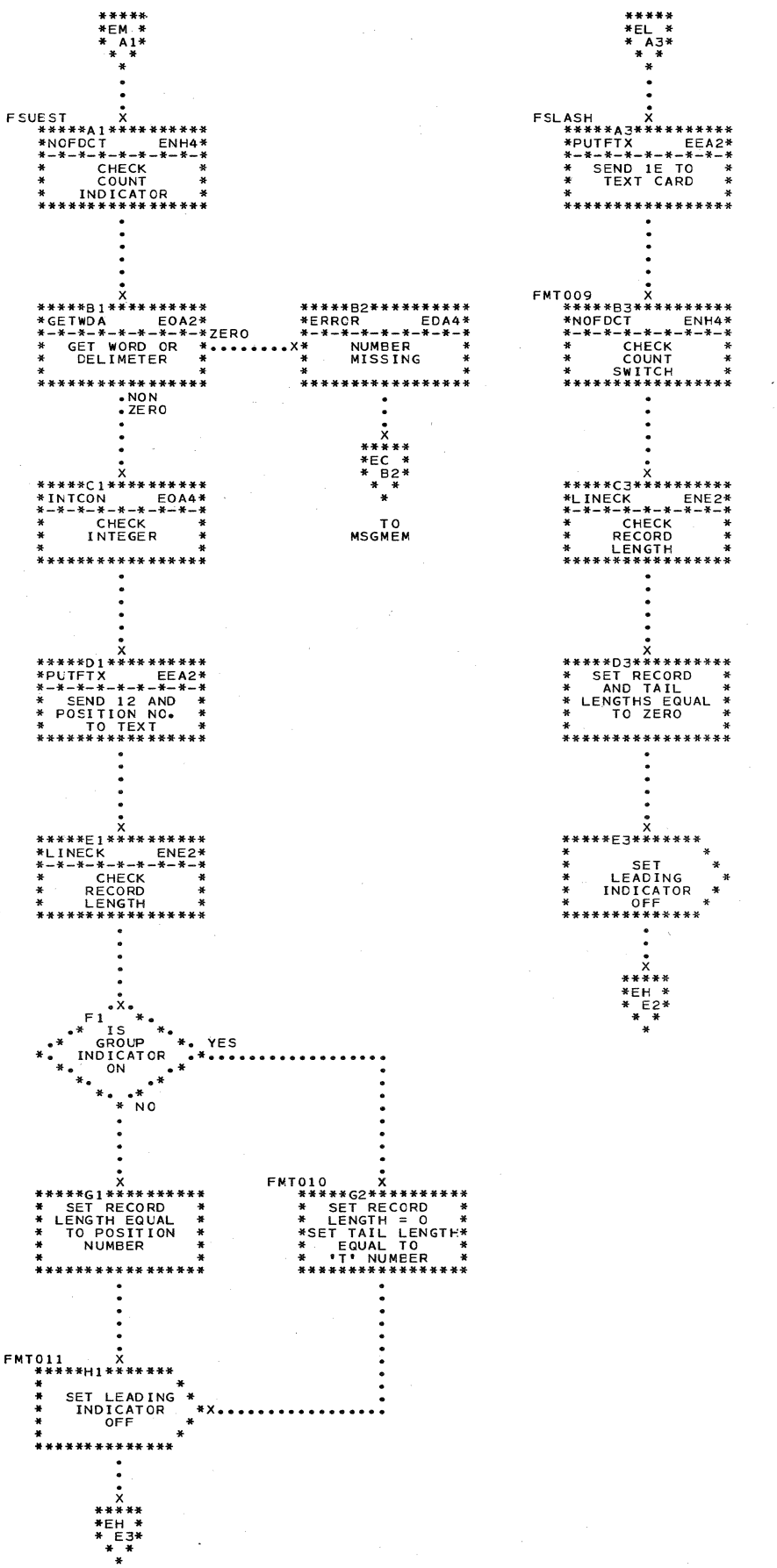


Chart EM. Subroutines T, FSLASH

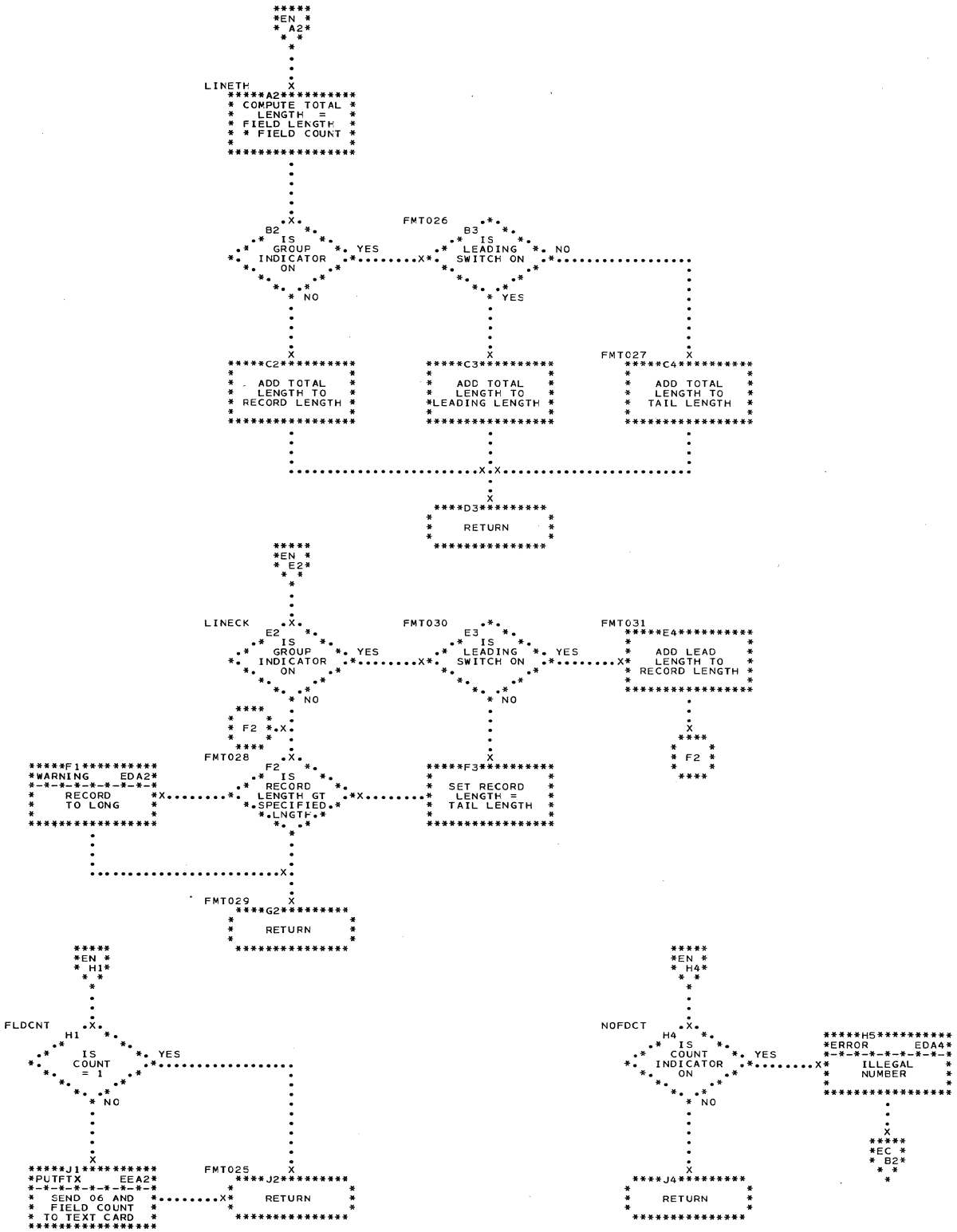


Chart EN. Subroutines LINETH, LINECK, FLDCNT, NOFDCT

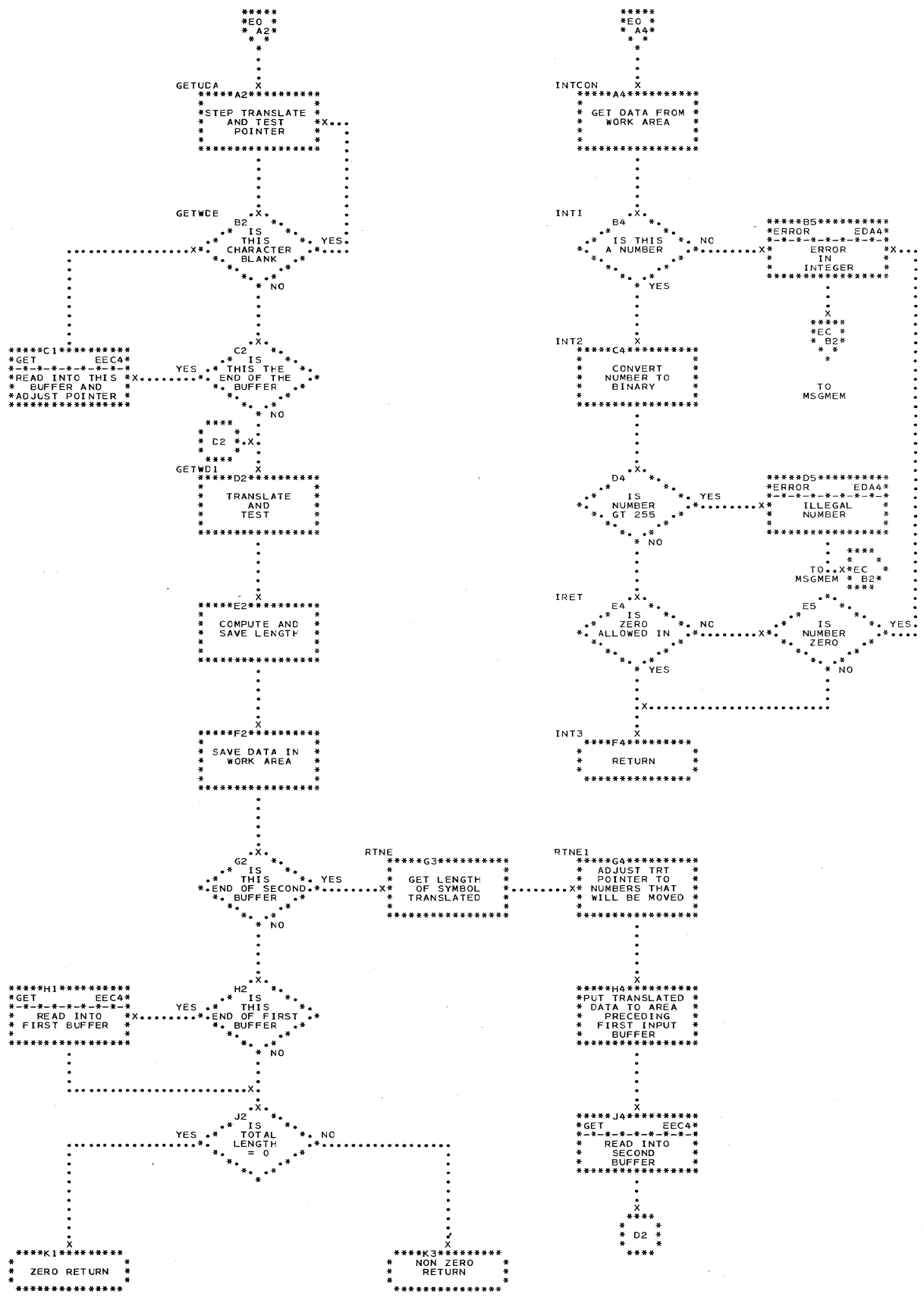


Chart EO. Subroutines GETWDA, INTCON

Phase 15 modifies input text and converts it to a more refined form by reordering the sequence of text words within the statements. The text words may be modified to a form closely resembling an instruction format. When the text words are modified, registers are assigned to the operands depending upon the operator. Argument lists for external or arithmetic statement function references are created by modifying the input text. In-line function references are processed by generating the appropriate instruction format text or in-line function call word. During the input text processing, errors pertaining to a DO loop, arithmetic IF statement, statement number, function argument, and operand usage and form are recognized, and the proper messages are given.

Chart 06, the Phase 15 Overall Logic Diagram, indicates the entrance to and exit from Phase 15 and is a guide to the overall functions of the phase.

ORDER OF OPERATIONS

Phase 15 implements an ordering of the operations within each statement of the text by reordering the sequence of text words.

The desired order is defined by a hierarchy of the specific operations as represented by adjective codes and determined by comparing two adjective codes. The text word is either processed, or stored in the operations or subscript table depending upon the hierarchy.

Operations Table

The operations table is a temporary storage area used during the ordering of operations within a statement for any text words referring to the operation. An exception is made for subscript text which is stored in the subscript table.

The operations table may contain no more than 50 entries. The entries in the operation table are accessed by a pointer to the last entry for the specific statement under consideration.

Subscript Table

The subscript table is used as a temporary storage area for subscript text. Each subscript entry in the subscript table is two words. There may be no more than 38 entries to the subscript table.

Forcing Scan

The forcing scan directs the ordering of the text words of the statement. It compares the forcing value of the various adjective codes 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 text word in the input text. The left forcing value applies to the adjective code within the text word in the operations table.

The adjective code of the first word of the statement has the highest forcing value of any adjective code except the end of statement indicator. This adjective code is entered into the operations table.

As a word of the input text is accessed, its right forcing value is compared to the left forcing value of the adjective code of the last word entered into the operation table. If the left forcing value is higher than the right forcing value of the current input text word, the current input text word is stored in the operations table. If the left forcing value is lower or equal, the current input text word is processed.

A word is uniquely processed depending upon its adjective code, and then written onto the output data set. In this way, the input data set is ordered when it leaves Phase 15 as the output data set.

ARGUMENT LISTS

When an adjective code indicating a call to an in-line, external, or arithmetic statement function is detected, a list of arguments is constructed. An exception is made for the SNGL and DBLE in-line functions which are processed by Phase 15.

For external or arithmetic statement functions, the argument list is preceded by a text word containing information to identify the specific function call. The first word of the argument list is a count word which indicates the number of arguments. It is followed by a text word for each argument.

For in-line functions not processed by Phase 15, only one text word is generated. The three parts of this text word are:

1. An in-line function call adjective code.
2. Registers assigned to the operand in the mode/type field.
3. A code to indicate the specific in-line function in the address pointer field.

During the processing of argument lists, a count of the total number of arguments is kept in the communications area. This count will be used by Phase 20.

TEXT WORD MODIFICATION

As each statement of Phase 15 is processed, the various text words are examined and modified. The contents of the adjective code field may be changed to an operation code which is determined by the required operation and the mode of the operands. The mode field is replaced by an appropriate register assignment.

Register Assignment

Registers are assigned by Phase 15 according to the adjective code that is encountered. Many operations and most function references require that certain or all operands be in registers.

There are eight registers (general registers 0, 1, 2, and 3; and floating point registers 0, 2, 4, and 6) assigned by Phase 15 in these cases. The type of register used depends upon the mode of the operation and operands.

When a register is required and one is not available, the contents of required registers are placed in the first available work area (save register technique).

ERROR CHECKS

As each statement is accessed and processed, specific error conditions are recognized. General format errors as well as errors connected with specific statements such as DO, arithmetic IF, statement number, or an argument list are noted. DO loops are examined to determine if the DO variable is redefined or if a DO loop is partially nested. Arithmetic IF statements are examined to determine if the arithmetic expression contains legal symbols. They are also examined to determine if more or less than three statement numbers have been specified. Statements numbers are examined to insure that they are defined and do not indicate transfers to nonexecutable statements. If a function subprogram is being compiled, a check is made to determine whether the function name is defined. The members of an argument list are examined to determine whether they are valid. If the particular list has a required length it is examined to determine if that list is of the required length. If an error condition is discovered, an error message is given.

ROUTINES/SUBROUTINES

The routines and subroutines within Phase 15 fall into four groups. The first group contains the PRESCN routine which accesses each statement and determines which routines of the second group will process that statement or begin the processing of that statement. The routines called by PRESCN form the second group.

The third group refers to the routines which are called by the FOSCAN routine (a routine in the second group).

The routines called by the second and third groups, other than FOSCAN, form the fourth group. Figure 45 illustrates these groups.

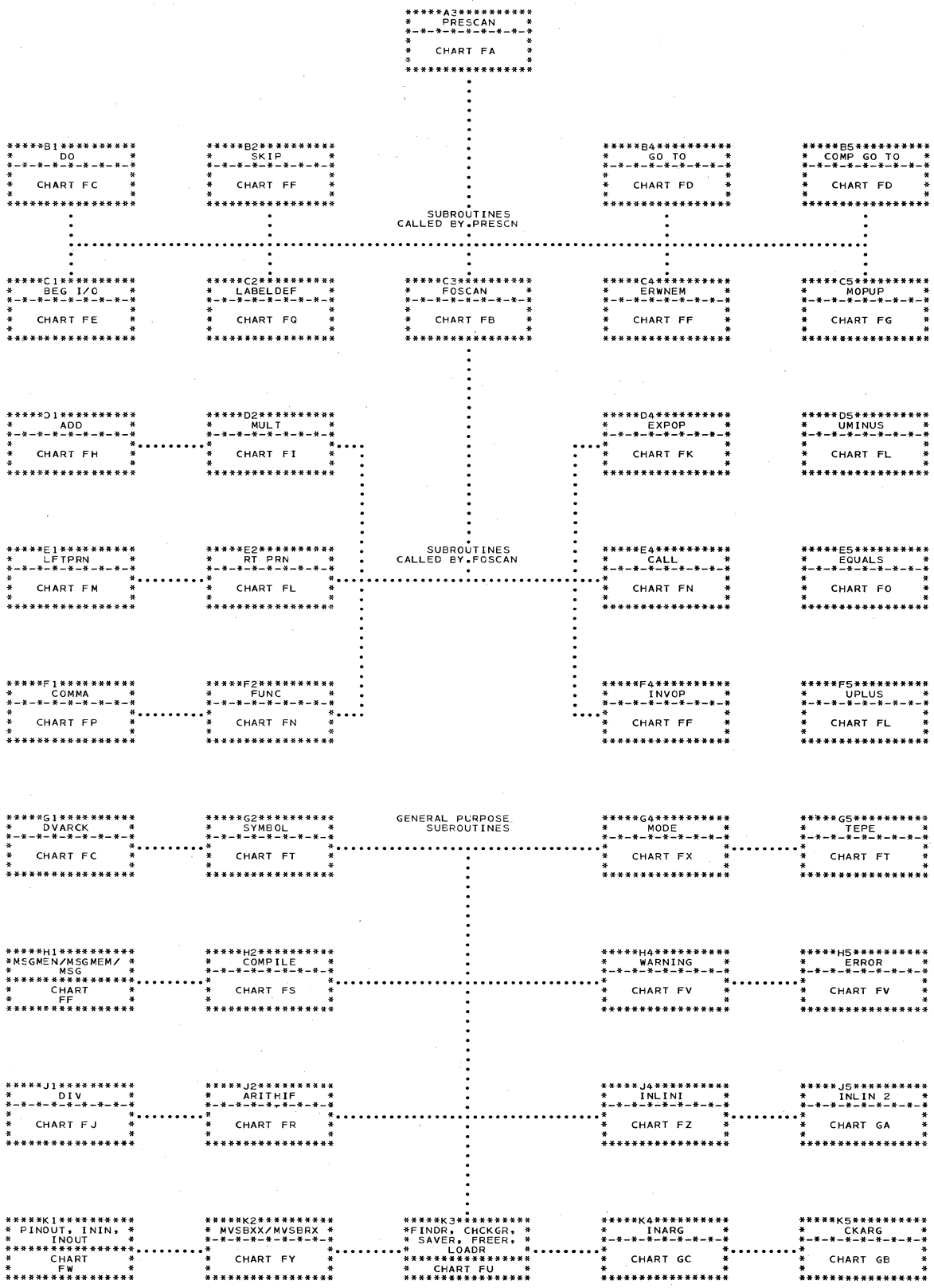


Figure 45. Organization of Phase 15

PRESCN Routine: Chart FA

The PRESCN (control) routine, identifies each statement type and passes control to the appropriate processing routine.

ENTRANCE: The PRESCN routine receives control from the FORTRAN System Director.

OPERATION: The PRESCN routine performs the necessary phase initialization. Each statement identification word is then accessed and the adjective code is examined to determine the appropriate processing routine for that statement.

EXIT: When the END statement indicating the end of the input text is sensed, control passes to the MOPUP routine. The MOPUP routine exits to the FORTRAN System Director to call in Phase 20.

FOSCAN Routine: Chart FB

The FOSCAN routine, which is the arithmetic scan, checks the syntax of the arithmetic, arithmetic IF, CALL, and ASF statements. This routine removes parentheses and orders the arithmetic expression according to the hierarchy of operations.

ENTRANCE: The FOSCAN routine is entered from the PRESCAN routine when an arithmetic, arithmetic IF, CALL, or ASF definition adjective code is detected.

CONSIDERATION: The forcing scan is used during the arithmetic scan. It is described in the Introduction to Phase 15.

OPERATION: The first text word of the statement is written onto the intermediate text output tape. The FOSCAN routine attempts to limit the value range of the adjective codes in the forcing value tables to increase the efficiency of working with the table. The FOSCAN routine equates the adjective code of any arithmetic, arithmetic IF, CALL, or ASF definition statement to a particular forcing adjective code value. The original adjective codes would cause a more extensive range of adjective codes in the forcing value table. The modified text word is placed into the operations table.

After the first word, as modified, is entered into the operations table, the next word of the input text is accessed and examined. If it is a subscript word, the subscript text is entered into the subscript table. The following word of the input text is then accessed and examined. When the word (in the operations table) containing the subscripted variable is processed, the related subscript text is accessed from the subscript table. The related subscript text is always the latest subscript in the subscript table.

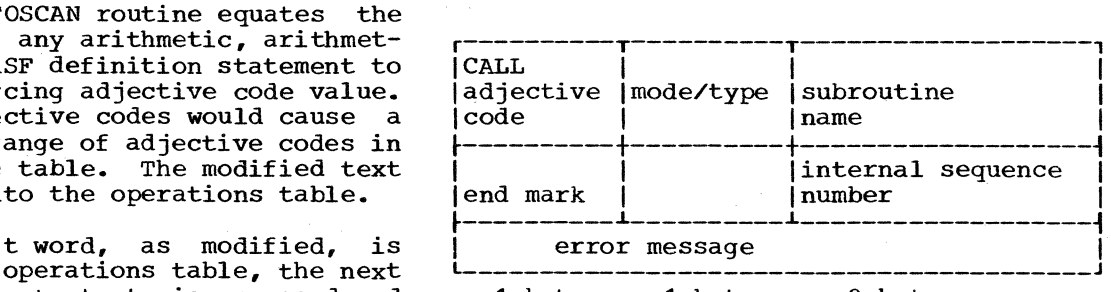
If the word accessed from the input text is not a subscript word, the right forcing value associated with that word is accessed and compared to the left forcing value of the latest operations table entry for that statement. If the right forcing value forces the left forcing value (according to the hierarchy set up by the forcing scan), the text is processed.

If the left forcing value is not forced by the right forcing value, the current word of the input text is entered into the operations table. The next word of the input text is then accessed.

If an attempt is made to enter information into the operations table when it is full, an error condition is recognized. An error message indicating that the statement is too long and should be subdivided is issued. The remainder of that statement is not processed.

The end mark is written on a work tape; all other words, including the non-processed words in the subscript table are not written out.

There is additional processing for the CALL and ASF definition statements. For a CALL statement, the name of the subroutine subprogram called is checked to determine if it has been defined. If the mode/type field of the word indicates an external library subprogram, the subroutine subprogram name is considered defined and processing continues. If the subprogram name is not defined, the statement is not processed. An error message, along with the end mark, are written onto the output buffer. For an undefined subprogram name, the associated CALL statement output is:



For an ASF definition, the ASF switch is set. This indicates to the various routines called by the arithmetic scan that an ASF definition statement is being processed.

Table 1. Right and Left Forcing Tables

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
- u	05	a (UMINUS)	01
end mark	00	never forced out	80
+ u	05	a (UPLUS)	01
ASF 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

Right and left forcing tables of the format in Table 1 are used to determine the right and left forcing values of the various operators.

Input/Output Formats: There are four statement types (arithmetic, arithmetic IF, CALL, and ASF) which may appear as input to the Phase 15 Arithmetic Scan. An arithmetic scan within the FOSCAN routine passes control to the associated operator routine to process the individual text words.

A simple arithmetic statement such as:

HERBIE = MACK - WINDY

appears in the input to Phase 15 as:

arithmetic adjective code	integer variable	a (HERBIE)
=	integer variable	a (MACK)
-	integer variable	a (WINDY)
end mark		internal statement number

1 byte 1 byte 2 bytes

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

The output from Phase 15 for this statement is:

arithmetic adjective code	integer variable		a (HERBIE)
L	register #3	variable	a (MACK)
S	register #3	variable	a (WINDY)
ST	register #3	variable	a (HERBIE)
end mark			internal statement number

1 byte 1 byte 2 bytes

The first operand, MACK, is loaded into register #3.

The second operand, WINDY, is subtracted from MACK.

The result is stored in the resultant field, HERBIE.

A complex arithmetic statement such as:

A (IOAD) = JETHRO * MOOSE + A(IOAD)

appears in the input to Phase 15 as:

arithmetic adjective code	real subscripted variable	a (A)	
SAOP	residual mode/type information	Offset	These two words contain the subscript text for A(IOAD).
p (subscript)		p (dimension)	
=	real variable	a (JETHRO)	
*	real variable	a (MOOSE)	
+	real subscripted variable	a (A)	
SAOP	residual mode/type information	Offset	These two words contain the subscript text for A(IOAD).
p (subscript)		p (dimension)	
end mark		internal statement number	
1 byte	1 byte	2 bytes	

The output from Phase 15 for this statement is:

arithmetic adjective code	real subscripted variable		a (A)
LE	register #6	subscripted variable	a (JETHRO)
ME	register #6	real variable	a (MOOSE)
SAOP	0	register #3	Offset
p (subscript)			p (dimension)
AE	register #6	real subscripted variable	a (A)
SAOP	0	register #3	Offset
p (subscript)			p (dimension)
STE	register #6	real subscripted variable	a (A)
end mark			internal statement number
1 byte	1 byte		2 bytes

Phase 15 zeros residual mode information and replaces residual type information with a work register.

These three words contain the subscript text for + A (IOAD).

Phase 15 zeros residual mode information and replaces residual type information with a work register.

These three words contain the subscript text for A (IOAD) =.

A simple arithmetic IF statement such as:

```
IF (MART) 11,5,63
```

appears in the input to Phase 15 as:

arithmetic IF adjective code	00	0000
(	integer variable	a (MART)
)	statement number	p (11)
,	statement number	p (5)
,	statement number	p (63)
end mark	00	internal statement number
1 byte	1 byte	2 bytes

The output from Phase 15 for this statement is:

arithmetic IF adjective code	00		0000
LE	register #0	variable	a (MART)
IF forcing adjective code	50		0000
)	statement number		p (11)
,	statement number		p (5)
,	statement number		p (63)
end mark			internal statement number

1 byte 1 byte 2 bytes

The value of the arithmetic portion of the arithmetic IF is loaded into register 0.

The mode field indicates the address pointer field indicates register 0.

A complex Arithmetic IF statement such as:

IF (IR(H) + M - 18) 6, 26, 64

appears in the input to Phase 15 as:

arithmetic IF adjective code	00	00 00
(	integer subscripted variable	a (IR)
SAOP	residual mode/type information	Offset
p (subscript)		p (dimension)
+	integer variable	a (M)
-	integer constant	a (18)
)	statement number	p (6)
,	statement number	p (26)
,	statement number	p (64)
end mark		internal statement number

1 byte 1 byte 2 bytes

These two words contain subscript information for IR(H).

The output from Phase 15 for this statement is:

arithmetic IF adjective code	00		0000
SAOP	0	register #3	offset
p(subscript)			p(dimension)
L	register #3	subscripted variable	a(IR)
A	register #3	variable	a(M)
S	register #3	constant	a(18)
IF forcing adjective code	register #3	1	0003
)	statement number		p(16)
,	statement number		p(16)
,	statement number		p(26)
end mark			internal statement number

1 byte 1 byte 2 bytes

Phase 15 zeros residual mode information and replaces residual type information with a work register. These three words contain the subscript information for IR(H).

The value of M is added to the value of IR(H) in register #3.

The value of 18 is subtracted from the value of IR(H)+M in register 3.

A 1 in the mode/type field indicates a register in the address pointer field. Register #3 will contain the value of the arithmetic portion of the arithmetic IF.

A CALL statement with no parameters:

```
CALL SUBRTN
```

appears in the input to Phase 15 as:

CALL adjective code	real subprogram	a(SUBRTN)
end mark		internal statement number

1 byte 1 byte 2 bytes

The output from Phase 15 for this statement is:

CALL adjective code	real subprogram	a(SUBRTN)
CALL forcing	real subprogram	a(SUBRTN)
00	00	0000
end mark		internal statement number
1 byte	1 byte	2 bytes

There is no argument.

A simple CALL statement:

```
CALL SUBRTN (A,1.0)
```

appears in the input to Phase 15 as:

CALL adjective code	real external subprogram	a(SUBRTN)
(	real variable	a(A)
,	real constant	a(1.0)
end mark		internal statement number
1 byte	1 byte	2 bytes

The output from Phase 15 for this statement is:

CALL adjective code	real external subprogram	a (SUBRTN)
CALL forcing	real external function	a (SUBRTN)
00	00	0002
(	real variable	a (A)
,	real constant	a (1.0)
end mark		internal statement number
1 byte	1 byte	2 bytes

There are two arguments.

A complex CALL statement:

CALL OAJK (A,B(I),C*2.+D)

appears in the input to Phase 15 as:

CALL adjective code	real subprogram	a (OAJK)
(	real variable	a (A)
,	real subscripted variable	a (B)
SAOP	residual mode/type information	Offset
p (subscript)		p (dimension)
,	real variable	a (C)
*	real constant	a (2.)
+	real variable	a (D)
)	00	0000
end mark		internal statement number
1 byte	1 byte	2 bytes

These two words refer to B(I).

The output from Phase 15 for this statement is:

CALL adjective code	real subprogram		a (OAJK)	
LE	register #6	variable	a (C)	
ME	register #6	constant	a (2)	The arithmetic expression is calculated first: the result is then stored in a work area.
AE	register #6	variable	a (D)	
STE	register #6	work area	work area	
00	0	0	0003	There are three arguments.
SAOP	register #3	register #3	Offset	These three words refer to B(I). A general register is assigned as a work register.
p (subscript			p (dimension)	
LA	B	type	a (B)	
,	real type	work area	a (work area)	The expression was calculated above and placed in a work area.
end mark			internal statement number	
1 byte	1 byte		2 bytes	

Exit: The FOSCAN routine passes control to various routines associated with certain operators encountered by Phase 15. These routines are: ADD, MULT, EXPON, UMINUS, UPLUS, RTPRN, LFTPRN, CALL, FUNC, EQUALS, INVOP, COMMA, and END.

DO Routine and Subroutine DVARCK: Chart FC

DO Routine

The DO routine examines the DO statement for a statement number that defines the last statement of the DO loop, for a DO variable not multi-defined, and for a DO loop nested to a depth no greater than 25.

ENTRANCE: The DO routine receives control from the PRESCN routine when a DO or implied DO adjective code is detected or to complete processing of a computed GO TO.

OPERATION: The referenced statement number and the DO variable are entered into the DO table. The statement number is checked to ensure that it is defined. If not, an error condition is noted. Subroutine DVARCK is called to check that this DO statement does not redefine an existing DO variable, or to check for a level of nesting greater than 25. If either condition occurs, an error is noted and control is passed to the MSGNEM/MSGMEM/MSG routine to eliminate the following text words of the statement.

If no error exists, subroutine PINOUT is used to place the input word onto the output data set. This portion of the processing is also used by the COMP GO TO and BEGIO routines.

EXIT: The DO routine passes control to the MSGNEM/MSGMEM/MSG routine when an end mark is detected.

SUBROUTINE CALLED: During execution the DO routine calls subroutines LAB and PINOUT.

Subroutine DVARCK

Subroutine DVARCK processes a DO or implied DO variable.

ENTRANCE: Subroutine DVARCK is entered from the BEGIO and DO routines.

OPERATION: Subroutine DVARCK checks for two possible errors. The first is a DO variable that has been defined as the DO variable for a DO loop within which this DO loop is defined, i.e., multi-definition. If this error is not present, the DO variable is entered into the DO or the IMPLIED DO table. If either table is full, a level of nesting greater than 25 is present and an error is indicated.

EXIT: After execution subroutine DVARCK returns control to the subroutine which called it. If any errors are detected, control passes to the MSGNEM/MSGMEM/MSG routine.

SUBROUTINE CALLED: Subroutine DVARCK calls the ERROR subroutine if any errors are detected.

COMP GO TO, GO TO Routines: Chart FD

COMP GO TO Routine

The COMP GO TO routine checks each statement number used in the computed GO TO.

ENTRANCE: The COMP GO TO routine is entered from the PRESCN routine when a computed GO TO is detected.

OPERATION: The COMP GO TO routine essentially examines the pointer field of a text word in a computed GO TO statement. When a statement number reference is encountered, subroutine LAB determines if that number is defined. Each text word that is processed by the COMP GO TO routine is put into the output data set by subroutine PINOUT.

EXIT: The COMP GO TO routine passes control to the DO routine.

GO TO Routine

The GO TO routine checks the statement number referenced by the GO TO statement.

ENTRANCE: The statement number referenced by the GO TO statement is checked using subroutine LAB to insure that it is defined. The current input word is then written onto the output data set.

EXIT: The GO TO routine passes control to the MSGNEM/MSGMEM/MSG routine to put out the remainder of the statement, including the end mark.

BEGIO Routine: Chart FE

The BEGIO routine processes the input/output lists of READ and WRITE statements.

ENTRANCE: The BEGIO routine is entered from the PRESCN routine when a BEGIO adjective code is detected.

OPERATION: When the statement is checked and an implied DO is found, the DO variable is processed by subroutine DVARCK. As each word of the statement is checked, it is put onto the output data set by subroutine PINOUT.

EXIT: When the end mark is detected, control is passed to the MSGNEM/MSGMEM/MSG routine.

SUBROUTINES CALLED: The BEGIO routine calls subroutines DVARCK and PINOUT.

ERWNEM, SKIP, MSGNEM/MSGMEM/MSG, INVOP Routines: Chart FF

These routines control the processing of miscellaneous text words in Phase 15.

ERWNEM Routine

The ERWNEM routine processes three adjective codes encountered in the PRESCN routine.

ENTRANCE: The ERWNEM routine is entered from the PRESCN routine when an end mark, ERROR, or WARNING adjective code is recognized in the PRESCN routine.

OPERATION: Subroutine PINOUT is referenced to put the word containing the end mark, WARNING, or ERROR adjective code into the output buffer. Subroutine PINOUT also updates both the input and output pointers.

EXIT: The ERWNEM routine returns control to the PRESCN routine.

SUBROUTINES CALLED: During execution, the ERWNEM routine calls subroutine PINOUT.

SKIP Routine

The SKIP routine begins the processing of the Return and Continue statements.

ENTRANCE: The SKIP routine is entered from the PRESCN routine when a RETURN or CONTINUE adjective code is detected.

OPERATION: Subroutine PINOUT is referenced to put one word onto the output buffer, and to update both the unput and output pointers.

EXIT: The SKIP routine passes control to the MSGNEM/MSGMEM/MSG routine to complete the processing of the statement.

SUBROUTINE CALLED: During execution the SKIP routine calls subroutine PINOUT.

MSGNEM/MSGMEM/MSG Routine

The MSGNEM/MSGMEM/MSG routine processes the remaining text words of a statement, and puts any ERROR and WARNING messages and/or any necessary ENDDO text onto the output data set.

ENTRANCE: The MSGNEM/MSGMEM/MSG routine is entered from all statement processing routines except the ERWNEM and LABEL DEF Routines.

OPERATION: The MSGNEM/MSGMEM/MSG routine has three entry points for the current word of the statement text. It is entered at MSGNEM if the current word does not have an end mark adjective code, MSGMEM if there may be an end mark adjective code, and MSG if there is an end mark adjective code.

At MSGNEM, the current input word is eliminated by updating the input buffer pointer without moving the current word to the output buffer area. Control is then passed to entry point MSGMEM to check the adjective code. If it is not an end mark, the text word is eliminated using MSGNEM and the next word is processed by MSGMEM. When an end mark indicator is found, control passes to entry point MSG to move the input word to the output data set. The output buffer pointer is updated; the input buffer pointer is not updated.

If a statement represented the end of one or several DO loops, ENDDO text words are generated for each loop. Any error or warning messages generated for this statement by Phase 15 are now written onto the output data set using the ISN found in the address pointer field of the text word whose adjective code is an end mark. The word containing the ISN can now be eliminated by updating the input buffer pointer.

EXIT: The MSGMEM/MSGNEM/MSG routine returns control to the PRESCN routine.

SUBROUTINES CALLED: The MSGNEM/MSGMEM/MSG routine calls subroutines ININ and INOUT.

INVOP Routine

The INVOP routine processes invalid adjective codes detected by the PRESCN routine.

ENTRANCE: The INVOP routine is entered from the PRESCN routine when an invalid adjective code is detected.

OPERATION: The INVOP routine calls subroutine ERROR to process an invalid adjective code message.

EXIT: The INVOP routine passes control to the MSGNEM/MSGMEM/MSG routine.

SUBROUTINES CALLED: The INVOP routine calls subroutine ERROR.

MOPUP Routine: Chart FG

The MOPUP routine performs the final processing for the phase.

ENTRANCE: The MOPUP routine is entered when the END statement is detected by the PRESCAN routine.

CONSIDERATION: The END statement may be followed on the input data set by error or warning message text words. The final text word contains all zeros. These text words, followed by an end of data set, are written on the output work tape.

During the phase, whenever it is necessary to utilize a register that is not available, an instruction is generated to load the contents of the register into a work area. A count is kept of the maximum size of the work area required at object time. This count is used to update the location counter in the FORTRAN communications area.

If the output does not exceed one output buffer, it is not written onto the tape but remains within main storage. A bit is set in the communications area to indicate the location of the Phase 15 output data set to Phase 20.

OPERATION: The final text words are written onto the output data set. An additional error message is written onto the output data set if a FUNCTION subprogram is being compiled, but the function name has not been defined.

The location counter is updated by the maximum size of the work area, and the work tapes are rewound.

EXIT: The MOPUP routine passes control to the FORTRAN System Director which passes control to Phase 20.

ADD Routine: Chart FH

The ADD routine prepares text referring to the ADD, SUBTRACT, MULTIPLY, and DIVIDE operators for the COMPILE routine. It determines which operands are registers and then, if necessary, interchanges the operands or assigns a register to the left operand.

CONSIDERATION: The word in the operations table which was forced by the FOSCAN routine is considered the current text word. Its address pointer field references the right operand. The pointer field of the preceding word references the left operand. The operator for both operands is in the adjective code field of the current word in the operations table.

ENTRANCE: The ADD routine is entered from the forcing scan of the FOSCAN routine when a text word, whose adjective code refers to an ADD or SUBTRACT operator is forced. The ADD routine is also entered from the MULT routine.

OPERATION: The ADD or SUBTRACT operands are checked for mixed-mode and legality of the symbol using subroutines MODE and SYMBOL, respectively.

The operands for ADD, SUBTRACT, MULTIPLY, and DIVIDE operators are processed together. If both the left and right operands are in registers, control is passed to the COMPILE routine to process an RR instruction.

If the right operand is not in a register but the left operand is, control is passed to the COMPILE routine to process an RX instruction. (The COMPILE routine, when entered for an RX instruction, assumes that the left operand is a register and the right operand is not. The text word is modified to meet this condition.)

If the right operand is in a register and the left operand is not, the RX condition could be met by interchanging the operands. This interchange is made for the operands of ADD and MULTIPLY operators because their order of operands is immaterial. Interchanging cannot be made for the SUBTRACT and DIVIDE operators.

If neither operand is in a register, or the right operand alone is in a register but is not the operand of an ADD or MULTIPLY operator, a register is assigned to the left operand. Prior to this assignment, the left operand is checked for a subscripted variable. If one is present, subroutine MVSBXX is called to process the subscripted variable.

Next, subroutine FINDR is referenced to assign a register to the left operand. All other subscript occurrences within operands are processed by Phase 15 in the COMPILE routine. After a register is assigned to the left operand, subroutine LOADR1 is called to generate an instruction to load the left operand into a register at object time. Now, the left operand can be referred to as a register.

EXIT: The ADD routine passes control to the COMPILE routine either to process an RR or an RX instruction.

SUBROUTINES CALLED: The ADD routine calls subroutines SYMBOL, MODE, MVSBXX, FINDR, and LOADR1.

MULT Routine: Chart FI

The MULT routine aids in the processing of the operands of the multiply and divide instructions.

ENTRANCE: The MULT routine is entered from the FOSCAN routine when the forcing scan forces a word out of the operations table whose adjective code indicates a multiply or divide instruction.

CONSIDERATION: The left and right operands of a multiply instruction are the multiplicand and the multiplier, respectively.

In multiplication, it is possible to interchange the multiplier and the multiplicand without changing the value of the product.

The multiply instruction for integer quantities requires that the multiplicand be in an odd register. The even register which precedes the multiplicand (the odd register) must be made available unless it contains the multiplier. Both even and odd registers are required for the multiplication procedure.

The multiply instruction for real quantities requires that at least the multiplicand (the left operand) be in a register.

OPERATION: The operands of a multiply or divide operation are checked for symbol validity by subroutine SYMBOL. Subroutine MODE assures that the modes agree. If the operands are real, control is passed to the ADD routine to place the left operand in a register.

For integer quantities which are operands of a divide operation, control is passed to the DIV routine. The MULT routine completes the processing for integer quantities which are the operands of a multiply instruction. If both operands are in registers, they are manipulated so that the left operand is in an odd register. Control is then passed to the COMPILE routine to process an RR instruction.

If neither operand is in a register, the left operand is placed into an odd register and the even register preceding the odd register is made available. Control is then passed to the COMPILE routine to process an RX instruction.

If either operand is in a register, the operands are switched and processed so that the left operand is in an odd register. Control is then passed to the COMPILE routine to process an RX instruction.

During the processing, the various registers are checked for availability by subroutine CHCKGR. If a register is required, but is not currently available, its contents are placed into a work area by subroutine CHCKGR.

EXIT: Control is passed to the ADD routine for real operands. Control is passed to the DIV routine for integer operands of a divide operator.

For integer operands of a multiply operator, control is passed to the COMPILE routine. If both fixed point operands of a multiply instruction are in a register at the completion of the MULT routine, control is passed to the COMPILE routine to process an RR instruction. If only the left integer operand of a multiply instruction is in a register at the completion of the MULT routine, control is passed to the COMPILE routine to process an RX instruction.

SUBROUTINES CALLED: The MULT routine calls subroutines MVSBXX, INOUT, FREER, CHCKGR, MODE, SYMBOL, and LOADR1.

DIV Routine: Chart FJ

The DIV routine processes integer operands of a divide operation.

ENTRANCE: The DIV routine is entered from the MULT routine.

CONSIDERATION: For integer division, the dividend must be in an even-odd register pair. The dividend of a divide operation is represented by the left operand; the divisor, by the right operand.

OPERATION: If the dividend is already in a register, the appropriate even or odd register is made available.

If the dividend is not in a register, the operands are processed for the presence of subscript expressions. Subroutine MVSBRX processes the left operand, while subroutine MVSBXX processes the left and/or right operands. The dividend (left operand) is then placed into an even-odd register pair.

EXIT: The DIV routine passes control to the COMPILE routine at one of three points depending upon whether the right operand is a register, is subscripted, or is neither subscripted nor a register.

SUBROUTINES CALLED: The DIV routine calls subroutines CHCKGR, FREER, MVSBXX, MVSBRX, and LOADR1.

EXPON Routine: Chart FK

The EXPON routine processes the text word whose adjective code indicates exponentiation.

ENTRANCE: The EXPON routine is entered from the FOSCAN routine when a word containing an exponentiation adjective code is forced out of the operation table by the forcing scan.

CONSIDERATION: The phrase A**B appears in the text input to Phase 15 as:

adjective code	real variable	a (A)
**	real variable	a (B)

1 byte 1 byte 2 bytes

The second word is forced out by the forcing scan. The adjective code for this word (**) indicates exponentiation.

OPERATION: The base and exponent are checked for validity using subroutines SYMBOL and CKARG, respectively. If the base is an integer number while the exponent is a real number, subroutine MODE converts the base to a real number.

Exponentiation requires library subroutines; therefore, specific registers are required. (This is identical to the processing done in the COMMA routine for an external subprogram reference.)

The mode of the base and the exponent determine the library subprogram to be called. An internal code to indicate the subprogram name is entered in the pointer field of the text word with an exponentiation adjective code.

The argument count in the FORTRAN communications area is incremented by 2 since exponentiation requires exactly two arguments. The argument list, consisting of the base and exponent, is written out.

If the exponentiation appears within an ASF definition the contents of register 14 (the linkage register) and register 9 (the ASF argument register) are saved. The instructions to restore registers 9 and 14 are generated on return from any function call.

EXIT: The EXPON routine passes control to the FOSCAN routine.

SUBROUTINES CALLED: The EXPON routine calls subroutines SYMBOL, MODE, and CKARG.

UMINUS, UPLUS, RTPRN Routines: Chart FL

UMINUS Routine

The UMINUS routine processes the operand of a unary minus and generates an instruction to reverse the sign of the operand.

ENTRANCE: The UMINUS routine is entered from the forcing scan of the FOSCAN routine when a text word whose adjective code refers to a unary minus is forced out of the operations table.

OPERATION: The operand of the unary minus is checked for validity by subroutine TYPE. If the operand is not a register, subroutine MVSBRX is called to check for and process a subscript expression in the operand. The operand is then loaded into a register.

When the operand is in a register, an instruction is generated to complement the register and complete processing the unary minus.

EXIT: The UMINUS routine passes control to the FOSCAN routine.

SUBROUTINES CALLED: The UMINUS routine calls subroutine TYPE, INOUT, FINDR, and LOADR1.

UPLUS Routine

The UPLUS routine deletes the unary plus adjective code word.

ENTRANCE: This routine is entered from the forcing scan of the FOSCAN routine when a unary plus adjective code is forced.

CONSIDERATION: The unary plus serves no logical function; however, it is not considered an error.

A unary plus only occurs in the text following a word whose adjective code represents an equal sign or a left parenthesis.

OPERATION: The word containing a unary plus is deleted from the input text. The information in the address pointer and mode/type fields is moved into those fields in the preceding word of the operation table.

The text would appear in the operation table as:

= or (	00	0000
unary +	mode/type	a (operand)

1 byte 1 byte 2 bytes

The text would be revised in the operation table to:

= or (	mode/type	a (operand)
1 byte	1 byte	2 bytes

EXIT: The UPLUS routine passes control to the FOSCAN routine.

RTPRN Routine

The RTPRN routine is only entered for an error condition and checks for a further error.

ENTRANCE: The RTPRN routine is entered from the forcing scan of the FOSCAN routine when a text word with a right parenthesis adjective code is forced out of the operations table.

CONSIDERATION: A right parenthesis forced out of the operations table by the forcing scan is an error. If the right parenthesis is followed by an operator, Phase 10 has generated a warning message.

OPERATION: If the right parenthesis is not followed by an operator, Phase 15 gives an illegal delimiter or symbol missing error message.

EXIT: The RTPRN routine returns control to the FOSCAN routine.

SUBROUTINES CALLED: The RTPRN routine calls subroutine ERROR.

LFTPRN Routine: Chart FM

The LFTPRN routine processes a left parenthesis.

ENTRANCE: The LFTPRN routine is entered from the FOSCAN routine when a text word with a left parenthesis adjective code is forced out of the operations table by the forcing scan.

CONSIDERATION: The three uses of a left parenthesis considered by the LFTPRN routine are: in a CALL statement, an arithmetic IF statement, or as a regular left parenthesis (i.e., when used to change the hierarchy of operations).

OPERATION: If a left parenthesis is part of a CALL statement, the argument is checked for validity and the argument count is set to 1. Control is passed to the COMMA routine to continue processing.

If a left parenthesis is part of an arithmetic IF statement, the parenthesized expression is checked to determine the following:

1. The symbol used in the arithmetic expression is valid.
2. The expression result is in a register.
3. The operation that placed the resultant value of the arithmetic expression in a register sets the condition code so the proper branch can be taken as a result of that value.

An IF forcing text word is entered into the output, it has the following format:

IF forcing adjective code	mode	indicator	register number
1 byte	1 byte		2 bytes

Register number indicates the register that contains the value of the arithmetic expression; indicator denotes whether the condition code is set or not.

If this is a regular left parenthesis used to change hierarchy, it is deleted from the operations table.

EXIT: There are three normal exits for the LFTPRN routine, depending upon the use of the left parenthesis. Within the CALL statement, control passes to the COMMA routine; within the arithmetic IF statement, control passes to the ARTHIF routine; for a regular left parenthesis, control is returned to the FOSCAN routine.

An error condition exists and control passes to the MSGNEM/MSGMEM/MSG routine if the left parenthesis is not one of the three normal occurrences, if there is an invalid symbol in an arithmetic IF, or if the left parenthesis is not forced by a right parenthesis indicating an invalid statement.

SUBROUTINES CALLED: The LFTPRN routine calls subroutines CKARG and ERROR.

FUNC, CALL, and END Routines: Chart FN

FUNC Routine

The FUNC routine processes one-argument functions.

ENTRANCE: The FUNC routine is entered from the FOSCAN routine when the forcing scan forces a text word with a FUNC adjective code out of the operations table.

CONSIDERATION: In-line, external, and ASF functions may have one argument.

OPERATION: The in-line functions are processed separately by the INLIN1 routine. An argument for ASF and external functions is processed by checking the validity of the argument using subroutine CKARG and setting up an argument count of 1.

EXIT: Control is passed to the COMMA routine to complete the processing of a one-argument function.

SUBROUTINES CALLED: The FUNC routine calls subroutine CKARG.

CALL Routine

The CALL routine processes the CALL statement.

ENTRANCE: The CALL routine is entered from the FOSCAN routine when a text word whose adjective code is a CALL forcing adjective code is forced out of the operations table.

CONSIDERATION: If a CALL statement has no arguments, the CALL forcing adjective code word indicates that a CALL is to be processed. If a CALL statement has arguments, either the LFTPRN, FUNC, or COMMA routine has determined that a CALL is to be processed and has processed it.

The CALL forcing adjective code word may take one of two forms. If the CALL has arguments, the word appears as:

CALL forcing adjective code	mode	0000
1 byte	1 byte	2 bytes

If the CALL has no arguments, the word appears as:

CALL forcing adjective code	mode	address of CALL subroutine
1 byte	1 byte	2 bytes

OPERATION: The address pointer field is examined to determine if the CALL has been processed. If the CALL has arguments, it has been processed and an exit is taken.

If the CALL has not been processed, it is a CALL with no arguments. There are two words written onto the output buffer, the first indicates a CALL and the subroutine to which the CALL is made; the second indicates no arguments. These two words appear as follows:

CALL adjective code	mode/type		a (CALL subroutine)
argument count adjective code	0	0	00 00
1 byte	1 byte		2 bytes

EXIT: The CALL routine passes control to the MSGNEM/MSGMEM/MSG routine to move the end mark to the output buffer.

END Routine

The END routine determines if the arithmetic IF, arithmetic, and ASF statements were processed.

ENTRANCE: The END routine is entered from the FOSCAN routine when a text word within Arith, Arith IF, or ASF adjective code is forced out of the operations table.

OPERATION: The END routine examines the address pointer field. If it is 0, the statement has been processed and the exit is taken. If it is not 0, the statement has not been processed, and an illegal statement error message is given. The exit is then taken.

EXIT: The END routine passes control to the MSGNEM/MSGMEM/MSG routine.

SUBROUTINE CALLED: The END routine calls subroutine ERROR.

EQUALS Routine: Chart FO

The EQUALS routine processes a text word containing an equal adjective code.

ENTRANCE: The EQUALS routine is entered from the FOSCAN routine when the forcing scan forces a word with an equal adjective code out of the operations table.

CONSIDERATION: The arithmetic and ASF definition statements contain an equal adjective code. The result of an ASF definition must be in register 0 (general or floating-point), while the result of an arithmetic statement is in the field which represents the symbol to the left of the equal sign in an arithmetic statement (the resultant field).

OPERATION: If the text word containing the equal adjective code is part of an arithmetic statement, the first part of the processing checks the resultant field. If an invalid symbol is represented in the resultant field, an error message is given, and control is passed to the MSGNEM/MSGMEM/MSG routine to eliminate the remainder of the statement.

If the resultant field is valid, subroutine TYPE is referenced to check the right operand which represents the result of the computations on the right side of the equal sign. If the right operand is invalid, an error message is given and control is passed to the MSGNEM/MSGMEM/MSG routine to eliminate the remainder of the statement. If the right operand is valid, the MODE routine is called to insure that the right operand and the resultant field modes are the same.

Because the mode of the resultant field determines the mode of the value resulting from the arithmetic operation, the mode of the right operand, if different, is converted to the resultant field mode. An instruction to store the right operand in the resultant field is then generated.

If the ASF switch, which is set on at the beginning of ASF definition statement processing in subroutine FOSCAN, indicates an ASF definition, different processing is followed. The type of the right operand is checked for an error by subroutine TYPE. If an error is found, an error message is given and the rest of the statement is eliminated using the MSGNEM/MSGMEM/MSG routine. Subroutine MODE compares the modes of the ASF and the right operand if the right operand is valid.

The result of the ASF must be in general register 0 if the ASF is an integer function and in floating point register 0 if it is a real function. If the result is not in the correct register, a load instruction is generated to put the result into the correct register.

The final processing for an equal adjective code word in an ASF definition is the generation of the return instruction. This allows the ASF coding to return to that portion of coding which referenced the ASF during the object program execution.

EXIT: At the completion of the EQUALS routine, control is returned to the FOSCAN routine. If errors were detected during the processing, control passes to the MSGNEM/MSGMEM/MSG routine.

SUBROUTINES CALLED: The EQUALS routine calls subroutines ERROR, TYPE, and MODE.

COMMA Routine: Chart FP

The COMMA routine processes an argument list.

ENTRANCE: The COMMA routine is entered from the FOSCAN routine when the forcing scan forces a word whose adjective code refers to a comma out of the operations table.

CONSIDERATION: A comma occurs in the input text only when an argument list is to be processed. This argument list may be part of an in-line function, external function, or ASF call. An argument list is set off in the original FORTRAN statement by a left parenthesis preceding the list and a following right parenthesis.

The processing for an ASF definition and call is basically the same as the external function call. The processing for an in-line function differs and is in the INLIN2 routine.

For a function call, general registers 0 and 1 and all floating-point registers in use are saved.

If an ASF definition is being processed, register 14 is saved before and restored after the function call. Register 14 is the linkage register for an ASF.

OPERATION: Each argument represented is checked by subroutine CKARG for validity. A count is kept of the number of arguments used in the argument list. This count is added to the count of arguments in a

counter within the FORTRAN communications area.

The required registers are examined for availability. If they are not available, subroutine SAVER is referenced to store the contents of the registers in a work area.

Certain error checks are made. If the function name is invalid, an error message is given. If an argument is followed by an end mark, a warning message is given to indicate a missing right parenthesis. If neither a right parenthesis nor an end mark follows the argument, an error message indicating an illegal delimiter is given.

EXIT: The COMMA routine passes control to the forcing scan in the FOSCAN routine.

SUBROUTINE CALLED: The COMMA routine calls subroutines CKARG, ERROR, and WARN.

LABEL DEF Routine, Subroutine LAB: Chart FQ

LABEL DEF Routine

The LABEL DEF routine checks the occurrence of statement numbers used to indicate the end of a DO loop.

ENTRANCE: The LABEL DEF routine is entered from the PRESCN routine when a statement number definition adjective code is encountered.

CONSIDERATION: As each DO definition is encountered, the ending DO loop statement number (the ENDDO) is entered into the DO table along with the DO variable. By checking the current statement number against the latest statement number in the DO table, it can be determined if this ENDDO is properly nested with respect to the other DO loops.

OPERATION: If the statement number definition is not referenced as an ENDDO, the text word is put onto the output data set using subroutine PINOUT. A statement number referenced as an ENDDO is checked against the entries in the DO table. If a DO nesting error is detected, an error message is given and the text word is put onto the output data set using subroutine PINOUT.

If the statement number is a legal ENDDO, an indicator is set for the MSGNEM/MSGMEM/MSG routine to generate the ENDDO word of text. The input word is then put onto the output data set using subroutine PINOUT.

EXIT: The LABEL DEF routine returns control to the PRESCN routine.

SUBROUTINES CALLED: The LABEL DEF routine calls subroutines PINOUT and ERROR.

Subroutine LAB

Subroutine LAB checks for legal statement number references.

ENTRANCE: Subroutine LAB is entered from the GOTO, COMP GOTO, ARITH IF, and DO routines.

OPERATION: A referenced statement number is checked to insure that the statement number has been defined and is not that of a FORMAT statement. Because a branch cannot be made to either an undefined statement number or a FORMAT statement, an error message is printed for each such reference.

EXIT: Subroutine LAB returns control to the subroutine which called it.

SUBROUTINE CALLED: Subroutine LAB calls subroutine ERROR if either error condition is found.

ARITH IF Routine: Chart FR

The ARITH IF routine processes the statement number portion of an arithmetic IF for two errors: undefined statement numbers and an incorrect number of statement numbers specified.

ENTRANCE: The ARITH IF routine is entered from the LFTPRN routine.

CONSIDERATION: Phase 10 may have truncated the text of an arithmetic IF due to various error conditions encountered (i.e., invalid delimiters or statement numbers). If less than three statement numbers are specified by the user, the input text appears as if truncated due to error conditions.

OPERATION: Each statement number is examined by subroutine LAB. If the statement number has been defined, it is put into the output text of Phase 15 and the next input text word is accessed using subroutine PINOUT. If the number was not defined, subroutine LAB does not return to the ARITH IF routine.

As each statement number is being processed, a count is kept. If it is other than three, an error message is entered for the statement. If the statement was trun-

cated by Phase 10, an error message is not given for the error condition represented by less than three statement numbers.

EXIT: The ARITH IF routine takes one of three exits: when an end of statement indicator is known to be, may be, or is not present.

SUBROUTINES CALLED: The ARITH IF routine calls subroutines LAB, PINOUT, and ERROR.

COMPILE Routine: Chart FS

The COMPILE routine creates text in the form of RR and RX instructions, moves subscript text from the subscript table to the output data set, increments the output buffer pointer, and decrements the pointer to the operations table.

ENTRANCE: The COMPILE routine is entered from the ADD, MULT, DIV, EXPON, UMINUS, UPLUS, and SYMBOL routines.

OPERATION: When text is to be created in the form of an RR instruction, the left operand becomes the resultant register. The FREER routine is then called to free the right operand register, and processing continues as for an RX instruction.

The right operand is checked. If it is subscripted, it is entered into the output text. If not, that part of the processing is skipped.

The operation code and the register number are entered into the current word of the operations table. The output buffer pointer is then incremented, and the pointer to the operations table is decremented.

EXIT: The COMPILE routine passes control to the FOSCAN routine.

SUBROUTINES CALLED: The COMPILE routine calls subroutines FREER and INOUT.

Subroutines SYMBOL and TYPE: Chart FT

These subroutines process operands for symbol validity.

Subroutine SYMBOL

This subroutine processes the left and right operands of an operator.

ENTRANCE: Subroutine SYMBOL is entered from the ADD, MULT, and EXPON routines.

OPERATIONS: Subroutine SYMBOL calls subroutine TYPE to check for errors in the left and right operands.

If an error is discovered in the right operand, control is passed to the COMPILE routine to decrement the pointer to the operations table. Subroutine TYPE gives an error message for the specific error. By decrementing the pointer to the operations table, the word being processed is deleted. The COMPILE routine passes control to the forcing scan in the FOSCAN routine.

If an error is detected in the left operand by subroutine TYPE, an error message is given. Subroutine SYMBOL eliminates the left operand from the operations table by overlaying it with the right operand. Control is then passed to the COMPILE routine.

EXIT: If there is no error detected in the left or right operand, control is returned to the routine which called subroutine SYMBOL. If there is an error, control is passed to the COMPILE routine.

Subroutine TYPE

Subroutine TYPE checks each symbol used as an operand for errors.

ENTRANCE: Subroutine TYPE is entered from the UMINUS, EQUALS, and INLIN1 routines and subroutine SYMBOL.

CONSIDERATION: A symbol, to be valid, must have a valid type code in the associated text word.

OPERATION: The type code associated with the symbol is examined to determine if the symbol is invalid or if the symbol is multi-defined. If either condition exists, or if the symbol is missing, an error message is given. This error message indicates the specific error encountered.

EXIT: If no errors are discovered, subroutine TYPE returns control to the routine which called it. If an error is detected, control is passed to the error return specified by the routine which called subroutine TYPE.

Subroutines FINDR, FREER, CHCKGR, SAVER, and LOADR1: Chart FU

These five subroutines perform the various register manipulations necessary during the Phase 15 text processing.

CONSIDERATION: There are eight registers (floating-point registers 0, 2, 4, and 6; general registers 0, 1, 2, and 3) available to Phase 15 for assignment as work registers. A record of register availability is kept using a 1-byte indicator. Each of the eight bits of the 1-byte indicator represents a different register (see Figure 46).

General Registers				Floating-point Registers			
3	2	1	0	6	4	2	0

Figure 46. 1-Byte Indicator

The register routines are utilized during the processing of the operands of arithmetic operators. They are also used to insure the availability of registers 0 and 1 for external references.

Subroutine FINDR

Subroutine FINDR finds a register and indicates that it is unavailable.

ENTRANCE: The FINDR routine is called by the ADD routine and subroutine MODE.

OPERATION: Subroutine FINDR accesses the first available general register for integer quantities and the first available floating-point register for real quantities.

EXIT: Subroutine FINDR returns control to the routine which called it.

Subroutine FREER

Subroutine FREER indicates that a register is available.

ENTRANCE: Subroutine FREER is entered from the COMPILE and INLIN2 routines and subroutines CKARG and MODE.

OPERATION: When it is determined which register is to be freed, that bit in the byte which indicates register availability is set to available.

EXIT: Subroutine FREER returns control to the routine or subroutine which called it.

Subroutine CHCKGR

Subroutine CHCKGR accesses a specific general register.

ENTRANCE: Subroutine CHCKGR is entered by the MULT, DIV, INLIN1, and INLIN2 routines and subroutine MODE.

OPERATION: Subroutine CHCKGR is entered to determine the availability of a specific general register.

When that register is found to be available, it is marked occupied and the return is made to the calling routine.

When that register is unavailable, control is passed to subroutine SAVER to make a register available by storing the contents of the specified register in a work area.

EXIT: Subroutine CHCKGR returns control to the routines which called it.

Subroutine SAVER

Subroutine SAVER stores the contents of a specified register into the next available area of the work area.

CONSIDERATION: When a register is required and it is not available, the contents of the register is stored into a work area.

ENTRANCE: Subroutine SAVER is entered from the COMPILE, COMMA, DIV, or FINDR routines.

OPERATION: The latest entry in the operation table is accessed. If it utilizes the register being treated, an instruction to store the contents of that register in the next available area of the work area is generated. This instruction word is moved to the output buffer.

If this entry does not utilize the register in question, the next entry is accessed and examined.

Note that if the register in question is used for a double-precision quantity, the work area is aligned on a double-word boundary.

EXIT: The SAVER subroutine returns control to the routines which called it.

SUBROUTINES CALLED: Subroutine SAVER calls subroutine INOUT.

Subroutine LOADR1

Subroutine LOADR1 enters an operand into a specific register.

ENTRANCE: Subroutine LOADR1 is entered from the ADD, MULT, DIV, EQUAL, LFTPRN, and INLIN1 routines and subroutines MODE and INARG.

OPERATION: Subroutine LOADR1 generates the instruction to load the contents of the left operand, being processed in the routine or subroutine which referenced subroutine LOADR1, into a specific register. The instruction word is then placed into the output buffer.

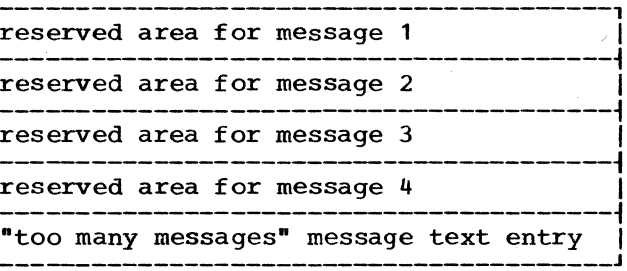
EXIT: Subroutine LOADR1 returns control to the routine which called it.

SUBROUTINES CALLED: Subroutine LOADR1 calls subroutine INOUT.

Subroutine WARN/ERROR: Chart FV

Subroutine WARN/ERROR is called when a warning or error condition is encountered during text processing. It generates the error or warning message text entry for the specific condition encountered.

CONSIDERATION: There is a reserved area for a maximum of four error and warning messages for any given statement. The end of the area contains a message that indicates more than four error and/or warning messages. The area has the following form:



Subroutine WARN/ERROR does not place the internal sequence number in the error and warning message text word save area. The internal sequence number is entered by the MSGNEM/MSGMEM/MSG routine after the word containing the end mark is processed.

ENTRANCE: Subroutine WARN/ERROR has two entry points. It is entered from the COMMA routine and subroutine CKARG, TYPE, and INARG at entry point WARN and from the INVOP, EQUAL, ARITH IF, RTPRN, COMMA, LFTPRN, and LABEL DEF routines and subroutines TYPE, LAB, INARG, and CKARG at entry point ERROR.

OPERATIONS: At entry point WARN, subroutine WARN/ERROR generates a warning message text word using the warning number passed to it by the calling routine. The warning message is then entered into the error and warning message area. If there are already four error and warning message text words, processing for this statement is terminated.

At entry point ERROR, subroutine WARN/ERROR computes the error number from the information passed by the calling routine. The error message text word is then constructed and placed in the error and warning message area. If there are already four error and warning message text words, processing for this statement is terminated.

When a warning condition occurs, control is returned to the calling routine. When an error condition occurs within a statement, control is normally returned to the MSGNEM/MSGMEM/MSG routine to eliminate the rest of that statement. In some instances, the MSGNEM/MSGMEM/MSG routine does not receive control to process any remaining error or warning conditions for the statement. This decision is made prior to calling subroutine WARN/ERROR.

EXIT: Subroutine WARN/ERROR returns control to the calling routine. If there are more than four error and warning messages detected in Phase 15, control is passed to the MSGNEM/MSGMEM/MSG routine.

Subroutines PINOUT, ININ, INOUT: Chart FW

These three subroutines alone or in combination perform the input/output operations for Phase 15.

Subroutine PINOUT

Subroutine PINOUT performs both input and output functions.

ENTRANCE: Subroutine PINOUT is entered from any Phase 15 routine whenever the current input word is to be put out.

OPERATION: Subroutine PINOUT moves the current input word to the output buffer. Subroutines ININ and INOUT are then called to update the input and output pointers respectively.

EXIT: Subroutine PINOUT returns control to the routine which called it.

SUBROUTINES CALLED: Subroutine PINOUT calls subroutines ININ and INOUT.

Subroutine ININ

Subroutine ININ updates the input buffer pointer and refills a buffer if necessary.

ENTRANCE: Subroutine ININ is called by subroutine PINOUT and the MSGNEM/MSGMEM/MSG routine.

CONSIDERATION: There are two buffers used for input.

OPERATION: The input buffer pointer is updated. A check for the end of the buffer is then made. If the end of the buffer has been reached, the buffer is refilled and the other buffer is made current.

Subroutine INOUT

Subroutine INOUT updates the output buffer pointer and empties a buffer if necessary.

ENTRANCE: Subroutine INOUT is entered from subroutines PINOUT, MSGNEM/MSGMEM/MSG, and MVSBXX, and the DIV, MULT, UMINUS, and save register routines.

CONSIDERATION: There are two output buffers.

OPERATION: The output buffer is updated and the end of the buffer is checked. If it has been reached, the buffer is written out and the other buffer is made current.

EXIT: Subroutine INOUT returns control to the subroutine or routine which called it.

Subroutine MODE: Chart FX

Subroutine MODE checks two operands and changes them, if necessary, so that both are the same mode.

ENTRANCE: Subroutine MODE is entered from the ADD, EXPON, EQUAL, INLIN1, and MULT routines.

CONSIDERATION: A hierarchy of modes exists (double-precision, real, and integer) with double-precision being the highest.

OPERATION: Subroutine MODE determines if there is a difference in the modes of the two operands under consideration. If there is, an appropriate in-line call word is generated to change the operand whose mode is lower in the hierarchy of modes to the higher mode.

There is one exception to the rule that the mode is changed according to the hierarchy of modes. This exception occurs when the operator is '='. In this case the right operand is adjusted to the left operand regardless of the hierarchy.

When subroutine MODE is entered from INLIN1, only the appropriate function call is generated.

Subroutines MVSBXX and MVSBRX: Chart FY

Subroutine MVSBXX

Subroutine MVSBXX processes a left operand subscripted variable when the right operand might also be a subscripted variable.

ENTRANCE: Subroutine MVSBXX is entered from the ADD, MULT, and DIV routines and subroutine MODE to process an existing left operand subscripted variable.

CONSIDERATION: If the right operand is not a subscripted variable and the left operand is a subscripted variable, the left operand subscripted variable is the last entry in the subscript table.

If the right and left operands are subscripted variables, the right operand subscripted variable is the last entry in the subscript table. The left operand subscripted variable is the next to last entry.

OPERATION: If the left operand is not a subscripted variable, no processing need take place, and an immediate return is made.

If the left operand is a subscripted variable and the right operand is not, the left operand is moved to the output buffer using subroutine INOUT. The pointer to the last entry in the subscript table is updated.

If the left and right operands are subscripted variables, the left operand is moved to the output buffer. After the left operand subscripted variable is moved from the subscript table, the right operand subscripted variable is moved to the place in the table previously occupied by the left operand. This avoids blank areas in the subscript table which would cause errors.

EXIT: Subroutine MVSBXX returns control to the routine which called it.

SUBROUTINE CALLED: During execution subroutine MVSBXX calls subroutine INOUT.

Subroutine MVSBRX

Subroutine MVSBRX is entered when only the left operand may be a subscripted variable. If the left operand is a subscripted variable, it is processed.

ENTRANCE: Subroutine MVSBRX is entered from the EQUALS, UMINUS, and DIV routines and subroutine MODE.

OPERATION: If the left operand is not a subscripted variable, no processing need occur and a return is made.

If the left operand is a subscripted variable, it is moved to the output buffer using subroutine INOUT.

EXIT: Subroutine MVSBRX returns control to the routine which called it.

SUBROUTINE CALLED: Subroutine MVSBRX calls subroutine INOUT.

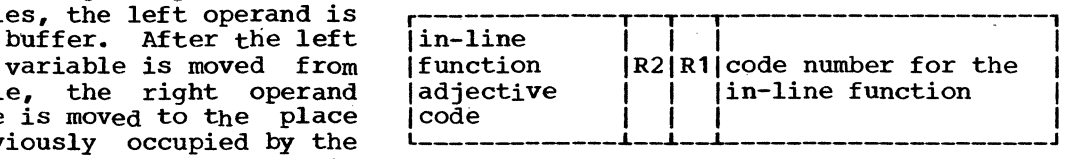
INLIN1 Routine: Chart FZ

The INLIN1 routine processes the single argument in-line functions.

ENTRANCE: The INLIN1 routine is entered from the FUNC routine.

CONSIDERATION: There are eight single-argument in-line functions: IFIX, FLOAT, DFLOAT, SNGL, DBLE, ABS, IABS, and DABS.

OPERATION: Instructions are generated to form the functions for the SNGL and DBLE in-line functions. For the IFIX, FLOAT, DFLOAT, IABS, ABS, and DABS functions, an in-line function call word is generated. This word is in the following format:



Depending upon the specific in-line function, up to three registers are made available by Phase 15. For ABS, IABS, and DABS only an argument register is required. This register is indicated in the R1 field, and the R2 field is zero.

For IFIX, FLOAT, and DFLOAT in-line functions, three registers are required: an argument register, a result register, and a work register. The argument register is indicated in R1, the result register in R2. The work register is R1-1; it is freed by Phase 15. If the mode of the argument is incorrect, an error message is given.

EXIT: The INLIN1 routine passes control to the COMMA routine. If there were errors in the argument, control is passed to the MSGNEM/MSGMEM/MSG routine.

SUBROUTINES CALLED: During execution the INLIN1 routine calls subroutines ERROR, MODE, and INARG.

Subroutine INLIN2: Chart GA

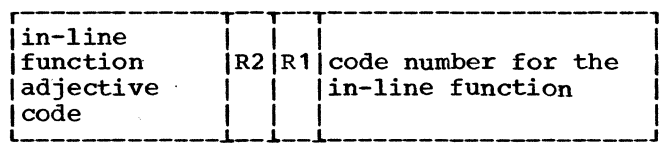
Subroutine INLIN2 processes an in-line function with two arguments.

ENTRANCE: Subroutine INLIN2 is entered from the COMMA routine.

CONSIDERATION: In an in-line function with two arguments, the result of the function is in the register assigned to the first argument. Due to this fact, the register assigned to the second argument should be indicated as free.

OPERATION: Both arguments are checked by subroutine INARG to determine if they are valid and in a register. Subroutine INARG will assign them to a register, if necessary. Then the register assigned to the second argument, R2, is indicated as free, because the R2 register is not used to contain the result.

An in-line function call word is generated. It has the following format:



If the in-line function does not have a code of an in-line function with two arguments, or if the argument is not of the same mode as the function, an error message is given.

EXIT: The INLIN2 routine normally passes control to the COMMA routine. If an error message is given an exception is made, and control is passed to the MSGMEM routine.

SUBROUTINES CALLED: The INLIN2 routine calls subroutines ERROR, INARG, and FREER.

Subroutine CKARG: Chart GB

Subroutine CKARG checks the argument in an external call for validity. It also assures that the arguments have an assigned storage location.

ENTRANCE: Subroutine CKARG receives control from the EXPON, LFTPRN, COMMA, and FUNC routines.

CONSIDERATION: The calling sequence for an external call requires that each argument have an assigned main storage address. An argument has an assigned address if it is a constant, a variable (not an ASF dummy variable or a subscripted variable with variable subscripts), or if it is in a work area.

OPERATION: Entering subroutine CKARG an argument has no assigned address. The argument is assigned to a register, if not already in one, and the contents of the register are stored into a work area (i.e., assigning an address). This procedure is also used to save the contents of a register when that register is to be used for another purpose.

Subscripted variables, which are used as arguments, remain in the subscript form, but are not assigned a main storage address.

If a symbol, which is not a valid argument, is used within an argument list, an error or warning message is given, depending upon the severity. If an argument is missing, an error message is given.

EXIT: The CKARG routine returns control to the routine which called it.

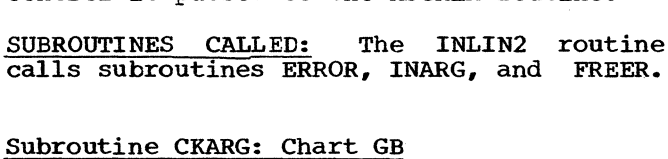
SUBROUTINES CALLED: Subroutine CKARG calls subroutines: FINDR, LOADR1, FREER, ERROR, and WARNING.

Subroutine INARG: Chart GC

Subroutine INARG processes an in-line function argument.

ENTRANCE: Subroutine INARG is entered from the INLIN1 and INLIN2 routines.

CONSIDERATION: An in-line function call word has the following form:



If there is only one argument, it appears in register R1. If there is a second argument, it appears in register R2. If there is no second argument, register R2 is zero.

Due to this form of the in-line function call text word, the arguments must be in a register.

OPERATION: The argument is examined to determine if a register assignment is required. If so, a register is assigned.

If the argument is a subscripted variable or a dummy subscripted variable, the mode/type fields are modified to agree so that, during the remainder of Phase 15, they are indistinguishable. The subscript table is updated to access the next subscript, which may be the next argument.

If an argument is missing or if an invalid argument is used, an error message is given. If an array or a dummy array is used as an argument, a warning message is given.

EXIT: Subroutine INARG returns control to the subroutine which called it. An exception is made if an error message is given. In that case, control is returned to the MSGNEM/MSGMEM/MSG routine.

SUBROUTINES CALLED: Subroutine INARG calls subroutines ERROR, WARNING, FINDR, and LOADR1.

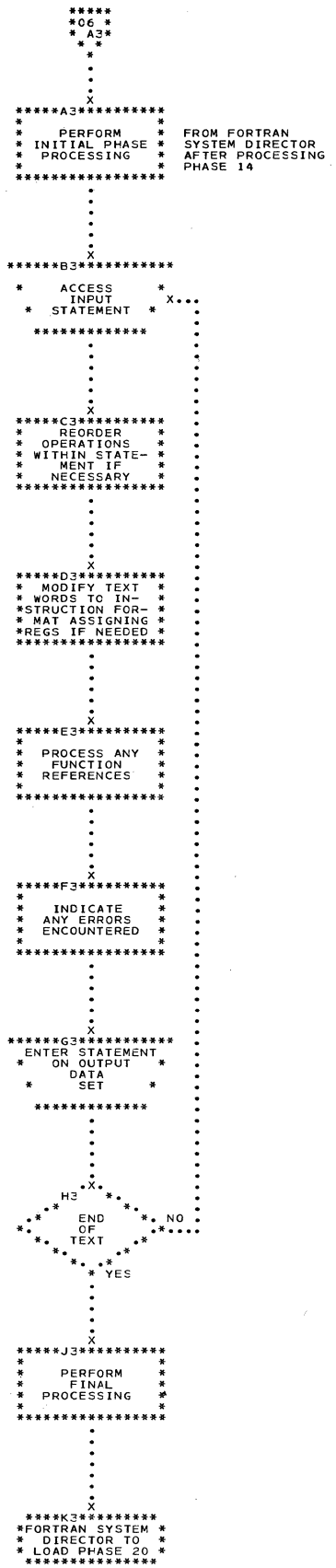


Chart 06. Phase 15 Overall Logic Diagram

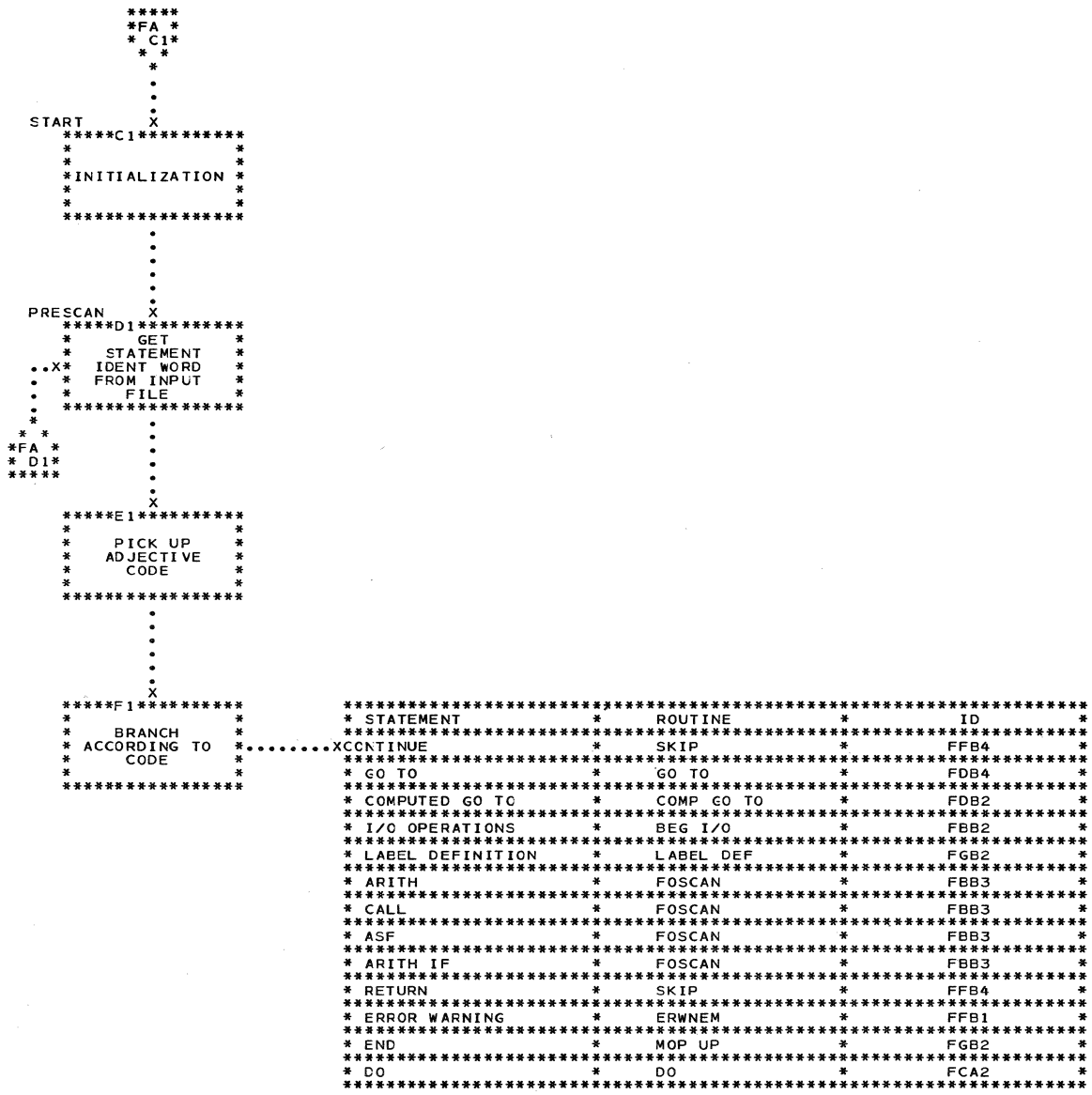
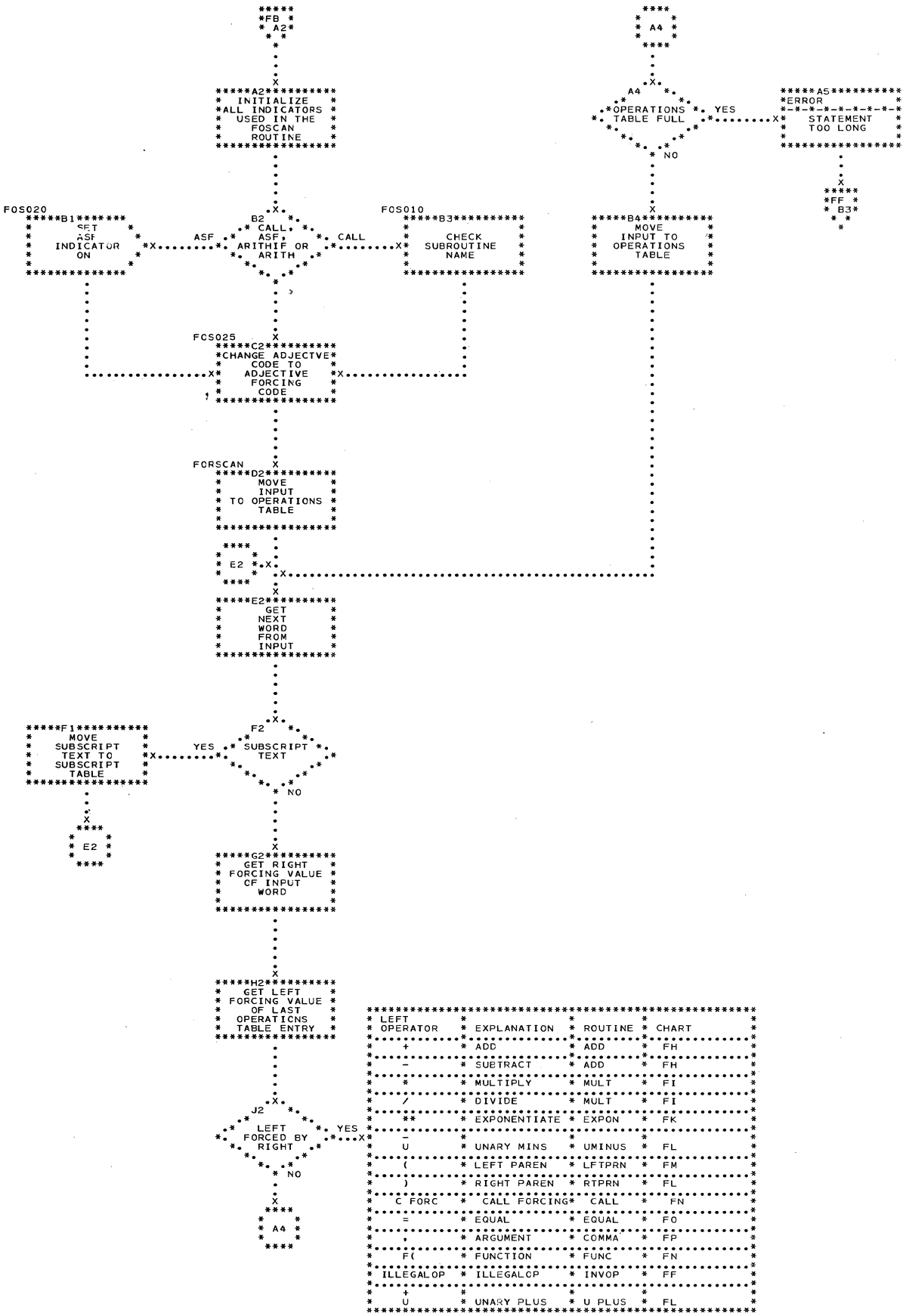


Chart FA. PRESCAN Routine



LEFT OPERATOR	EXPLANATION	ROUTINE	CHART
+	ADD	ADD	FH
-	SUBTRACT	ADD	FH
*	MULTIPLY	MULT	FI
/	DIVIDE	MULT	FI
**	EXPONENTIATE	EXPON	FK
-			
U	UNARY MINS	UMINUS	FL
(	LEFT PAREN	LFTPRN	FM
)	RIGHT PAREN	RTPRN	FL
C FORC	CALL FORCING	CALL	FN
=	EQUAL	EQUAL	FO
,	ARGUMENT	COMMA	FP
F(	FUNCTION	FUNC	FN
ILLEGALOP	ILLEGALOP	INVOP	FF
+			
U	UNARY PLUS	U PLUS	FL

Chart FB. FOSCAN Routine

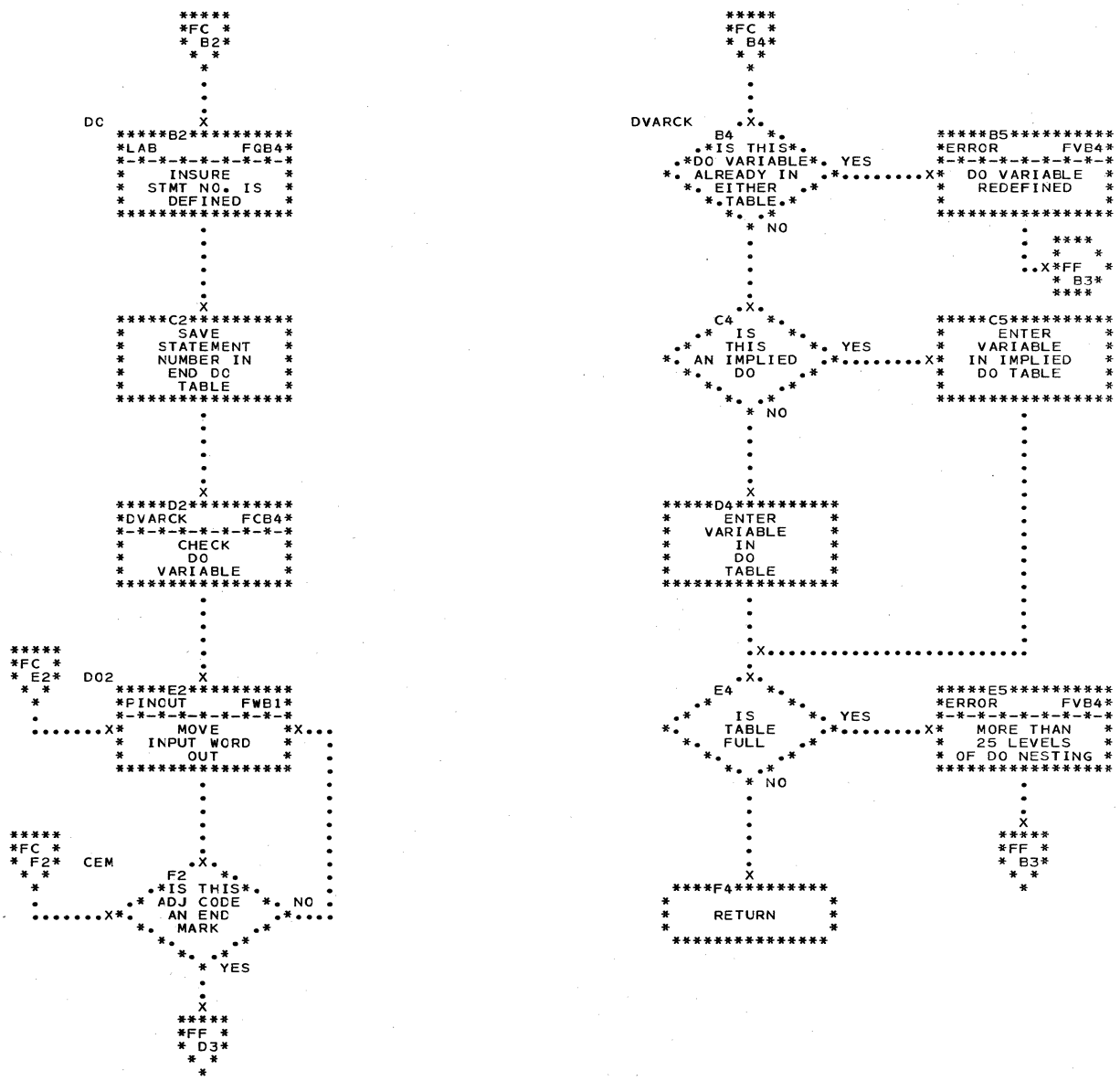


Chart FC. DO Routine, Subroutine DVARK

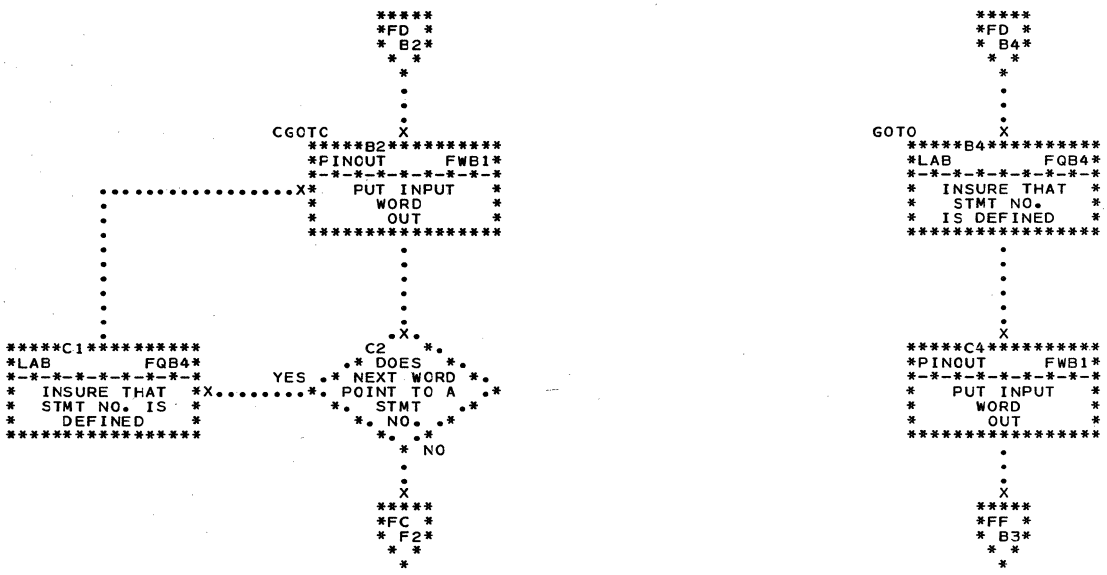


Chart FD. COMP GO TO, GO TO Routines

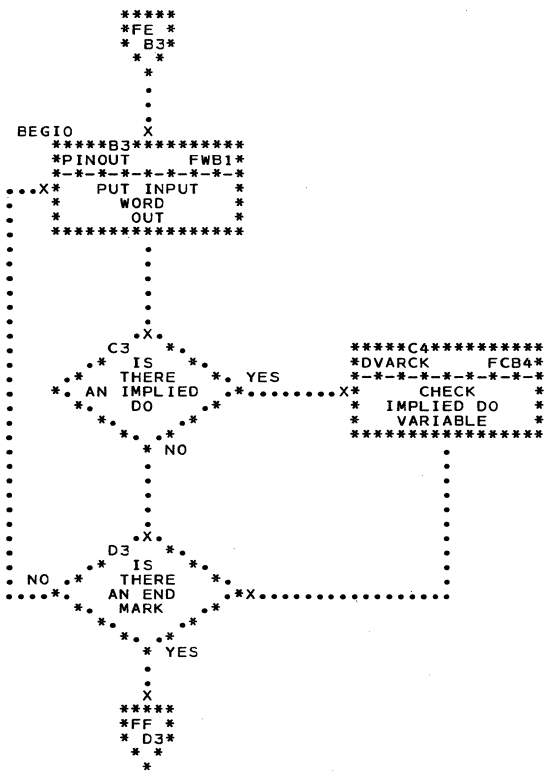


Chart FE. BEGIO Routine

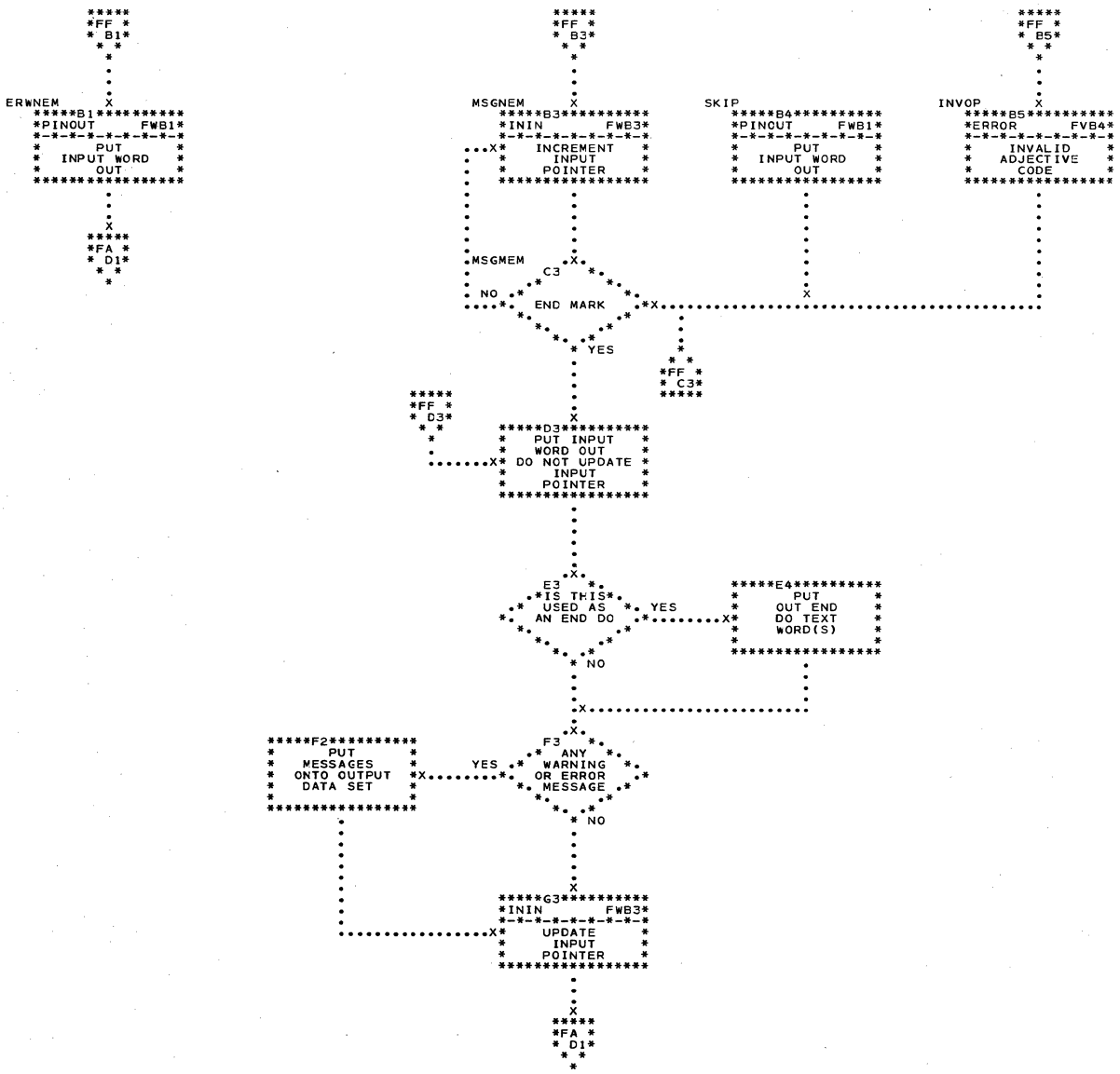


Chart FF. ERWNEM, SKIP/MSGNEM/MSGMEM/MSG/INVOP Routines

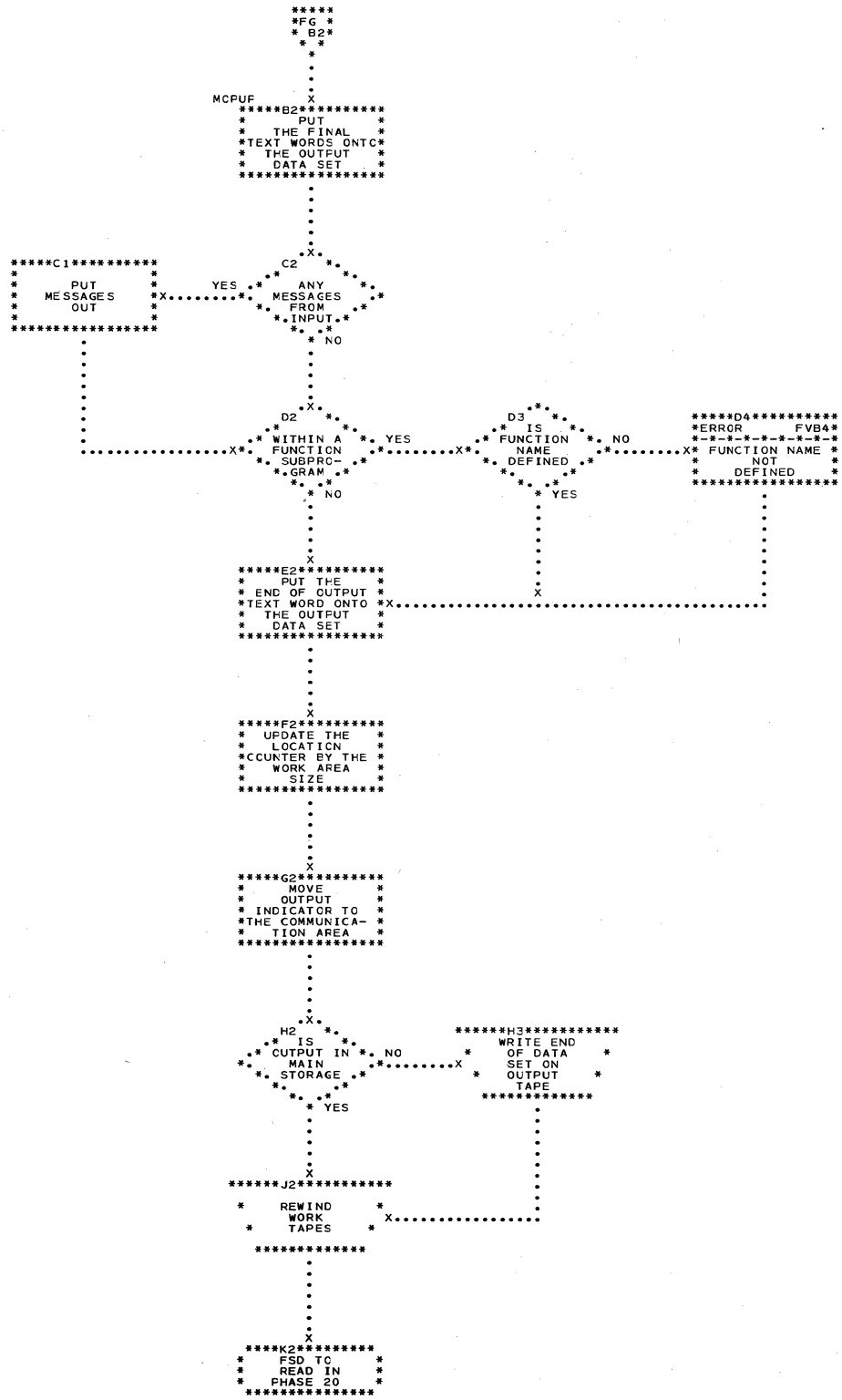


Chart FG. MOPUP Routine

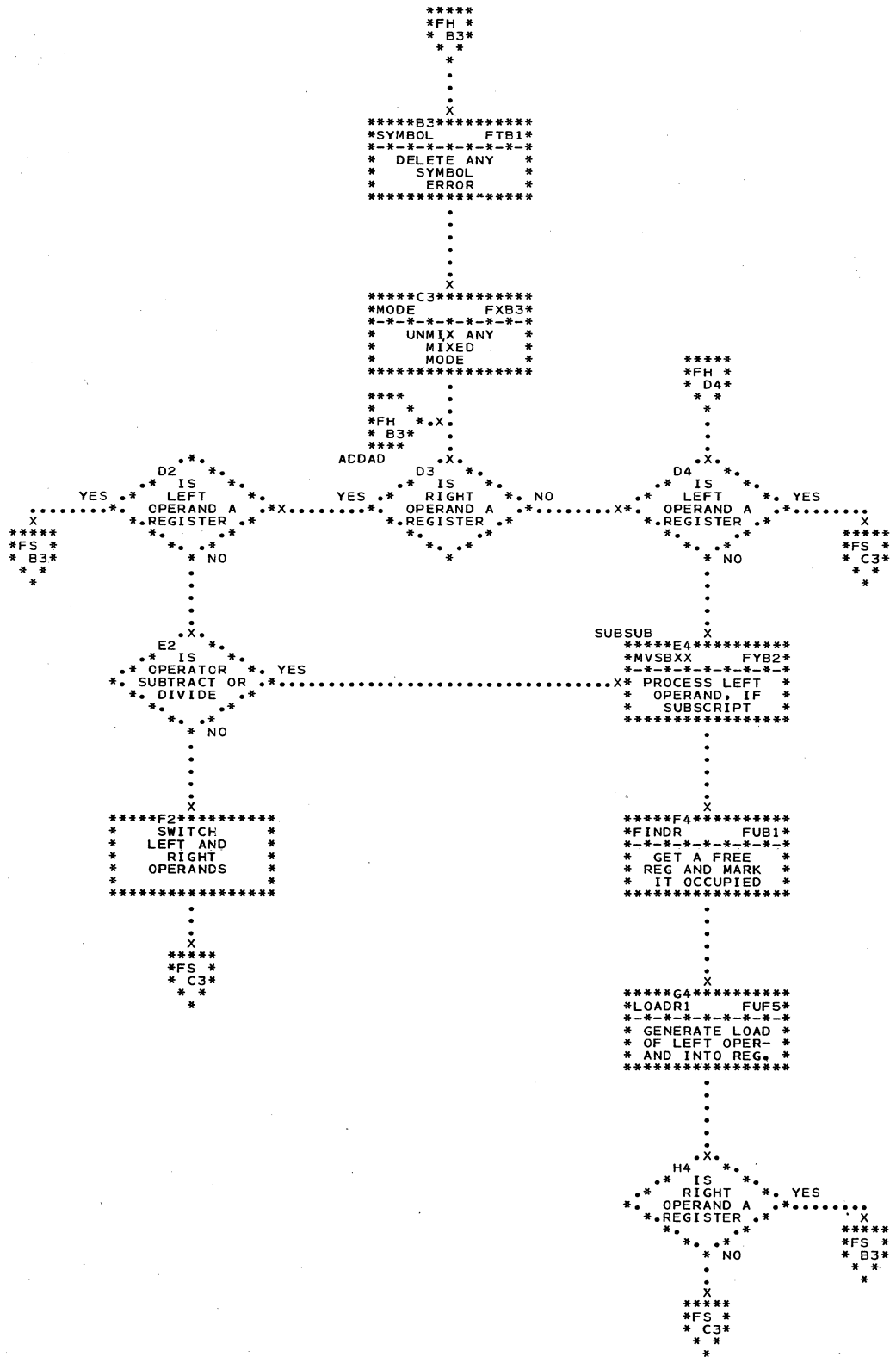


Chart FH. ADD Routine

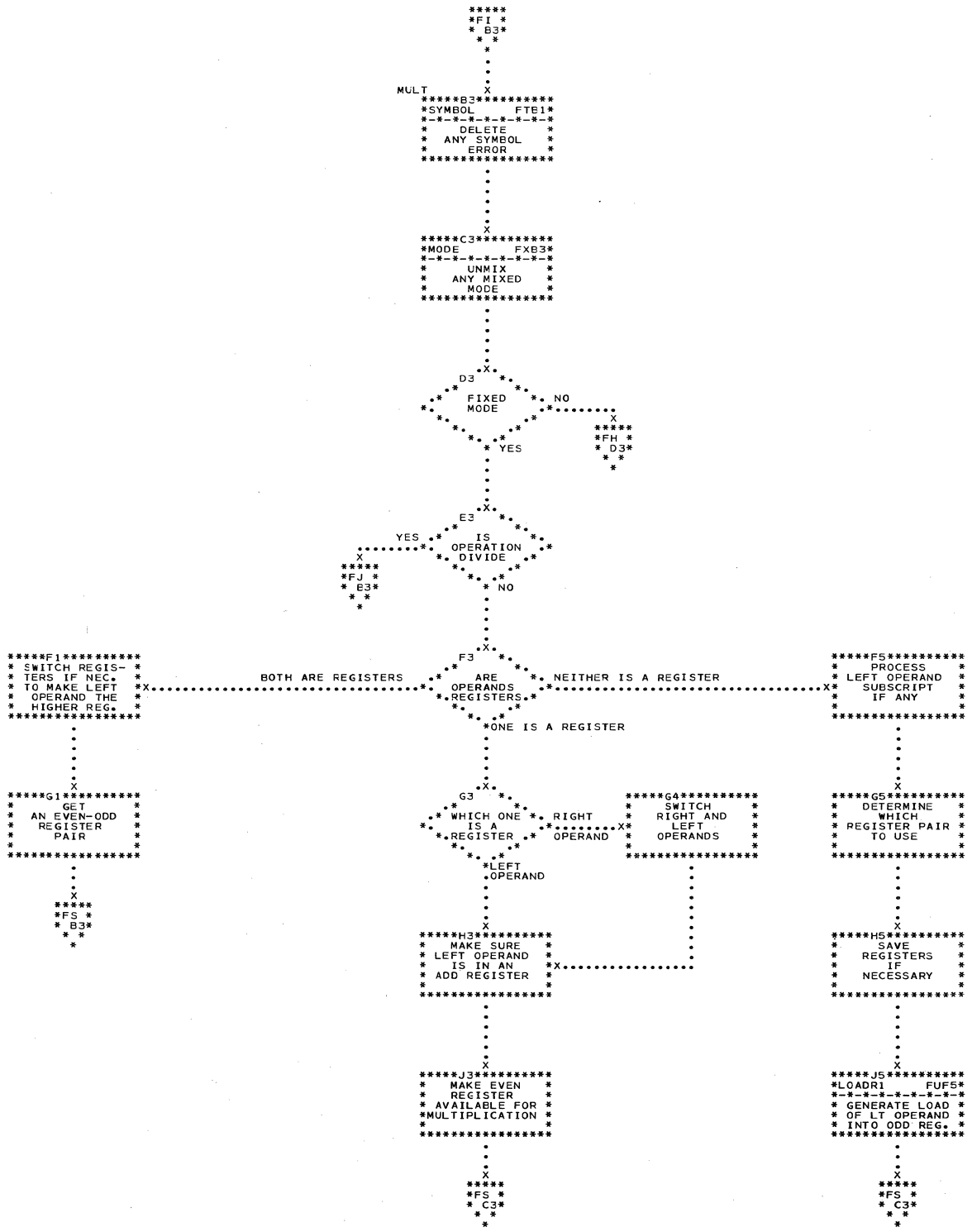


Chart FI. MULT Routine

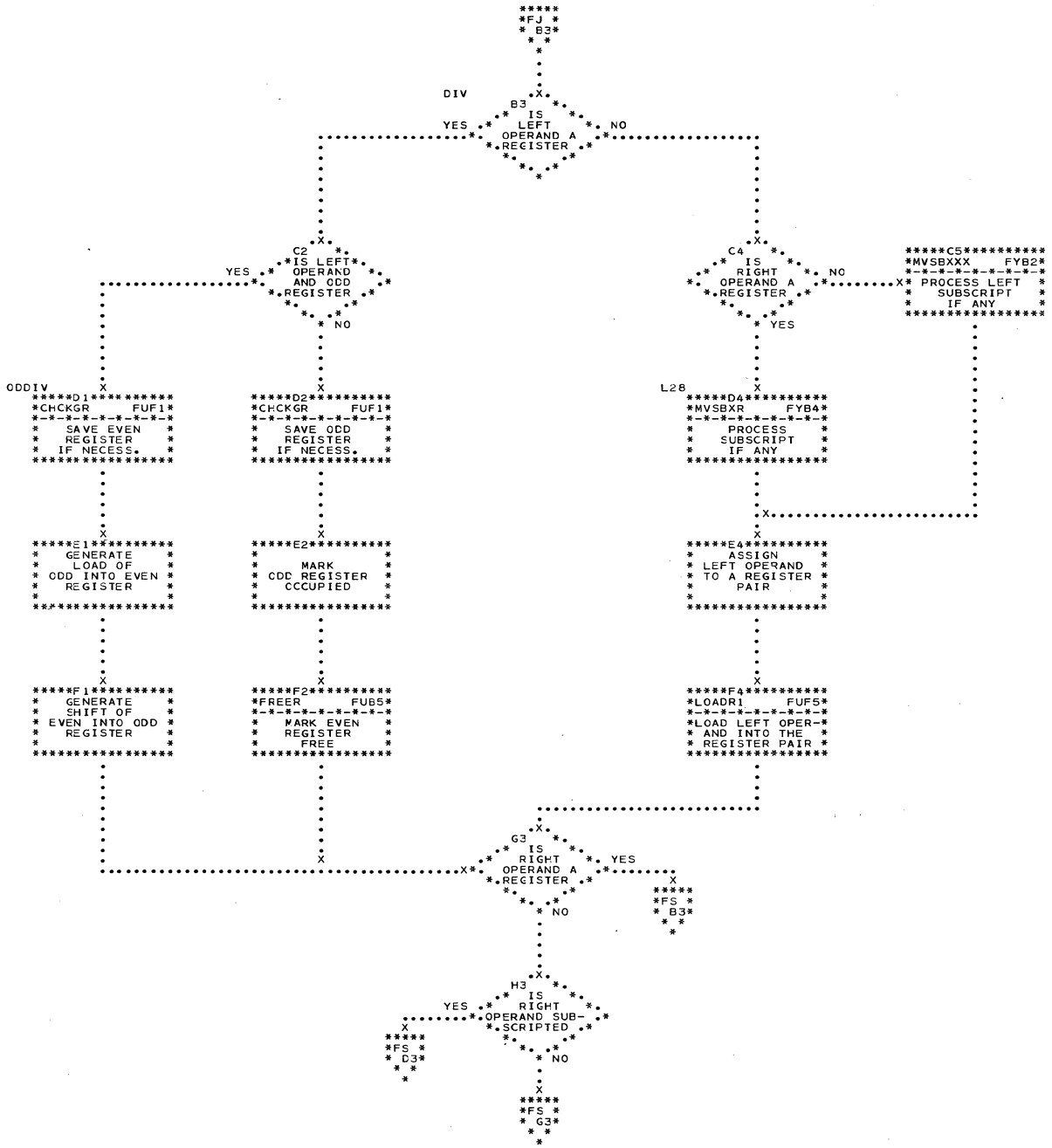


Chart FJ. DIV Routine

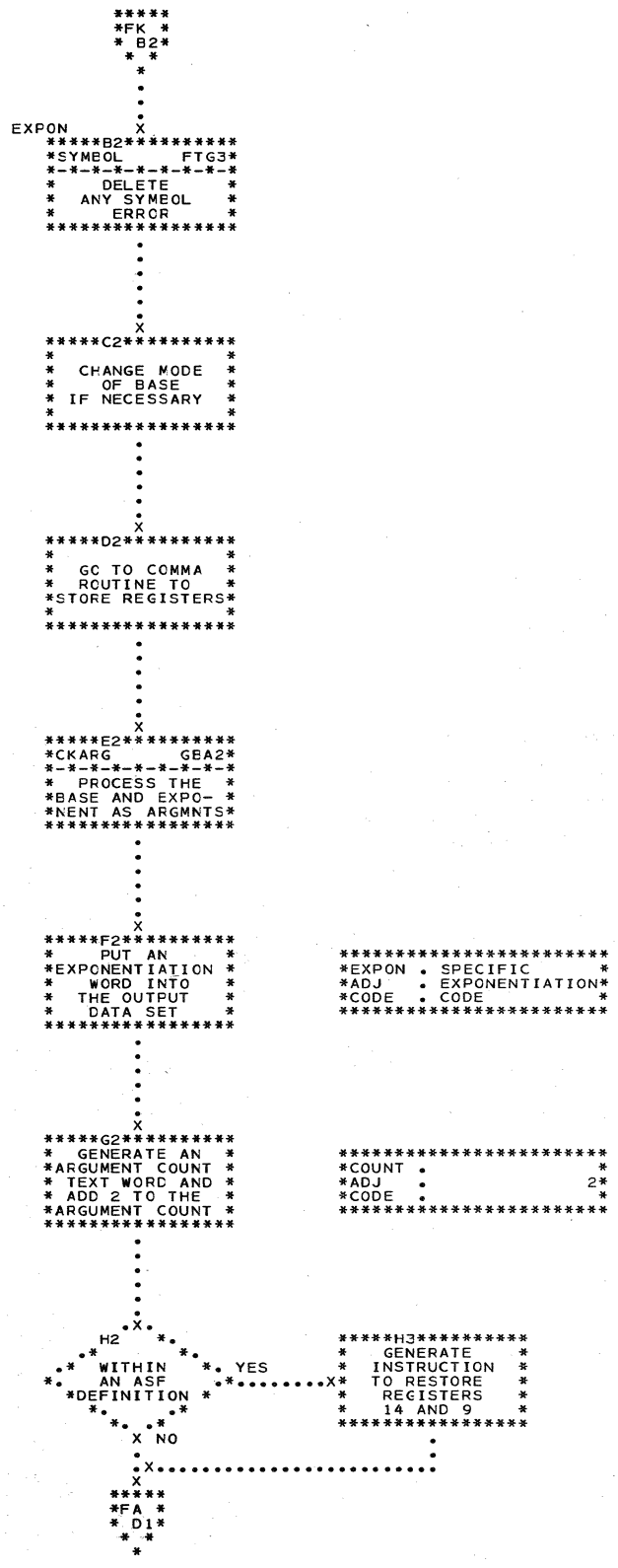


Chart FK. EXPON Routine

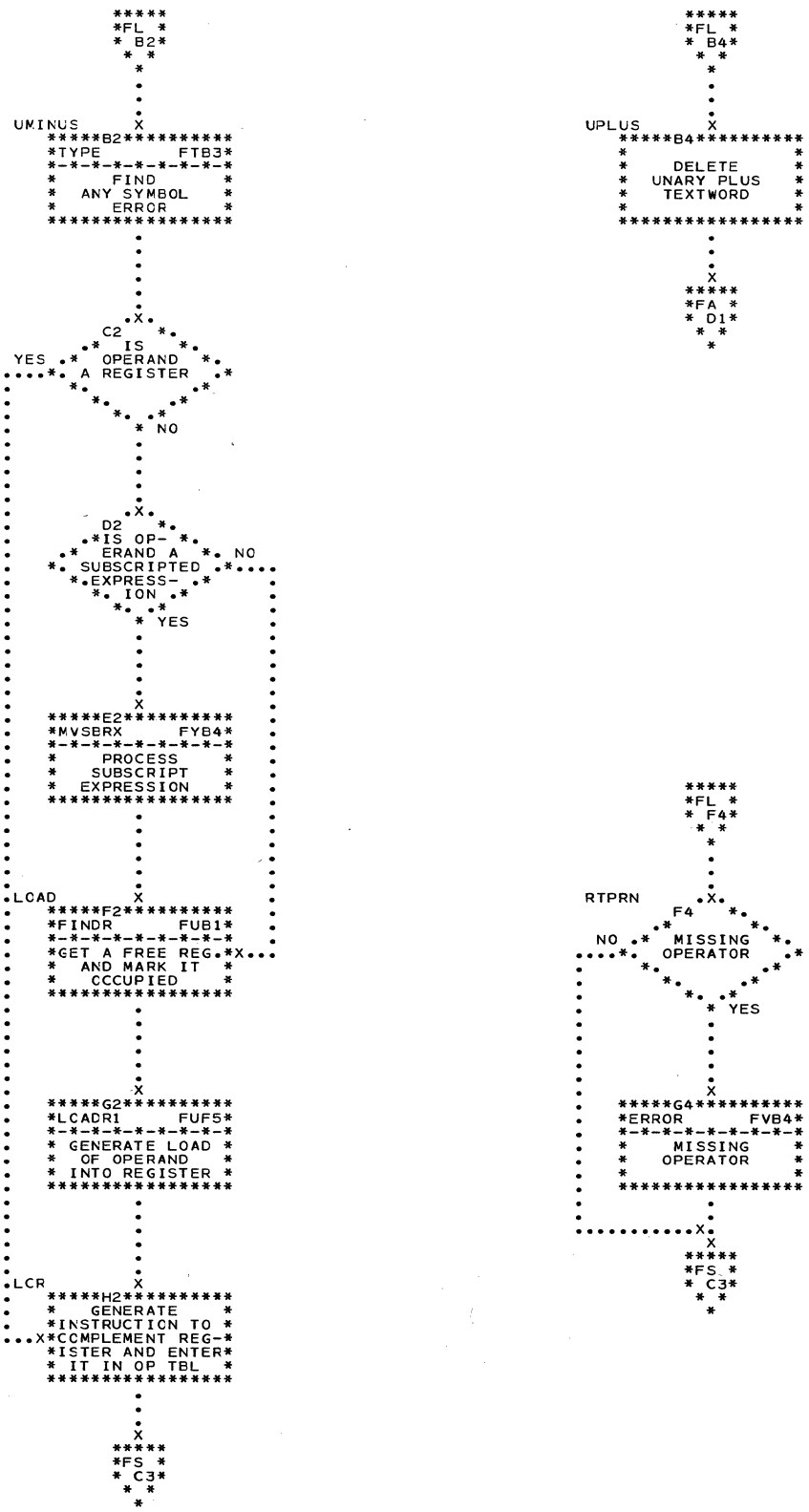


Chart FL. UMINUS, UPLUS, RTPRN Routines

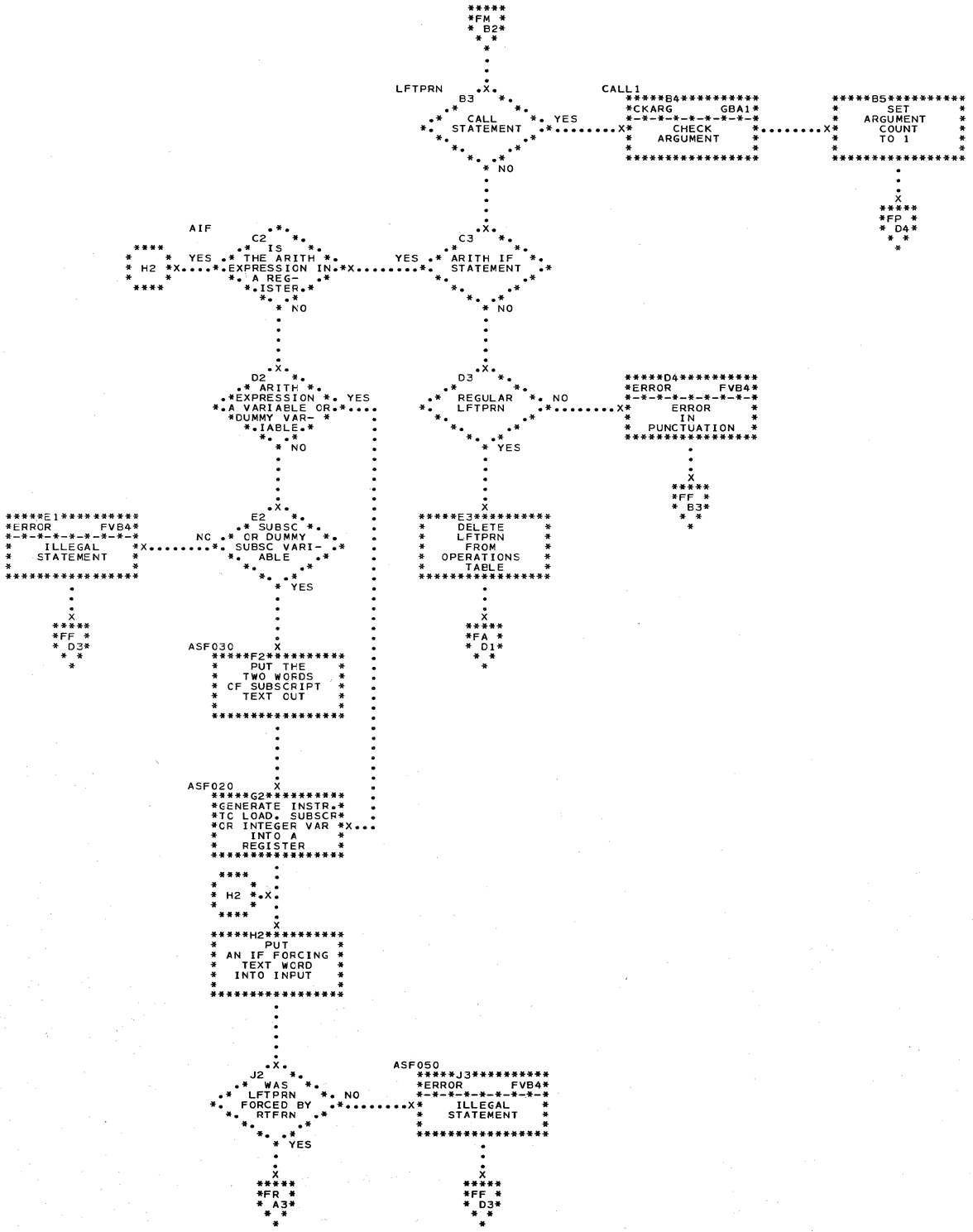
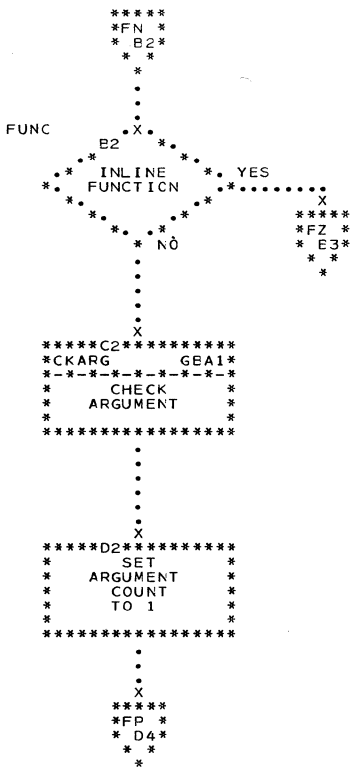
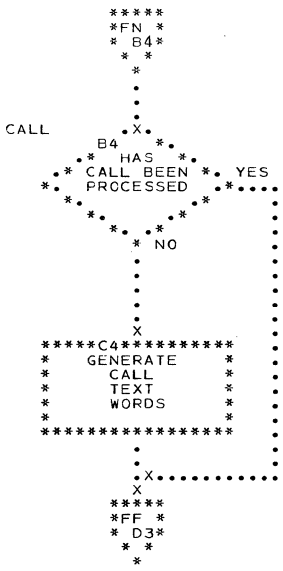


Chart FM. LFTPRN Routine





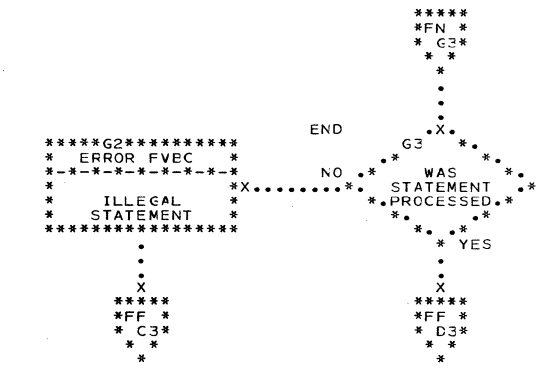


Chart FN. FUNC, CALL, END Routines

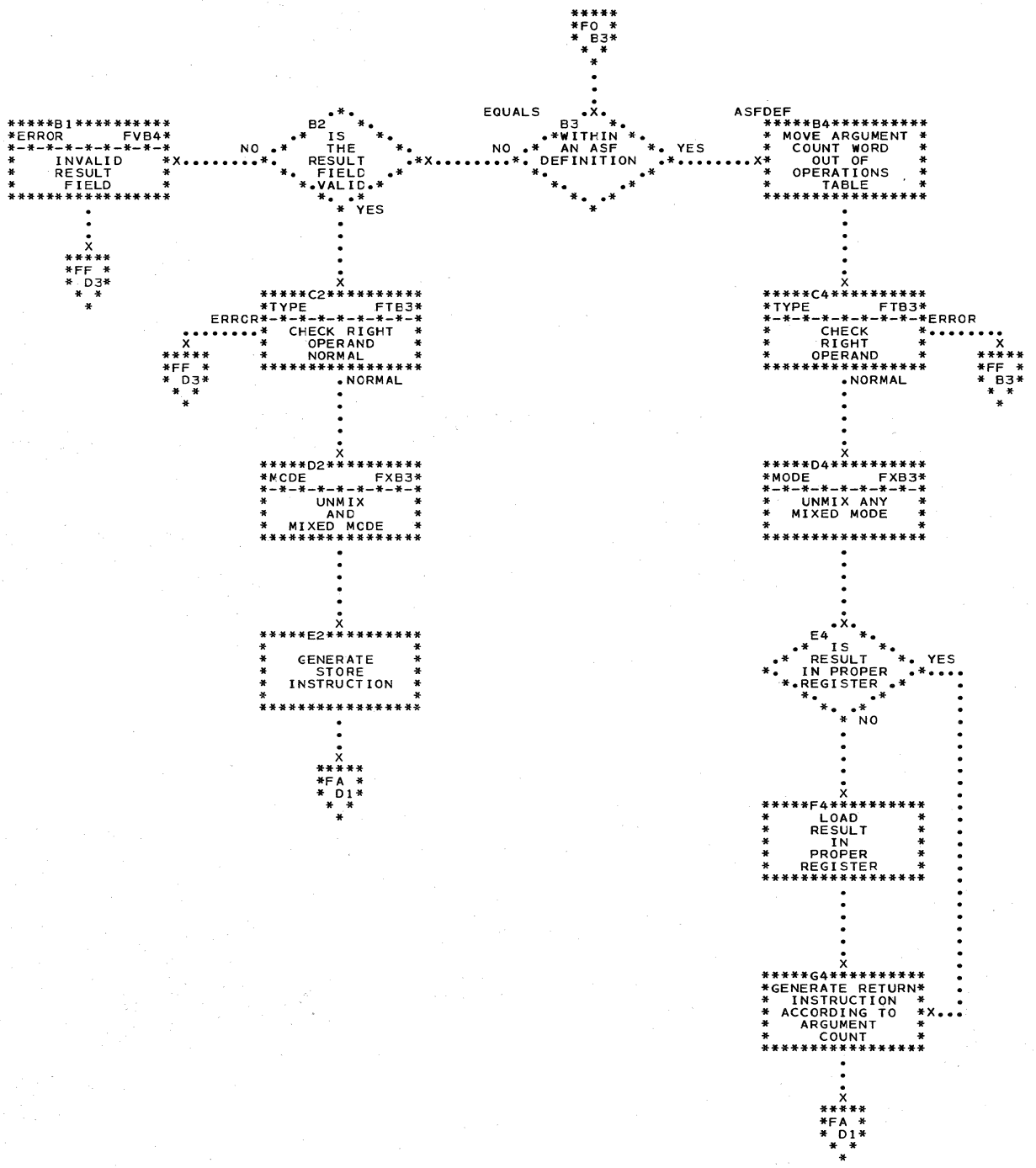


Chart FO. EQUALS Routine

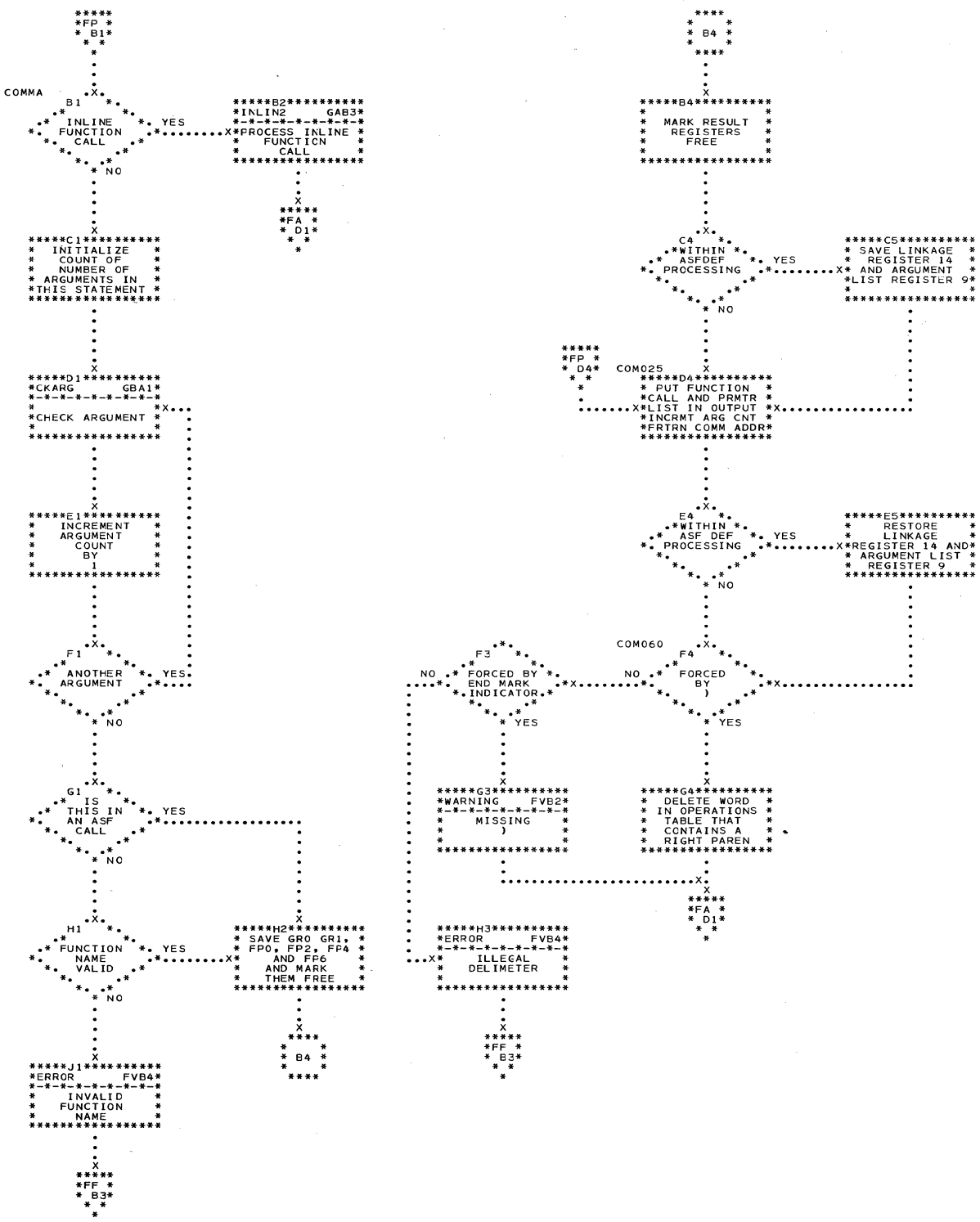


Chart FP. COMMA Routine

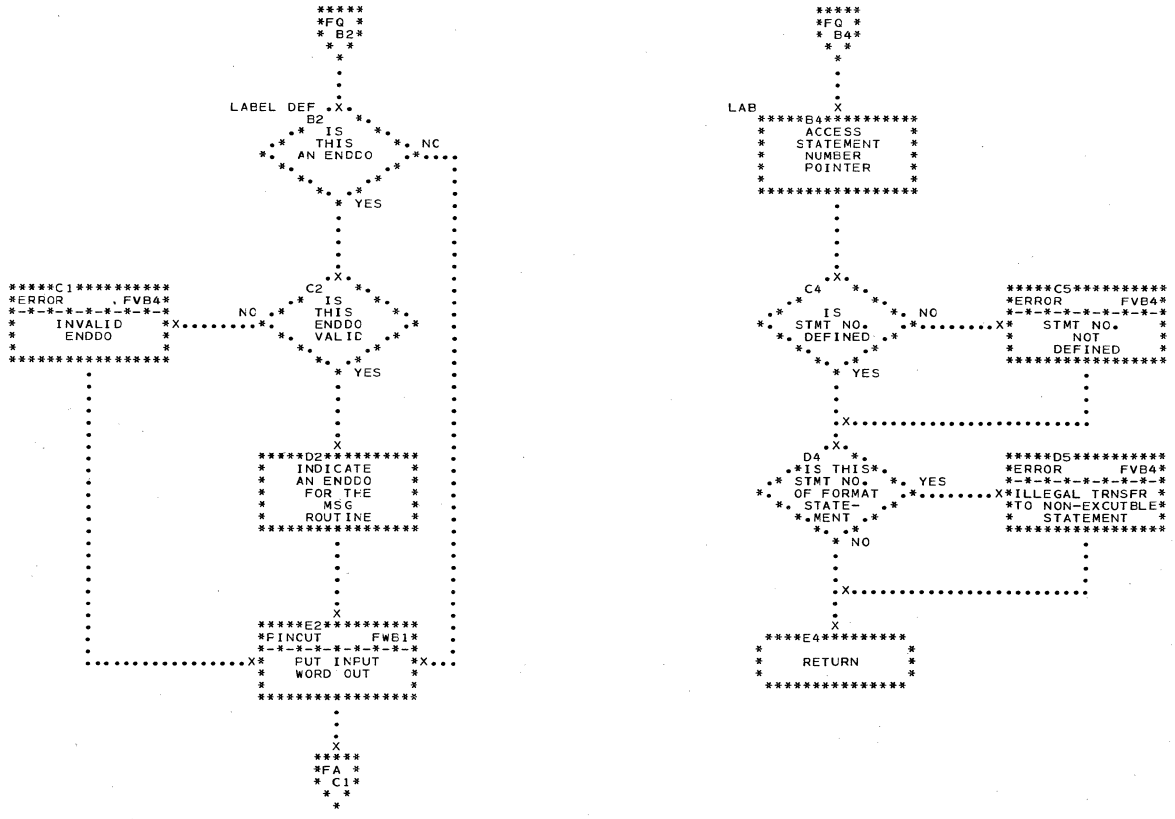


Chart FQ. LABEL DEF Routine, Subroutine LAB

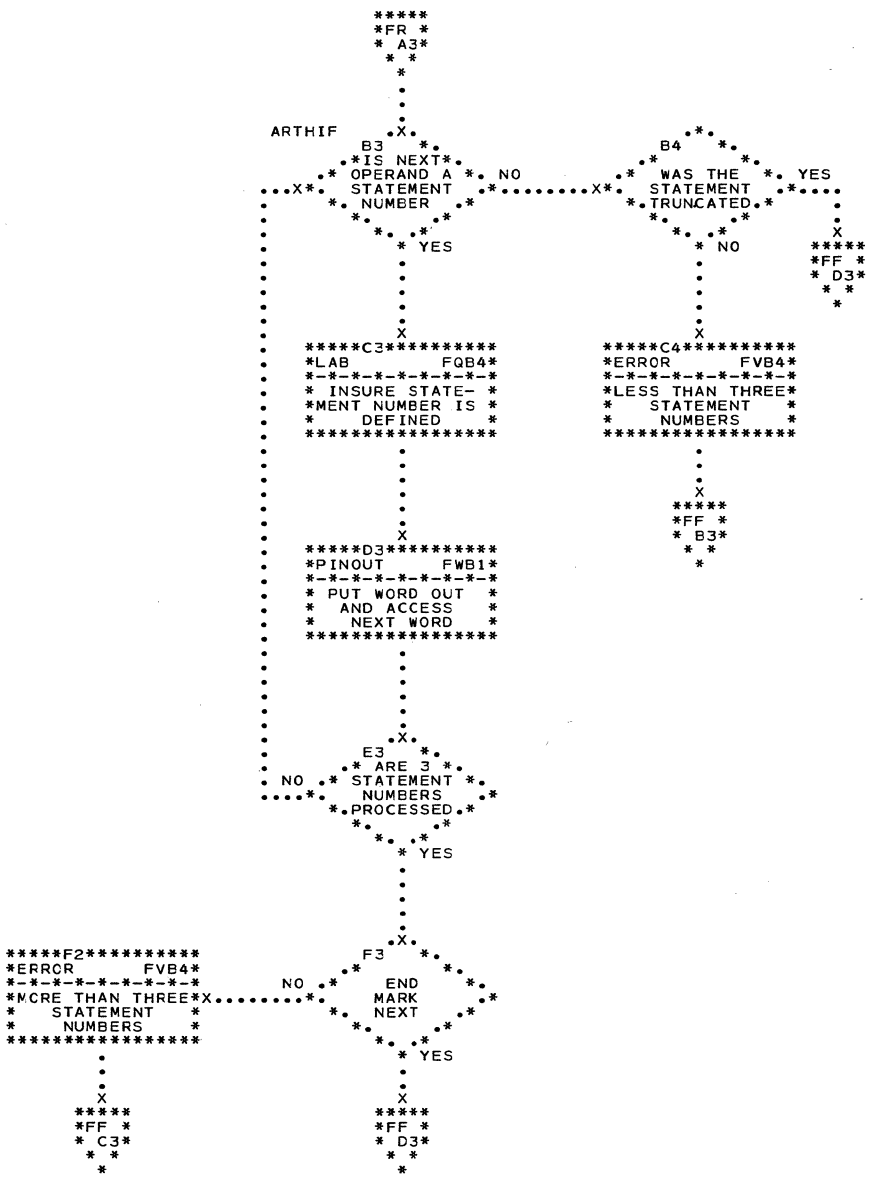


Chart FR. ARITH IF Routine

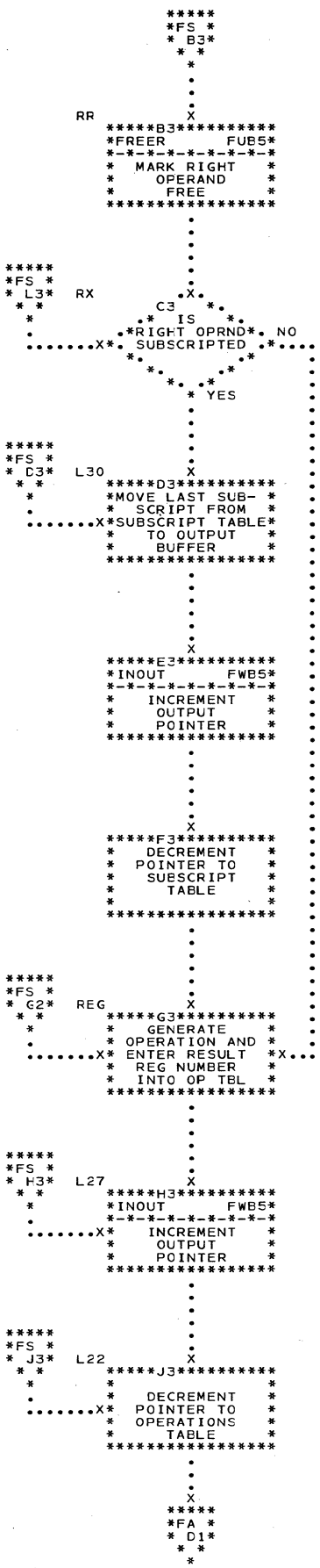


Chart FS. COMPILE Routine

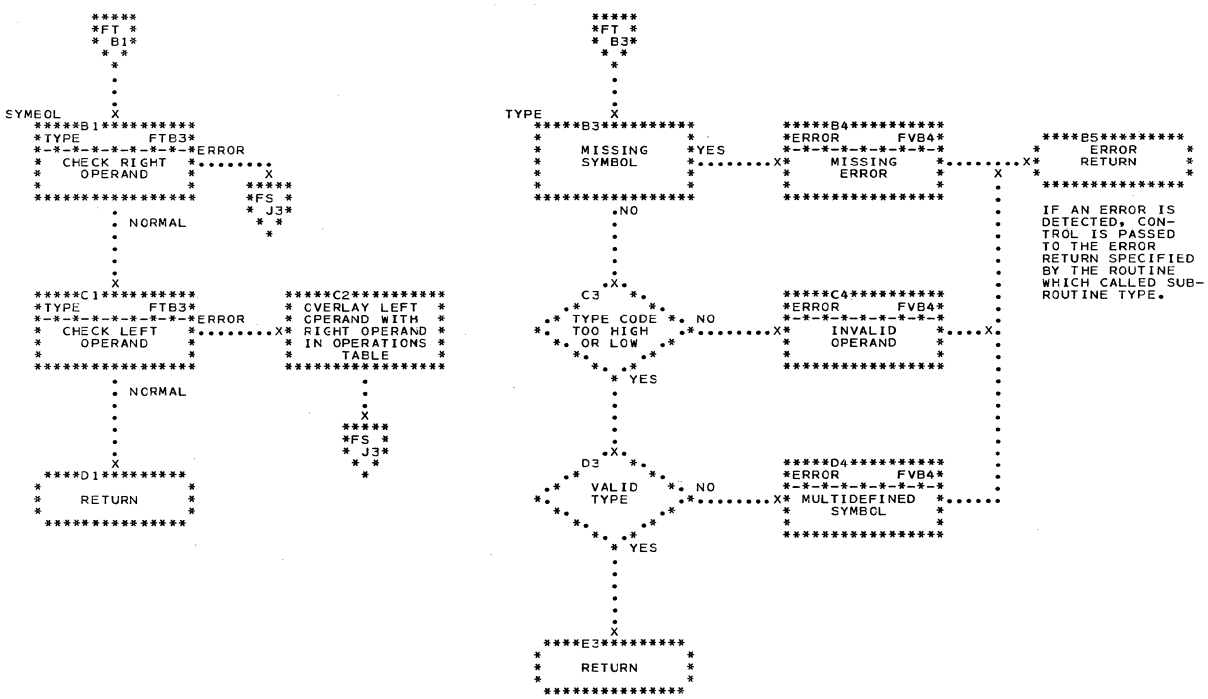


Chart FT. Subroutines SYMBOL, TYPE

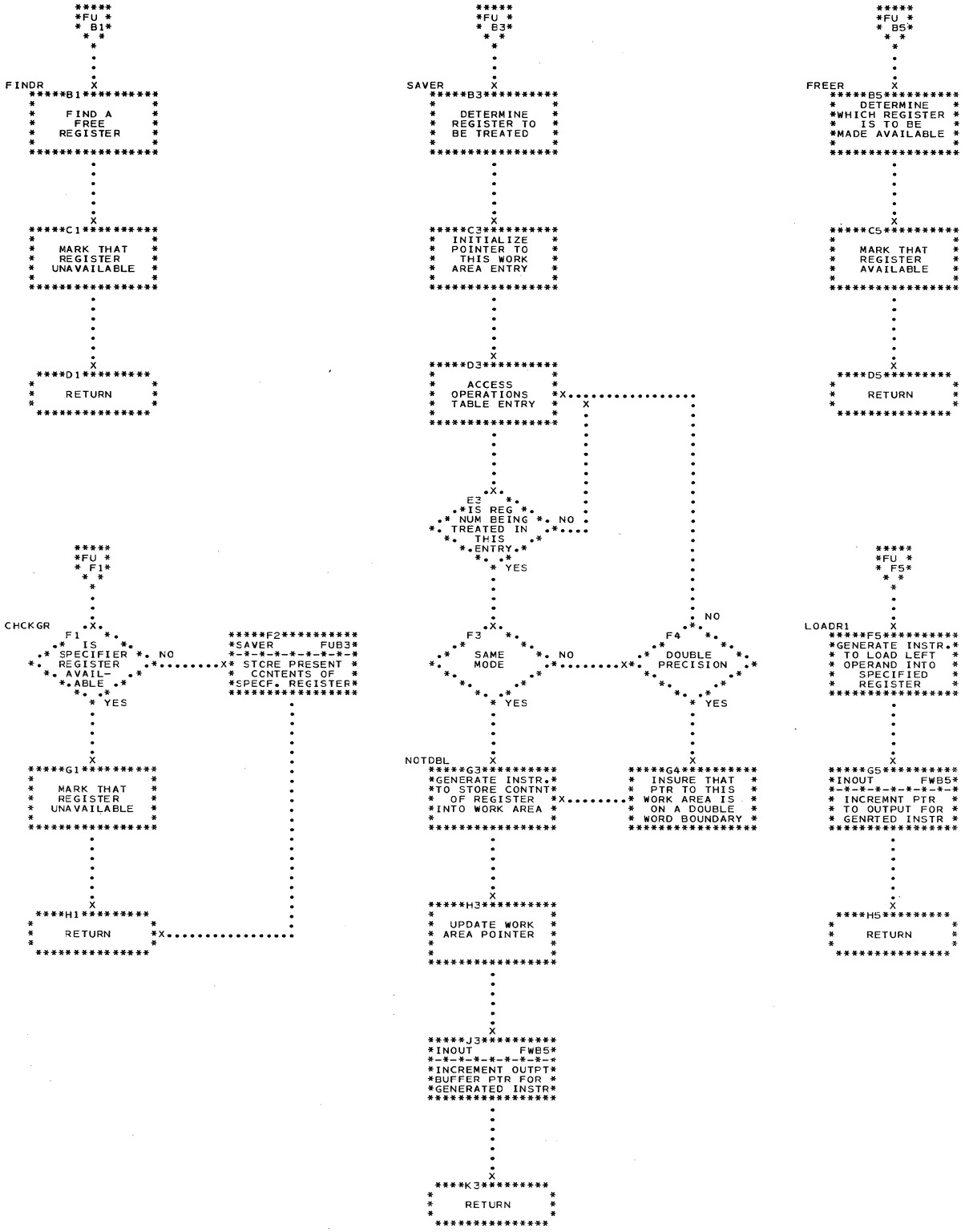


Chart FU. Subroutines FINDR, CHCKGR, SAVER, FREER, LOADR1

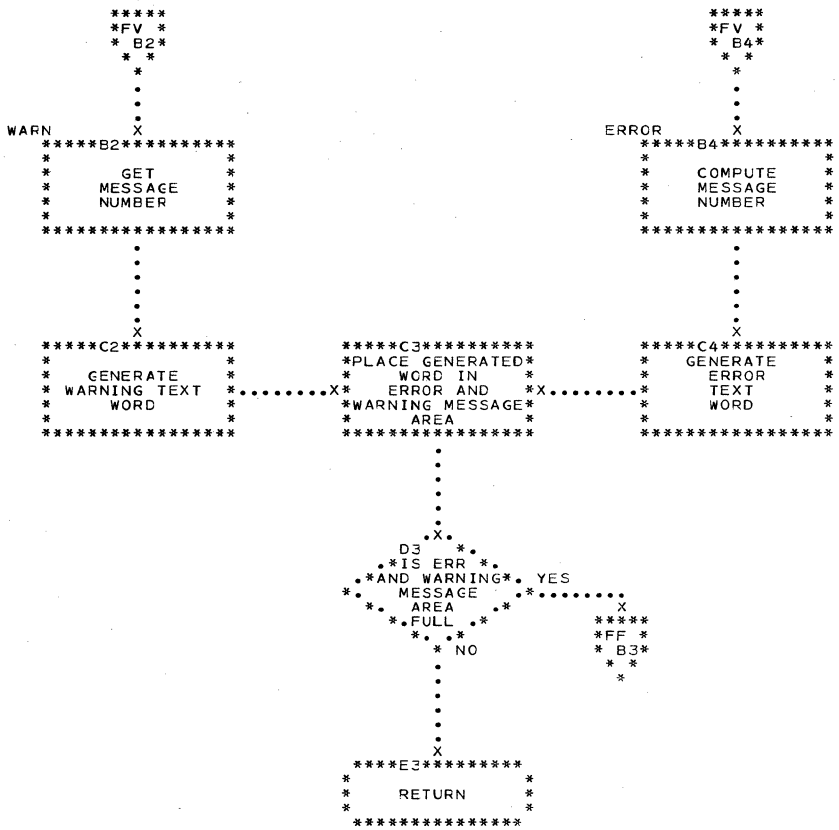


Chart FV. Subroutine WARN/ERROR

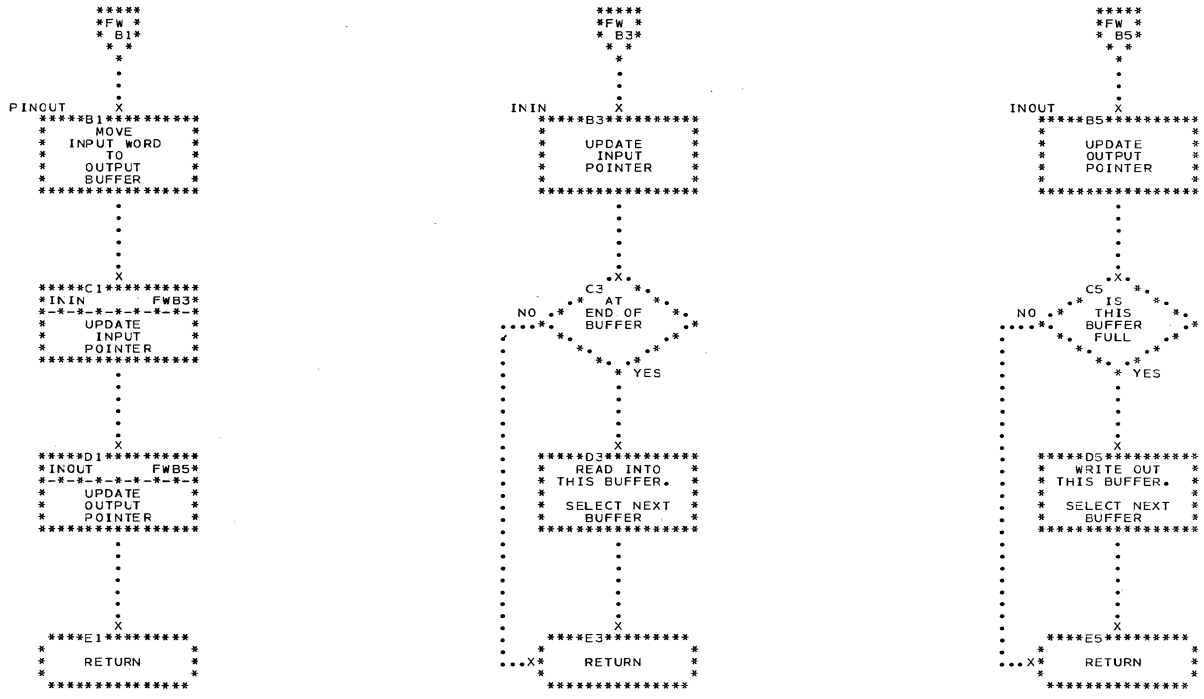


Chart FW. Subroutines PINOUT, ININ, INOUT

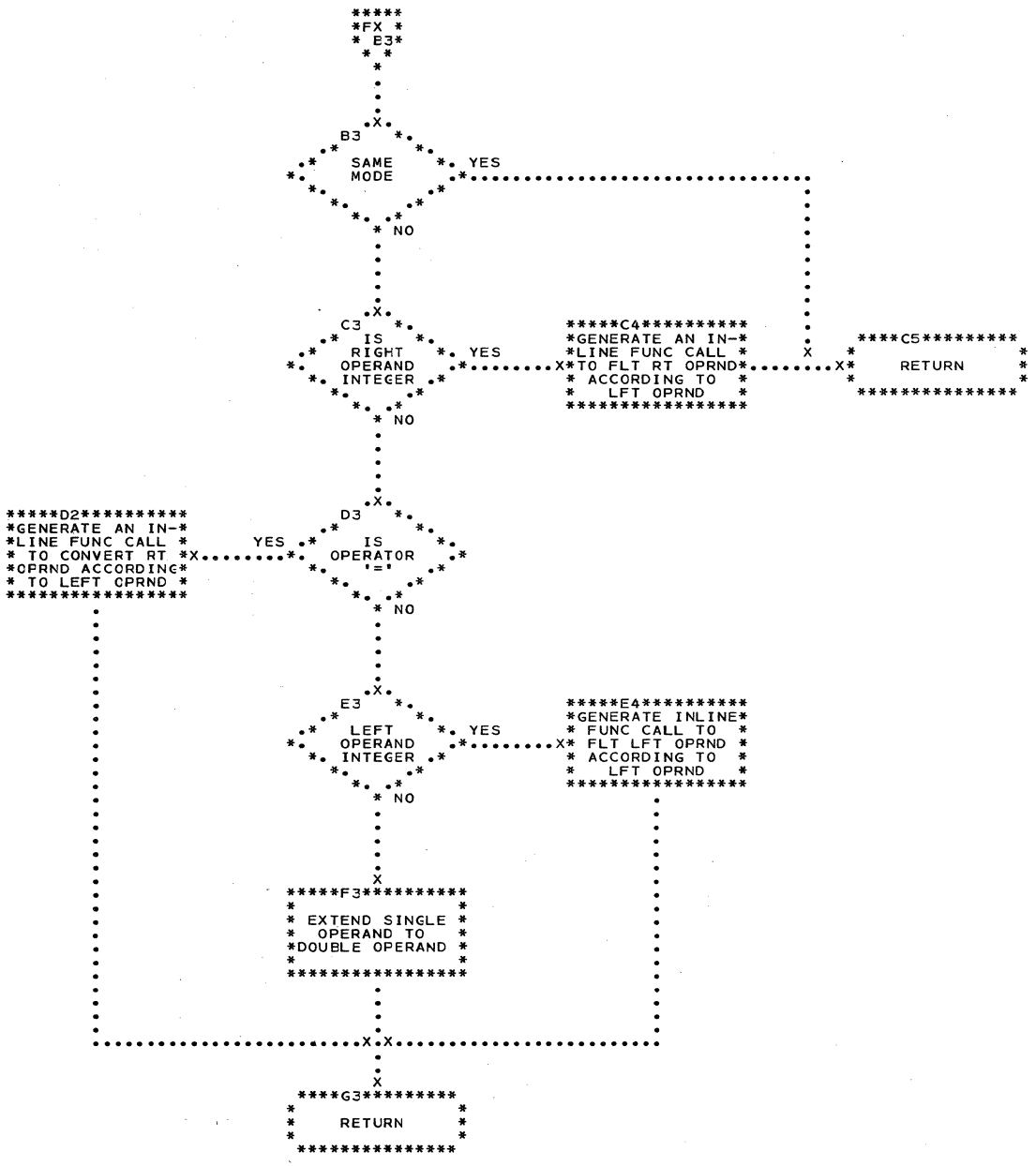


Chart FX. Subroutine MODE

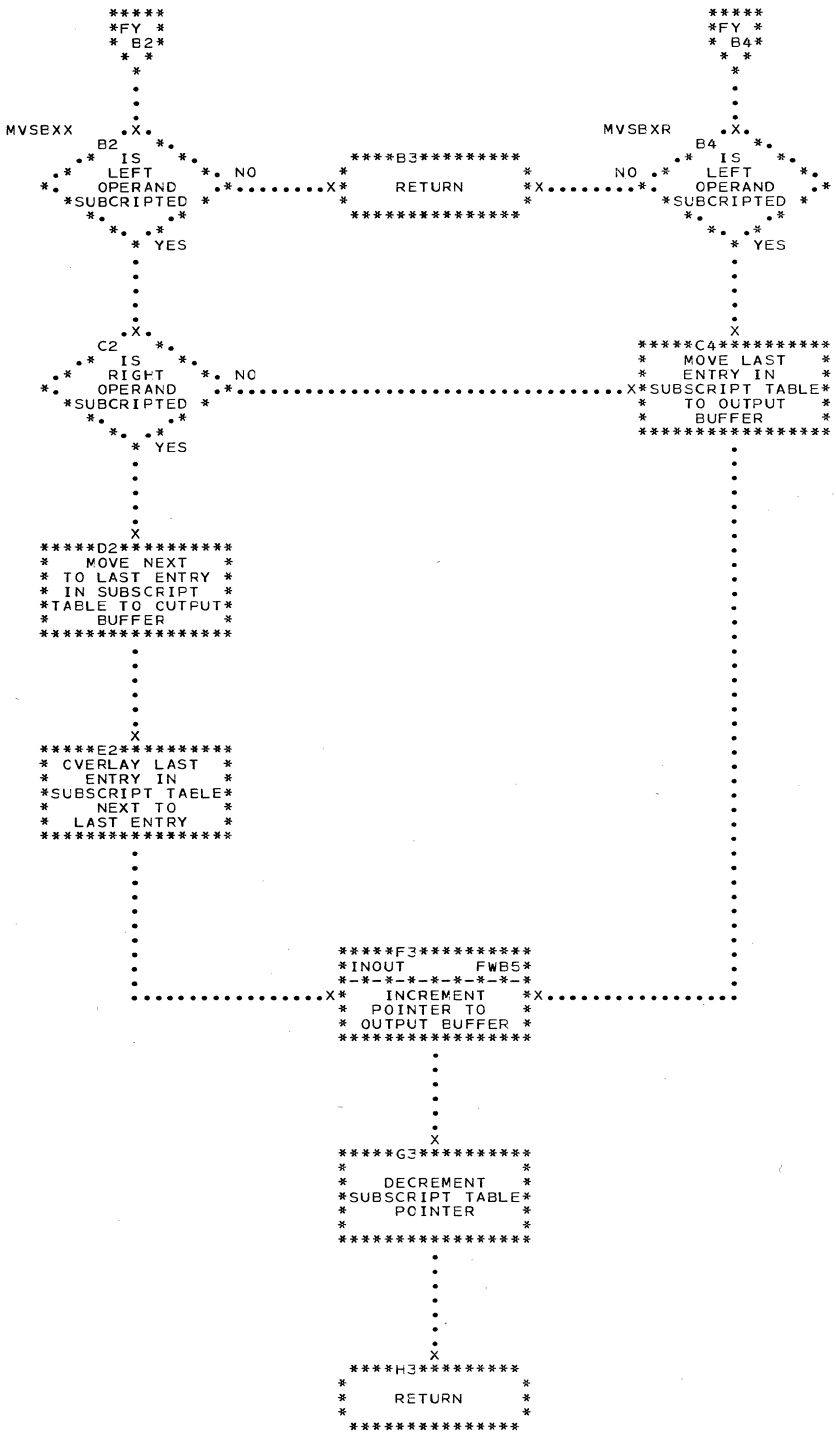


Chart FY. Subroutines MVSBXX, MVSBXR

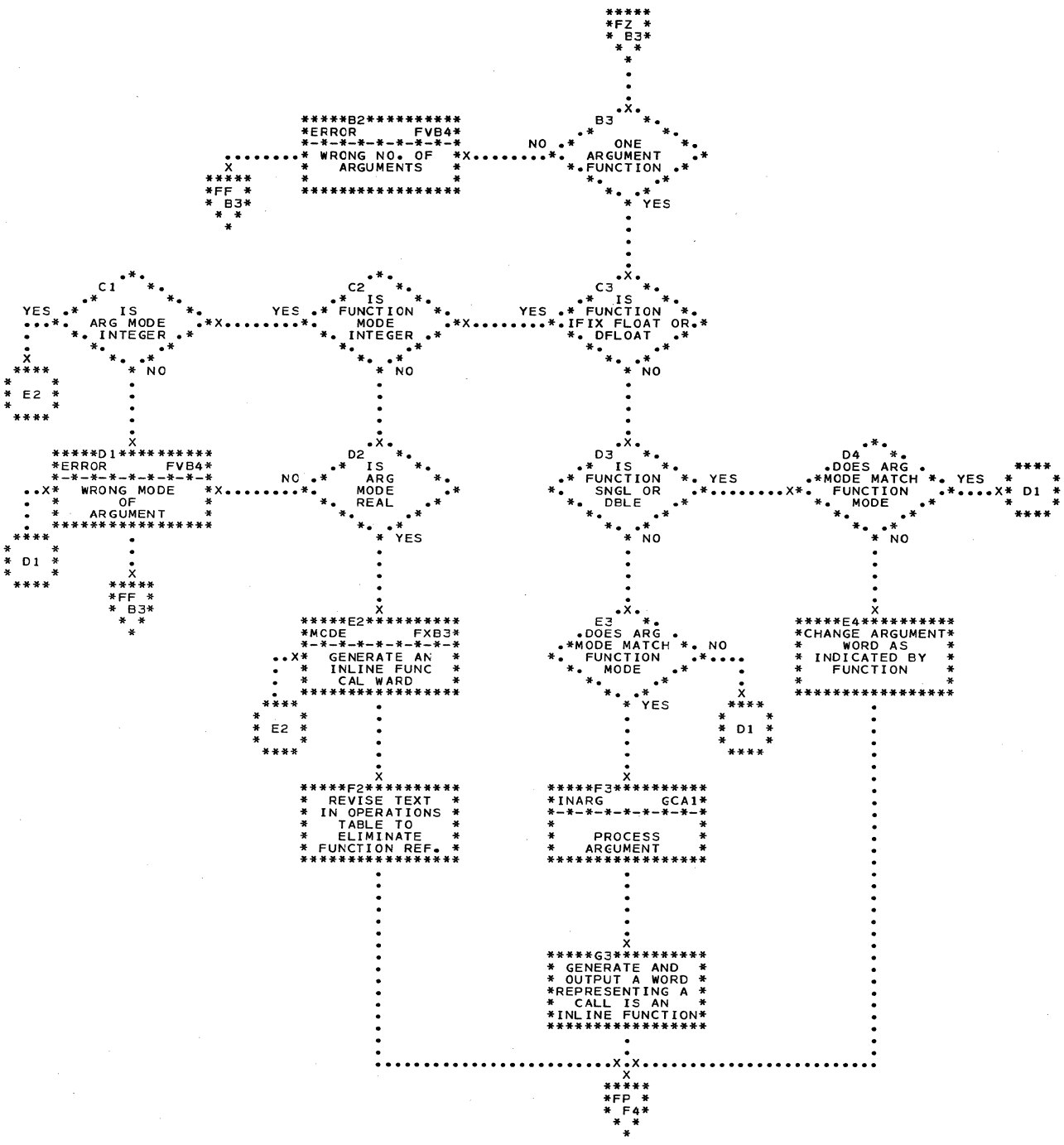


Chart FZ. INLIN1 Routine

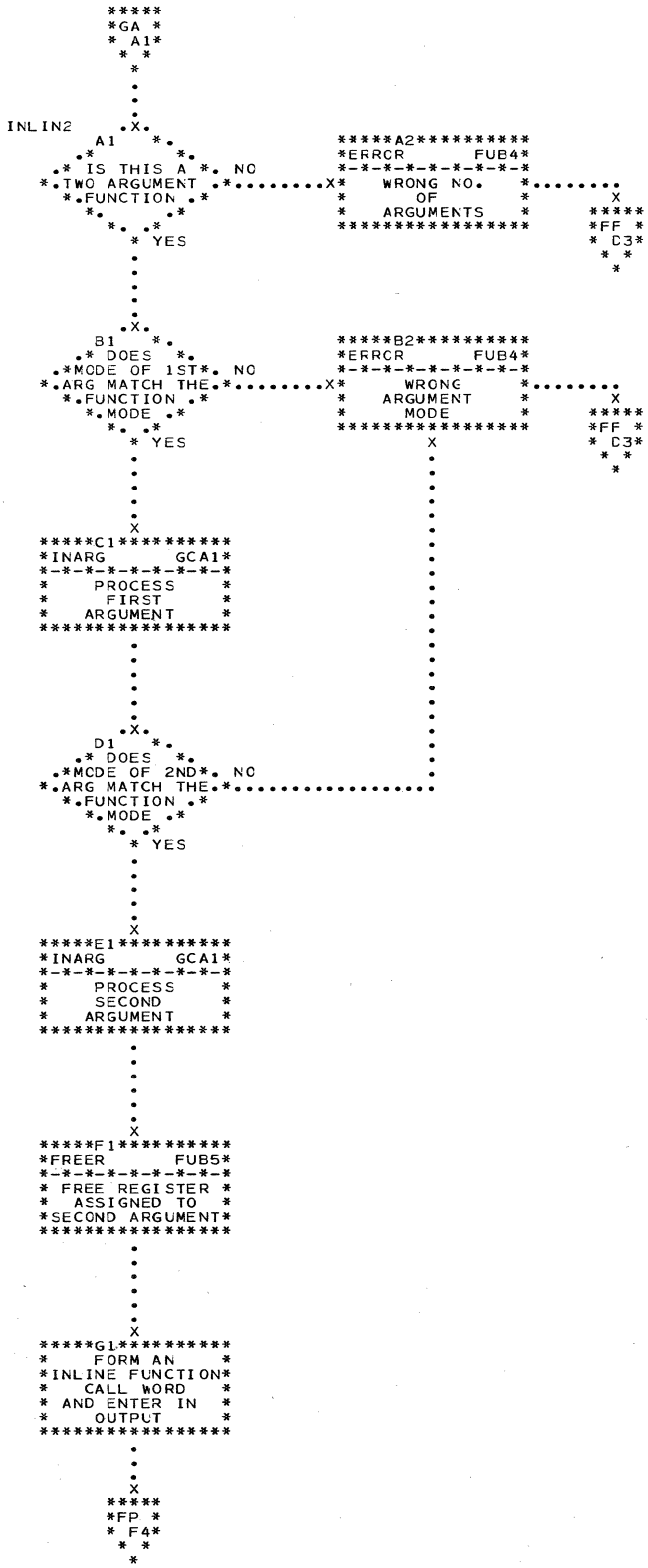


Chart GA. INLIN2 Routine

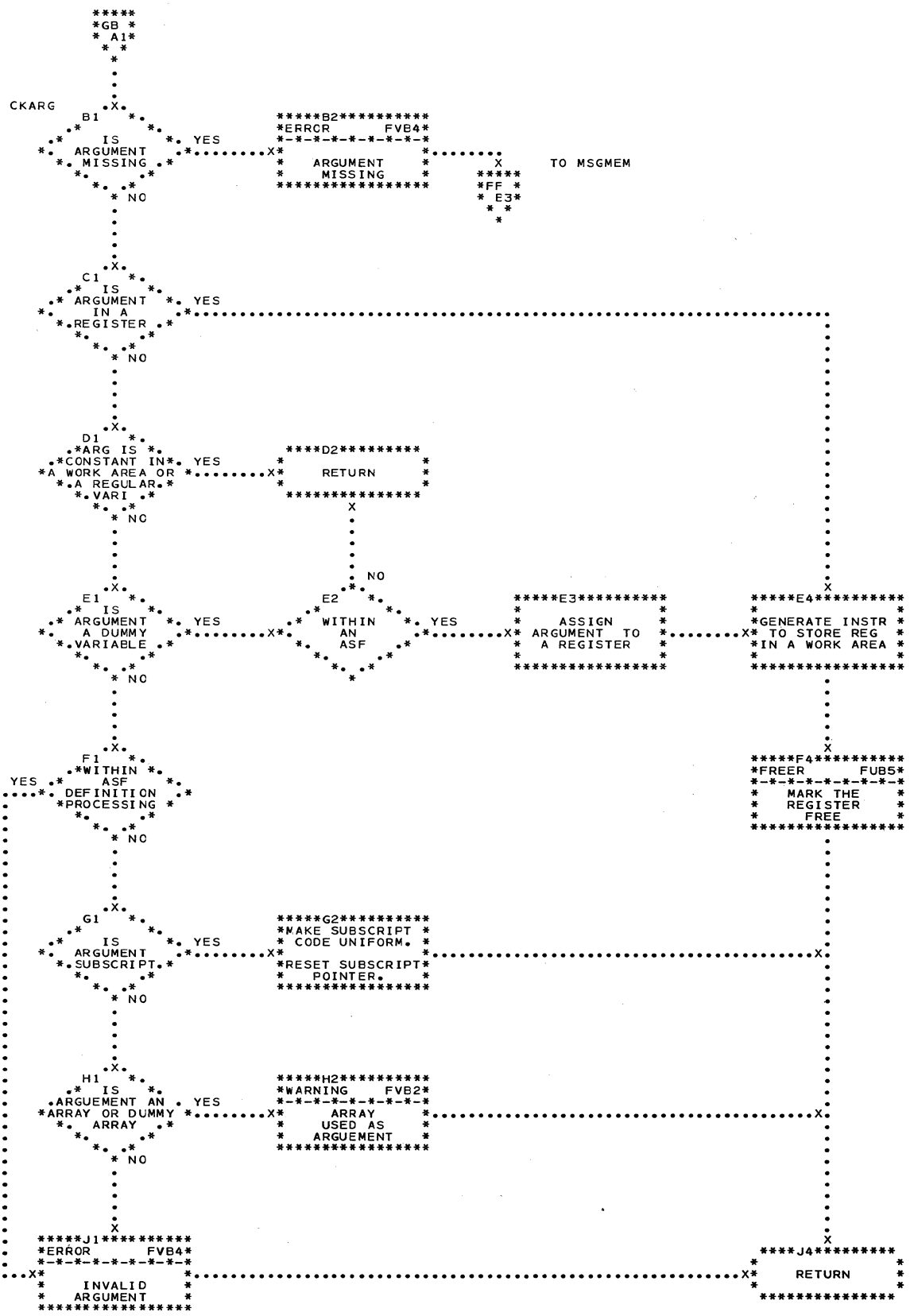


Chart GB. Subroutine CKARG

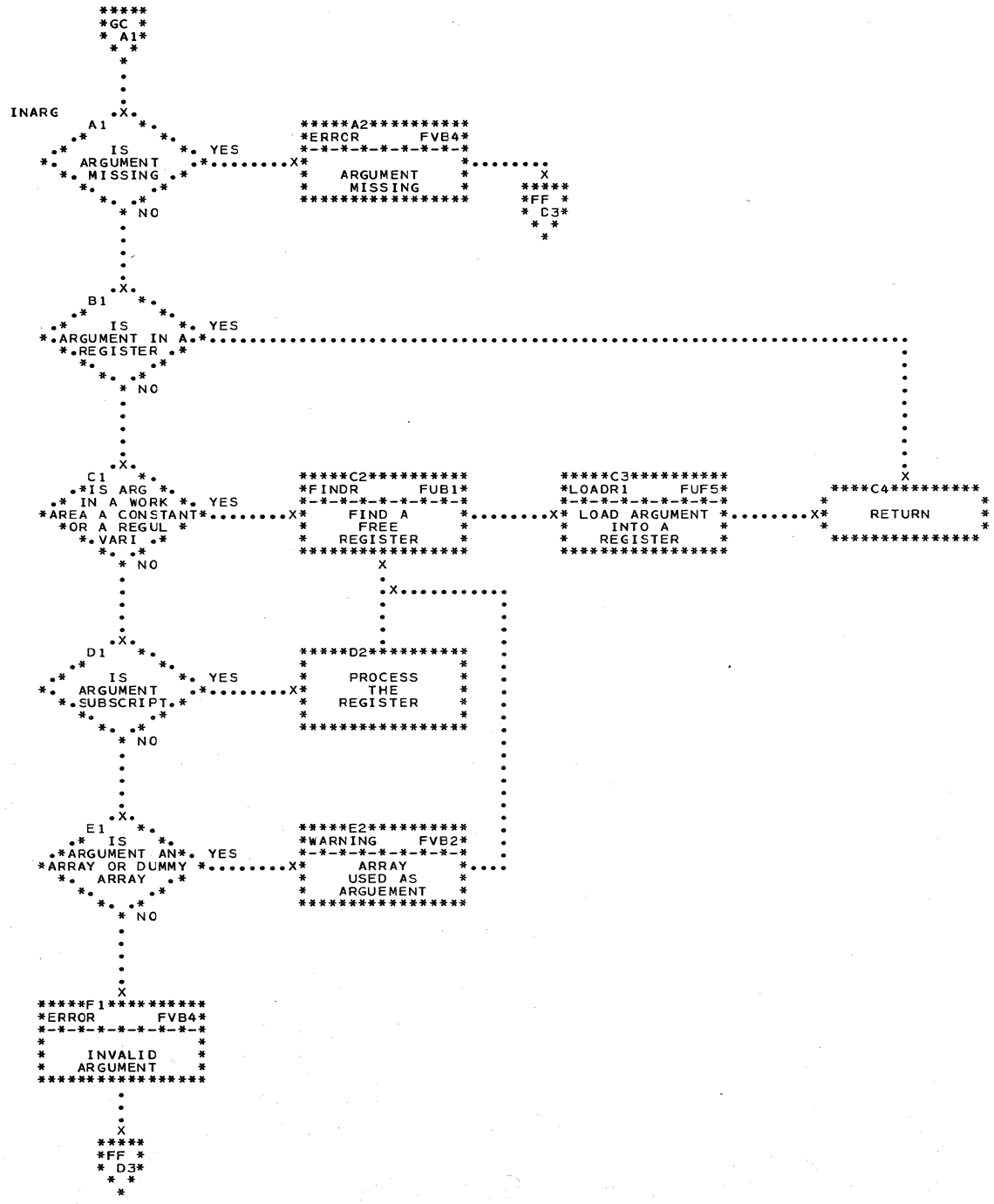


Chart GC. INARG Routine

Phase 20 increases the efficiency of the object program by decreasing the amount of computation associated with subscript expressions. Phase 20 in addition, performs miscellaneous functions, such as the generation of ESD and RLD records for any required library exponentiation subroutines and for any reference to IBCOM.

Chart 07, the Phase 20 Overall Logic Diagram, indicates the entrance to and exit from Phase 20 and is a guide to the overall functions of the phase.

SUBSCRIPT OPTIMIZATION

A subscript expression can reoccur frequently in a FORTRAN program. Recomputation at each occurrence is time-consuming and results in an inefficient object program. Therefore, Phase 20 performs the initial computation of any given subscript expression and assigns a register which ,at object time, contains the results of the computation. Phase 20 then modifies the text (previously referred to as intermediate text in order to differentiate it from COMMON and EQUIVALENCE text) for subsequent occurrences of the expression. This text modification (optimization) essentially replaces the computation of the subscript expression with a reference to its initial computation (that is, to the assigned register). The text for each subsequent occurrence of the subscript expression can be modified in this manner as long as the values of the integer variables in the expression remain unchanged.

Index Mapping Table

The index mapping table, used to aid the implementation of subscript optimization, 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 used for assignment to a subscript expression is determined during the initialization process for Phase 20. The format for an entry in the index mapping table is shown in Figure 47.

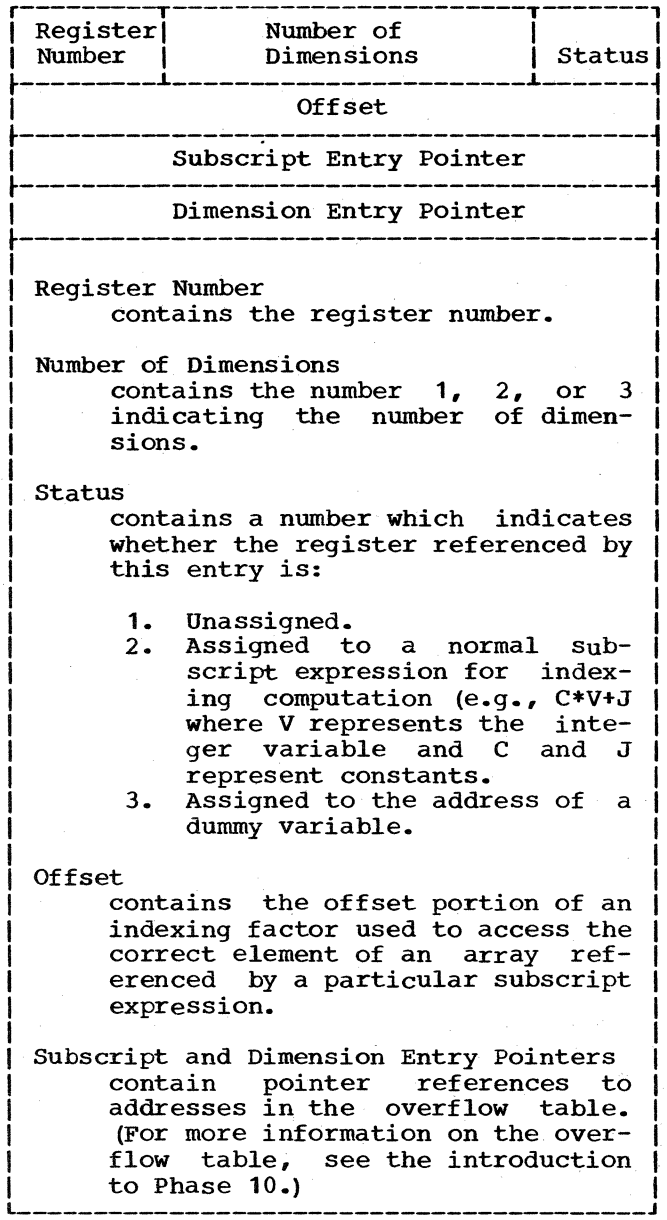


Figure 47. Index Mapping Table Format

Statements Subject to Optimization

Before Phase 20 can attempt subscript optimization, it must first find text that can include subscript expressions. Thus, a search of text is made for:

1. Arithmetic statements.
2. IF statements.

3. CALL statements.
4. I/O lists (treated as statements in text).

When Phase 20 encounters one of these statements containing a subscripted variable, the optimization process begins. The index mapping table is used to determine if the subscript expression of the subscripted variable has been previously encountered.

SUBSCRIPT TEXT INPUT: The text input to Phase 20 for a subscript expression is shown in Figure 48.

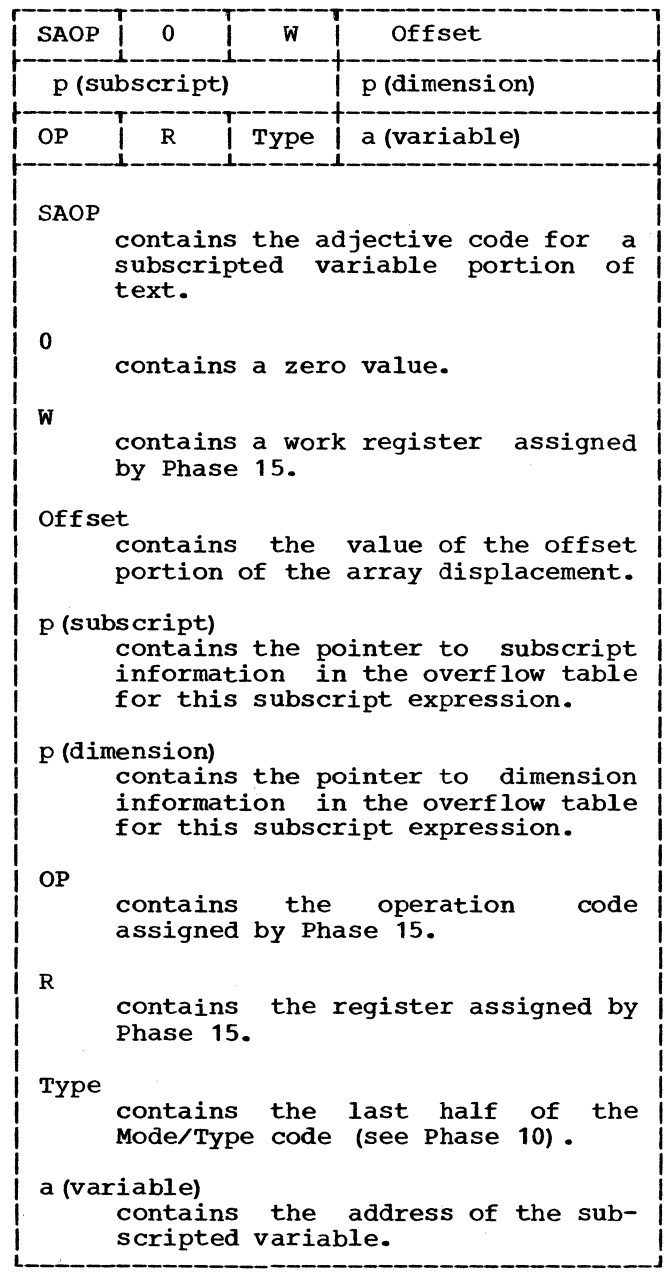


Figure 48. Subscript Text Input Format

Register Assignment

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 for assignment, the register that is currently assigned to the subscript expression of the least dimension is reassigned to the current subscript expression.

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

Generation of Literals

Phase 20 generates literals associated with the array displacement represented by any given subscript expression. The calculation of an array displacement is explained in Appendix C. This explanation includes a discussion of the offset and CDL (constant, dimension, length) portions of the array displacement. Literals are generated by Phase 20 under the following conditions:

1. Phase 20 adds the offset portion of the array displacement to the displacement of the variable address. A resulting value outside the addressable range of 0 through 4095 bytes causes Phase 20 to make the offset a literal. The generation of an offset literal allows Phase 25 to produce instructions without assigning a new base register whose contents are unknown.
2. Phase 20 generates a literal for each component of the CDL portion of the array displacement. For example, the value of C2*D1*L (the CDL component associated with the second dimension of an array displacement) is generated as a literal by Phase 20. Literals associated with the CDL portion of the array displacement are used by Phase 25 in its generation of machine language instructions. (If the first component of the CDL portion of the array displacement is a power of 2, that power, instead of the address for the literal C1*L, is placed in text.)

Subscript Text Output

Subscript text output from Phase 20 depends on the previous optimization of the subscript expression. Three adjective codes used to indicate the different conditions that can be present in subscript text output and the conditions are explained in the following paragraphs.

SAOP 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 the variable address displacement. Subscript text output associated with an SAOP adjective code is shown in Figure 49.

SAOP	N	W	Offset
p (subscript)			a (C1*L)
a (C2*D1*L)			a (C3*D1*D2*L)
OP	R	X	a (variable)

SAOP
contains an adjective code designating the form of 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 which combine to form the CDL portion 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 computation by Phase 20.

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

Note: All other entries are as defined in Figure 48.

Figure 49. Subscript Text Output From Phase 20 - SAOP Adjective Code

XOP ADJECTIVE CODE: This code indicates that the subscript expression has not been previously optimized, 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 variable address. The subscript text output associated with an XOP adjective code is shown in Figure 50.

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)

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

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

Note: All other entries are as defined in Figures 48 and 49.

Figure 50. Subscript Text Output From Phase 20 - XOP Adjective Code

AOP ADJECTIVE CODE: This code indicates that the subscript expression has been previously optimized. The subscript text output associated with an AOP adjective code is shown in Figure 51.

AOP	0	B	Offset
OP	R	X	a (variable)

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

0
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 Figure 48 and 49.

Figure 51. Subscript Text Output From Phase 20 - AOP Adjective Code

Special Considerations

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.

SUBSCRIPTED DUMMY VARIABLE: In addition to normal optimization, a base register is assigned to any dummy variable to access the variable during the execution of the object program. The assignment is entered in the index mapping table.

Preceding subscript text output associated with a dummy variable, an instruction to load the address of the variable into the base register is generated and placed in the output buffer. Because that base register, at object time, contains the address of the variable, the displacement portion of the text address field is set to zero. The base register number replaces the register portion of the text address field.

CONSTANT SUBSCRIPT EXPRESSION: Phase 20 does not assign a register to a constant subscript expression whose value is within the addressable range of 0 through 4095 bytes.

Subscript text output with an AOP adjective code is modified for a constant subscript expression, as follows:

1. The field following the AOP adjective code contains an F instead of a zero.
2. The X field can contain a temporary register, if an offset literal was generated.

Statements That Affect Optimization

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

DO AND READ STATEMENTS: These statements sometimes cause the redefinition of an integer variable that exists in a subscript expression. (Arithmetic statements are also processed for a possible redefinition of an integer variable.) Any integer variable that is redefined (i.e., changes value) becomes a bound variable. For example, in the DO statement:

```
DO 25 K=1,1000
```

the integer variable K is a bound variable.

Any encounter of a bound variable causes Phase 20 to examine subscript expressions assigned a register in the index mapping table. A bound variable in a subscript expression invalidates any previous computation for that expression and causes the register assigned to that expression to be deleted.

ARITHMETIC, CALL, AND IF STATEMENTS: Any subprogram argument that is an integer variable causes redefinition of all subscript expressions containing that integer variable.

REFERENCED STATEMENT NUMBER: When a statement number is referenced by other statements (e.g., a GO TO statement), Phase 20 does not know if the value of previously encountered integer variables can still be used by subscript expressions containing those variables. Because any given integer variable may now be a bound variable, Phase 20 considers a statement number referenced by some other pertinent statement as a point of definition and deletes all register assignments to subscript expressions in the index mapping table.

END STATEMENT: The END statement indicates the end of Phase 20 processing. If the end of the input tape is reached before the END statement, compilation is aborted because input to Phase 20 is limited to one tape.

ESD/RLD RECORDS

When Phase 20 encounters exponentiation, it references one of the library exponentiation subroutines. Because this reference is external to the program being compiled, Phase 20 generates text, ESD, and RLD records for the first encounter of any given exponentiation subroutine.

References to IBCOM, CGOTO, and IBERR also cause Phase 20 to generate ESD and RLD records.

STORAGE MAP

The storage map for Phase 20 is shown in Figure 52.

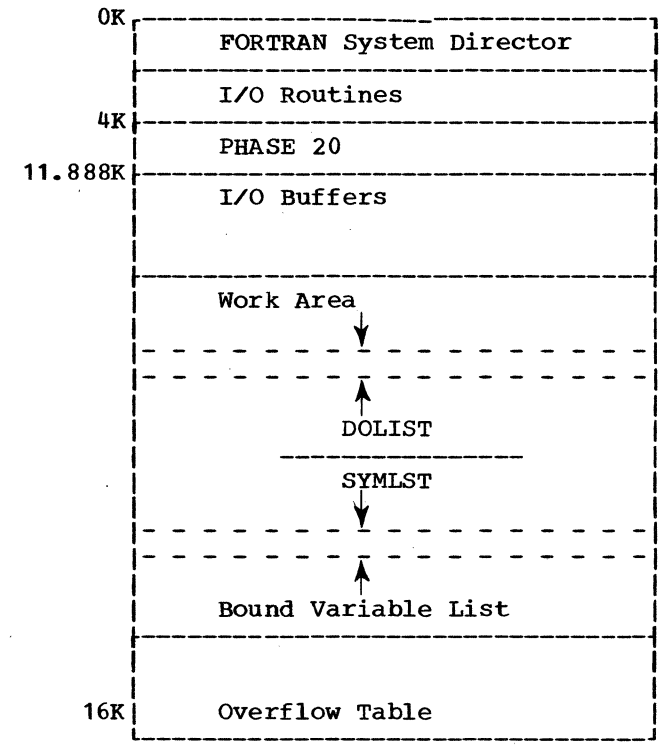


Figure 52. Storage Map for Phase 20

ROUTINES/SUBROUTINES

The routines and subroutines that perform subscript optimization, process subscript text input and output, and generate ESD and RLD records are shown in Figure 53.

The routines and subroutines are categorized as follows:

1. Charts HA and HB: Phase 20 initialization and control.
2. Charts HC through HJ: Processing routines for the various statements.
3. Charts HK through HT: Optimization routines.
4. Charts HU through ID: General purpose utility subroutines.

INIT Routine: Chart HA

The INIT routine performs the required initialization for Phase 20.

ENTRANCE: The INIT routine receives control from the FORTRAN System Director.

OPERATION: An indicator is set in the communications area to indicate to the FORTRAN System Director that Phase 20 is in control. The location counter used in the assignment of addresses to literal constants generated during Phase 20 is initialized to the location counter value in the communications area. The INIT routine determines the first register number available for assignment in Phase 20 by using the contents of that location counter. The beginning address of the index mapping table is determined according to this register.

The INIT routine also opens the input and output data sets, primes the first input buffer area, and determines the initial address of the work area and the bound variable list. This list contains the addresses of all current bound variables.

The INIT routine initializes the addresses for the ESD, RLD, and text card output areas. The program name is then moved to all three output areas. If the program being compiled is a main program, an ESD and RLD record are immediately put onto the output data set.

EXIT: The INIT routine exits to the Control routine for Phase 20.

Control Routine: Chart HB

The Control routine controls Phase 20 processing.

ENTRANCE: The Control routine receives control from the INIT routine.

OPERATION: The initial text word for a statement is accessed and its adjective code is checked to determine if optimization is to be performed. Optimization occurs in Phase 20 if the adjective code indicates an arithmetic, IF, or CALL statement, or an I/O list substatement. Upon encountering the code for one of these statements, the Control routine moves the entire text for the associated statement to the work area, and the adjective code in each word is examined for exponentiation at this time. Exponentiation requires an external reference to one of the library exponentiation subroutines. The text word that contains an adjective code for exponentiation appears as follows:

Code	00	000N
1 byte	1 byte	2 bytes

where N is any integer constant whose value depends upon the required library exponentiation subroutine.

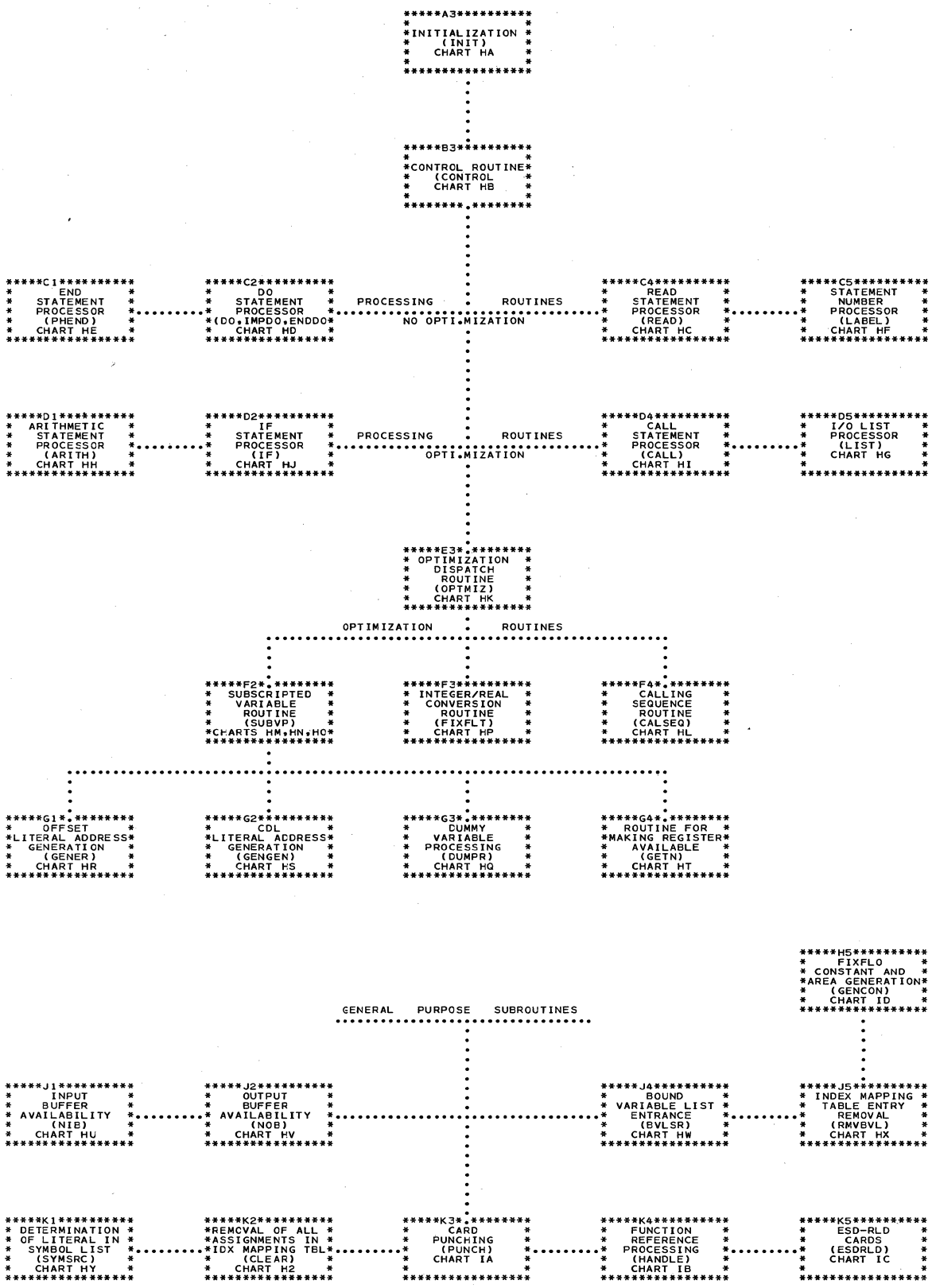


Figure 53. Organization of Phase 20

The ESDRLD routine assigns the address at which the particular library subroutine is located at object time, and then places that address in the pointer field of the text word. The first encounter of any given library subroutine causes text, ESD, and RLD records to be generated unless the ESD table indicates that this subroutine has been used previously. When the entire text for the statement under consideration is moved to the work area, control is passed to the associated routine which processes the text.

If the adjective code in the initial text word for a statement does not indicate that optimization is to be performed, a further check is made of the adjective code. Each of these codes causes control to be passed to an associated routine that may do either of the following:

1. Generate certain information used by the subscript optimization process (e.g., a DO variable is placed on the bound variable list).
2. Refine the text to increase the efficiency of Phase 25 in producing machine language instructions.

Upon completion of its processing, the referenced routine either returns control to the Control routine to move the text to the current output buffer area, or includes the text in its own processing. Subroutine NOB is requested to make an output buffer area available for the next word, as each word is placed in the output buffer area.

Any adjective codes not previously mentioned in this discussion cause the Control routine to move the text to the current output buffer area.

EXIT: Control is passed from the Control routine to one of the following routines depending on the adjective code:

1. LIST routine.
2. ARITH routine.
3. IF routine.
4. CALL routine.

ROUTINES CALLED: During execution the Control routine references the READ, DO, ENDDO, and PHEND routines, and subroutines NIB and NOB.

READ Routine: Chart HC

The READ routine indicates that the I/O list is part of a READ statement. If the I/O list references items that are external to the program being compiled, text, ESD and RLD records are put onto an output data set.

ENTRANCE: The READ routine receives control from the Control routine.

OPERATION: An indicator is set to notify the LIST routine that the current I/O list is part of a READ statement. The statement is put on to the output data set, word by word. A check is then made for the END I/O list. If it is encountered, the indicator is set off.

EXIT: The READ routine exits to the CONTROL routine to process the next statement.

SUBROUTINES CALLED: During execution the READ routine references the ESDRLD routine.

DO, IMPDO, and ENDDO Routines: Chart HD

The DO, IMPDO, and ENDDO routines perform DO statement processing. The processing involved in each of the three routines is limited.

CONSIDERATION: Each iteration of a DO loop increments the DO variable and causes that DO variable to become bound.

DO Routine

The DO routine processes a DO statement.

ENTRANCE: The DO routine receives control from the Control routine.

OPERATION: Upon entry to the DO routine, a DO loop indicator is turned on. (This indicator, upon interrogation within SUBVP, indicates that the current statement is within a DO loop.) To indicate any nested DO loops, the count of the current number of DO loops is incremented by 1. A list of subscript expressions within the DO loop (DO list) is cleared.

The DO variable of the current DO statement is accessed and control is given to subroutine BVLSR to place the DO variable on the bound variable list. Because of this new entry on the bound variable list, subroutine RMVBVL is used to remove register assignments for subscript expressions which utilize that DO variable in the index mapping table.

EXIT: The DO routine exits to the Control routine.

SUBROUTINES CALLED: During execution the DO routine references subroutines BVLSR and RMVBVL.

IMPDO Routine

The IMPDO routine processes the DO variable of an implied DO statement.

ENTRANCE: The IMPDO routine receives control from the LIST or Control routine.

OPERATION: Upon entry to the IMPDO routine, the DO loop indicator is turned on; the count of the current number of DO loops is incremented by 1; and the DO list is cleared.

Subroutine BVLSR places the DO variable of an implied DO statement on the bound variable list, and subroutine RMVBVL removes register assignments from the index mapping table for subscript expressions that utilize the DO variable.

EXIT: The IMPDO routine exits to the OPTMIZ or Control routine.

SUBROUTINES CALLED: During execution the IMPDO routine references subroutines BVLSR and RMVBVL.

ENDDO Routine

The ENDDO routine ensures that the end of a DO loop is recognized.

ENTRANCE: The ENDDO routine receives control from the Control routine.

CONSIDERATION: The word containing an ENDDO adjective code (indicating the end of a DO loop) may be followed by an end mark. If it is, the next statement is accessed. If it is not followed by an end mark, the Control routine would (without ENDDO processing) continue the scan for the end mark, and an entire statement might not be processed.

OPERATION: The ENDDO routine decrements the count of the number of DO loops encountered. If the count equals zero, the DO loop indicator is turned off.

ENDDO is interpreted as an end mark and returns to a point in the Control routine at which it is assumed an end mark had been found.

EXIT: The ENDDO routine passes control back to the Control routine.

PHEND Routine: Chart HE

The PHEND routine completes the processing performed by Phase 20.

ENTRANCE: The PHEND routine receives control from the Control routine, when the adjective code for an END statement is encountered.

CONSIDERATION: The END statement is not to be confused with the end mark indicator, which merely indicates the conclusion of a given statement.

OPERATION: When the END statement is reached, the contents of the text, ESD, and RLD buffers are put onto an output data set, and the END statement word is moved to the output buffer. Additional text beyond the END statement (due to an error condition in the END statement) is also moved to the output buffer. Miscellaneous closing procedures are performed, including a branch into subroutine NOB to write the contents of the output buffer on the work tape. Upon return to the PHEND routine, the output work tapes are rewound. The communications area is updated according to the contents of the location counter whose value has changed since Phase 20 assigned addresses to generated literals.

EXIT: If the error flag in the communications area is set off (indicating that any errors are minor), the PHEND routine passes control to Phase 25. If the error flag in the communications area is set on, PHEND exits according to the following conditions:

1. If the GOGO bit in the communications area is set (indicating compilation of the object program is to continue regardless of the presence of errors), control is passed to Phase 25.
2. If the GOGO bit in the communications area is not set (implying that compilation of the object program is not to continue), control is passed to Phase 30.

SUBROUTINES CALLED: During execution the PHEND routine references subroutines NIB, NOB, and PUNCH.

LABEL Routine: Chart HF

The LABEL routine removes all current register assignments from subscript computations for pertinent referenced statement numbers.

ENTRANCE: The LABEL routine receives control from the Control routine when a statement number is being processed.

CONSIDERATION: A statement number referenced by other statements represents a point of definition, unless it is either the statement number for a FORMAT statement or only referenced by a DO statement.

OPERATION: The LABEL routine determines if a point of definition exists by checking the status of appropriate bits in the Usage field of the overflow table entry for the statement number (see the introduction to Phase 10). If a point of definition exists, subroutine CLEAR sets the status of all entries in the index mapping table to unassigned; all registers are then available.

EXIT: The LABEL routine exits to the Control routine to continue processing the statement referenced by the statement number.

SUBROUTINE CALLED: During execution the LABEL routine references subroutine CLEAR.

LIST Routine: Chart HG

The LIST routine processes the I/O list substatement of a READ statement.

ENTRANCE: The LIST routine receives control from the Control routine when an I/O list substatement is detected.

CONSIDERATIONS: An I/O list substatement is part of a statement; however, it is treated within text as if it were a statement.

When an I/O operation is caused by a READ statement, the value of the variables in the I/O list substatement changes; (i.e., they become bound). When a READ statement is encountered by the Control routine, control is passed to the READ routine to set a READ statement indicator for the LIST routine. In this way, the LIST routine knows that the current I/O list substatement is associated with a READ statement.

OPERATION: If the READ statement indicator is on, it is then set off in preparation for the processing of subsequent I/O list substatements. During the processing which follows, the LIST routine searches for integer variables that become bound as a result of the read that occurs because of the READ statement. The LIST routine, as it processes an I/O list substatement within a READ statement, checks the mode code for 'Integer.' If 'Integer' is found, the corresponding integer variable is accessed and control is passed to subroutine BVLSR to place that variable on the bound variable list.

When the end mark indicator is reached, the OPTMIZ routine optimizes the I/O list substatement. When control is returned to the LIST routine, all registers assigned to subscript expressions with bound variables are made available by calling subroutine RMVBVL.

When the READ statement indicator is off, the List routine detects that the current I/O list is not associated with a READ statement. Control is passed to OPTMIZ to optimize the I/O list. Return from OPTMIZ, in this case, causes a direct exit from the LIST routine.

EXIT: The LIST routine exits to the Control routine to process the next statement.

ROUTINES CALLED: During execution the LIST routine references the OPTMIZ routine and subroutines BVLSR and RMVBVL.

ARITH Routine: Chart HH

The ARITH routine optimizes an arithmetic statement.

ENTRANCE: The ARITH routine receives control from the Control routine.

CONSIDERATION: The symbol preceding an equal sign in an arithmetic statement (the initial symbol) changes value during the execution of the object program. If this symbol is an integer variable, it is a bound variable. Function arguments may also be bound variables.

OPERATION: The ARITH routine accesses and examines a text word for a function reference. If such a reference is found, the ARITH routine passes control to subroutine HANDLE, which sets an indicator. This indicator is examined by the ARITH routine after optimization to determine if a function is referenced in the arithmetic statement and to set the exit from ARITH, accordingly. HANDLE also places any function argument that is an integer variable on the bound variable list.

If no function reference is found by the ARITH routine, the text word is checked for an operation code of 'Store Integer.' When this code is present, subroutine BVLSR places the associated integer variable (the initial symbol of the arithmetic statement) on the bound variable list.

When the end mark for the statement is encountered, the OPTMIZ routine is used to optimize the statement. The point of return from the OPTMIZ routine is used to check the status of the indicator set by HANDLE. If this indicator is on, control is passed to the CALL routine. At this time, a variable which is both in COMMON and part of a subscript expression within the index mapping table causes that subscript expression to be deleted from the index mapping table.

Regardless of the indicator setting, subroutine RMVBVL is used to make available those registers assigned to subscript expressions with bound variables.

EXIT: The ARITH routine exits either to the CALL routine as a result of the indicator set by HANDLE being set on or to the Control routine to process the next statement.

ROUTINES CALLED: During execution, the ARITH routine references the HANDLE and OPTMIZ routines and subroutines BVLSR and RMVBVL.

CALL Routine: Chart HI

The CALL routine causes optimization of a CALL statement.

ENTRANCE: The CALL routine receives control from the Control routine. It is also entered at a midpoint from the ARITH and IF routines when a function CALL is part of an arithmetic or IF statement.

CONSIDERATION: Integer variables which are arguments in a CALL statement or are located in COMMON, may become bound as the result of a subprogram execution.

OPERATION: The mode code of each parameter is examined. If an integer type code is found, subroutine BVLSR places the corresponding integer variable on the bound variable list.

When the end mark indicator is encountered, the OPTMIZ routine optimizes the CALL statement. When control is returned to any CALL routine, the variable which is both in COMMON and part of a subscript expression within the index mapping table causes that subscript expression to be deleted from the index mapping table.

Subroutine RMVBVL then removes all register assignments in the index mapping table from those subscript expressions containing bound variables.

EXIT: The CALL routine exits to the Control routine to begin processing the next statement.

ROUTINES CALLED: During execution, the CALL routine references the OPTMIZ routine and subroutines BVLSR and RMVBVL.

IF Routine: Chart HJ

The IF routine optimizes the arithmetic expression of an IF statement.

ENTRANCE: The IF routine receives control from the Control routine.

CONSIDERATION: The arithmetic expression of the IF statement may contain a function reference and, therefore, function arguments which may be bound variables.

OPERATION: The IF routine accesses and examines a text word for a function reference. If such a reference is found, the IF routine passes control to subroutine HANDLE which sets an indicator. This indicator is examined by the IF routine, after optimization, to determine if a function is referenced in the IF statement. HANDLE also places any function argument that is an integer variable on the bound variable list.

The IF routine processes the IF statement text until it encounters an IF forcing adjective code (set in Phase 15) denoting the end of the arithmetic expression or an end of statement indicator. When this code is encountered, the OPTMIZ routine is used to optimize the arithmetic expression.

The point of return from the OPTMIZ routine is used to check the status of the indicator set by HANDLE. If this indicator is on, it is turned off and control is passed to a point within the CALL routine. At this time, a variable which is both in COMMON and part of a subscript expression within the index mapping table causes that subscript expression to be deleted from the index mapping table.

EXIT: The IF routine exits either to a point within the CALL routine as a result of the indicator set by HANDLE being set on, or to the Control routine to process the statement numbers in the IF statement.

ROUTINE CALLED: During execution, the IF routine references the OPTMIZ routine and subroutine HANDLE.

OPTMIZ Routine: Chart HK

The OPTMIZ routine is a dispatch routine for further processing of statements which are to be optimized.

ENTRANCE: OPTMIZ receives control from LIST, ARITH, CALL, and IF routines when optimization is required.

OPERATION: For statements which are to be optimized, the OPTMIZ routine scans the associated text (which is in the work area) for specific adjective codes. These adjective codes indicate various processing as follows:

1. The adjective code for external function and arithmetic statement function CALL causes control to be given to the CALSEQ routine to perform processing connected with the argument list.
2. The adjective code for IFIX, FLOAT, and DFLOAT causes control to be given to the FIXFLO routine to initialize an area and a constant to be used in the conversion between real and integer quantities.
3. The adjective code for a subscripted variable causes control to be given to subroutine SUBVP to perform optimization with respect to any subscript expressions.
4. The adjective code for an implied DO causes control to be given to the IMPDO routine to process the DO variable of the implied DO.
5. The adjective code for an end DO causes control to be given to the ENDO routine to ensure that the end of a DO loop is recognized.

In all three cases control is returned to the OPTMIZ routine to complete the processing of the statement.

An end mark indicator causes the end mark text word to be moved to a current output buffer area. Control is passed to subroutine NOB to assure the availability of an output buffer area for the next output word. Control is then returned to the routine that referenced the OPTMIZ routine.

If the adjective code does not indicate any of the preceding conditions, the text word is moved to the current output buffer area. Control is passed to subroutine NOB to assure the availability of an output buffer area for the next output word. The text scan is then continued.

EXIT: The OPTMIZ routine returns control to the calling routine.

ROUTINES CALLED: During execution the OPTMIZ routine references the CALSEQ, FIXFLO, and SUBVP routines and subroutine NOB.

CALSEQ Routine: Chart HL

The CALSEQ routine assigns addresses to the argument list entries and causes the relevant ESD, RLD, and text information to be entered on the respective cards.

ENTRANCE: The CALSEQ routine is entered from the OPTMIZ routine.

CONSIDERATION: The arguments of a CALL statement are accessed by a program external reference. Therefore, RLD cards as well as text cards are required.

OPERATION: If this is a CALL with no arguments, the CALL text word is placed onto the output data set, and a return to the OPTMIZ routine is made.

Each argument in the argument list is accessed and is entered onto a text card with an object time address. If the argument is subscripted, the Phase 20 subscript processing is accomplished by subroutine SUBVP. For subscript variables used as arguments, a zero address is entered into the text cards.

RLD cards are punched for non-subscripted arguments. An argument in COMMON is assigned an external symbol identification (ESID) of 02. An argument not in COMMON is assigned an ESID of 01.

An indication of the last argument in the argument list is made, unless that argument is subscripted. For a subscripted argument the indication is made at object time, because the subscript calculation at object time would destroy a compile-time indicator.

EXIT: The CALSEQ routine returns control to the OPTMIZ routine.

SUBROUTINES CALLED: Subroutine CALSEQ calls the SUBVP routine, and subroutines CALTXT and CALRLD.

Subroutine SUBVP: Charts HM, HN, HO

The SUBVP routine optimizes a subscript expression.

ENTRANCE: The SUBVP routine receives control from the OPTMIZ routine or subroutine CALSEQ.

CONSIDERATION: The subscript text input and output are discussed in the introduction to Phase 20.

OPERATION: The SUBVP routine initially determines if the current subscript expression is constant. A constant subscript expression does not require a register assignment.

For other than a constant subscript expression, a register assignment is made in the index mapping table. If no register is available, control is given to the GETN routine to free the register assigned to the subscript expression with the least dimension.

At this point in the process, when a dummy subscripted variable is encountered, control is given to subroutine DUMPR to process that dummy variable. If the subscripted variable is not a dummy, then subsequent processing of the subscript expression depends on whether that expression is constant or has already been optimized at a previous encounter. Either of the two preceding conditions causes the generation of subscript text output with an adjective code of AOP, unless this is the first occurrence of the expression within a DO loop. If it is, the expression is left in SAOP form.

If the subscript expression is not constant or has not been previously optimized, either subscript text output with an adjective code of SAOP or subsequent text output with an adjective code of XOP is generated. The form of generated text depends on whether or not a literal must be produced for the offset portion of the array displacement.

EXIT: The SUBVP routine returns control to the routine that referenced it.

SUBROUTINES CALLED: During execution, the SUBVP routine references the GETN and DUMPR routines and subroutines GENER, GENGEN, and NOB.

FIXFLO Routine: Chart HP

The FIXFLO routine assigns an area and a constant to be used by the library subroutines IFIX and IFLOAT.

ENTRANCE: The FIXFLO routine receives control from the OPTMIZ routine whenever a FUNCTION adjective code is encountered within the text of the statement to be optimized.

OPERATION: For a call either to IFIX or IFLOAT, a text word containing the address of a constant and a work area is generated. If the constant has previously been assigned an address, that address is used. If the work area has been generated previously, the address of that area is used. Subroutine GENCON moves the constant or the work area definition to the text card area and puts a card image on an output data set, if necessary.

EXIT: The FIXFLO routine exits to the OPTMIZ routine.

SUBROUTINES CALLED: The FIXFLO routine references subroutines GENCON and NOB.

DUMPR Routine: Chart HQ

The DUMPR routine processes subscripted dummy variables.

ENTRANCE: The DUMPR routine receives control from subroutine SUBVP.

CONSIDERATION: No more than three dummy variables are allowed in the index mapping table at any given time.

OPERATION: The DUMPR routine determines if the dummy variable in text is currently assigned a register in the index mapping table. If one is assigned, it can be used for subsequent processing by subroutine DUMPR.

If no register has previously been assigned to the dummy variable, one must now be assigned. If three dummy variables are already assigned registers in the index mapping table, one of these registers is reassigned to the current dummy subscripted variable.

However, if less than three dummy variables are assigned registers in the index mapping table, the first unassigned register (if one exists) is assigned to the current dummy variable. The entry with the least dimension is spilled (made available) if all registers were previously assigned.

After a register is assigned to the current dummy variable, a 'load base register' instruction is generated and placed in the output area. This instruction precedes the text for the dummy subscripted variable. The register number used for the instruction is placed in text.

EXIT: The DUMPR routine returns control to the SUBVP routine.

Subroutines GENER, GENGEN: Chart HR

Subroutine GENER

Subroutine GENER provides linkage to subroutine GEN for the generation of offset literals.

ENTRANCE: Subroutine GENER receives control from the SUBVP routine.

CONSIDERATION: When subroutine SUBVP encounters a value outside the range of 0 through 4095 (the largest number physically used as an offset) by the addition of the offset (calculated in Phase 10) to the displacement (determined in Phase 15), subroutine GENER is called to generate an offset literal. The adjective code is changed from SAOP to XOP or, in the case of an AOP, indicators are set to indicate to Phase 25 that an offset literal is generated and replaces the offset in text.

OPERATION: Upon entry into subroutine GENER, the offset to be generated as a literal is passed as a parameter to subroutine GEN, which generates the literal.

GENER modifies the text to indicate that an offset literal is generated and to supply the address of that literal.

EXIT: Subroutine GENER returns control to the SUBVP routine.

SUBROUTINE CALLED: During execution GENER references subroutine GEN.

Subroutine GENGEN

Subroutine GENGEN provides linkage to subroutine GEN for the generation of CDL literals.

ENTRANCE: Subroutine GENGEN receives control from the SUBVP routine.

OPERATION: Upon entry in subroutine GENGEN, the CDL portion to be generated as a literal is passed as a parameter to subroutine GEN, which generates the literal.

EXIT: Subroutine GENGEN returns control to subroutine SUBVP.

SUBROUTINES CALLED: During execution, subroutine GENGEN references subroutine GEN.

Subroutine GEN: Chart HS

ENTRANCE: Subroutine GEN receives control from subroutine GENER or GENGEN.

OPERATION: Subroutine GEN determines if the passed literal is already in the symbol list. Presence of that literal in the table causes subroutine SYMSRC to return the address of that literal to subroutine GEN.

If the literal is not in the symbol list, GEN must assign the address of the literal using the location counter and produce the corresponding text card output. Subroutine GEN uses the current contents of the location counter as the address of the offset literal. This address, along with the offset literal, is placed in the source symbol table and the symbol list.

GENER moves the offset literal to the text card area. If the area is full, PUNCH is accessed.

EXIT: Subroutine GEN returns control to the calling routine.

SUBROUTINES CALLED: During execution, subroutine GEN references subroutines SYMSRC and PUNCH.

GETN Routine: Chart HT

The GETN routine makes a register available.

ENTRANCE: GETN receives control from the SUBVP routine.

CONSIDERATION: When subroutine SUBVP prepares to assign a register to the current subscript expression, there may be no registers available. Control is then passed to subroutine GETN which accesses a register.

OPERATION: Each entry in the index mapping table is examined until an entry with the least dimension is found. The entry is deleted and the corresponding register is made available. The available register number is then placed in a specified field that can be referenced by subroutine SUBVP.

EXIT: GETN returns control to subroutine SUBVP at the point at which it is called. The two exception exits in GETN are:

1. To EXRTN within SUBVP if the Mode/Type code field indicates a constant subscript and a generated literal.

2. To CLOSE within the PHEND routine if there are no entries for subscript expressions in the index mapping table.

Note: Exception exit 1 is taken after a register is found and its number is returned by a location, but before the index mapping table entry is deleted.

Subroutine NIB: Chart HU

Subroutine NIB updates a current input buffer pointer.

ENTRANCE: Subroutine NIB receives control from the Control, READ, and PHEND routines.

CONSIDERATION: There are two input buffers.

OPERATION: A buffer pointer is updated until the end of the current buffer is reached. The next buffer is then accessed for processing purposes and the first buffer is refilled. If an end of tape indicator is detected during the read, an error indicator is set. This indicator is examined within NIB, immediately before any read is executed.

EXIT: Subroutine NIB returns control to the Control routine. An exception exit to the CLOSE routine is made if there is an error in the previous read.

Subroutine NOB: Chart HV

Subroutine NOB keeps a buffer area available for output.

ENTRANCE: Subroutine NOB receives control from the Control, PHEND, OPTMIZ, INIT, AOP, FIXFLO, READ, SUBVP, and DUMPR routines and subroutine CALSEQ.

CONSIDERATION: There are two output buffers.

OPERATION: The output buffer pointer is updated until the end of the current buffer is reached. At that time, the alternate buffer is made current and the full buffer is written on the work tape.

EXIT: Subroutine NOB returns control to the routine that referenced it.

Subroutine BVLSR: Chart HW

Subroutine BVLSR enters bound variables on the bound variable list.

ENTRANCE: Subroutine BVLSR receives control from the DO, IMPDO, LIST, ARITH, and CALL routines.

OPERATION: If the bound variable is not currently on the bound variable list, it is now entered. (See the introduction to Phase 20.)

EXIT: Subroutine BVLSR returns control to the routine which referenced it.

Subroutine RMVBVL: Chart HX

Subroutine RMVBVL removes register assignments from the index mapping table for subscript expressions containing bound variables.

ENTRANCE: Subroutine RMVBVL receives control from the DO, IMPDO, LIST, ARITH, and CALL routines.

OPERATION: If the bound variable list is empty, there is no processing and an immediate exit is taken. If the bound variable list is not empty, the status of each index mapping table entry is examined. If the associated register is assigned to a subscript expression, that expression is checked against the bound variable list. If a bound variable is in the expression, the status of the associated index mapping table entry is set to unassigned. The exit occurs after the entire index mapping table is processed.

EXIT: Subroutine RMVBVL returns control to the routine which referenced it.

Subroutine SYMSRC: Chart HY

Subroutine SYMSRC determines if a literal is in the symbol list.

ENTRANCE: Subroutine SYMSRC receives control from subroutine GEN.

CONSIDERATION: The format of the symbol list is as follows:

literal	address
literal	address
literal	address

The address of this table is indicated in the FORTRAN communications area.

OPERATION: If the literal is in the symbol list, the address of the literal is returned to the subroutine which referenced it. If the literal is not in the table, a zero function is returned.

EXIT: Subroutine SYMSRC returns control to subroutine GEN.

Subroutine CLEAR: Chart HZ

Subroutine CLEAR removes all entries from the index mapping table at a point of definition to insure the availability of all registers used for subscript expressions.

ENTRANCE: Subroutine CLEAR receives control from the LABEL routine.

CONSIDERATION: The format of the index mapping table is described in the introduction to Phase 20 (see Figure 20.1).

OPERATION: The status of each entry in the index mapping table is examined. If the status is unassigned (i.e., the register is available), the next entry is accessed. If the status is assigned, it is changed to indicate that the register is now available. The next entry is then accessed.

EXIT: Subroutine CLEAR returns to the LABEL routine.

Subroutine PUNCH: Chart IA

Subroutine PUNCH processes a card buffer.

ENTRANCE: Subroutine PUNCH receives control from the PHEND routine and subroutine GENER.

OPERATION: A Supervisor Call instruction is directed to the FORTRAN System Director to have the contents of the card buffer punched, if the DECK option is specified. If the NODECK option is specified, no cards are punched. If GO is requested, the contents of the buffer are written on the GO tape.

EXIT: Subroutine PUNCH returns either to the PHEND routine or subroutine GENER.

Subroutine HANDLE: Chart IB

Subroutine HANDLE processes function references in an arithmetic statement or in an arithmetic expression of an IF statement.

ENTRANCE: Subroutine HANDLE receives control from either the ARITH or IF routine.

CONSIDERATION: Integer variables which are arguments in a function call or are located in COMMON, may become bound as the result of a subprogram execution.

OPERATION: Subroutine HANDLE sets an indicator for the ARITH or IF routine to indicate that a function call occurred within the statement being processed. The on condition of this indicator later causes an exit to be taken from the calling routines to a location within the CALL routine to initiate a search of the index mapping table for variables in COMMON.

HANDLE places any integer parameters of the function call on the bound variable list.

EXIT: Subroutine HANDLE returns control to the routine that referenced it.

SUBROUTINE CALLED: During execution, subroutine HANDLE references subroutine BVLSR.

Subroutine ESDRLD/CALRLD/CALTXT: Chart IC

This subroutine generates ESD, RLD, and text cards during Phase 20, whenever necessary.

ENTRANCE: Subroutine ESDLRD/CALRLD/CALTXT is entered at ESDRLD whenever a reference to an external symbol is made. Subroutine ESDRLD/CALRLD/CALTXT is entered at CALRLD whenever an argument list is processed. Subroutine ESDRLD/CALRLD/CALTXT is entered at CALTXT whenever text information is to be entered for an argument list.

CONSIDERATION: References to an external symbol occur in the OPTMIZ routine for references to IBCOM, IBERR, and CGOTO, and in the ARITH routine for references to the various exponentiation routines.

The ESD table contains the address of each external symbol and an area into which the address of the respective routine will

be loaded (the address constant). This field is initially zero. The ESD table has the following format:

address	symbol
	a (IBCOM)
	a (FRXPI)
	a (CGOTO)
	a (FRXPR)
	a (FIXPI)
	a (FDXPI)
	a (FDXPD)
	a (IBERR)
2 bytes	2 bytes

OPERATION: When subroutine ESDRLD/CALRLD/CALTXT is entered at ESDRLD, the ESD table is checked to determine if the symbol has been referenced. If not, an address constant address is assigned to the symbol and entered into the ESD table. ESD, text, and RLD entries in their respective areas are then set up. If any card areas are full, subroutine ESDPUN is referenced to place the contents of the full areas on an output data set.

When subroutine ESDRLD/CALRLD/CALTXT is entered at entry point CALRLD, an RLD entry is made in the current RLD area. If both RLD areas are not full, a return is made. If the RLD areas are full, the text, ESD, and full RLD card areas are placed on an output data set by subroutine ESDPUN.

When subroutine ESDRLD/CALRLD/CALTXT is entered at entry point CALTXT, a text card entry is made. If this entry does not fill the text card area, a return is made. If the text card area is full, the text card area along with the ESD area and any RLD area that is full are placed on an output data set.

EXIT: Subroutine ESDRLD/CALRLD/CALTXT returns control to the calling routine.

SUBROUTINES CALLED: During execution subroutine ESDRLD/CALRLD/CALTXT calls subroutine ESDPUN.

Subroutine GENCON: Chart ID

Subroutine GENCON moves the constant or the work area definition from the FIXFLO routine into the text card area.

ENTRANCE: Subroutine GENCON receives control from subroutine FIXFLO.

OPERATION: The text card area is examined. If a double word will not fit in that area, a text card is punched using subroutine PUNCH. The constant or the work area definition is then placed at the beginning of the text card area. A test is made to determine if the text card area is full. If it is full a card is punched; if not, no card is punched.

EXIT: Subroutine GENCON passes control to FIXFLO.

SUBROUTINE CALLED: During execution, subroutine GENCON references subroutine PUNCH.

Subroutine ESDPUN: Chart IE

Subroutine ESDPUN performs the card output for Phase 20.

ENTRANCE: Subroutine ESDPUN is entered from subroutine ESDRLD/CALRLD/CALTXT.

CONSIDERATION: There is a count in the FORTRAN communications area of the number of cards punched.

OPERATION: The current card number and card image are moved into an available card buffer. If no source errors exist and the DECK option is specified, a card is punched; or if a GO option is specified, a card image is placed onto the GO tape. However, if source errors do exist, neither the GO nor DECK option are checked and an immediate return is made.

EXIT: Subroutine ESDPUN returns control to the subroutine that referenced it.

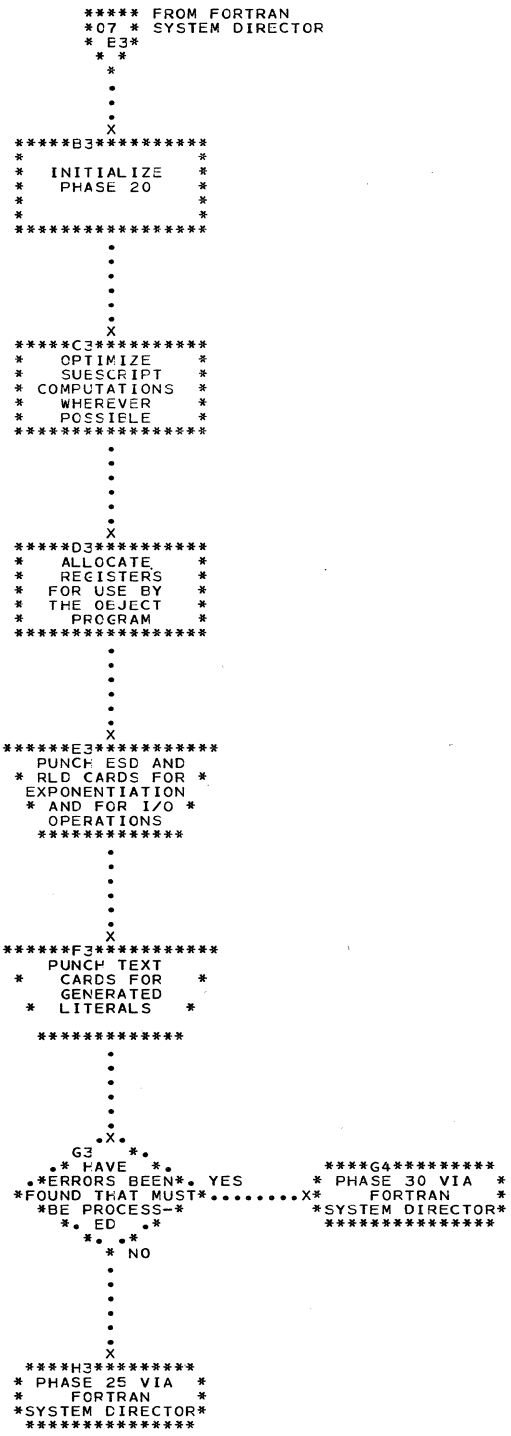


Chart 07. Phase 20 Overall Logic Diagram

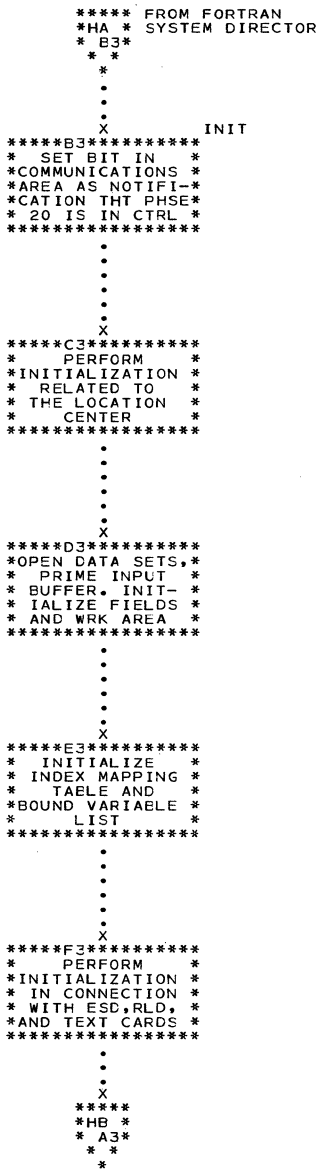
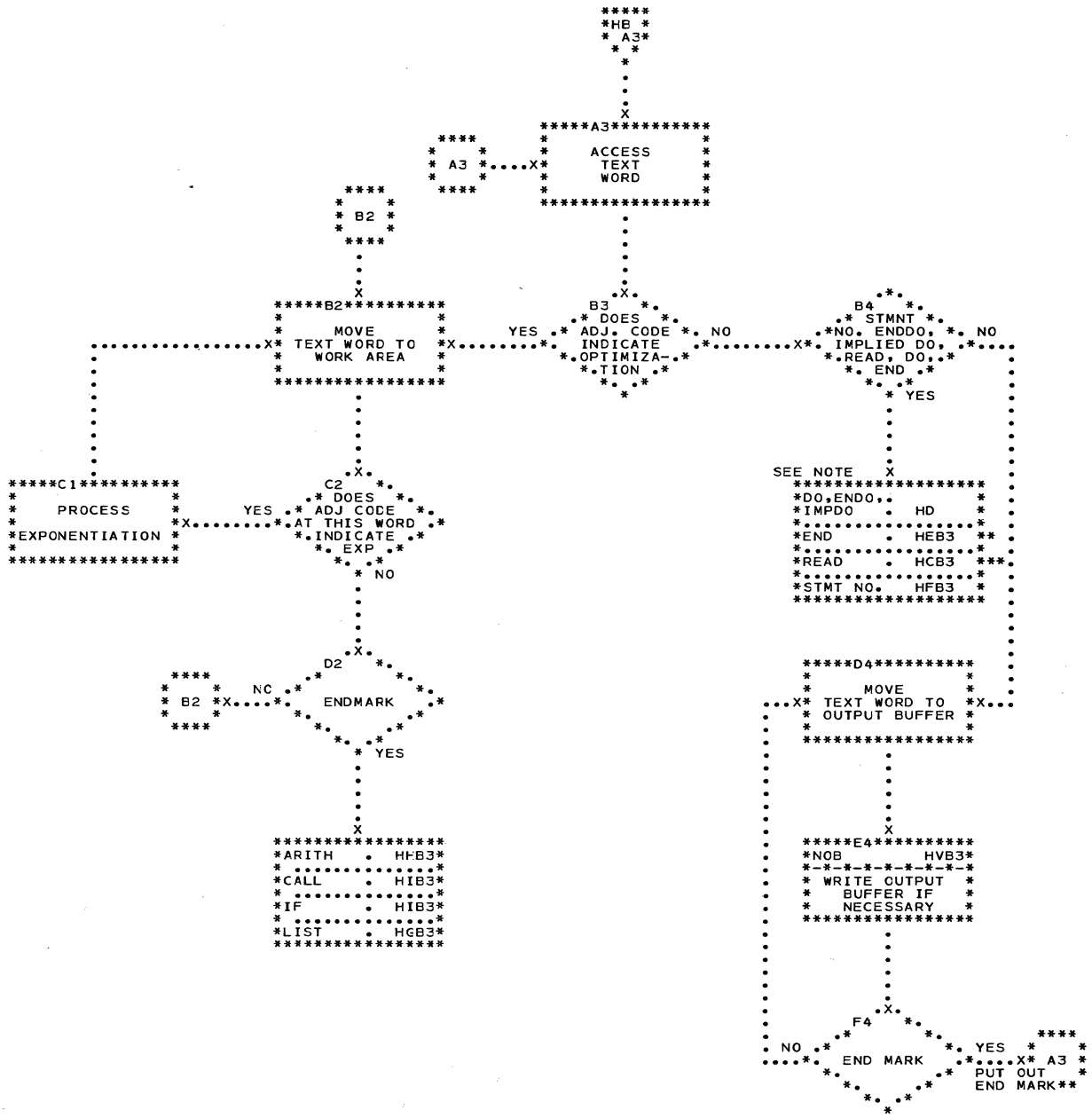


Chart HA. INIT Routine



* END - EXIT FROM PHASE 20 OCCURS WHEN THE END STATEMENT IS PROCESSED. THE EXIT CONDITIONS ARE EXPLAINED IN THE TEXT FOR THE END STATEMENT ROUTINE.

**READ - THE READ ROUTINE RETURNS TO A3 TO ACCESS A WORD, NOT TO D4.

Chart HB. CONTROL Routine

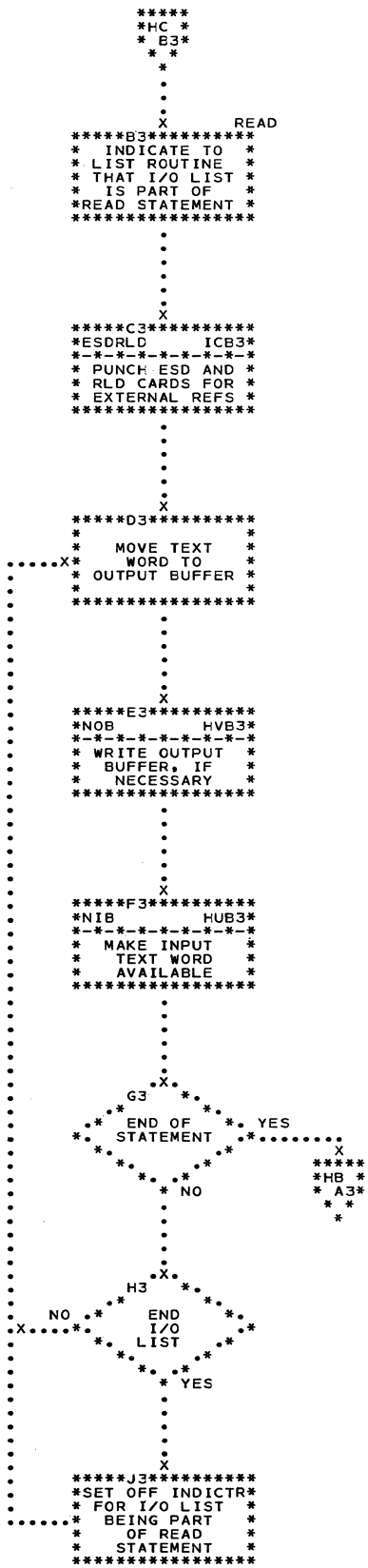


Chart HC. READ Routine

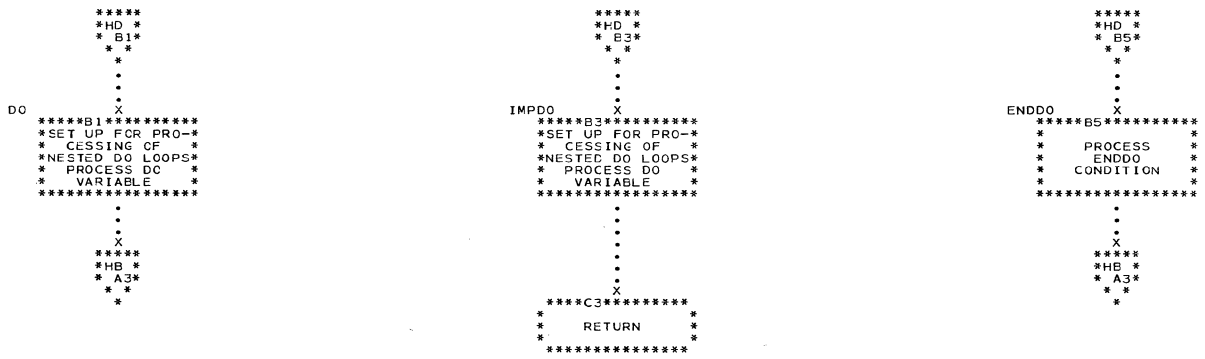


Chart HD. DO/IMPDO/ENDDO Routines

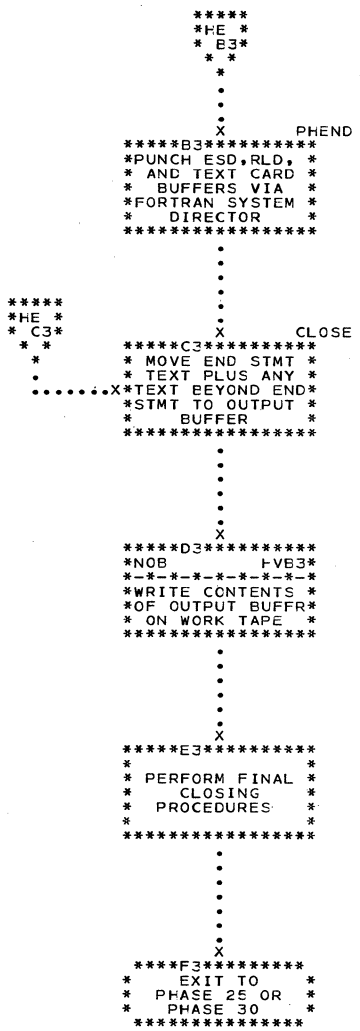


Chart HE. PHEND Routine

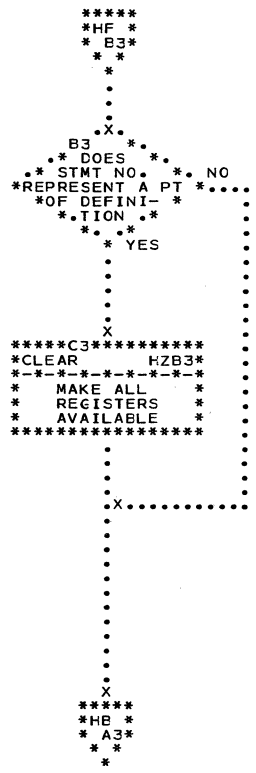


Chart HF. LABEL Routine

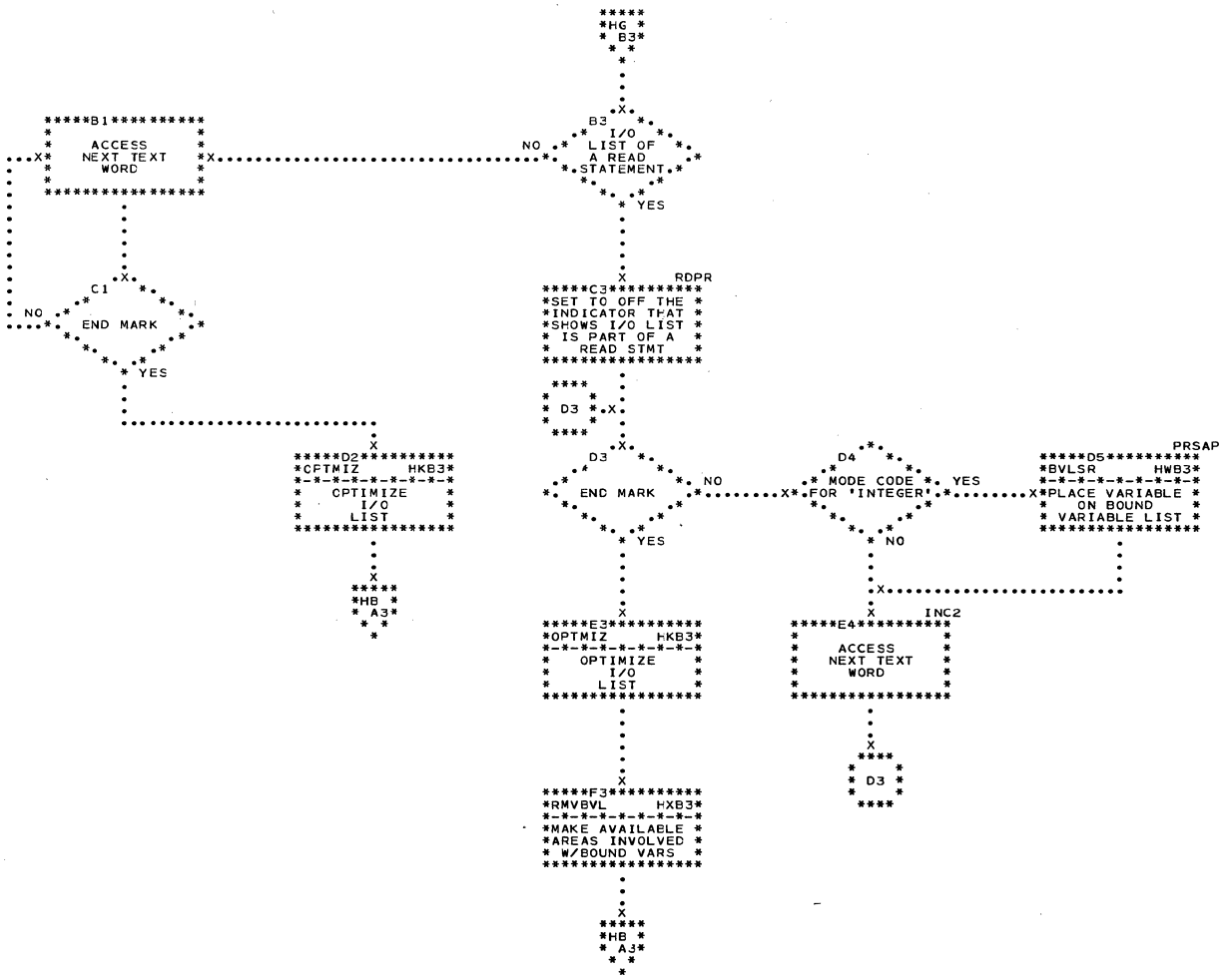


Chart HG. LIST Routine

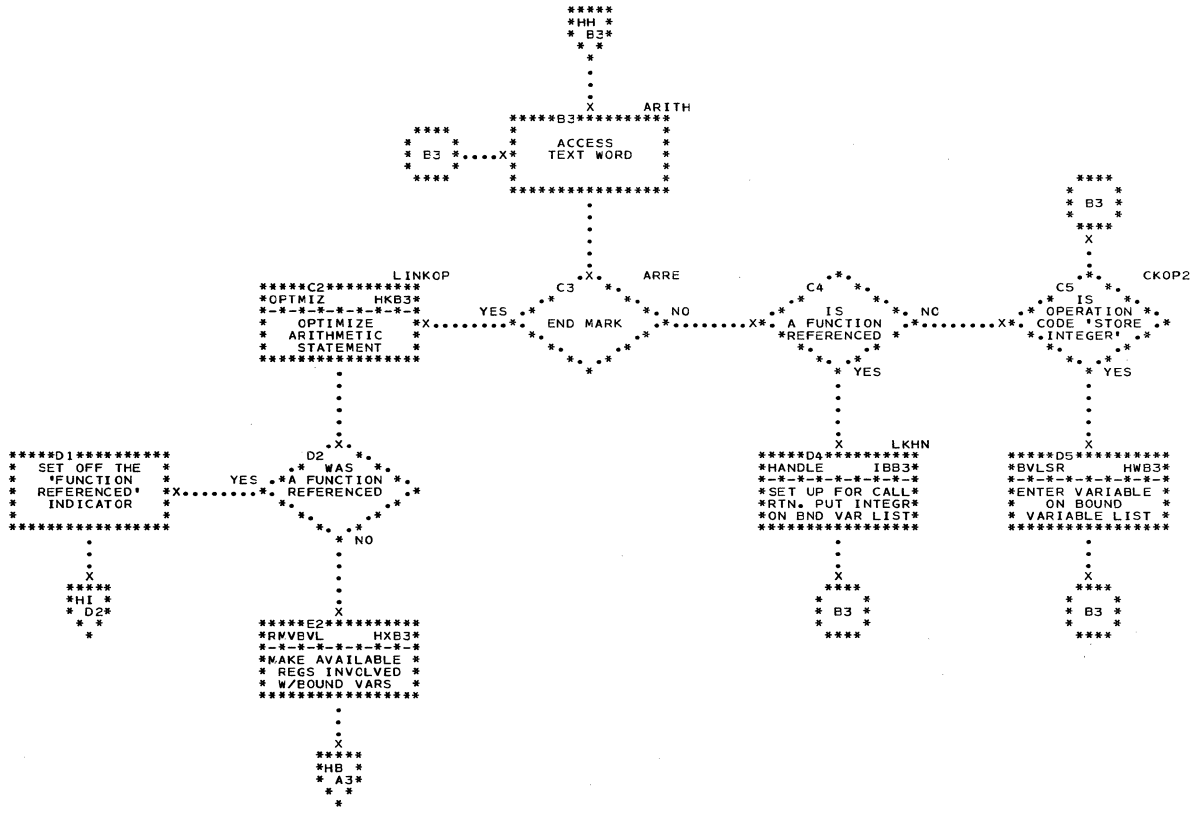


Chart HH. ARITH Routine

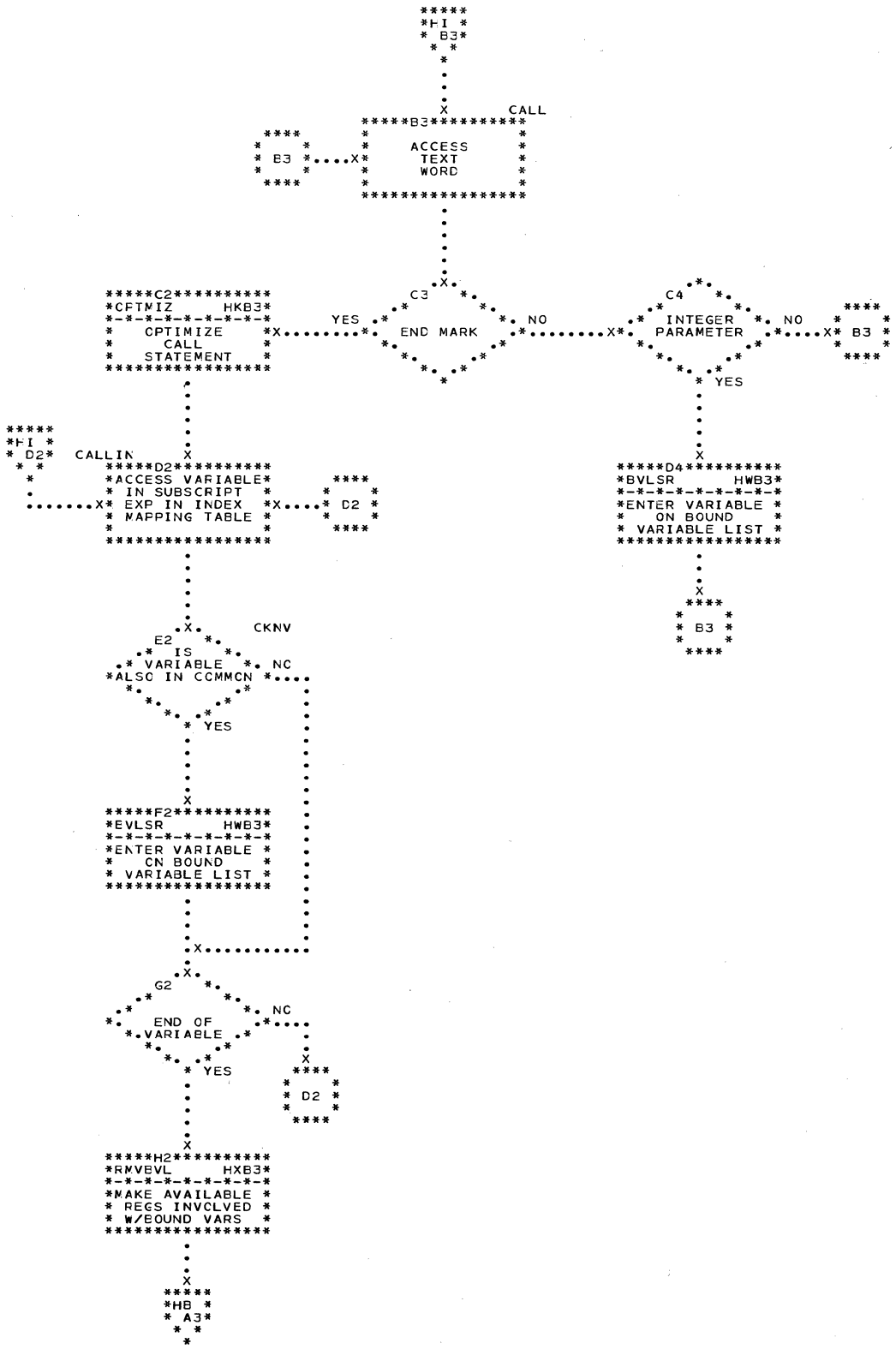


Chart HI. CALL Routine

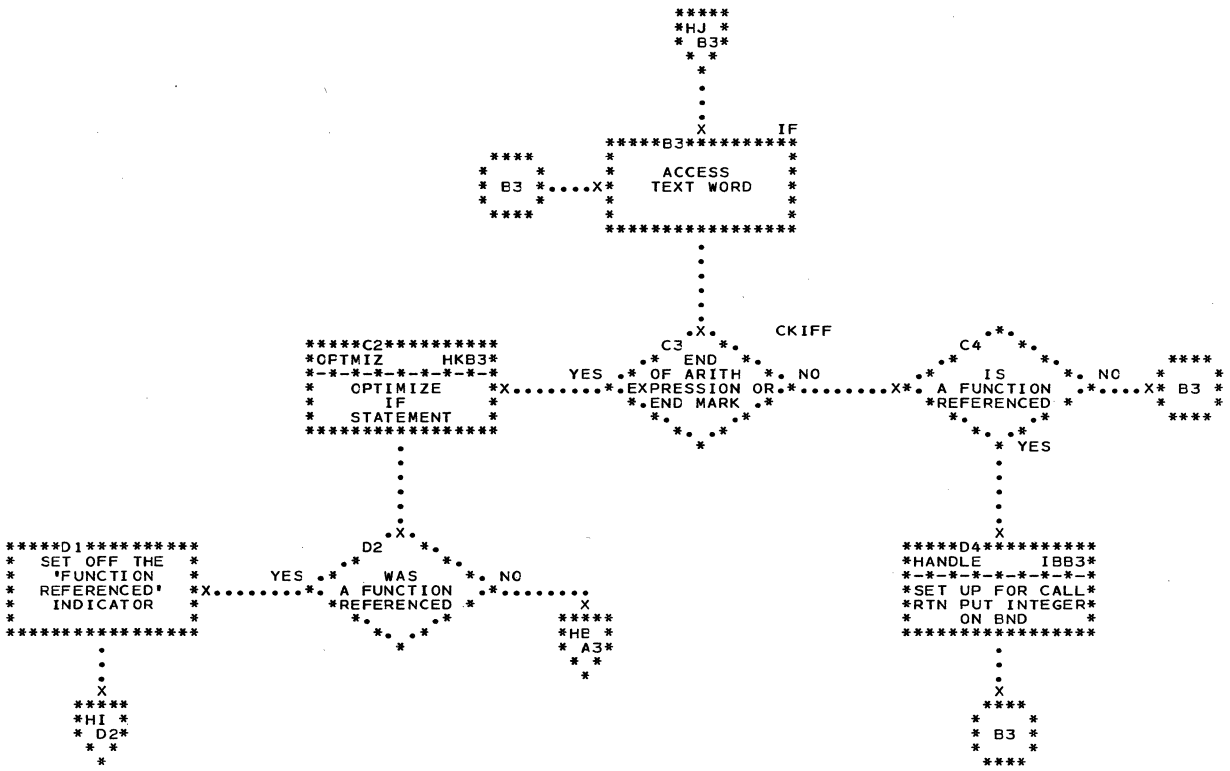


Chart HJ. IF Routine

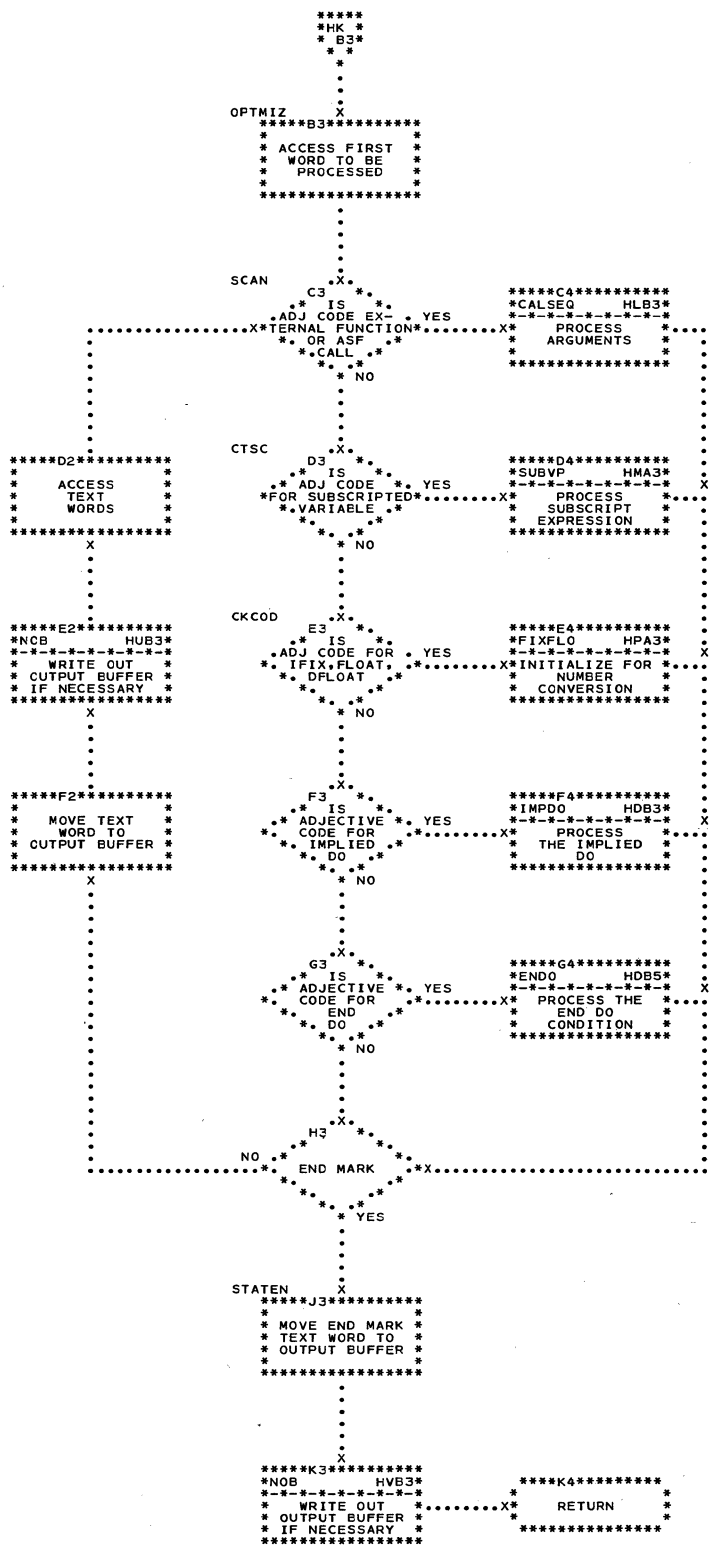


Chart HK. OPTMIZ Routine

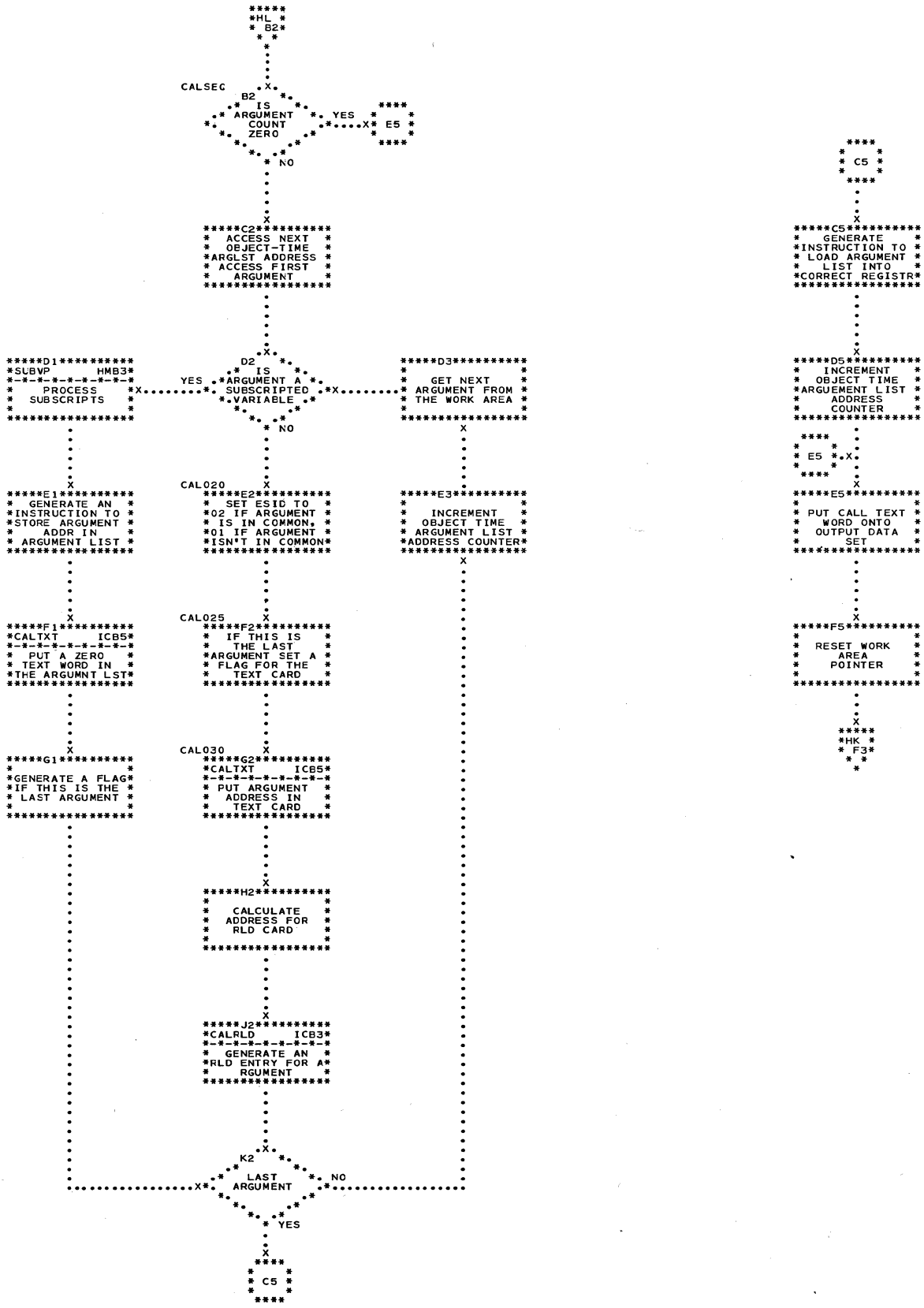


Chart HL. CALSEQ Routine

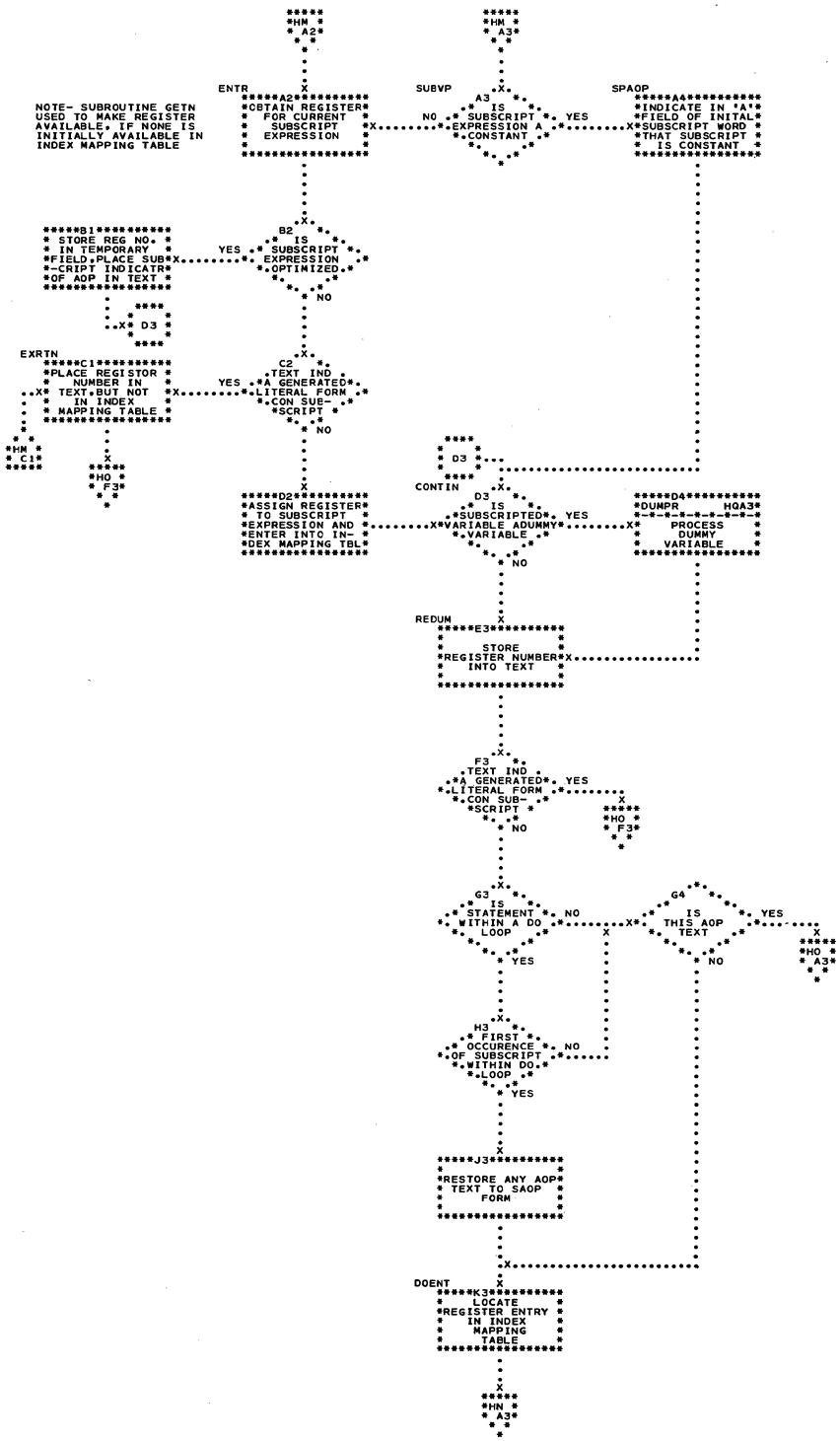


Chart HM. Subroutine SUBVP (1)

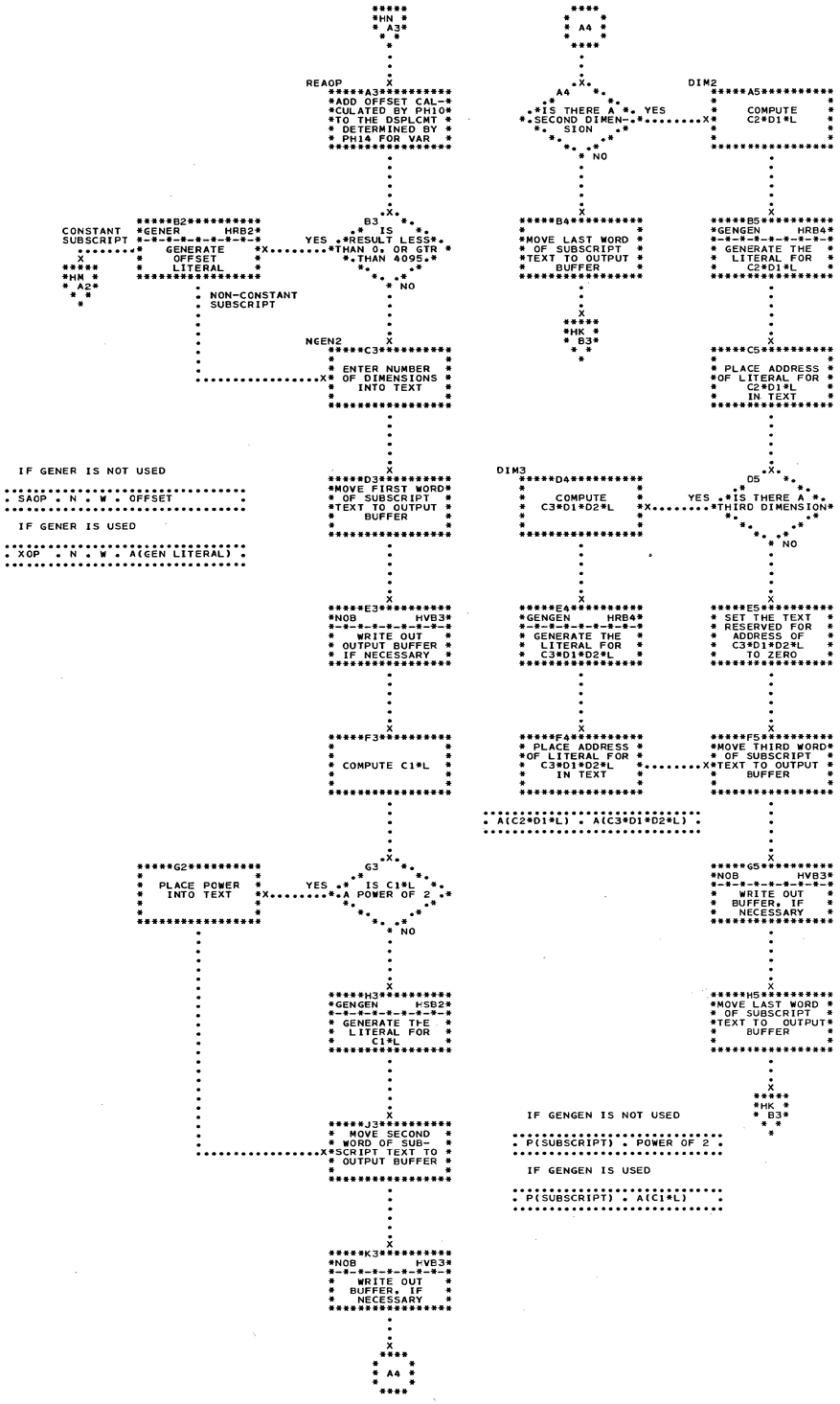
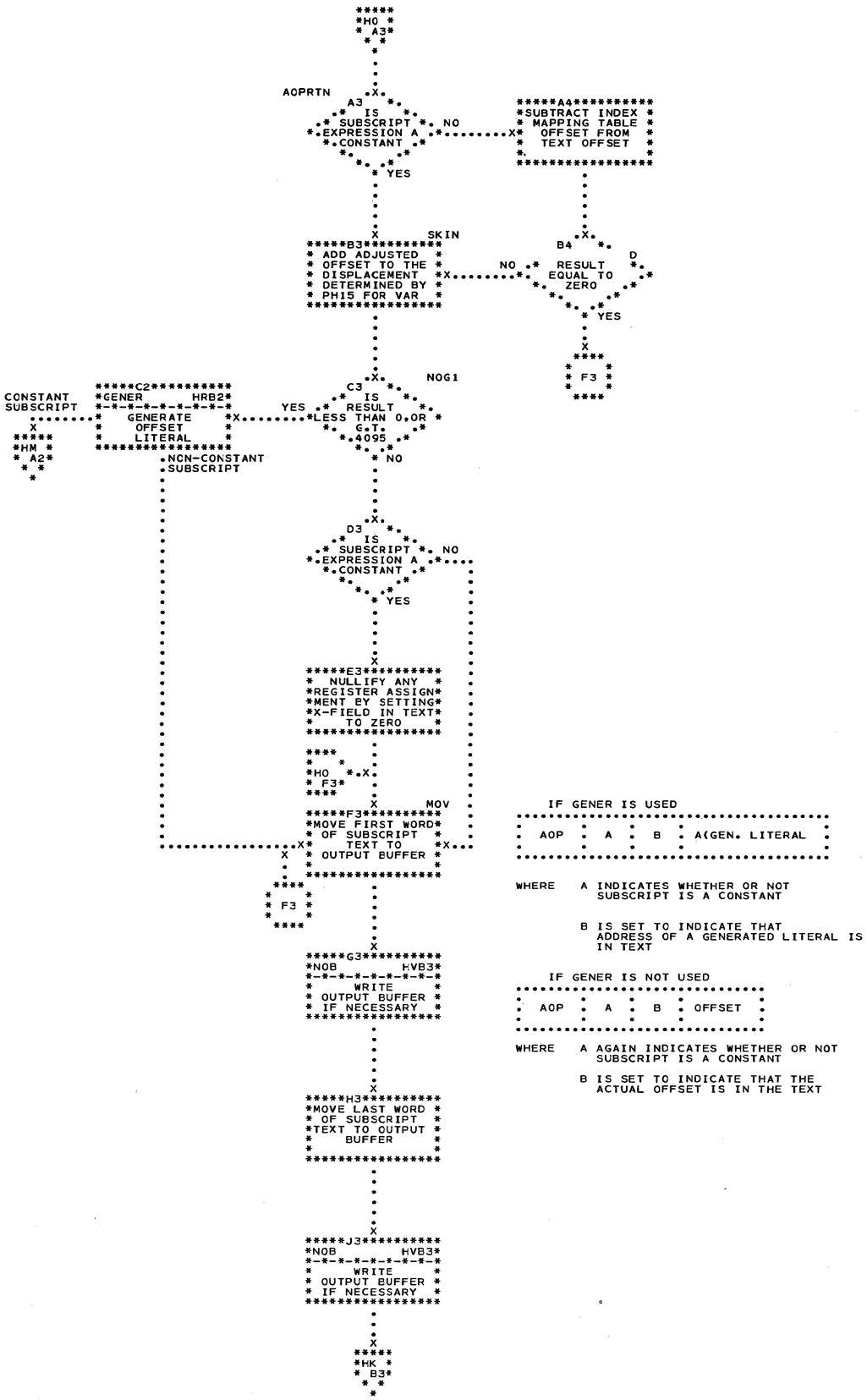


Chart HN. Subroutine SUBVP (2)



IF GENER IS USED

AOP	A	B	A(GEN. LITERAL

WHERE A INDICATES WHETHER OR NOT SUBSCRIPT IS A CONSTANT

B IS SET TO INDICATE THAT ADDRESS OF A GENERATED LITERAL IS IN TEXT

IF GENER IS NOT USED

AOP	A	B	OFFSET

WHERE A AGAIN INDICATES WHETHER OR NOT SUBSCRIPT IS A CONSTANT

B IS SET TO INDICATE THAT THE ACTUAL OFFSET IS IN THE TEXT

Chart HO. Subroutine SUBVP (3)

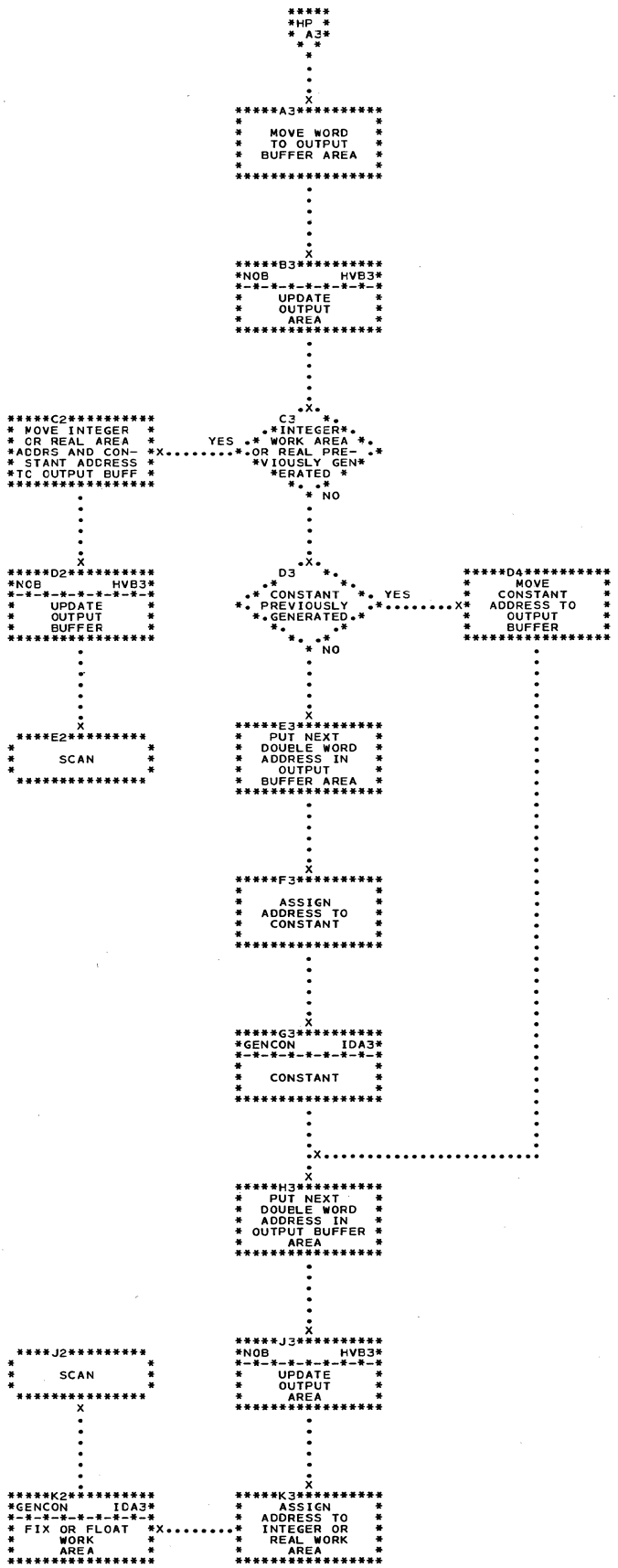


Chart HP. FIXFLO Routine

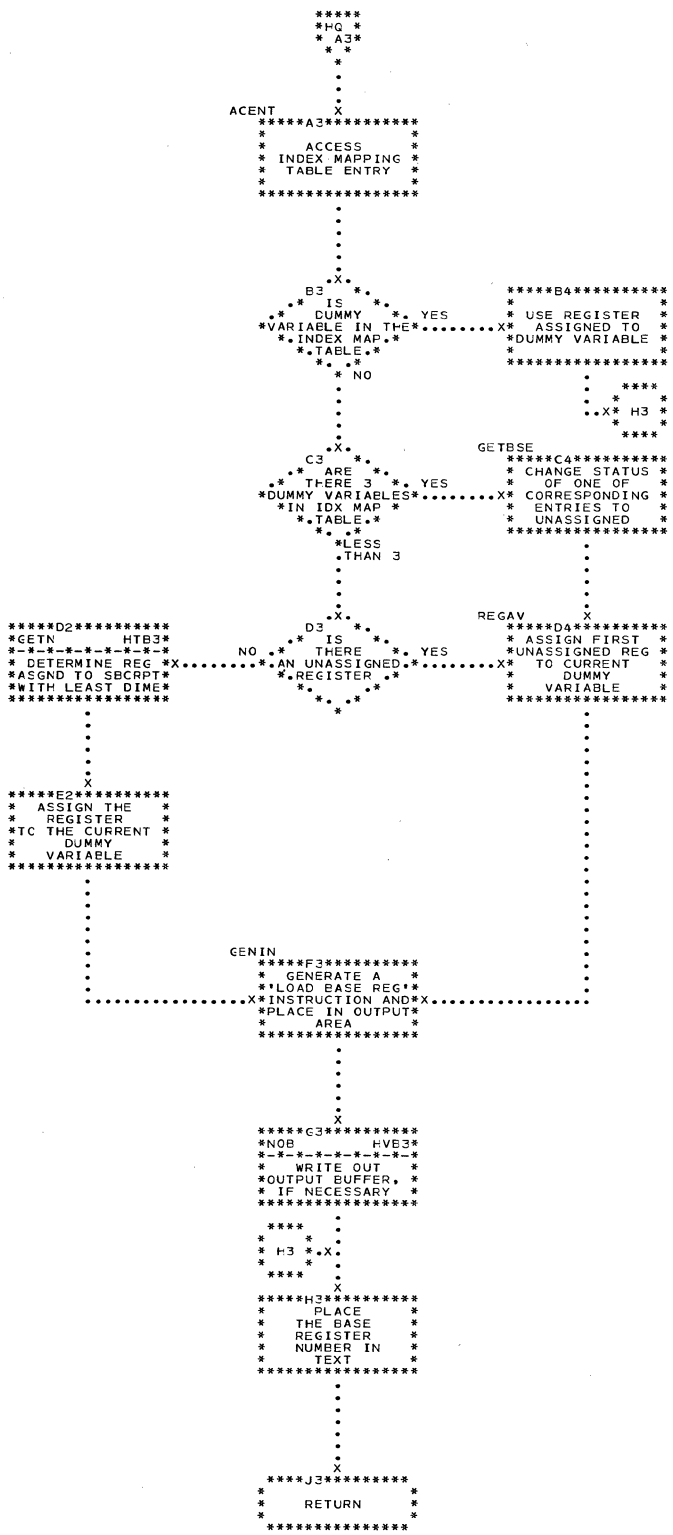


Chart HQ. DUMPR Routine

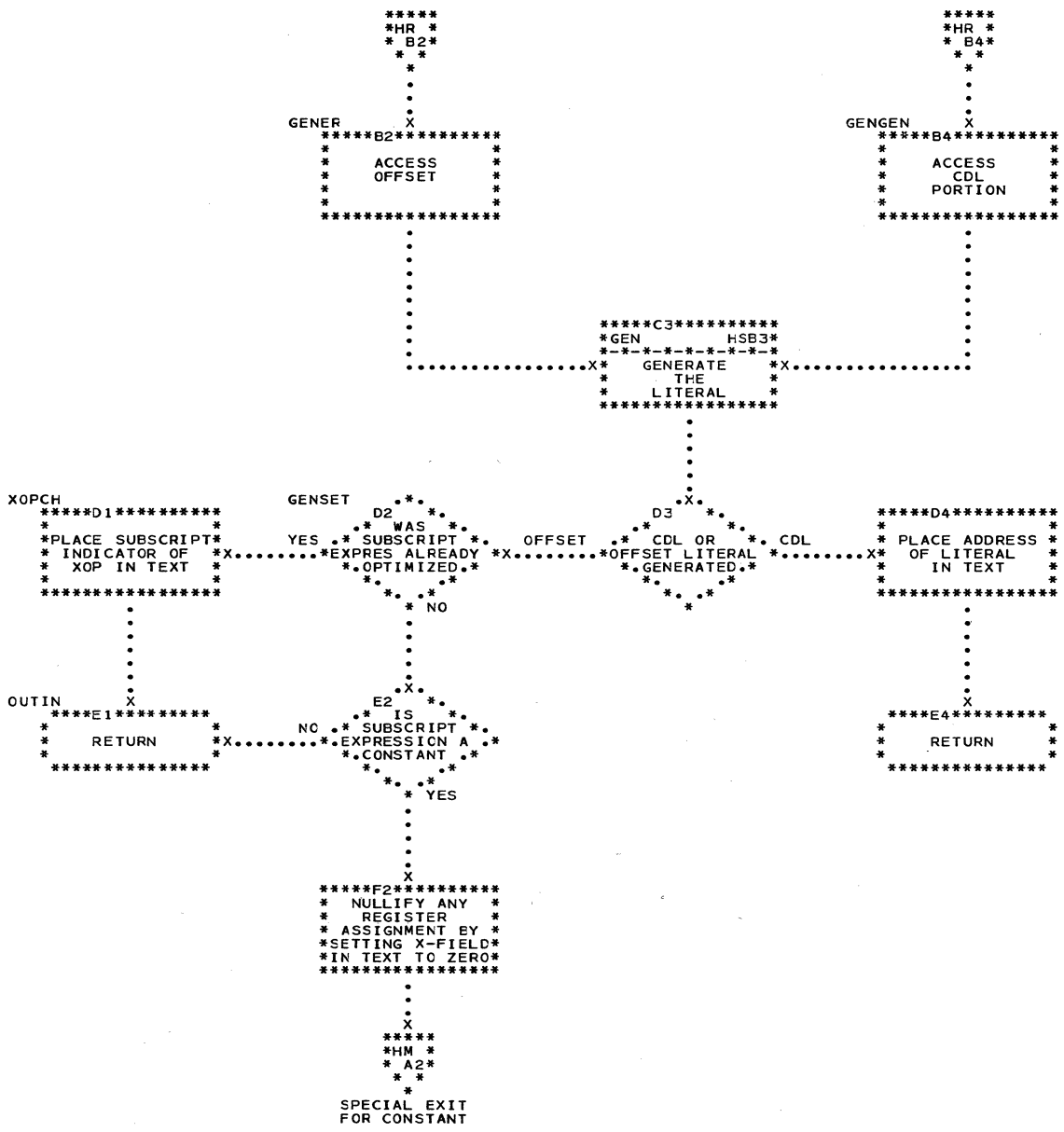


Chart HR. Subroutines GENER, GENGEN

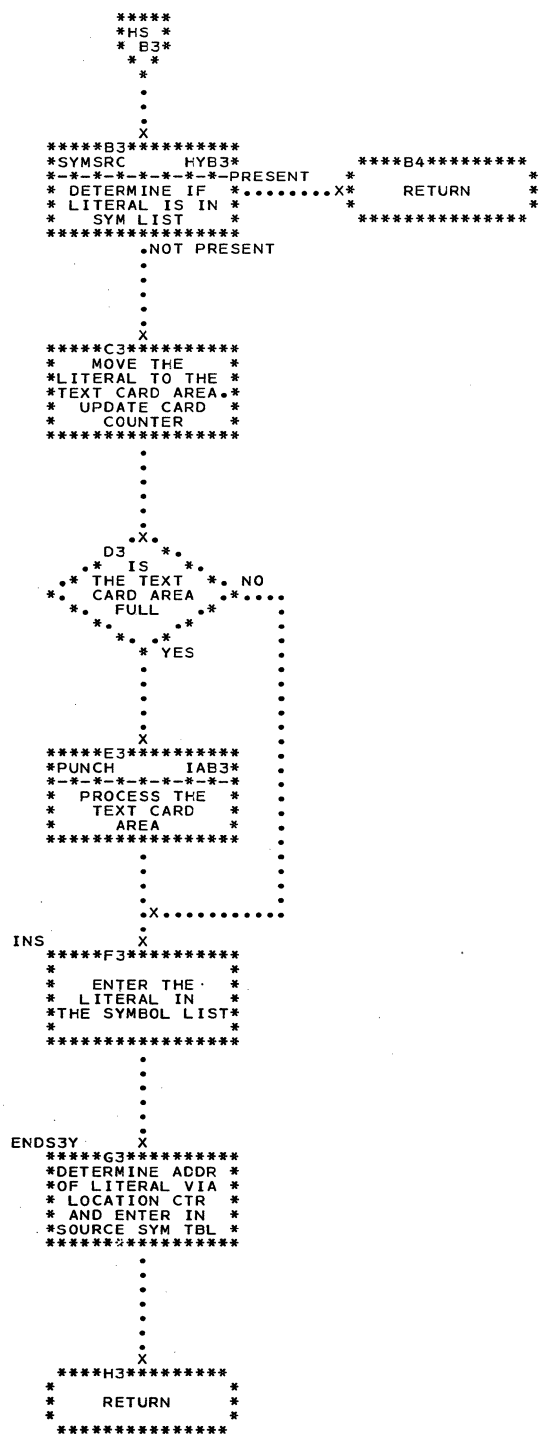


Chart HS. Subroutine GEN

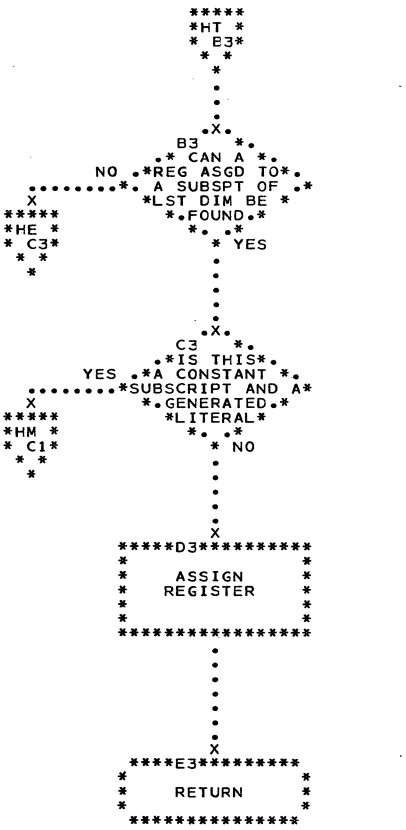


Chart HT. GETN Routine

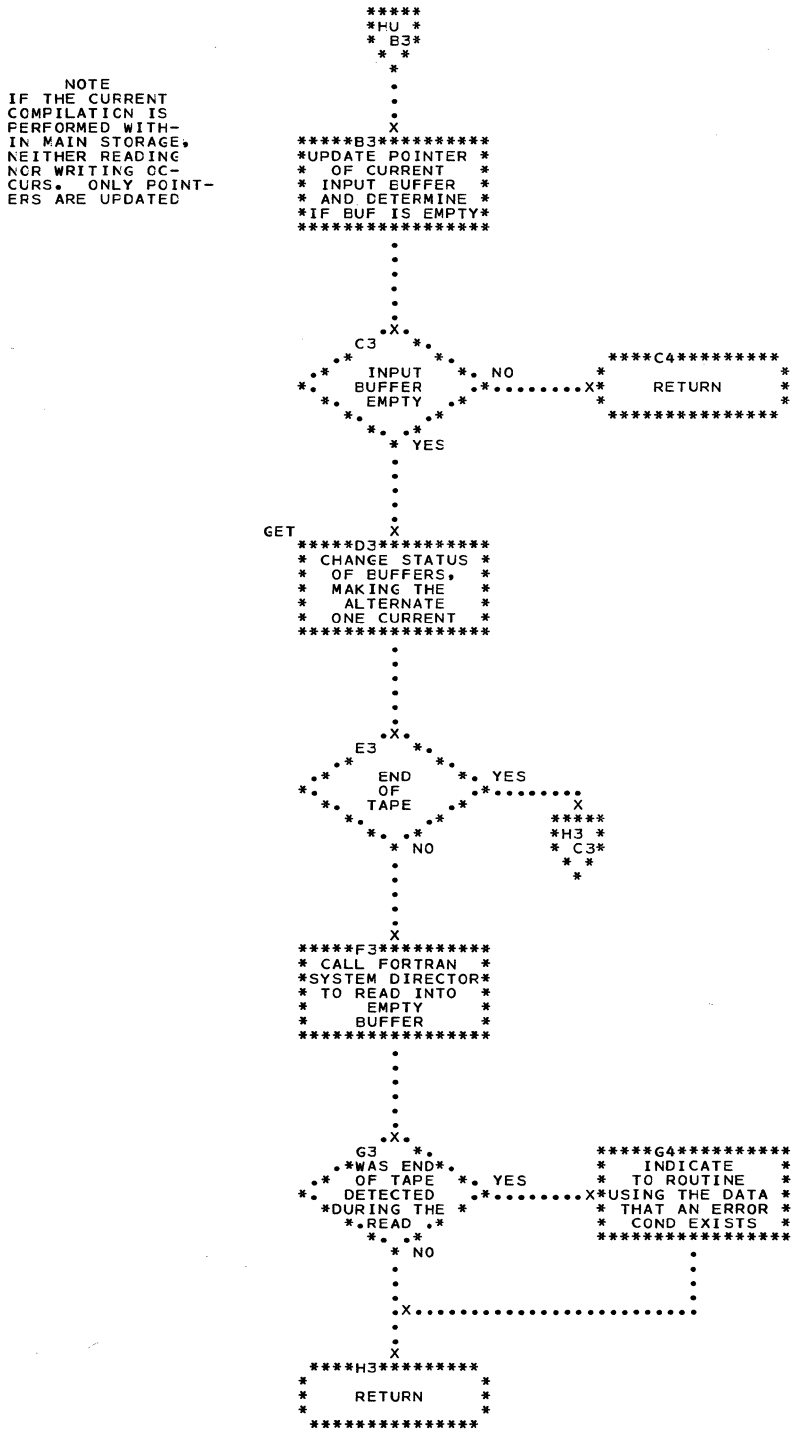


Chart HU. Subroutine NIB

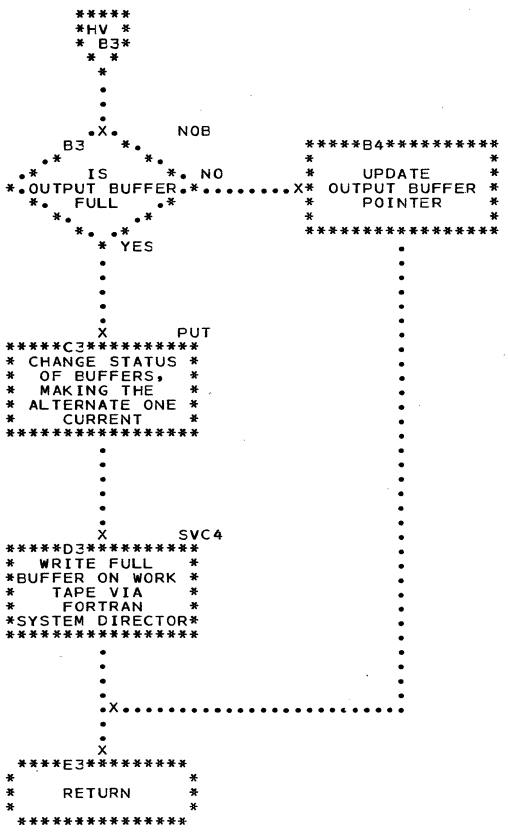


Chart HV. Subroutine NOB

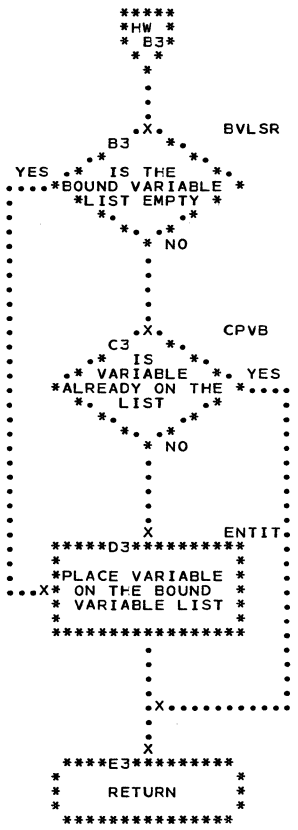


Chart HW. Subroutine BVLSR

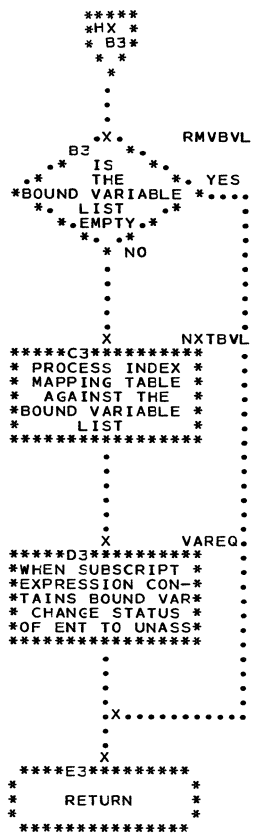


Chart HX. Subroutine RMVBVL

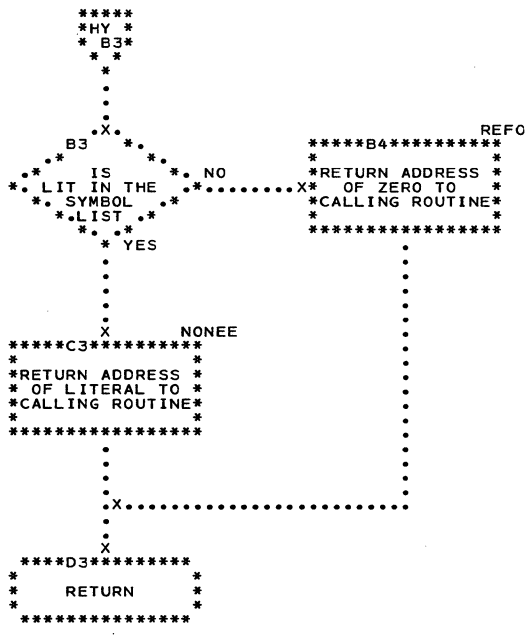


Chart HY. Subroutine SYMSRC

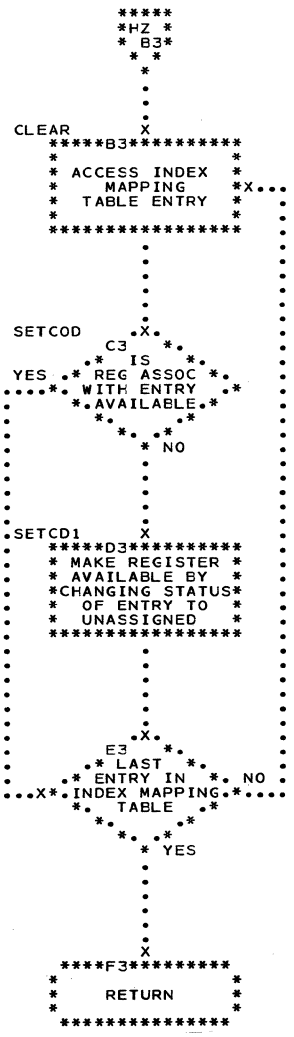


Chart HZ. Subroutine CLEAR

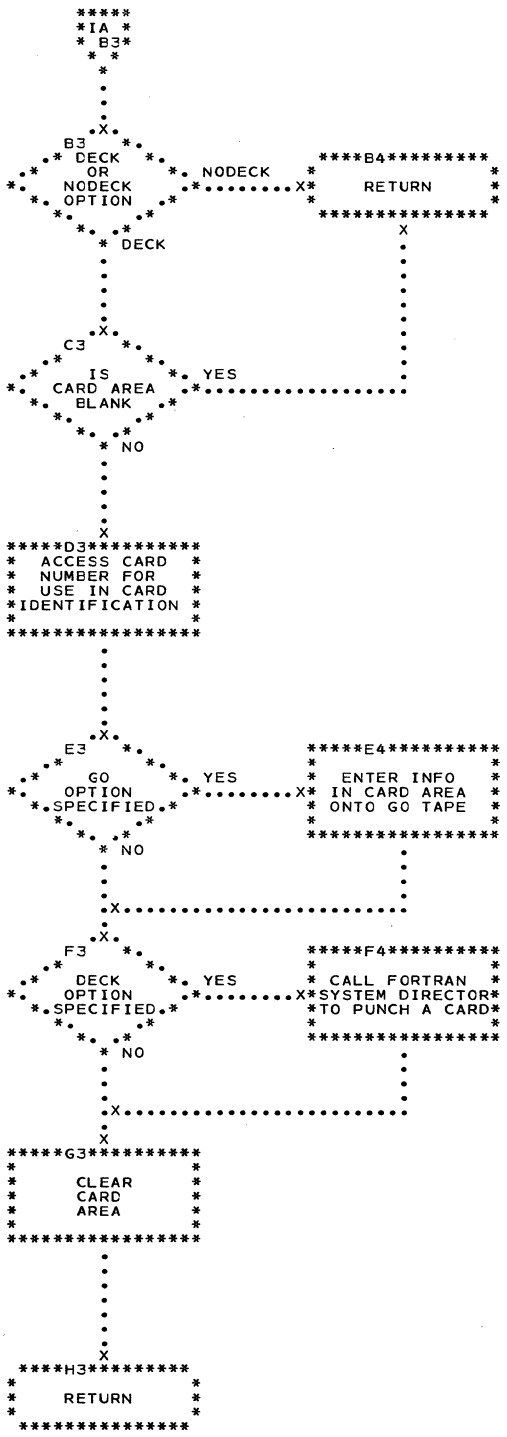


Chart IA. Subroutine PUNCH

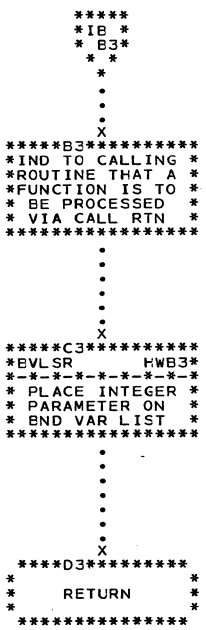


Chart IB. Subroutine HANDLE

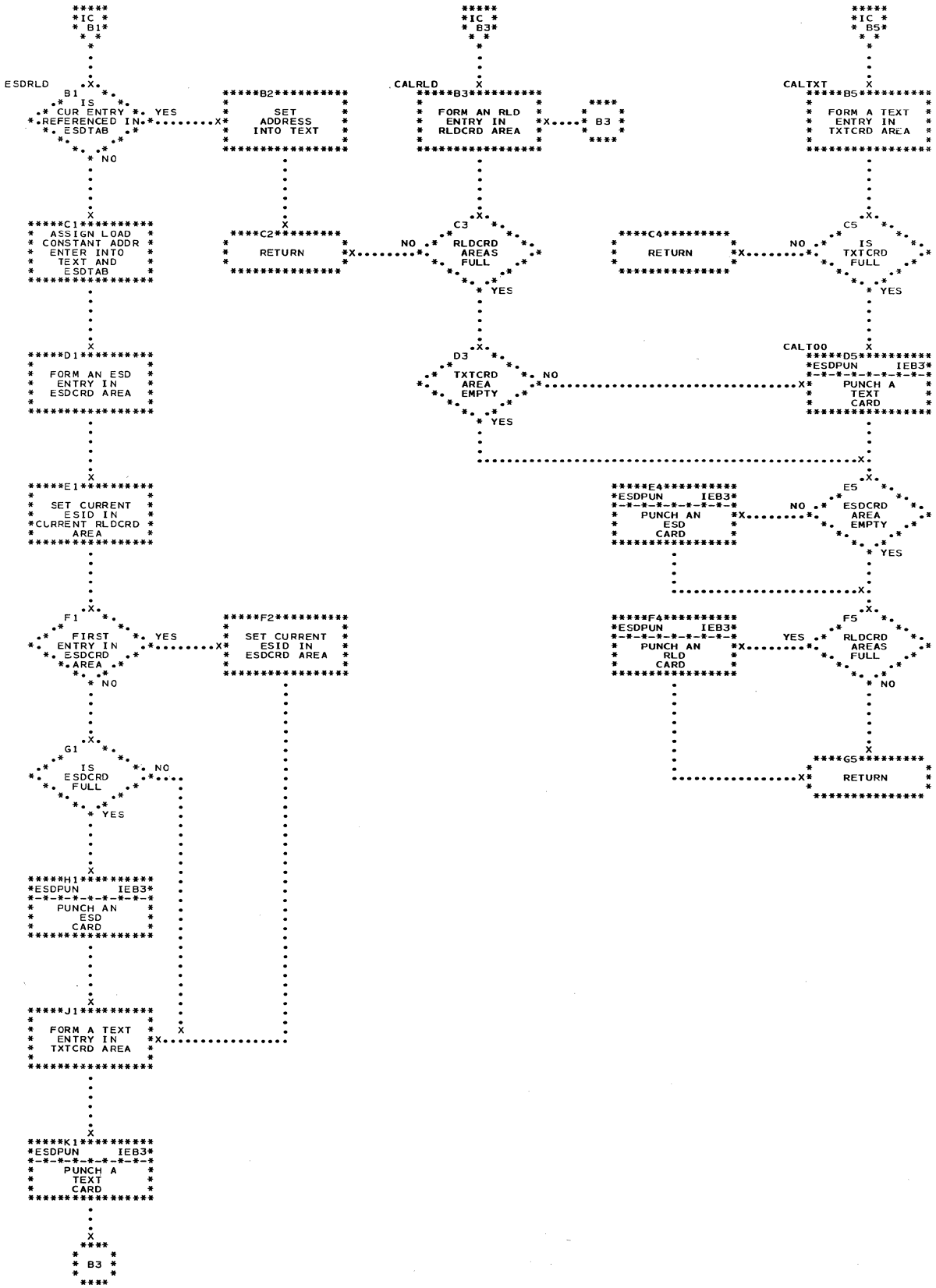


Chart IC. Subroutine ESDRLD/CALRLD/CALTXT

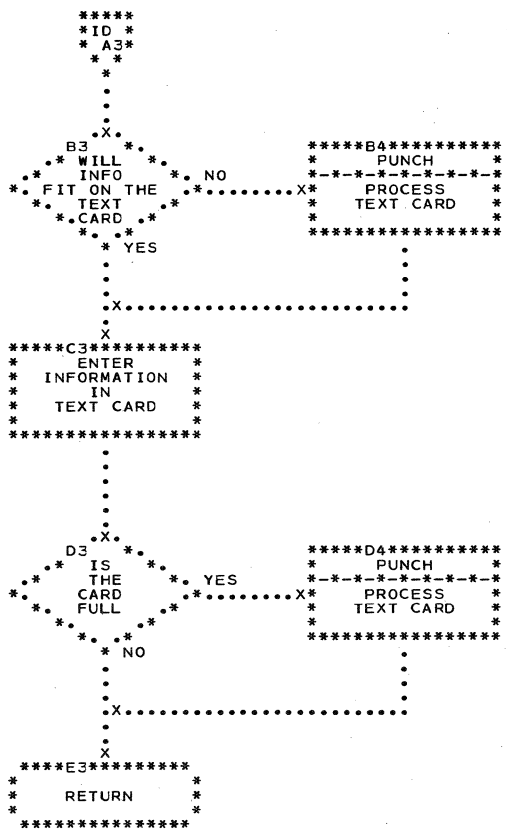


Chart ID. Subroutine GENCON

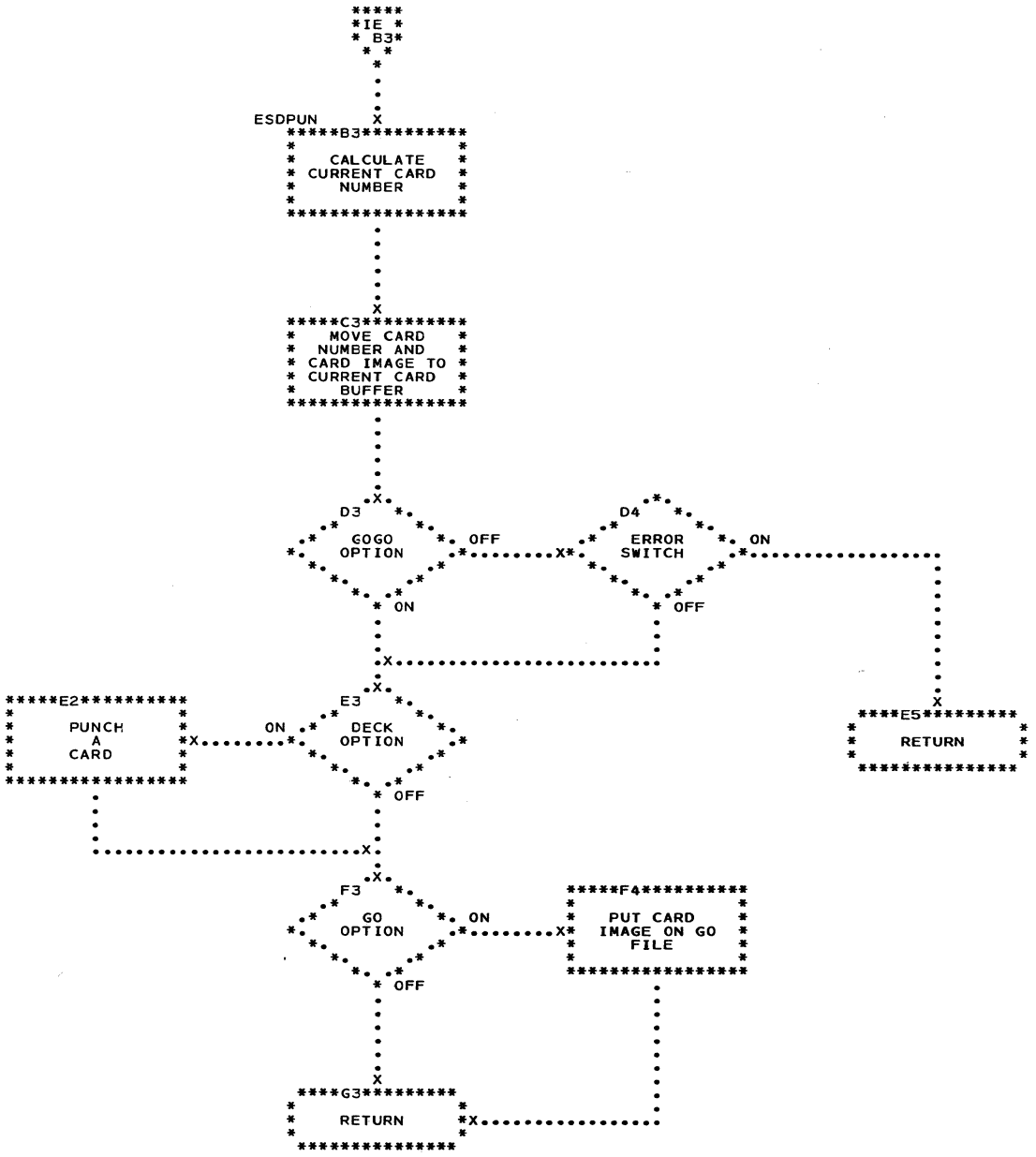


Chart IE. Subroutine ESDPUN

Phase 25 creates the object coding for the FORTRAN source program from the intermediate text entries and the overflow table. Instructions are generated and punched in the object deck if the DECK option is specified and/or written on the GO tape if the GO option is specified. Addresses are assigned to entries in the branch list which is written on the output data sets.

Phase 25 accesses entries in the intermediate text and checks the adjective code to determine the type of the entry. The adjective code determines which Phase 25 subroutine processes the entry. The processing subroutine generates the instructions.

Chart 08, the Phase 25 Overall Logic Diagram, indicates the entrance to and exit from Phase 25, and is a guide to the overall functions of the phase.

If the adjective code for a statement number definition is recognized, the contents of the location counter are inserted in the branch list entry for the statement number. The relative numbers for statement numbers in the branch list were established in Phase 14. Another branch list, created by Phase 25, contains the addresses of the beginning of arithmetic statement function expansions and addresses to control branching with a DO loop.

Phase 25 completes the generation of the base value table. Each address assigned to a base register is placed in this table.

When the intermediate text entry for the END statement is recognized, both branch list tables and the base value table are entered into the output data set. All three tables must be relocatable. All entries in these tables are entered in RLD cards, as well as text cards.

When Phase 25 has completed its execution, a test is made for an error or warning condition within the program. If one exists, Phase 25 calls the FORTRAN System Director to load Phase 30. If an error or warning did not exist in the compilation, the FORTRAN System Director is called to load the Control Card routine.

OBJECT PROGRAM TABLES

Several tables are used by the object program to execute the instructions generated by Phase 25. These tables are assembled in their final form in Phase 25.

Branch List Table for Statement Numbers

Phase 14 allocated storage for a branch list table. Each statement number, not a FORMAT statement number but referenced by a GO TO, Computed GO TO, IF, or DO statement, was assigned a relative number in this branch list table. This relative number was placed in the chain field of the dictionary entry in the overflow table.

When an entry for a statement number definition is recognized by Phase 25, the overflow table entry is accessed, and the relative number is used to assign a position to the statement number in the branch list. The value of the location counter is placed in this position in the branch list table. The next instruction generated by Phase 25 is the first instruction for the referenced statement.

The following instructions are generated for the portion of a FORTRAN statement that references the statement number:

```
L     1,address in the branch list
BCR   15,1
```

The first instruction loads the address of an entry in the branch list into general register 1; the second instruction branches to the address placed in general register 1.

Branch List Table for ASF Definitions and DO Statements

A second branch list table is generated by Phase 25 for arithmetic statement function expansions and DO statements. A number assigned to each arithmetic statement function by Phase 14 is used to assign locations in the second branch list table for each arithmetic statement function expansion. Phase 25 inserts the address of the first instruction in the arithmetic statement function expansion in this loca-

tion in the branch list. Any statement referencing the ASF uses the number of the ASF to find the address of the beginning of the ASF expansion.

Phase 25 also assigns each DO statement a location in the branch list. The address of an instruction near the beginning of the DO loop is entered in that location in the branch list. Object program instructions located at the end of the DO loop access this location in the branch list and branch to the address in the location.

The format for the second branch list is illustrated in Figure 54.

address of ASF expansion 1
address of ASF expansion 2
. . .
address of ASF expansion N
address of instruction in DO loop 1
address of instruction in DO loop 2
. . .
address of instruction in DO loop M

Figure 54. Branch List Table 2

Base Value Table

The base value table (see Figure 55) is generated by other phases of the FORTRAN compiler and Phase 25. An object program can use only general registers 4, 5, 6, and 7 as base registers. When the object program is entered, these registers are initialized with values from the base value table. If a base register other than 4, 5, 6, or 7 is used in an object program, the table is used to take special action. The value for each base register used by the object program is inserted in the base value table. The first entry in the base value table is the value placed in register 4; the second refers to register 5, etc.

value placed in the first base register used to access data in COMMON
. . .
value placed in the last base register used to access data in COMMON
value placed in the first base register used to access data in the object program
. . .
value placed in the last base register used to access data in the object program

Figure 55. Format of the Base Value Table

For a program which uses registers 4 and 5 to access COMMON and registers 6, 7, 8, 9, 10, and 11 to access data and instructions in the object program, the base value table takes the values shown in Figure 56.

The value 20480 should be entered in general register 11. However, register 7 is the last register available for use as a base register. Until Phase 25 is executed, nothing has been done to correct this situation. The spill technique is implemented in the instructions generated by Phase 25. If an intermediate text entry indicates that a base register other than 4, 5, 6, or 7 is used to access data, an instruction is generated to load the value into general register 7, and 7 is used as the base register in the instruction.

Epilog Table

A subprogram may have only variables, arrays, or other subprograms used as parameters. A subprogram accesses a variable by

Register	4	5	6	7	8	9	10	11
Value	0	4096	0	4096	8192	12288	16384	20480

Figure 56. Values in a Base Value Table

moving the value of the variable from the calling program to the subprogram. An array or a subprogram is accessed by moving the address of the array or subprogram used as a parameter from the parameter list in the calling program to the subprogram. The result of the operation performed by the subprogram on array or another subprogram is in the locations allocated to the subprogram or array in the calling program. The result of the operation performed by the subprogram on the variable is in the subprogram itself.

The epilog table (see Figure 57) is generated to return the value of variables used as parameters to the calling program. Phase 25 generates an epilog table when a FUNCTION or a SUBROUTINE adjective code is recognized. The epilog table exists only at compile time.

L_1	S_1	$address_1$
	.	
	.	
	.	
L_n	S_n	$address_n$

Figure 57. Format of the Epilog Table

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

The instructions generated by the RETURN entry in the intermediate text access the epilog table to return the value of variables to the calling program.

INSTRUCTION GENERATION

Phase 25 accesses the intermediate text and generates instructions by analyzing the intermediate text. A FORTRAN object program makes use of all five formats for System/360 instructions - RR, RX, RS, SI, and SS. Intermediate text entries for operations within arithmetic expressions are almost in final form while other text entries must be thoroughly analyzed before instructions can be generated.

Arithmetic Expressions

The text words generated by Phase 15 for arithmetic expressions contain all the elements for the RX format instruction. The op-code, result register, base register, and the displacement have been supplied. If an index register is used, it is in connection with an array, and the operations must be generated by Phase 25 to adjust the index register. These intermediate text entries are denoted by adjective codes 40 through 8F and are processed by the Phase 25 subroutine RXGEN.

Intermediate Text Entries for Other Statements

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

A number of instructions are assembled for Phase 25. These instructions are not used to perform any operation; they are used as constants or literals by Phase 25 to generate instructions. These "skeleton instructions" are always assembled with an op-code. They may have a register and an address.

OUTPUT

Phase 25 inserts the generated instructions, branch lists, and the base value table into text card images. RLD card images are created for all entries in the branch lists and the base value table. The card images are then written on the GO data set, if the Compile and Go or GOGO option is specified, and/or punched into the card deck, if the DECK option is specified.

STORAGE MAP

The storage map for Phase 25 is shown in Figure 58.

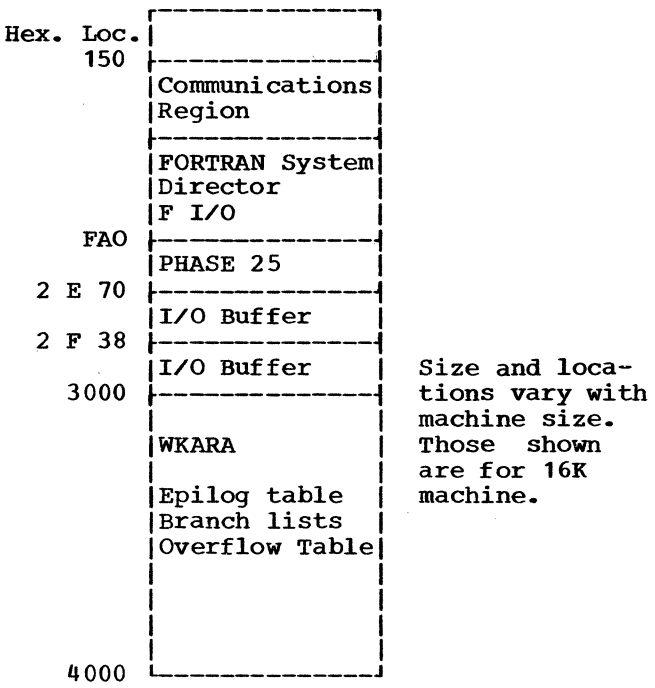


Figure 58. Storage Map for Phase 25

SUBROUTINES

The five types of subroutines used in Phase 25 are the initialization, classification, processing, input/output, and generation subroutines.

Subroutines INITIALIZATION (KA) and ENTRY (KU) initialize Phase 25 and generate object program instructions to initialize the object program.

Subroutine PRESCN (KB) classifies the intermediate text entries by adjective code.

The subroutines described in chart descriptions KC through KT process the intermediate text entries for various adjective codes and begin generating the instructions.

Subroutines BASCHK/RXOUT, RROUT (KX), and GENBC (KV) generate the object program instruction and initialize the input/output subroutines to enter instructions in text cards.

Subroutines GET (KW) and TXTEST, RLDTXT, TXTOUT (KZ) supervise the input/output for Phase 25.

Subroutine INITIALIZATION: Chart KA

Subroutine INITIALIZATION initializes Phase 25. It calculates the size of tables used by Phase 25, initializes the input/output data sets, and sets indicators for spill base registers.

ENTRANCE: Subroutine INITIALIZATION is entered from the FORTRAN System Director (FSD) after the FSD has read Phase 25 into main storage.

OPERATION: Subroutine INITIALIZATION establishes base registers for Phase 25. It then calculates the sizes of the branch table for statement numbers and the branch table for arithmetic statement functions. The buffer pointers are initialized, and the GOGO option is checked.

If the GOGO option is on, the error subroutine for the object program is set up to handle any errors in the object program. If, during object execution the program attempts to execute the coding generated from an erroneous FORTRAN statement, a branch is generated to the object error subroutine.

The input buffers are primed. An address constant is calculated for use in assigning addresses to object program instructions. Because these addresses must be relative to zero, the constant is subtracted from the current value of the location counter each time an address is to be assigned to an instruction.

Subroutine INITIALIZATION then initializes the first text card image. It inserts the address at which the first instruction is to be loaded, and computes constants used in the double-buffering technique.

If a spill base register is necessary, an indicator is set to indicate to the other subroutines in Phase 25 that a spill base register must be used. If COMMON has used all the base registers for a FORTRAN program (registers 4 through 7), an indicator is set to indicate a type 3 program. A type 3 program uses all available base registers (4, 5, 6, and 7) to reference COMMON in the object program. A type 3 program presents the problem of not having a base register to establish addressability for instructions. Subroutine ENTRY resolves this problem. Subroutine ENTRY is then referenced to generate the coding for the object program to initialize itself.

EXIT: Subroutine INITIALIZATION exits to subroutine PRESCN to begin processing the first intermediate text word.

SUBROUTINES CALLED: Subroutine INITIALIZATION references subroutine:

1. ENTRY to assemble coding for an object program to initialize itself.
2. GET (READXT entry) to prime input buffers.
3. HEADNG to print a page heading, if necessary.

Subroutine PRESCN: Chart KB

Subroutine PRESCN passes control to a Phase 25 subroutine to process the intermediate text entries for a statement.

ENTRANCE: Subroutine PRESCN is entered at two points. The first entry point, PRESCN, is used if subroutine PRESCN must access an intermediate text word. The subroutines which enter PRESCN under this condition are: ASFDEF, ASFUSE, ASFEXP, ENDIO, TRGEN, CGOTO, RETURN, LABEL, RXGEN, FUNGEN, IOLIST, FIXFLT, SIGN, DIM, ABS, and STOP/PAUSE.

The second entry point, NOGET, is used if the entering subroutine has accessed an intermediate text word, and subroutine PRESCN must not access another intermediate text word. The subroutines entering PRESCN under this condition are: INITIALIZATION, ARITHI, DO1, ENDDO, SAOP, AOP, SUBRUT, RDWRT, and STOP/PAUSE.

OPERATION: Subroutine PRESCN uses the adjective code for the intermediate text entry to determine which Phase 25 subroutine should process the next series of intermediate text entries. Using shifting operations in the registers, subroutine PRESCN determines how to branch to a subroutine. If the adjective code is less than or equal to 25, (hexadecimal), one branch table is accessed. If the adjective code is between 25 and 8F, control is passed directly to subroutine RXGEN/LM/STM. If the adjective code is greater than or equal to 8F, another branch table is accessed.

EXIT: Subroutine PRESCN exits to the Phase 25 subroutine determined by the adjective code.

SUBROUTINE CALLED: During execution, subroutine PRESCN references subroutine GET to access another intermediate text word.

Subroutine RXGEN/LM/STM: Chart KC

Subroutine RXGEN/LM/STM processes intermediate entries with adjective codes between 25 and 8F (hexadecimal) and entries for store multiple and load multiple instructions in an ASF expansion.

ENTRANCE: Subroutine RXGEN/LM/STM is entered from subroutine PRESCN.

OPERATION: Subroutine RXGEN/LM/STM tests the next entry being processed for an arithmetic operation within an arithmetic statement function. If the entry is for an arithmetic operation, control passes to subroutine ASFEXP to enter the instruction in an arithmetic statement function expansion. If the entry is not an arithmetic operation in an ASF, a test is made to determine if the entry is to be developed into an ASF linkage instruction, which uses only general registers 14 and 15.

If the text entry is neither for arithmetic operation nor a linkage instruction, it is not part of an ASF expansion. A test is made to determine if a register-to-register (RR) or a register-to-storage (RX) instruction is to be generated. For an RR instruction, the second register is inserted in the text word, and subroutine RROUT is called to generate the instruction. If the text entry does not indicate that an RR instruction is to be generated, subroutine BASCHK is called to generate an RX instruction.

An indication of an ASF linkage instruction causes a test to be made to determine if a branch instruction should be generated from the intermediate text word. If so, another test is made to determine if this is the last ASF in the object program. If it is, the address of the first instruction in the program that is neither an ASF nor part of the initialization for the object program is entered in the branch table so the last instruction in the object program initialization may branch around any ASF to the first instruction. Subroutine RXGEN/LM/STM then tests for an RR instruction.

If the instruction is not a branch instruction, it is an instruction referencing a dummy variable. Subroutine RXGEN/LM/STM zeros the bit in the instruction set in Phase 20 to indicate that this intermediate text entry dealt with passing parameters. The subroutine calls subroutine RXOUT to generate the instruction.

When subroutine RXGEN/LM/STM is entered to process the load or store multiple register instructions, a text entry designating multiple register instructions has

been made with the object program address allocated to storing registers. RXGEN/LM/STM inserts the load or store multiple operation code in the text word, inserts registers 2 and 3 in the instruction, and calls subroutine BASCHK to generate the instruction.

EXIT: Subroutine RXGEN/LM/STM exits to subroutines:

1. PRESCN to process another intermediate text entry.
2. ASFEXP to process an intermediate text entry for an arithmetic operation in an ASF.

SUBROUTINES CALLED: RXGEN/LM/STM calls subroutines BASCHK/RROUT and RXOUT to generate object program instructions.

Subroutine LABEL: Chart KD

Subroutine LABEL processes intermediate text entries for statement numbers definitions.

ENTRANCE: Subroutine LABEL is entered from subroutine PRESCN when an intermediate text entry for a statement number is encountered.

OPERATION: Subroutine LABEL inserts the statement number in the print buffer. It then loads the contents of the location counter in a register, subtracts the address constant computed in subroutine INITIALIZATION from the register, and inserts the result in the print buffer as the address of the statement number.

The buffer pointer is updated and checked to determine if the end of the print buffer is reached. If so, the FORTRAN System Director is called to print the contents of the buffer.

If the statement number is referenced by an executable statement, its entry in the overflow table is accessed for the branch list number. This number is used to compute the position for the statement number in the branch list. The address for the statement number location in the object program instructions is inserted in the branch list.

EXIT: Subroutine LABEL exits to subroutine PRESCN to process the next intermediate text word.

SUBROUTINES CALLED: Subroutine LABEL references the FORTRAN System Director to print statement numbers in the storage map.

Subroutines TRGEN, CGOTO: Chart KE

Subroutine TRGEN

Subroutine TRGEN generates branching instructions for unconditional GO TO statements.

ENTRANCE: Subroutine TRGEN is entered from subroutine PRESCN when subroutine PRESCN encounters an unconditional GO TO adjective code, and from subroutine RETURN to generate an unconditional branch.

OPERATION: Subroutine TRGEN accesses the overflow table pointer in the intermediate text word. The relative number in the branch table is in the overflow table statement number entry. Subroutine TRGEN uses the relative number in the branch table to compute the address of the branch table entry for the statement number. In the object program, the address of the object program instruction for the statement that defined the statement number is located at the branch table address computed by subroutine TRGEN.

Subroutine TRGEN generates an instruction to load the address stored in the branch table in a general register, and an instruction to branch to the address contained in the register.

EXIT: Subroutine TRGEN exits to subroutine PRESCN if an unconditional GOTO is generated, or subroutine RETURN, if RETURN instructions are generated.

SUBROUTINES CALLED: During execution subroutine TRGEN references the following subroutines:

1. BASCHK/RXOUT to generate the load instruction.
2. RROUT to generate the branch instruction.

Subroutine CGOTO

Subroutine CGOTO processes intermediate text entries for computed GO TO statements. It generates the calling sequence for the computed GO TO library subroutine.

ENTRANCE: Subroutine CGOTO is entered from subroutine PRESCN when a computed GO TO adjective code is recognized.

OPERATION: Subroutine CGOTO accesses intermediate text entries until it finds the variable for the statement. Instructions are then generated to load the variable into general register 2 and to load the beginning address of the computed GO TO subroutine in a general register. A branch and link instruction is generated to the address contained in the register.

The address of the beginning of the branch list is inserted in the object program immediately following the BALR instruction; then, the numbers of parameters in the call are inserted. Then the relative number for each statement number is inserted in the object program. The computed GO TO library subroutine uses the beginning address of the branch table and the list of relative numbers to branch to the correct instruction in the object program.

EXIT:
Subroutine CGOTO exits to subroutine PRESCN to process the next intermediate text word.

SUBROUTINES CALLED: During execution subroutine CGOTO calls the following subroutines:

1. BASCHK to generate load instructions.
2. RROUT to generate the branch and link instruction and insert the list of relative numbers in the object program.
3. ARGOUT to insert the beginning address of the branch list in the object program.

Subroutines DO1, ENDDO: Chart KF

Subroutine DO1

Subroutine DO1 sets up a DO table for a DO statement and establishes one instruction to initialize a DO loop and another instruction to store the value of the DO variable.

ENTRANCE: Subroutine DO1 is entered from subroutine PRESCN when a DO or implied DO adjective code is recognized.

OPER TION: Subroutine DO1 makes entries in the DO table which consists of 25 8-byte entries. Each entry contains four 2-byte fields and has the following format:

increment or address of variable representing increment	test value or address of variable representing test value	address of DO variable	displacement of branch list entry

Subroutine DO1 enters the increment, test value, and address of the DO variable in the DO table. DO1 also checks the initializing value of the DO loop for an immediate DO parameter. If that value is for an immediate parameter, a load address instruction is generated to load the initial value for the DO loop in a general register. If it is not, a load instruction is generated to load the parameter from its location in main storage into the general register.

An address in the branch list is computed for the DO loop. Each DO is assigned a number, and the branch list location is computed using this number. The contents of the location counter are placed in the branch list location. The location counter contains the address of the next instruction to be generated.

An instruction is then generated to store the register containing the DO variable in the location of the DO variable.

For the statement:

```
DO 12 I = J, 100, 3
```

these instructions are generated by subroutine DO1:

```
L 0, J

ST 0, I
```

The address of the store instruction is entered in the branch list.

EXIT: Subroutine DO1 exits to subroutine PRESCN to process the first intermediate text entry within the DO loop.

SUBROUTINE CALLED: Subroutine DO1 calls subroutine BASCHK/RXOUT to generate instructions.

Subroutine ENDDO

Subroutine ENDDO generates instructions to end a DO loop.

ENTRANCE: Subroutine ENDDO is entered from subroutine PRESCN when an end DO adjective code is recognized.

OPERATION: Subroutine ENDDO initializes the skeleton instruction and then determines if the increment value is an immediate DO parameter or a variable. If it is an immediate DO parameter, a load address instruction is generated to load the parameter in register 2. If it is a variable, a load instruction is generated to load the variable into general register 2.

Subroutine ENDDO then checks the test value of the DO loop for an immediate DO parameter. If it is, a load address instruction is generated to load the test value into general register 3. If it is not, a load instruction is generated to load the test value from its location in main storage into general register 3.

Subroutine ENDDO generates an instruction to load the DO variable into register 0. The location of the branch list entry for the DO loop is computed and inserted into a skeleton instruction. A load instruction is generated to load the address in the branch list entry in register 1.

Subroutine RXOUT is then called to construct a BXLE instruction. This instruction is used to increment the DO variable and test if the DO variable has reached its test value. The instruction generated is BXLE 0,2,0(1). For the statement:

DO 25 I=J,100,3

these instructions are generated by subroutine ENDDO:

```
LA     2,3
LA     3,100
L      0,I
L      1,address in branch list
BXLE   0,2,0(1)
```

EXIT: Subroutine ENDDO exits to subroutine PRESCN.

SUBROUTINE CALLED: Subroutine ENDDO calls subroutine BASCHK/RXOUT to generate instructions.

Subroutine ARITHI: Chart KG

Subroutine ARITHI processes arithmetic IF statements in Phase 25.

ENTRANCE: Subroutine ARITHI is entered from subroutine PRESCN when PRESCN recognizes an IF forcing adjective code.

CONSIDERATION: The intermediate text input for an IF statement in Phase 25 has the following format:

Adjective Code	Mode/Type Code	Address
IF Forcing	mode	R
	statement number	p(1)
	statement number	p(2)
	statement number	p(3)
end mark		internal statement number
adjective code	mode/type	address or p(4)

The symbols p(1), p(2), and p(3) represent overflow table pointers to the first, second and third statement numbers, respectively, used in the IF statement. If an arithmetic expression was used as the argument for the IF statement, Phase 15 has assembled intermediate text entries for the computation of the value of the expression ahead of the entry containing the forcing IF adjective code.

These entries are processed by other subroutines in Phase 25. The entries for the expression were designed so that the result is placed in the register R, indicated in the first text word. The instruction generated from the last text word before the IF adjective code insures that the result of the operation is placed in register R.

OPERATION: Subroutine ARITHI accesses the text words depicted in the diagram and places them in a work area. Subroutine ARITHI tests an indicator set in the low order portion of the mode/type field in the IF forcing entry. This indicator indicates if the instruction generated from the text entry immediately preceding the IF forcing entry set the condition code. If the instruction did not set the condition code, a load and test instruction is generated to set the code to test for a negative, zero, or positive expression value.

The remainder of subroutine ARITHI optimizes and generates the branching instructions for the IF statement (a load instruction to load register 1 from the branch list and a branch on condition code to the address placed in the register). The mask for the branch instruction is set by subroutine ARITHI according to the con-

ditions represented by the statement numbers in the IF statement.

The optimization is concerned with the number of branch instructions generated. It is affected by the following conditions:

1. Two of the statement numbers are equal.
2. The entry following the end mark entry for an IF statement is the statement number definition for one of the statement numbers referenced by the IF statement.

One branch instruction is generated for two equal statement numbers if the entry following the end mark entry is the definition entry for one of the statement numbers referenced by the IF statement.

For example, the FORTRAN statements

```
   IF (I) 1,1,2
1  X = A + 1.0
     .
     .
     .
2  X = A + 1.0
```

generate the following instructions for the IF statement:

```
L     0,I
LTR   0,0
L     1,STNO2
BCR   2,1
```

Two branch instructions are generated for two equal statement numbers referenced by the IF statement if the entry following the end mark entry is not the definition entry for any of the statement numbers in the IF statement.

For example, the FORTRAN statements

```
    IF    (A) 1,1,2
69 LOAD = 2.0 + 2.0
         .
         .
         .
1   X   = A+1,0
```

generate the following instructions for the IF statement:

```
LE    0,A
LTER  0.0
L     1,STNO1
BCR   13,1
L     1,STNO2
BCR   2, 1
```

Two branch instructions are also generated for unequal statement numbers if the entry following the end mark entry for an IF statement is the definition entry for one of the statement numbers referenced by the IF statement.

For example, the FORTRAN statements

```
   IF  (A) 1,2,3
1  X = A+1.0
     .
     .
     .
3  X = A-1.0
     .
     .
     .
2  X = A
```

generate the following instructions:

```
LE    0,A
LTER  0.0
L     1,STNO2
BCR   8,1
L     1,STNO3
BCR   2,1
```

Three branch instructions are generated for unequal statement numbers if the entry following the end mark entry is not the definition entry for any of the statement numbers referenced by the IF statement.

EXIT: Subroutine ARITHI exits to subroutine PRESCN to process the next entry in the intermediate text.

SUBROUTINES CALLED: Subroutine ARITHI calls subroutines; GENBC to generate the branch instructions and RXOUT to generate a load and test instruction.

Subroutine RDWRT: Chart KH

Subroutine RDWRT processes the entries in the text for the READ/WRITE BACKSPACE, REWIND, and END FILE adjective codes, the FORMAT statement number, and the reference to the data set reference number.

ENTRANCE: Subroutine RDWRT is entered by subroutine PRESCN when PRESCN recognizes the READ/WRITE, BACKSPACE, REWIND, or END FILE adjective codes.

OPERATION: Subroutine RDWRT determines if this is a READ/WRITE statement using a FORMAT statement. If it is, the FORMAT indicator is set on. If no FORMAT statement is associated with this input/output statement, two text words are accessed. If a FORMAT statement is associated with the statement, three text words are accessed.

All input/output operations are processed by IBCOM. Subroutine RDWRT generates

a calling sequence to enter and pass parameters to IBCOM. Subroutine RDWRT generates an instruction to load the address of IBCOM in a general register. The adjective code for the statement is used to generate a displacement for a branch instruction. The displacement is the distance in bytes between the I/O subroutine represented by the adjective code and the beginning of IBCOM. A branch and link instruction is generated to branch to the particular subroutine in IBCOM.

The symbol referencing the data set reference number is checked for a data set reference number or an integer variable. A word is placed in the object program following the branch and link instruction indicating the type of symbol referencing the data set reference number, and containing the address of the integer variable or the data set reference number, itself.

If the I/O statement requires a FORMAT, the address of the FORMAT information is accessed from the overflow table and inserted in the object program as a parameter passed to IBCOM.

EXIT: Subroutine RDWRT exits to subroutine PRESCN to process the next intermediate text word.

SUBROUTINE CALLED: Subroutine RDWRT calls the following subroutines:

1. GET to access intermediate text words.
2. BASCHK to generate the load instruction.
3. RXOUT to generate the branch and link instruction and to generate the parameter containing the data set reference number.
4. ARGOUT to insert addresses in the object program.

Subroutine IOLIST: Chart KI

Subroutine IOLIST processes each member in the I/O list.

ENTRANCE: Subroutine IOLIST is entered from subroutine PRESCN when it detects a variable in an I/O list designated by a left parenthesis, right parenthesis, or a comma.

CONSIDERATION: An instruction was generated in subroutine RDWRT to load a register with the starting address of IBCOM. The address in this register is not altered by any of the I/O processing. Instructions in subroutine IOLIST are generated assuming this condition.

OPERATION: Subroutine IOLIST determines if the symbol entered in the I/O list is an array. If it is not an array, the indicator set in subroutine RDWRT is tested to determine if the symbol in the list requires a FORMAT. If it does not require a FORMAT, a displacement is computed for subroutine FIOLN in IBCOM, and a branch and link instruction is generated to call that IBCOM subroutine. If the symbol does require a FORMAT, the displacement is computed for subroutine FIOLF in IBCOM and a branch and link instruction is generated to call that IBCOM subroutine.

To complete the calling sequence for either FIOLN or FIOLF, a parameter word is generated. This word contains the length of the variable (4 for real or integer, 8 for double-precision), an indexing register if the variable is subscripted, and the base-displacement address of the variable in the I/O list.

When this subroutine was entered, a test was made to determine if the item in the I/O list is an array. If the item is an array, the next text word is accessed, and the indicator set in subroutine RDWRT is tested to determine if the array requires a FORMAT. If the array does require a FORMAT, the displacement is calculated for subroutine FIOAF in IBCOM, and a branch and link instruction is generated to subroutine FIOAF. If the array does not require a FORMAT, a displacement is computed for subroutine FIOAN in IBCOM, and a branch and link instruction is generated to FIOAN.

If the item in the I/O list is an array, two parameter words are generated. The first word contains the beginning address of the array; the second contains the length and number of the elements in the array.

EXIT: Subroutine IOLIST exits to subroutine PRESCN after the intermediate text entry is processed.

SUBROUTINES CALLED: During execution, subroutine IOLIST calls the following subroutines:

1. RXOUT to generate branch and link instructions and to insert the length and number of elements in a parameter word for an array.
2. ARGOUT to insert the address of the beginning of an array in the object program.
3. BASCHK to generate the parameter word for variables in an I/O list.

Subroutine ENDIO: Chart KJ

Subroutine ENDIO processes the end I/O entry in the intermediate text.

ENTRANCE: Subroutine ENDIO is entered from subroutine PRESCN when an end I/O list adjective code is recognized.

OPERATION: Subroutine ENDIO tests the indicator set by subroutine RDWRT to determine if the I/O list requires a FORMAT. If it does, the displacement for subroutine FENDF in IBCOM is calculated, and a branch and link instruction is generated to call subroutine FENDF. If the I/O list does not require a FORMAT, the displacement for subroutine FENDN in IBCOM is calculated, and a branch and link instruction to call subroutine FENDN is generated. The FORMAT indicator is set off.

EXIT: Subroutine ENDIO exits to subroutine PRESCN to process the next intermediate text entry.

SUBROUTINE CALLED: Subroutine ENDIO calls subroutine RXOUT to generate the branch and link instructions.

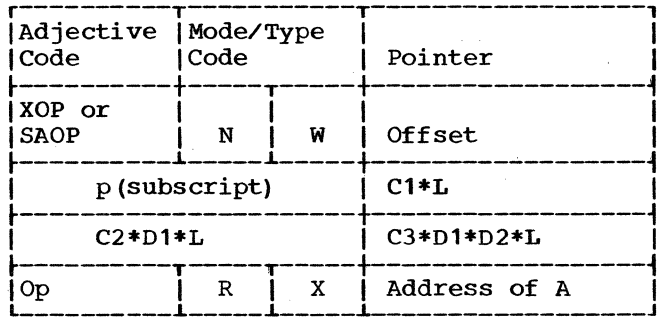
Subroutines SAOP, AOP: Chart KL

Subroutine SAOP

Subroutine SAOP processes intermediate text entries for subscript calculation if the entire subscript indexing factor must be calculated.

ENTRANCE: Subroutine SAOP is entered from subroutine PRESCN when an SAOP or XOP adjective code is recognized.

CONSIDERATION: The intermediate text entries for subscripted variables are in the format as described in Phase 20.



where:

N = number of dimensions
W = work register
R = register containing the result of the operation
X = index register

OPERATION: Subroutine SAOP gets the next three intermediate text words, stores them in a work area, and begins generating instructions for the first subscript parameter.

If the first parameter is a constant, no instructions are generated for it; the first parameter was included in the offset. If it is not a constant, a test is made to determine if a literal was generated for C1*L. If a literal was generated, an instruction is generated to load X with C1*L. Then, an instruction is generated to multiply the contents of X by V1. If a literal was not generated, the product C1*L was found to be a number of the form 2 and the number p was entered in the pointer field of the intermediate text by Phase 20. V1 is loaded into X. An instruction is generated to multiply the contents of X by shifting left p bits.

A test is made to determine if there is more than one subscript parameter in the subscript expression. If there is another parameter, a test is made to determine if the second parameter is a constant. If it is, no code is generated for this parameter. If the second parameter is not a constant, an instruction is generated to load C2*D1*L into the work register. Another instruction is generated to multiply the contents of W by the value V2. The next instruction generated adds the contents of W to the contents of X. The dimension count is decreased by 1 and a test is made for a third dimension.

The third dimension instructions are similar to those generated for the second dimension. C3*D1*D2*L is loaded into W; W is multiplied by V3, and the contents of W are added to X.

When all the instructions have been generated for the variables in the subscript expression, a test is made to determine if the instruction is part of an I/O list. If it is, an indicator is set for the I/O subroutines to handle this condition.

A test is then made to determine if the offset is a literal. If it is, an instruction is generated to add the literal to the contents of X. If the offset is not a literal, the offset is added to the displacement portion of the instruction to be executed for the subscripted variable.

Condition	Instruction	Alternate Instructions
First variable	L X=(C1*L) MH X,V1+2	L X,V1 SLA X,p
Second variable	L W,=(C2*D1*L) MH W,V2+2 AR X,W	
Third variable	L W,=(C3*D1*D2*L) MH W,V3+2 AR X,W	
Offset	A X,=(offset)	no instruction
Operation	op R,A(X)	op R,A+offset(X)

For the general form illustrated above, these instructions are generated:

EXIT: Subroutine SAOP exits to subroutine PRESCN to assemble the instruction for the subscripted variable.

SUBROUTINES CALLED: During execution subroutine SAOP calls subroutines BASCHK/RXOUT and RROUT to generate instructions.

Subroutine AOP

Subroutine AOP processes entries in the intermediate text for subscript calculations that do not use variables.

ENTRANCE: Subroutine AOP is entered from subroutine PRESCN when an AOP adjective code is recognized.

CONSIDERATION: The intermediate text entries for a subscripted variable with an AOP adjective code are in this format as described in Phase 20:

Adjective Code	Mode/Type Code	Pointer
AOP	00	offset
OP	R X	address of A

The offset may be in the form of an actual constant or a literal with the address of the literal in the intermediate text entry. The AOP entry is made for variables with a constant subscript, or for a subscript calculation that has been eliminated by Phase 20.

OPERATION: The text word containing the operation performed on the subscripted variable is accessed and a test is made to determine if the offset is a literal. If it is not, the offset is added to the displacement portion of the address in the instruction portion of the literal.

If the offset was a literal, a test is made to determine if the offset is for a constant subscript or was generated by Phase 20 for use with a previously calculated subscript expression. If it is for a constant subscript, an instruction is generated to load the offset into X. If it is for use with a previously calculated expression, the offset is really an adjusting factor for X, and the offset is added to the contents of X.

A test is then made to determine if the entry is in an I/O list. If it is, the I/O list indicator is set on for reference by the I/O subroutines.

In summary, no subscript calculations are generated for an offset that is not a literal. The instruction for the operation takes the form:

```
op    R,A+offset
```

For a literal, one instruction is generated for subscript calculation. If the offset is for a constant subscript, the instruction:

```
L     X,=(offset)
```

is generated. For an offset used as an adjustment factor on a previously calculated expression, the instruction:

```
A     X,=(offset)
```

is generated. The instruction containing the operation takes the form:

```
op    R,A(X)
```

EXIT: Subroutine AOP exits to subroutine PRESCN to generate the instruction containing the operation on the subscripted variable.

SUBROUTINES CALLED: During execution subroutine AOP calls subroutine BASCHK to generate the instruction for adjusting X with the offset.

Subroutines ASFDEF, ASFEXP, ASFUSE: Chart KM

Subroutine ASFDEF

Subroutine ASFDEF processes the first word for an arithmetic statement function definition.

ENTRANCE: Subroutine ASFDEF is entered from subroutine PRESCN when the ASF definition adjective code is recognized.

OPERATION: Subroutine ASFDEF accesses the ASF number in the pointer field of the intermediate text entry and calculates the address of this number in the second branch table. The operations defining the arithmetic statement function follow this text word. The contents of the location counter are inserted at the address in the ASF table just calculated.

EXIT: Subroutine ASFDEF exits to subroutine PRESCN to get the first word of the arithmetic statement function expansion.

Subroutine ASFEXP

Subroutine ASFEXP processes intermediate text entries for arithmetic operations in an arithmetic statement function expansion.

ENTRANCE: Subroutine ASFEXP is entered from subroutine RXGEN/LM/STM when the intermediate text entry for an arithmetic operation in an ASF expansion is recognized.

OPERATION: An instruction is generated by subroutine RXGEN immediately before calling the instructions for an ASF. This places the address of the parameter list for an ASF reference in a general register. The parameter list contains the addresses of the parameters used in the ASF reference.

Subroutine ASFEXP accesses the pointer field in the intermediate text word to find which parameter is used in this text word. It generates an instruction to load the address for the parameter in general register 15.

Subroutine ASFEXP accesses the operation field and the work register assigned to the operation in the intermediate text word. It then generates an instruction to perform the operation, placing the result in the work register. The displacement field of this instruction contains a zero. The base register is general register 15, because the first generated instruction placed the address of the parameter in general register 15.

For example, if the text word,

Operation Code	Mode/Type Code	Pointer
AE	60	1008

is generated by Phases 15 and 20, the following instructions are generated by Phase 25:

```
L     15, 8(9)
AE    6, 0(15)
```

The displacement 8 in the first instruction is accessed from the pointer field in the text word. The work register 6 is accessed from the mode/type field, and the operation AE is accessed from the adjective code field in the text word. In this example, register 9 is used to pass the address of the parameter list.

EXIT: Subroutine ASFEXP exits to subroutine PRESCN to process the next text word.

SUBROUTINE CALLED: During execution subroutine ASFEXP calls subroutine RXOUT to generate object program instructions.

Subroutine ASFUSE

Subroutine ASFUSE generates the instructions to call an arithmetic statement function.

ENTRANCE: Subroutine ASFUSE is entered from subroutine PRESCN when the adjective code for an ASF usage is recognized.

OPERATION: Subroutine ASFUSE accesses the pointer field of the text word and gets the ASF number assigned to the called function. The number is used to compute the address of the entry in the ASF table. This address is entered in a load instruction, which is generated to load the address from

the ASF table of the first instruction in the ASF expansion in general register 15.

Subroutine ASFUSE then generates an instruction BALR 14, 15 to branch to the first instruction in the ASF expansion.

EXIT: Subroutine ASFUSE exits to subroutine PRESCN to process the next intermediate text word.

SUBROUTINES CALLED: During execution subroutine ASFUSE calls subroutines RXOUT and RROUT to generate the load instruction and the branch instruction, respectively.

Subroutine SUBRUT: Chart KN

Subroutine SUBRUT processes FUNCTION and SUBROUTINE header cards. It builds the epilog table for the subprogram and generates instructions to pass the parameters in the calling program to the subprogram.

ENTRANCE: Subroutine SUBRUT is entered from subroutine PRESCN when the adjective code for a FUNCTION or SUBROUTINE header card is recognized.

OPERATION: Subroutine SUBRUT computes the address of the epilog table, in order that the epilog table can be constructed.

Subroutine SUBRUT gets the next text word and tests it for an end mark entry. If it is an end mark, the end of the intermediate text entries for the subprogram has been reached. Subroutine SUBRUT then generates a branch instruction to bypass instructions that will be generated for any ASF in the subprogram.

If the next text word is not an end mark, a test is made to determine if the text word is a dummy array. If it is, a move instruction is generated to move the address of the dummy from the parameter list in the calling program to the field that has been allocated to the dummy symbol in the subprogram. During object execution the address of the parameter list is passed to the subprogram in a general register. In Phase 25 the displacement of the parameter from the beginning of the parameter list is obtained by maintaining a count of the number of dummy names accessed by subroutine SUBRUT.

If the dummy name is a variable, its value, not its address, is passed to the subprogram. If the dummy name is a function, the address of an address constant is passed to the subprogram. A load instruction and a move instruction are generated. The mode of the variable is tested, and the number 4 for real or integer, or 8 for double-precision variables is saved. The displacement of the parameter from the beginning of the parameter list is inserted in the load skeleton instruction. A load instruction is then generated to access the address of the variable in the main program from the parameter list and put it in a general register. For example, subroutine SUBRUT may generate the instruction:

```
L       2,DISP2(1)
```

where 1 is the general register in which the address of the parameter list was placed by the calling program, and DISP2 is the displacement of the second variable from the beginning of the parameter list. The result of this instruction is the placement of the address of the second parameter in the calling program in general register 2.

A move instruction is generated to move the variable used as a parameter in the calling program to its location as a dummy variable in the subprogram. The address assigned to the dummy variable is placed in the move skeleton instruction and the move instruction is generated. For example, subroutine SUBRUT generates the following instruction after the load instruction:

```
MVC     DUMMY2(4) , 0(2)
```

where DUMMY is the location of the dummy variable in the subprogram and, if it is double-precision, its field length is 8.

The information for a dummy variable is placed in the epilog table, which is used by instructions generated by subroutine RETURN to return the value of dummy variables to the main program. The length of the variable, parameter list displacement, and the address of the variable in the main program are placed in the epilog table. The displacement for the parameter list is updated, and subroutine SUBRUT gets the next text entry to determine if it is an end mark.

EXIT: Subroutine SUBRUT exits to subroutine PRESCN when an end mark indicating the end of the parameter list is recognized.

SUBROUTINES CALLED: During execution, subroutine SUBRUT calls the following subroutines:

1. GENBR to generate a branch around the ASF expansions.
2. RXOUT or RROUT to generate move instructions.
3. RXOUT to generate load instructions.
4. GET to access intermediate text words from the input buffer.

Subroutine RETURN: Chart KO

Subroutine RETURN processes any intermediate text entries for a RETURN statement.

ENTRANCE: Subroutine RETURN is entered from subroutine PRESCN when a RETURN adjective code is recognized.

OPERATION: If any previous RETURNS have been processed for the subprogram, subroutine RETURN generates instructions for the following RETURNS to branch to the coding generated for the first RETURN.

If no previous RETURNS have been generated for the subprogram, the coding for the RETURN must be generated. The address of the RETURN coding is placed in the branch list, so that following RETURNS generated in the subprogram may branch to the same RETURN coding. Instructions are then generated to load the pointer to the parameter list in a general register and to return the value of the parameters to the caller.

If the subprogram is a FUNCTION subprogram, an instruction is generated to return the value of a function in a general register.

When the subprogram was entered, instructions were generated to save the registers of the calling program in main storage. The pointer to this area is placed in a general register. If another subprogram is called within a subprogram, this pointer must be saved in main storage. When the subprogram is returning to the calling program, this pointer must be reloaded into a register so that the registers for the calling program can be restored.

Instructions are then generated to restore the registers to the same values they contained when the subprogram was called.

The exit branch is then generated to return control to the calling program. The address to which the subprogram should return was left in register 14 by the branch and link instruction of the calling program.

EXITS: Subroutine RETURN exits to subroutine PRESCN after the instructions for a RETURN have been generated.

SUBROUTINES CALLED: During execution subroutine RETURN calls the following subroutines:

1. RROUT, RXOUT, and BASCHK to generate instructions for the first RETURN in a subprogram.
2. TRGEN if a RETURN has already been generated to generate an unconditional branch to the coding for the first RETURN in the subprogram.

Subroutine FUNGEN/EREXIT: Chart KP

Subroutine FUNGEN/EREXIT processes all in-line and library function calls. If the GOGO option is on, subroutine FUNGEN/EREXIT is used to generate the coding where source program errors occurred.

ENTRANCE: Subroutine FUNGEN/EREXIT is entered from subroutine PRESCN when:

1. An in-line function is called.
2. A library function is called.
3. An exponentiation operation is to be performed.
4. An intermediate text entry for an error is encountered.

OPERATION: Subroutine FUNGEN/EREXIT is entered at two points. The first entry point, FUNGEN, tests for an in-line function.

If the function is in-line, subroutine FUNGEN/EREXIT uses the function identification number to access a branch table and find the address of the subroutine used to generate the coding.

The second entry point, EREXIT, causes subroutine FUNGEN/EREXIT to access the address of the library subroutine IBERR from the communications area.

If the function is not in-line, subroutine FUNGEN/EREXIT generates instructions to load the address of the library function from its address constant in a general register, and an instruction to branch to the address loaded in the register.

EXIT: Subroutine FUNGEN/EREXIT exits to subroutine PRESCN after generating the linkage instructions for a library subroutine, or the proper subroutine for generating a specific in-line function.

SUBROUTINES CALLED: During execution subroutine FUNGEN/EREXIT calls subroutines BASCHK and RROUT to generate the instructions for linkage to a library subroutine.

Subroutines FIXFLT, GNBC6: Chart KQ

Subroutine FIXFLT

Subroutine FIXFLT generates instructions for the in-line functions IFIX, FLOAT, and DFLOAT.

ENTRANCE: Subroutine FIXFLT is entered from subroutine FUNGEN/EREXIT when the function number in the intermediate text word indicates one of the in-line functions IFIX, FLOAT, or DFLOAT.

CONSIDERATION: Phase 15 enters intermediate text words for in-line functions. One word forces the Phase 25 subroutine RXGEN/LM/STM to generate an instruction to insert the argument in a floating-point register. The second word contains the adjective code for an in-line function, the number of a floating point register (R1) and a fixed point register (R2), and the number of the in-line function in the pointer field. Phase 15 generates text words to insure that RXGEN generates instructions to initialize R1 and R2, and possibly a third fixed point register whose number is R2-1.

OPERATION: Subroutine FIXFLT determines if the in-line function is IFIX. If it is, the following instructions are generated:

```
AW    R1,CONST
STD   R1,WORK
L     R2,WORK+4
BALR  15,0
BC    10,6(15)
LNR   R2,R2
```

Phase 15 generated a text word that insured that the floating-point word was loaded into R1. The add unnormalized instruction adds the constant 4E00000000000000 to R1. This insures that the integer portion of the number is located in the low order portion of the number. The store double and load instructions load the low order portion of this number into R2. The BALR instruction is generated to insure addressability for the next generated instruction. The add unnormalized instruction sets the condition code, and the three instructions following it do not change the code. A branch on condition code instruction is generated to determine if the original floating-point number is negative. If it is, the instruction following the BC instruction is executed and the number in the general register is made negative by the load negative instruction. Otherwise, the branch on condition code instruction branches around the load negative instruction.

If the in-line function is not IFIX, the instructions for FLOAT and DFLOAT are generated. The instructions for either of these subroutines are the same. The text entries made by Phase 15 before and after the text word containing the in-line function number cause RXGEN to generate instructions for either single precision or double precision variables.

The instructions generated by subroutine FIXFLT/GENBC6 are:

```
LPR   R2-1,R2
ST    R2-1,WORK+4
LD    R1,CONST
AD    R1,WORK
LTR   R2,R2
BALR  15,0
BC    10,6(15)
LNER  R1,R1
```

Phase 15 entered a text word word which forced RXGEN to generate an instruction to load the argument of the function in R2. Subroutine FIXFLT generates an instruction to load positive the value in R2 into R2-1. The value in R2-1 is stored in a work area. The constant 4E00000000000000 is then loaded into register R1, and the contents of the work area are added to register R1. A load and test instruction is generated to determine if the integer number is negative or positive, and the same three instructions are generated for DFLOAT and FLOAT that were generated for IFIX.

EXIT: Subroutine FIXFLT exits to subroutine PRESCN to process the next intermediate text entry.

SUBROUTINES CALLED: Subroutine FIXFLT calls the following subroutines:

1. BASCHK/RXOUT and RROUT to generate instructions.
2. GNBC6 to generate the branch instructions.

Subroutine GNBC6

Subroutine GNBC6 generates branch instructions for the in-line functions.

ENTRANCE: Subroutine GNBC6 is entered from subroutines FIXFLT, SIGN, and DIM.

OPERATION: When subroutine GNBC6 is entered, the calling subroutine has insured that instruction has been generated to set the condition code.

Subroutine GNBC6 generates two instructions:

```
BALR  15,0
BC    M,6(15)
```

where M is the mask for the instruction passed by the subroutine that called GNBC6. These two instructions have the effect of

```
BC    M,*+6
```

EXIT: Subroutine GNBC6 exits to the subroutine that called it.

SUBROUTINES CALLED: Subroutine GNBC6 calls subroutines RXOUT and RROUT to generate instructions.

Subroutine SIGN, DIM, ABS: Chart KR

Phase 25 enters intermediate text words for the in-line functions SIGN, ISIGN, DSIGN, DIM, IDIM, DABS, ABS, and IABS. The first words entered force subroutine RXGEN to generate instructions to load the arguments for the function into general registers, if the mode of the function is integer, or into floating point registers, if the mode of the function is real or double precision. The next word generated by Phase 25 contains the in-line function adjective code, the register numbers (R1 and R2) in the mode/type field, and the in-line function number in the pointer field.

Subroutine SIGN

Subroutine SIGN processes the intermediate text entry for the in-line functions DSIGN, ISIGN, and SIGN.

ENTRANCE: Subroutine SIGN is entered from subroutine FUNGEN.

OPERATION: Subroutine SIGN is entered at three points, one each for the in-line functions SIGN, ISIGN, and DSIGN. At each entry point, the mode is set for the instructions to be generated by subroutine SIGN.

The instructions generated for the in-line function ISIGN by subroutine SIGN are:

```
LPR   R1,R1
LTR   R2,R2
BALR  15,0
BC    10,6(15)
LNR   R1,R1
```

The first argument is made positive by the load positive instruction, and a load and test instruction is generated to determine the sign of the second argument. The branch and link instruction is generated to insure addressability for the next instruction.

A branch on condition code is then generated to determine the result of the load and test instruction. If the result is negative, the next instruction (a load negative instruction to change the sign of the first argument to negative) is executed. If the result of the load and test instruction is positive or 0, the branch on condition code instruction branches around the load negative instruction.

EXIT: Subroutine SIGN exits to subroutine PRESCN to process the next word in the intermediate text.

SUBROUTINES CALLED: During execution subroutine SIGN calls the following subroutines:

1. RROUT to generate the register-to-register instructions.
2. GNBC6 to generate the branch instructions.

Subroutine DIM

Subroutine DIM generates the coding for the in-line functions DIM and IDIM.

ENTRANCE: Subroutine DIM is entered from subroutine FUNGEN.

OPERATION: Subroutine DIM may be entered at two points, one for the in-line function DIM and the other for IDIM. At each entry point, the mode is set for the instructions to be generated by subroutine DIM.

The instructions generated for the in-line function DIM by subroutine DIM are:

```
SER   R1,R2
BALR  15,0
BC    2,6(15)
SER   R2,R2
```

The second argument is subtracted from the first argument. A BALR instruction is generated to insure addressability for the next generated instruction. The condition code is set by the subtract instruction, and a branch on condition code is generated to test if the difference is negative. If it is, the next instruction generated is executed. This instruction zeros the contents of R2. If the difference is positive, this instruction is skipped, and the next instruction is executed.

The same instructions are generated for IDIM, except that the registers used are general registers, and the op codes generated are for fixed point instructions.

EXIT: Subroutine DIM exits to subroutine PRESCN to process the next text word.

SUBROUTINES CALLED: Subroutine DIM calls the following subroutines:

1. RROUT to generate the register-to-register operations.
2. GNBC6 to generate the branch instructions.

Subroutine ABS

Subroutine ABS generates the coding for the in-line functions ABS, DABS, and IABS.

ENTRANCE: Subroutine ABS is entered from subroutine FUNGEN.

OPERATION: Subroutine ABS is entered at three points, one each for the in-line functions ABS, DABS, and IABS. At each entry, the mode is set for the instruction to be generated by subroutine ABS.

Subroutine ABS generates an instruction to load positive the contents of R1 into R1. For example, the instruction

```
LPR   R1,R1
```

makes the contents of register R1 positive. The register type (fixed or floating point) and the op code depend on the in-line function called.

EXIT: Subroutine ABS exits to subroutine PRESCN to process the next intermediate text word.

SUBROUTINE CALLED: Subroutine ABS calls subroutine RROUT to generate the load positive instruction.

Subroutine STOP/PAUSE: Chart KS

Subroutine STOP/PAUSE generates instructions for STOP and PAUSE statements in the object program.

ENTRANCE: Subroutine STOP/PAUSE is entered by subroutine PRESCN when PRESCN recognizes a STOP or PAUSE adjective code.

OPERATION: Subroutine STOP/PAUSE gets the relative location of the STOP or PAUSE in IBCOM. This location is the distance in bytes from the beginning of IBCOM for the STOP or PAUSE instructions.

The location of the beginning address of IBCOM in the object program is accessed and inserted into a skeleton instruction. An instruction is generated to load the beginning address of IBCOM into register 15. A base-displacement instruction is generated to branch and link to STOP or PAUSE in IBCOM. The displacement is the distance in bytes from the beginning of IBCOM; the base register used is register 15.

A constant indicating the number of bytes in the halt number is generated along with the number itself. This constant immediately follows the branch and link instruction.

EXIT: Subroutine STOP/PAUSE exits to subroutine PRESCN to process the next intermediate text entry.

SUBROUTINES CALLED: Subroutine STOP/PAUSE calls the following subroutines:

1. BASCHK to generate the load instruction.
2. RXOUT to generate the branch and link instruction.
3. RROUT to generate the constant.

Subroutine END: Chart KT

Subroutine END processes the intermediate text entry for the END card. It generates instructions to restore registers and branch to the start of the program. Subroutine END punches text cards and RLD cards for the branch lists and the base value table, and creates the END card.

ENTRANCE: Subroutine END is entered from subroutine PRESCN when an END adjective code is recognized.

OPERATION: Subroutine END determines if the program is a main program. If it is, it tests further to determine if a STOP instruction was included in the program. If no STOP was generated, a call is generated to call the IBCOM subroutine IBEXIT.

The location counter is saved for the computation of the size of the program, and the text card data set for the object program instructions is closed. The text card addresses and the byte count for the card are set to zero. The following instructions and constants are generated:

```
DC    CL7'PROGRAM'
DC    XL1'07'
STM   14,12,SAVE
LM    4,N,BASVAL
BC    15,START
```

The two generated constants are required by the calling sequence. The first is the program name. It is always seven bytes in length. The second constant is the number of bytes in the program name. The STM instruction stores the general registers in the save area. The LM instruction loads all base registers for the program being executed in the general registers 4 through N, the highest base register used. A branch is generated to the first instruction to be executed by the program.

The print buffers for the statement numbers are closed, and the END subroutine is initialized to enter the branch list for statement numbers in text cards. All statement number entries in the branch list are then written on the output data sets. After all statement numbers have been put into text cards, subroutine END is initialized to put all branch list entries for arithmetic statement functions and branch addresses generated for DO loops in the text cards. All branch list entries are then entered into text cards.

After the entries have been made, RLD cards are punched for entries in the branch list. The statement number entries are punched first, and then RLD cards are punched for the ASF and DO branches.

A text card is then set up in the proper format for all base registers used for the object program. The addresses for the registers used to access COMMON are entered in the card first, followed by the base register addresses used to access data and coding in the program. The card containing these addresses is then put on the output data set. All base registers can be contained in one card. The base register addresses are inserted in an RLD card to insure relocatability. The registers used for COMMON are given an ESID of 2, and the registers used by the program are given an ESID of 1.

A text card buffer is then cleared for the END card of the object deck. The information identifying the END card is inserted in the buffer. A test is made to determine if the program is a subprogram. If it is not, the entry point is inserted in the text card. For main programs this entry point is eight. For all programs, the size of the program is entered in the end card. The text card is put out; the size of COMMON and the size of the program are printed.

EXIT: Subroutine END exits to the FORTRAN System Director:

1. To call the Control Card routine, if no errors or warnings have been detected.
2. To call Phase 30 if error/warning conditions have been detected during the compilation.

SUBROUTINES CALLED: Subroutine END calls the following subroutines:

1. BASCHK/RXOUT to generate instructions.
2. TXTOUT to put out the END card.

Subroutine ENTRY: Chart KU

Subroutine ENTRY generates instructions in the object program to initialize the object program.

ENTRANCE: Subroutine ENTRY is called by subroutine INITIALIZATION.

OPERATION: Subroutine ENTRY generates the first executable instructions in the object program. Subroutine ENTRY saves the location counter before it generates an object code so the instruction that transfers control to the object program branches to the correct address.

If a type 3 program is being compiled, an instruction to set up addressability of the program is generated. The FORTRAN System Director establishes addressability for the object program in register 15, but the object program does not use register 15 as the base register. Subroutine ENTRY generates an instruction to load the contents of register 15 into register 8, establishing addressability with register 8 for the initialization instructions in the object program.

If the program being compiled is a subprogram, an instruction is generated to save the address of the parameter list in main storage. The address of the parameter list is passed to the subprogram in general register 1. Object programs use register 1 as a work register; therefore, general register 1 must be saved for the epilog.

If the program is a main program, a call is generated to subroutine IBFINT in IBCOM. This subroutine is executed by the main program to initialize program status words for interruption. If the program is a subprogram, it does not have to call IBFINT because the main program has established the program status words.

If the program contains any arithmetic statement function (ASF), subroutine ENTRY generates a branch around the object code for the ASFs. The ASFs appear immediately after the coding to initialize the object program.

If any external calls were included in the program, the following instructions, which save and set up new save area pointers, are generated:

```
LR      12,13
LA      13,SAVE
ST      13,8(13)
ST      12,4(12)
```

EXIT: Subroutine ENTRY exits to the subroutine that called it.

SUBROUTINES CALLED: During execution subroutine ENTRY references subroutines RROUT and RXOUT to generate object program instructions.

Subroutine GENBC: Chart KV

Subroutine GENBC generates the branching instructions for the instructions generated by an IF statement.

ENTRANCE: Subroutine GENBC is called by subroutine ARITHI when it is processing IF statements.

OPERATION: Subroutine GENBC accesses the address of the beginning of the branch list, and the branch list number for the statement number in its entry in the overflow table. A load instruction is then generated to load the address in a general register.

The mask for the branch on condition code is accessed in main storage. The mask was inserted in a location in storage by subroutine ARITHI. Subroutine GENBC then generates a branch on condition code instruction.

EXIT: Subroutine GENBC returns to the subroutine that called it.

SUBROUTINES CALLED: During execution subroutine GENBC calls the following subroutines:

1. BASCHK to generate the load instruction.
2. RROUT to generate the branch on condition code instruction.

Subroutine GET: Chart KW

Subroutine GET updates the input buffer pointer, and initializes action to read another intermediate text record, if a buffer has been exhausted.

ENTRANCE: Subroutine GET is called by subroutines PRESCN, FIXFLT, IOLIST, RDWRT, CGOTO, SUBRUT, ARITHI, and DO1 to access intermediate text words.

OPERATION: Subroutine GET updates the buffer pointer, and tests for the end of the buffer. If it has not been reached, control passes to the subroutine that called subroutine GET.

If the end of the buffer has been reached, a test is made to determine if the compilation being executed had no need to write the intermediate text on an output tape; that is, the buffers are large enough to contain all the intermediate text records for the compilation. If this is an in-storage compile, the pointer to the second buffer in the double-buffer system is set in a register to access intermediate text words.

If the compilation is not in-storage, a test is made to determine if the end of the input data set has been reached. If it has, control is passed to the subroutine that called subroutine GET.

If the end of the data set has not been reached, a supervisor call is issued to read a text record into the buffer just exhausted. There are two returns to subroutine GET from the supervisor call. One entrance is used if the end of data set has been reached. If this entrance is used, subroutine GET sets an indicator for the end of the text data set.

The other return from the supervisor call is used if the end of data set is not reached. The intermediate text record is read into the buffer just exhausted, and the buffer pointer is set to process the intermediate text words in the second buffer.

EXIT: Subroutine GET exits to the subroutine that called it.

SUBROUTINES CALLED: The FORTRAN System Director is referenced to read an intermediate text record into an input buffer.

Subroutines BASCHK/RXOUT, RROUT: Chart KX

Subroutine BASCHK/RXOUT

Subroutine BASCHK/RXOUT processes RX instructions generated by the Phase 25 subroutines.

ENTRANCE: Subroutine BASCHK/RXOUT is called by subroutines IOLIST, ENDIO, RDWRT, TRGEN, CGOTO, SUBRUT, FUNGEN/EREXIT, STOP/PAUSE, GENBC, DO1, ENDDO, ASFEXP, ASFUSE, END, RXGEN/LM/STM, ENTRY, FIXFLT, RETURN, SAOP, and AOP.

OPERATION: Subroutine BASCHK/RXOUT is entered at two points. The first entry, BASCHK, is used to determine if a spill base register was used when storage was assigned to symbols in Phases 12 and 20. It tests for a spill base register in the program. If there is none, BASCHK/RXOUT branches to the portion of coding labeled RXOUT. If a spill base register is used in the compilation, BASCHK/RXOUT tests if the instruction to be generated uses a spill base register. If it does not, BASCHK/RXOUT branches to RXOUT.

If a spill base is used in the instruction, a test is made to determine if the last used the same spill base register. If it did not, the linkage register and the instruction pointer that were entered by the subroutine that called BASCHK/RXOUT are saved, because RXOUT must be entered to generate an instruction to load register 7 and the linkage registers for Phase 25 must be saved. A test is made to determine if the program used 16K bytes for COMMON. If it did, 8 is subtracted from the displacement of the instruction that is to be generated.

The load instruction is generated to load register 7 with the base value generated for the spill base register. Subroutine RXOUT is called to generate the load instruction; the registers for linkage and the instruction pointer are saved. When RXOUT was called, it generated its own linkage and returned to a point in subroutine RXOUT where the original registers were restored.

Subroutine BASCHK/RXOUT changes the base in the original instruction to be generated to base register 7.

The portion of the coding called RXOUT begins. The link register is saved, the instruction length is set to 4, and subroutine TXTEST is entered to enter the instruction in a text card.

EXIT: Subroutine BASCHK/RXOUT exits to subroutine TXTEST.

SUBROUTINES CALLED: Subroutine BASCHK/RXOUT calls subroutine RXOUT to load a spill base address in register 7.

Subroutine RROUT

Subroutine RROUT generates register-to-register instructions.

ENTRANCE: Subroutine RROUT is entered from subroutines RDWRT, SAOP, TRGEN, CGOTO, FUNGEN/EREXIT, STOP/PAUSE, GENBC, ASFUSE, RXGEN/LM/STM, SIGN, DIM, ABS, ENTRY, and RETURN.

OPERATION: The link register is saved for subroutine TXTEST, and the instruction length is set to 2.

EXIT: Subroutine RROUT exits to subroutine TXTEST to insert the instruction in a text card.

Subroutines TXTEST, RLDTXT, and TXTOUT: Chart KZ

Subroutine TXTEST

Subroutine TXTEST moves an instruction to the output buffer and determines if that instruction must be put out on the GO tape and/or punched in a card.

ENTRANCE: Subroutine TXTEST is entered by subroutines RROUT and BASCHK/RXOUT. Subroutine TXTEST may also be entered by subroutine RLDTXT, if TXTEST previously determined that an instruction can not fit on a card.

OPERATION: If neither the GO, GOGO, nor the DECK options are specified, subroutine TXTEST returns control to the subroutine that called subroutines RROUT or BASCHK/RXOUT. If one of these options is on, subroutine TXTEST updates the location counter by the length of the instruction. TXTEST then determines if the instruction can fit on a text card. If it cannot, the location counter is decremented by the length of the instruction, and an indicator is set for subroutine RLDTXT to indicate that this condition has occurred. Subroutine RLDTXT is entered to put out the card image.

If the instruction can fit on the card, it is moved to the card output buffer. The

count of the number of bytes in the card buffer and the buffer pointer are updated by the number of bytes in the instruction. A test is made to determine if the end of the buffer has been reached. If it has, subroutine RLDTXT is called to output the card image.

EXIT: Subroutine TXTEST exits to the subroutine that called it or to subroutine RLDTXT, if it is determined that the instruction being processed will not fit in this card image.

SUBROUTINES CALLED: Subroutine TXTEST calls subroutine RLDTXT if the end of the buffer is reached after an instruction has been inserted in a card image.

Subroutine RLDTXT

Subroutine RLDTXT inserts the byte count in the card to be put out, and initializes a new buffer.

ENTRANCE: Subroutine RLDTXT is entered by subroutine TXTEST under two conditions. It is entered as an open subroutine if an instruction must be inserted in a card and the instruction will not fit in the card; it is entered as a closed subroutine if the instruction is inserted in the card, and the end of the buffer has been reached.

OPERATION: Subroutine RLDTXT inserts the byte count into the card image to be put out, and calls subroutine TXTOUT to put out the card image. The buffer pointers are switched to start inserting instructions in a new buffer. The new buffer is cleared, the address constant is subtracted from the address currently in the location counter, and the result is inserted in the new text card buffer.

The indicator set in subroutine TXTEST is tested to determine whether the subroutine shall pass control to subroutine TXTEST or to the subroutine that called subroutines RROUT or BASCHK/RXOUT.

EXIT: Subroutine RLDTXT exits to subroutine TXTEST.

SUBROUTINES CALLED: Subroutine RLDTXT calls subroutine TXTOUT to put out the card image.

Subroutine TXTOUT

Subroutine TXTOUT inserts the card sequence number and the program identification in the card image output buffer, and outputs the card image on either the GO tape and/or in a text card.

ENTRANCE: Subroutine TXTOUT is called by subroutine RLDTXT.

OPERATION: Subroutine TXTOUT inserts the card sequence number and the program identification in the card image output buffer. If the GO or GOGO option is specified by the user, the card is written on the GO tape. If the DECK option is specified, the card image is punched in a text card.

EXIT: Subroutine TXTOUT returns to the subroutine that called it.

SUBROUTINES CALLED: The FORTRAN System Director is referenced to write the card image on the GO tape or to punch the card image in a text card.

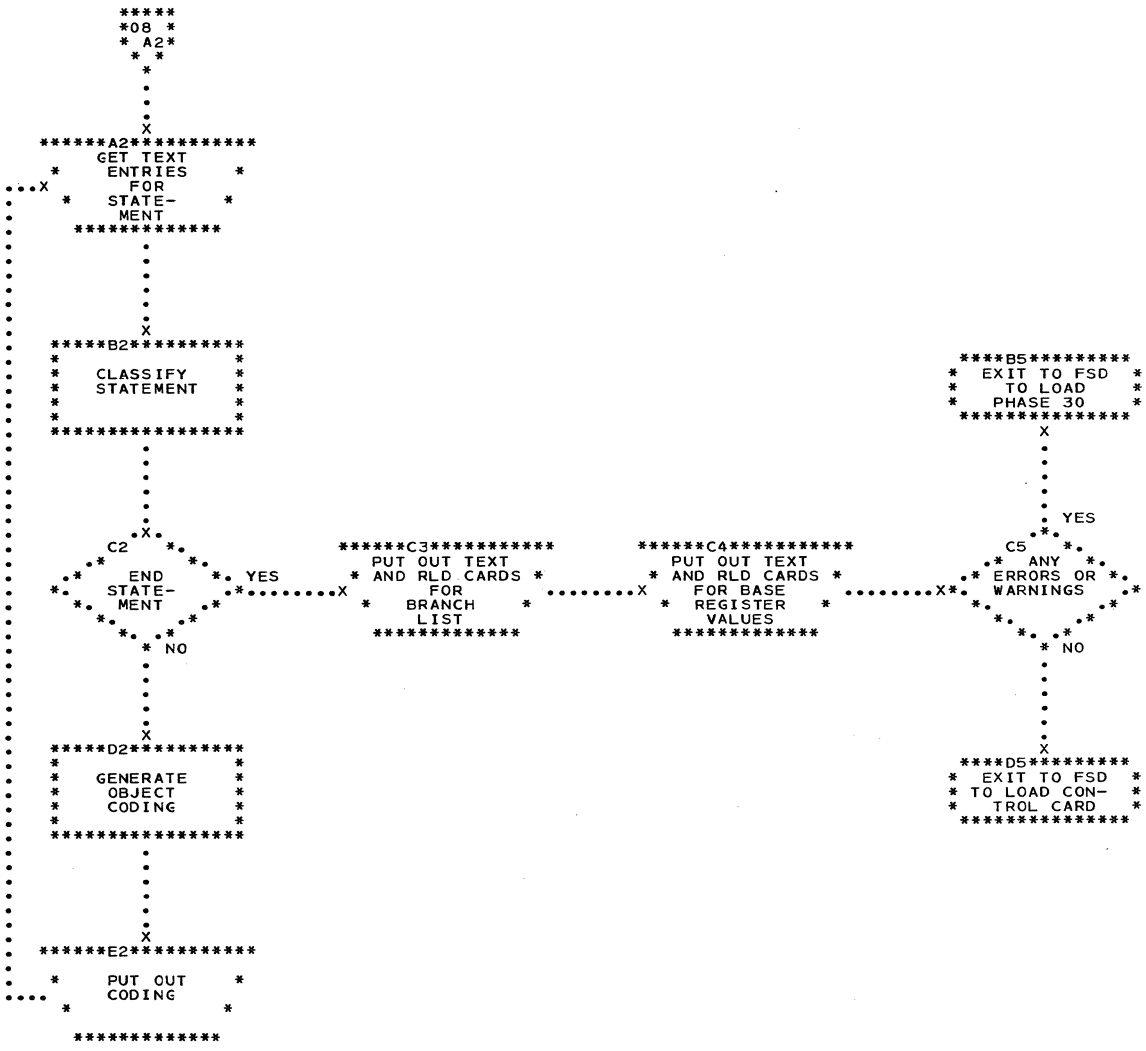


Chart 08. Phase 25 Overall Logic Diagram

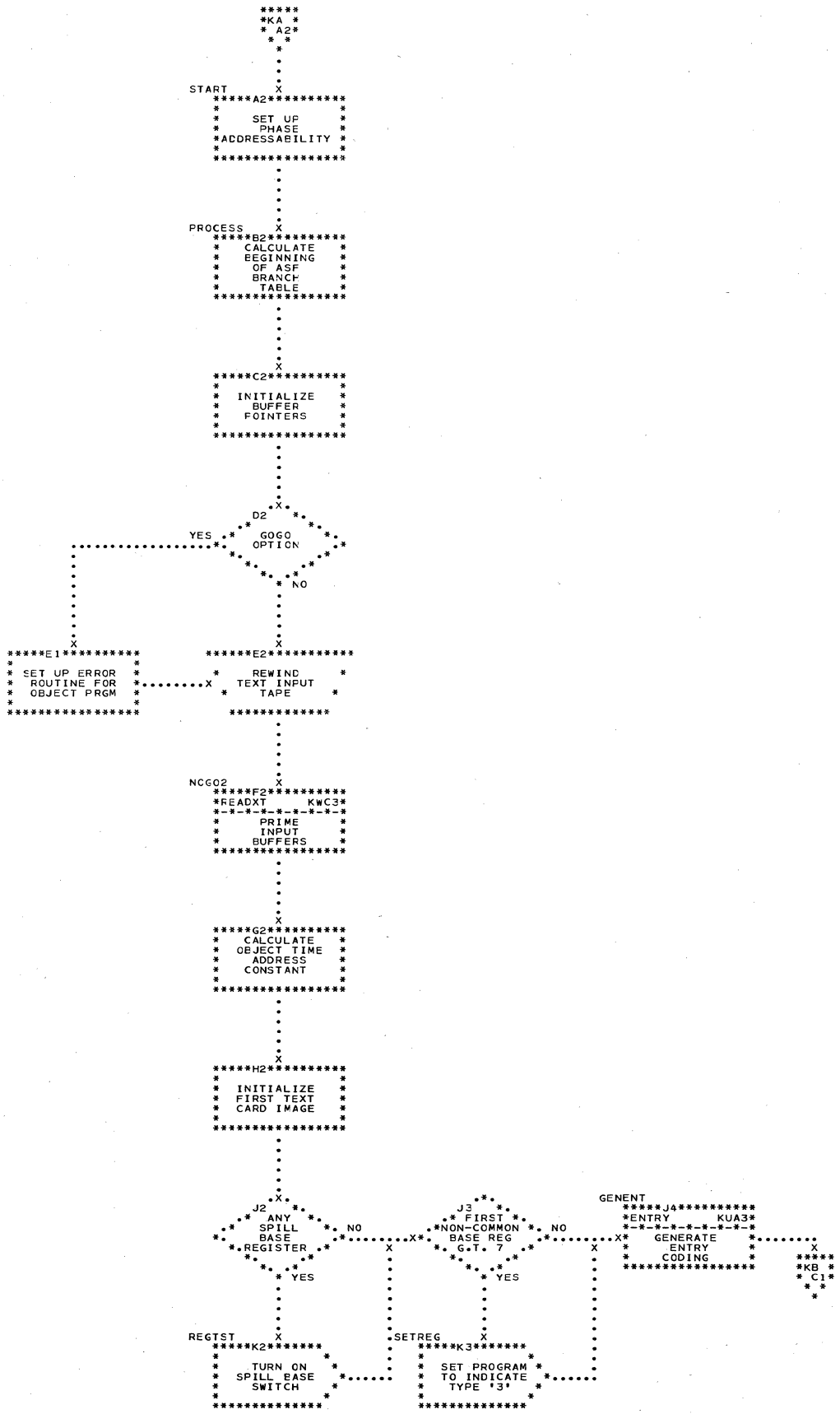


Chart KA. Subroutine INITIALIZATION

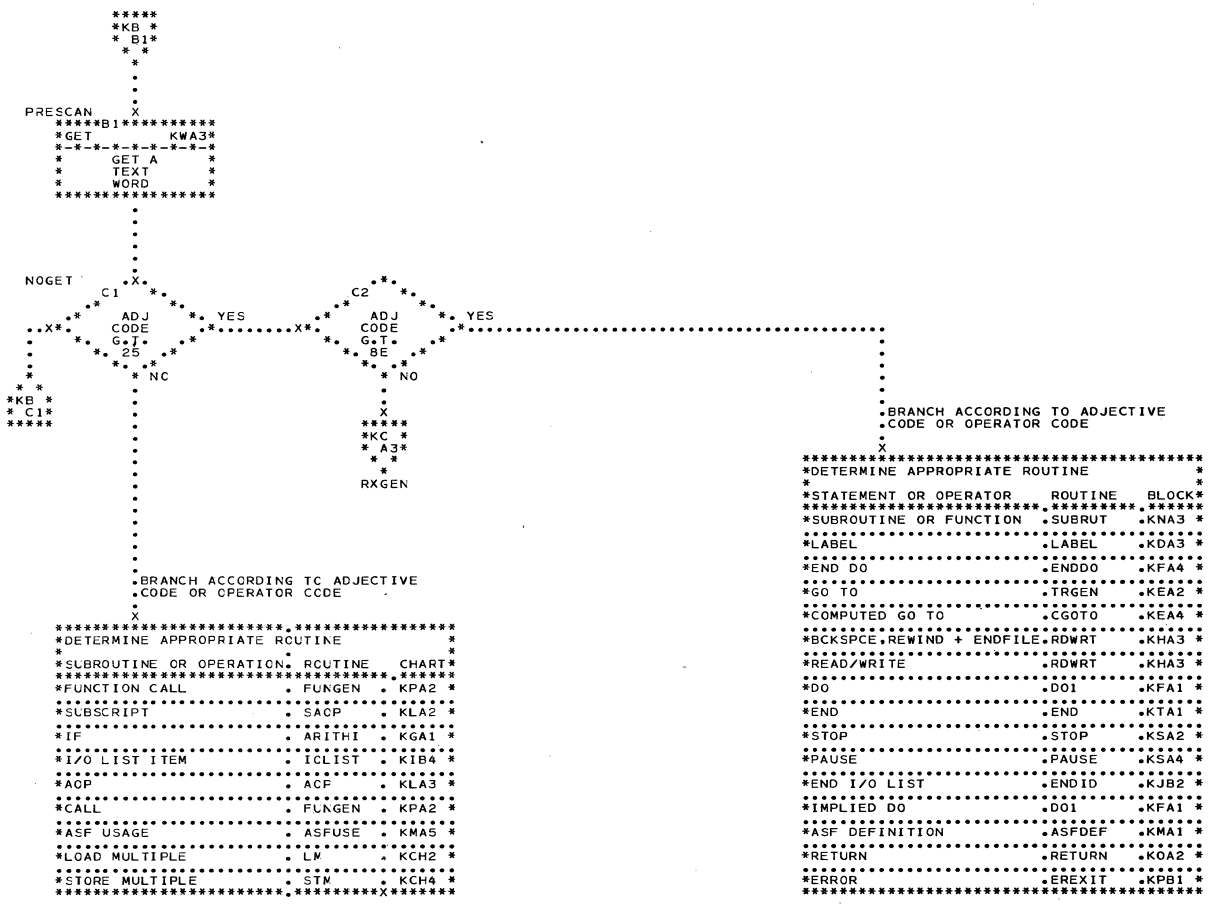


Chart KB. Subroutine PRESCAN

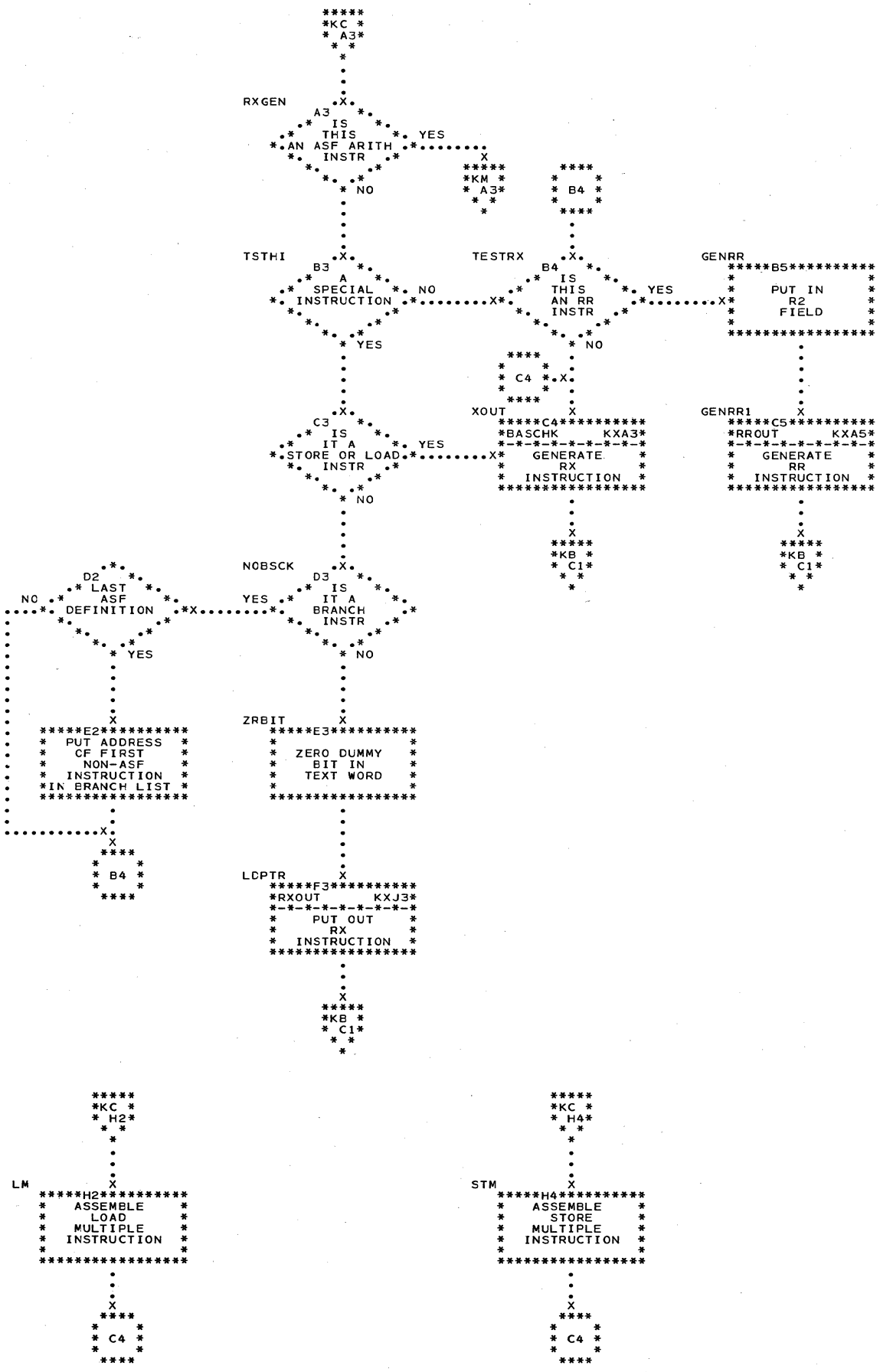


Chart KC. Subroutine RXGEN/lm/stm

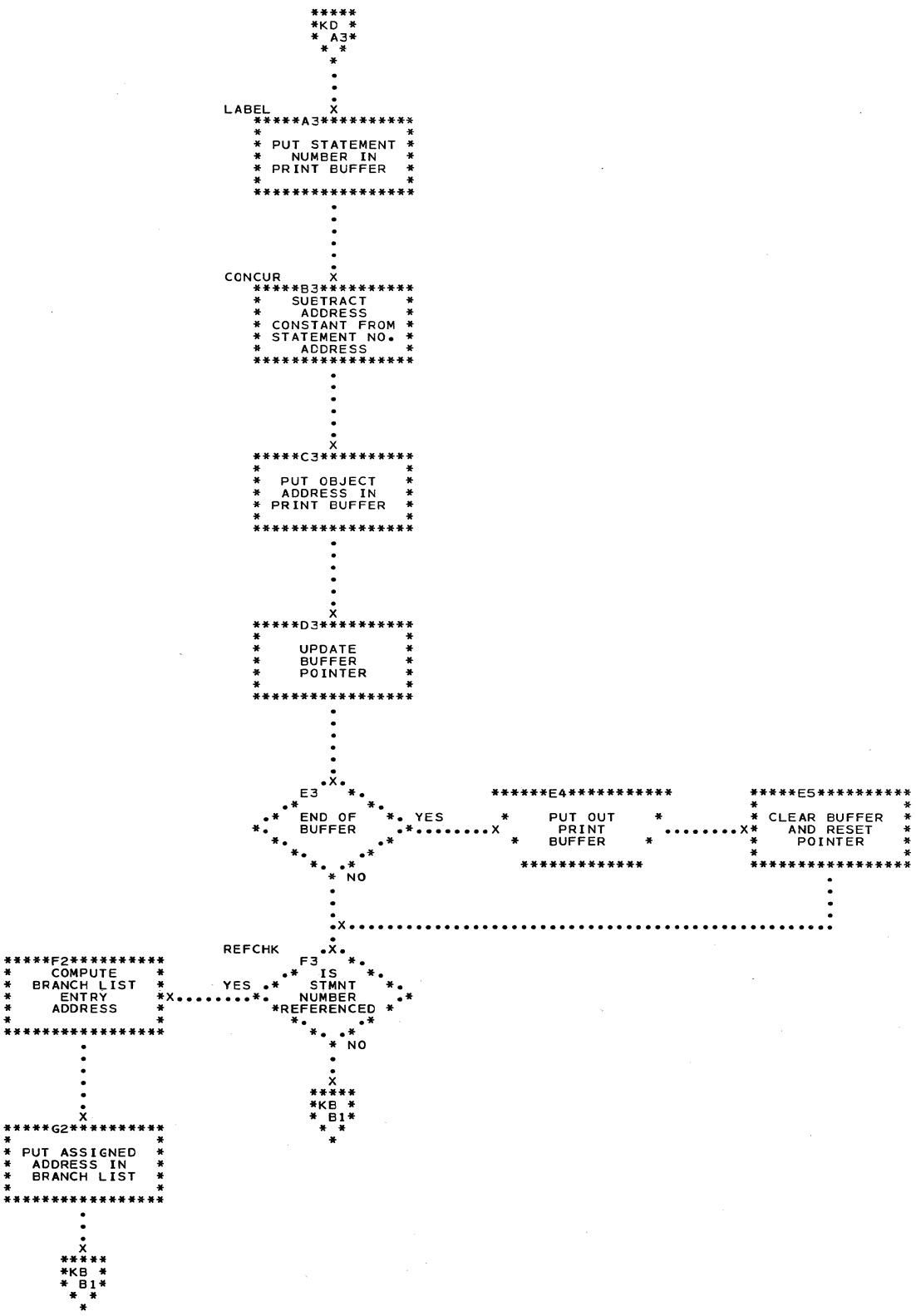


Chart KD. Subroutine LABEL

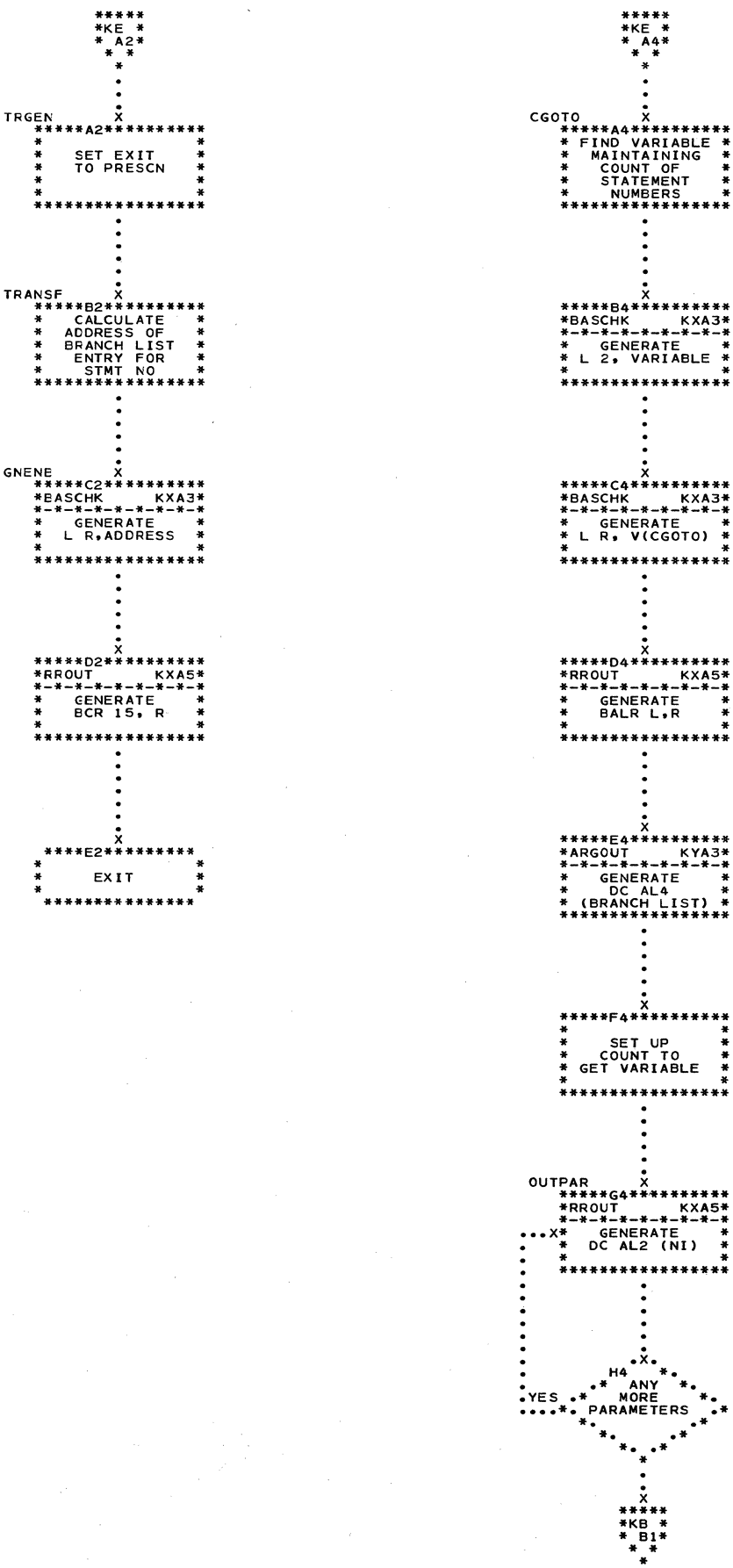


Chart KE. Subroutines TRGEN, CGOTO

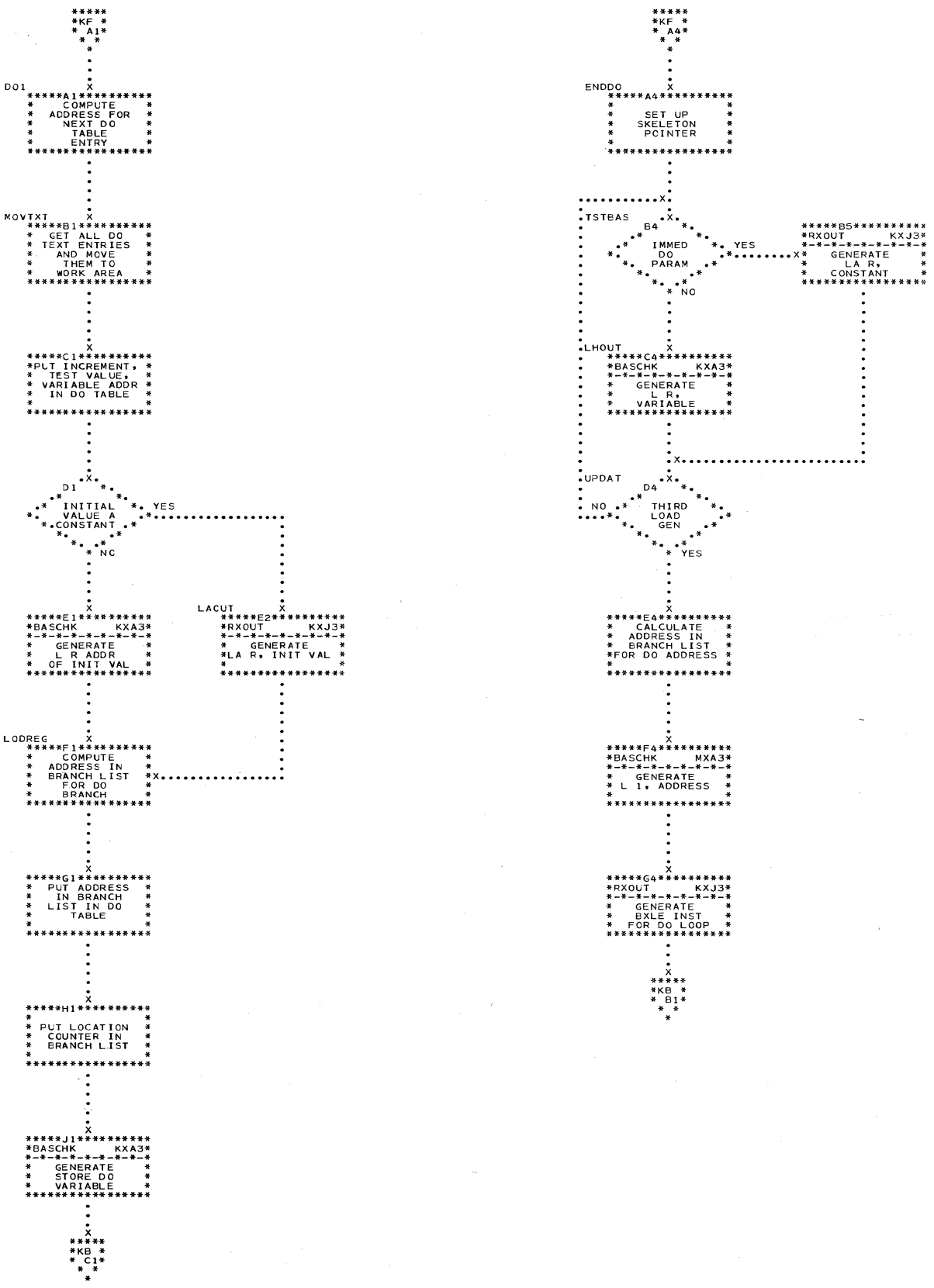


Chart KF. Subroutines DO1, ENDDO

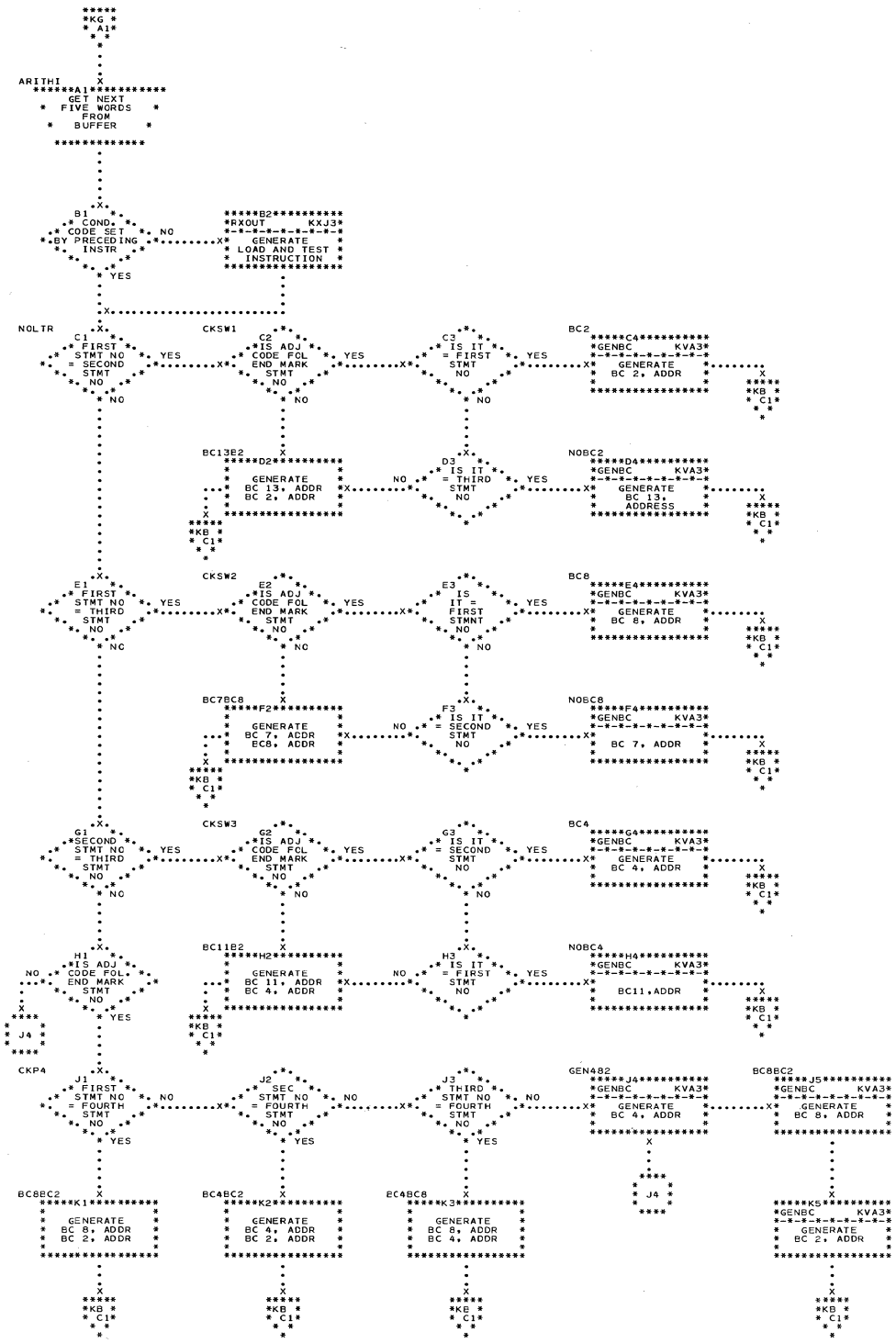


Chart KG. Subroutine ARITH1

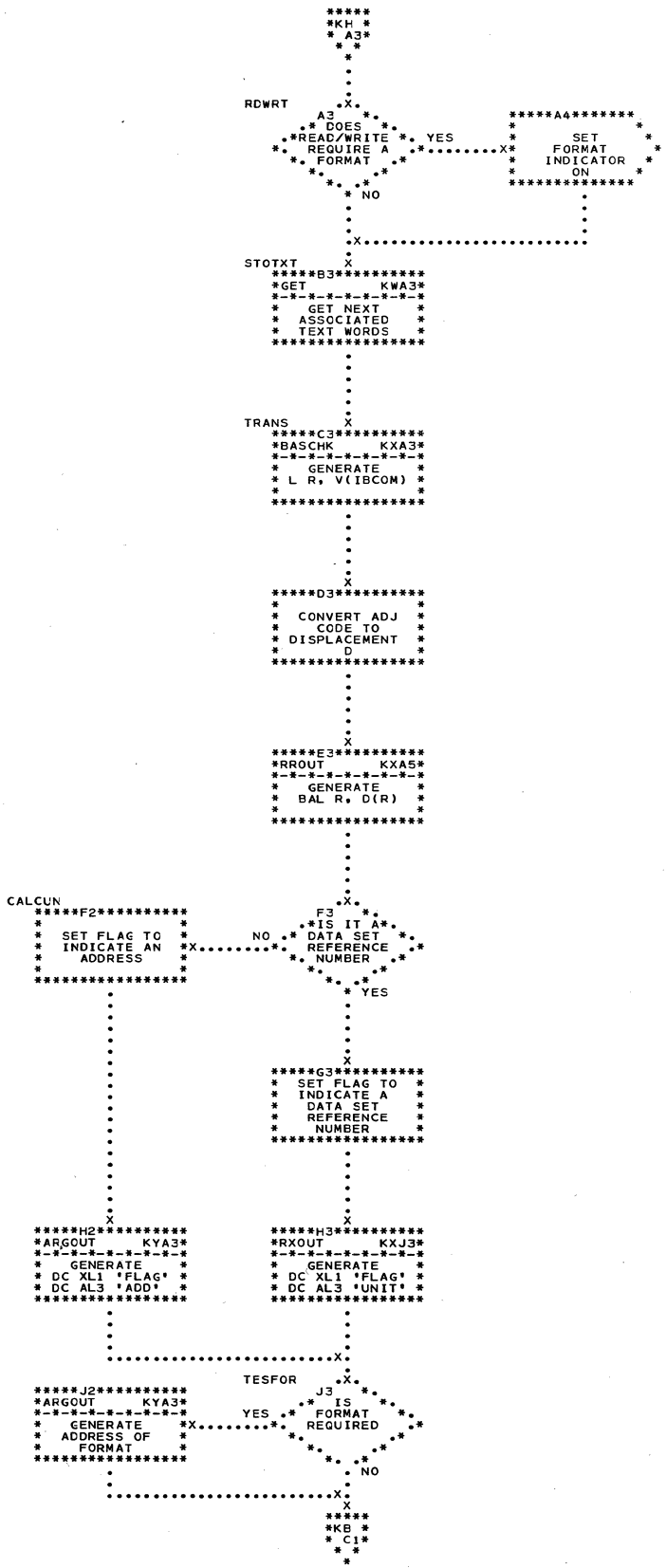


Chart KH. Subroutine RDWRT

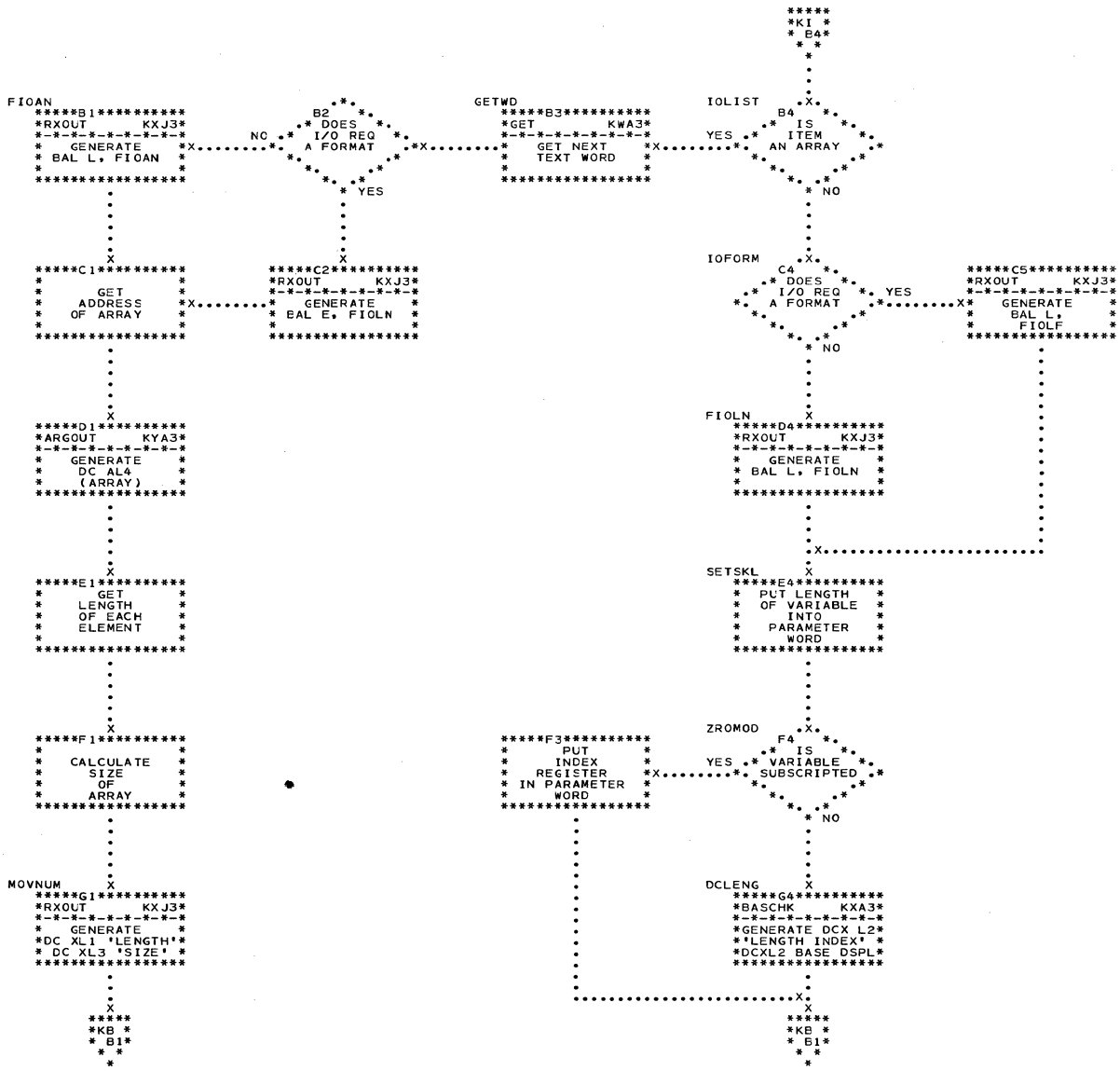


Chart KI. Subroutine IOLIST

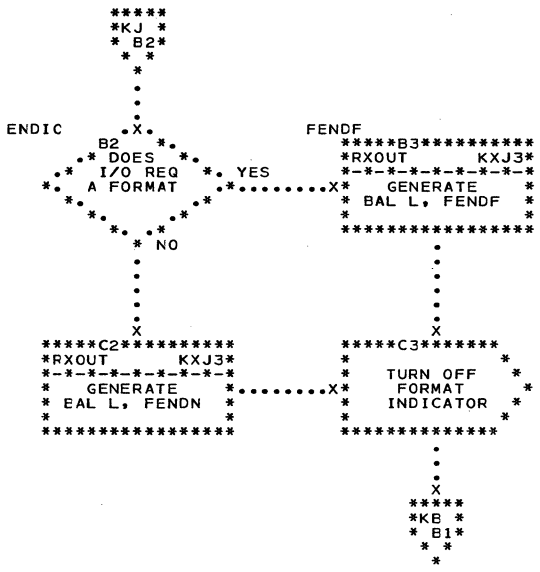


Chart KJ. Subroutine ENDI0

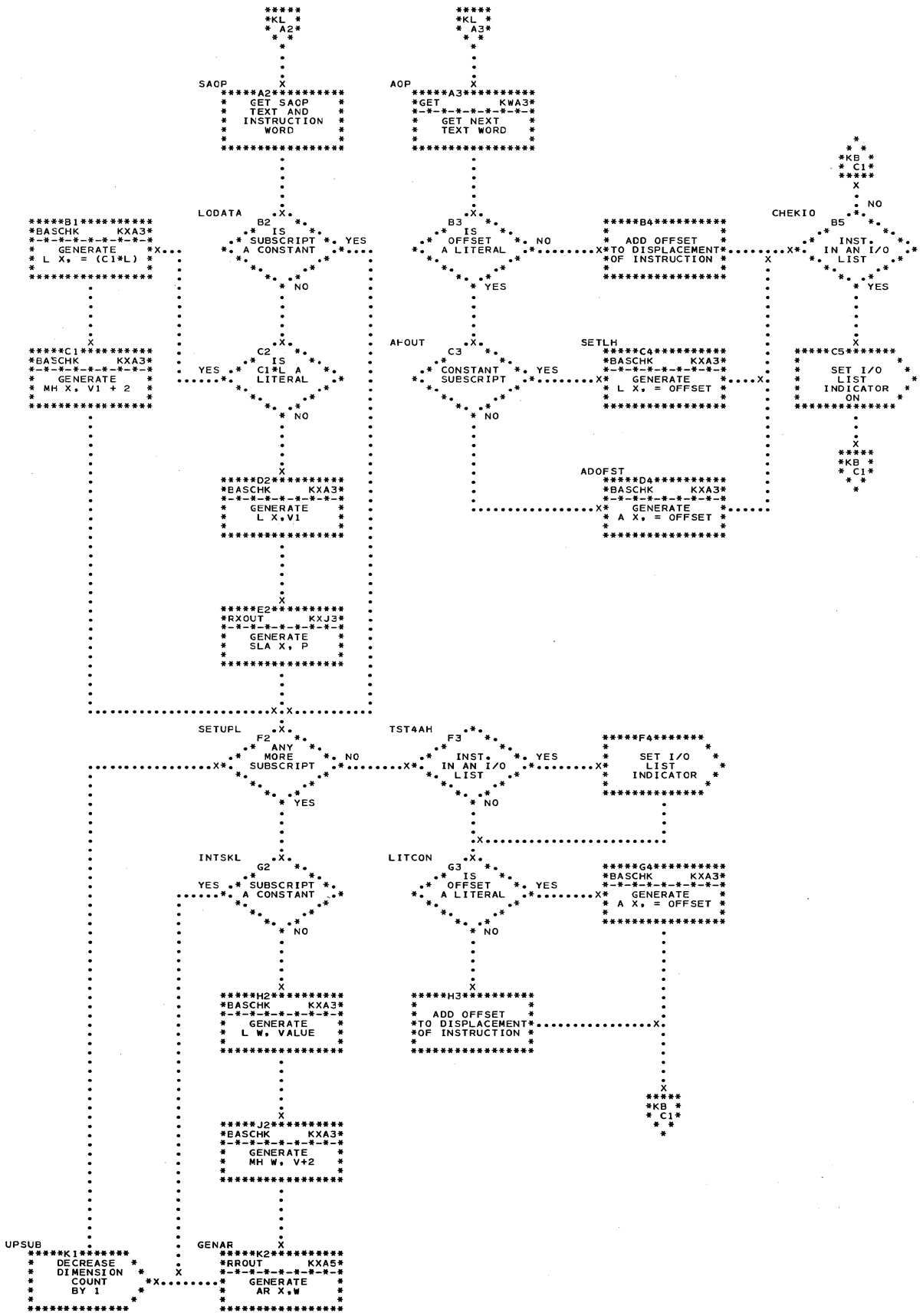


Chart KL. Subroutines SAOP, AOP

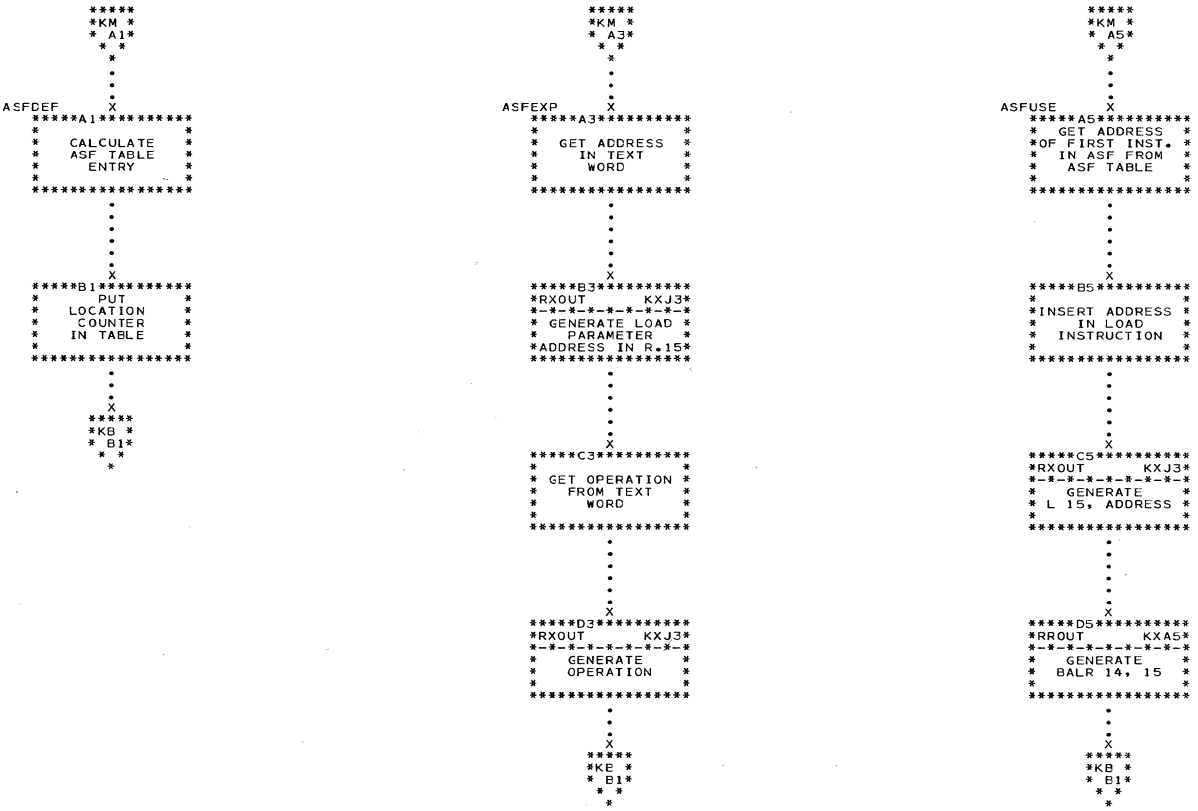


Chart KM. Subroutines ASFDEF, ASFEXP, ASFUSE

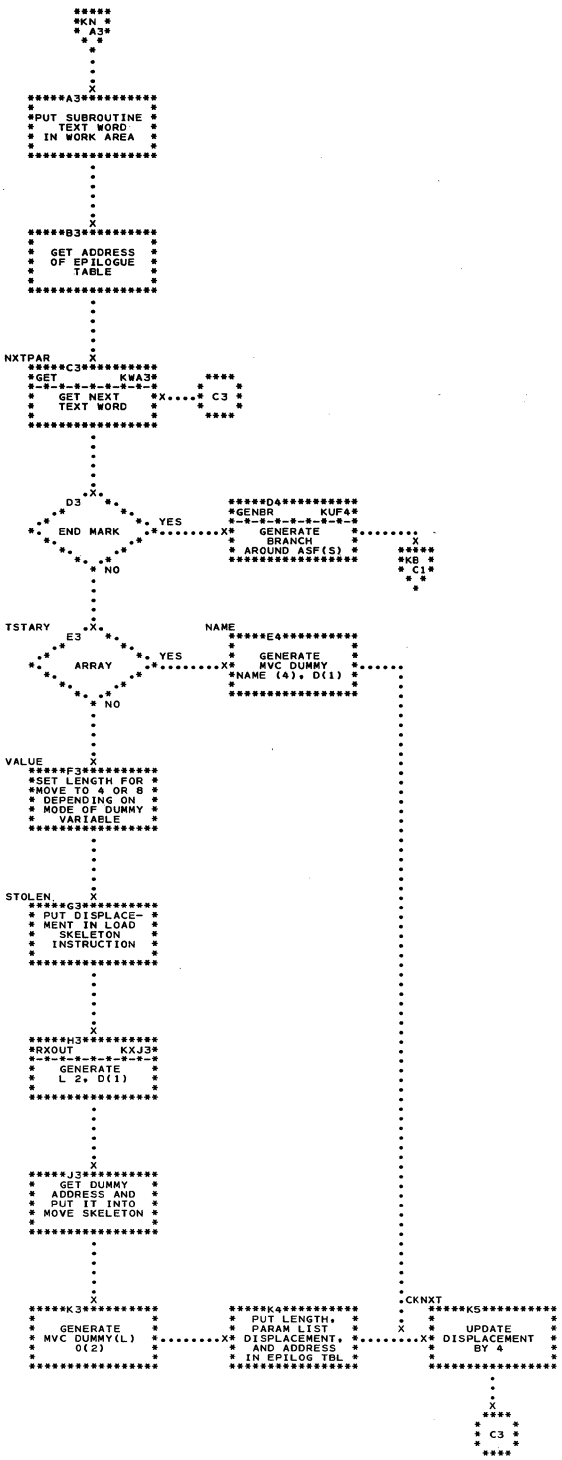


Chart KN. Subroutine SUBRUT

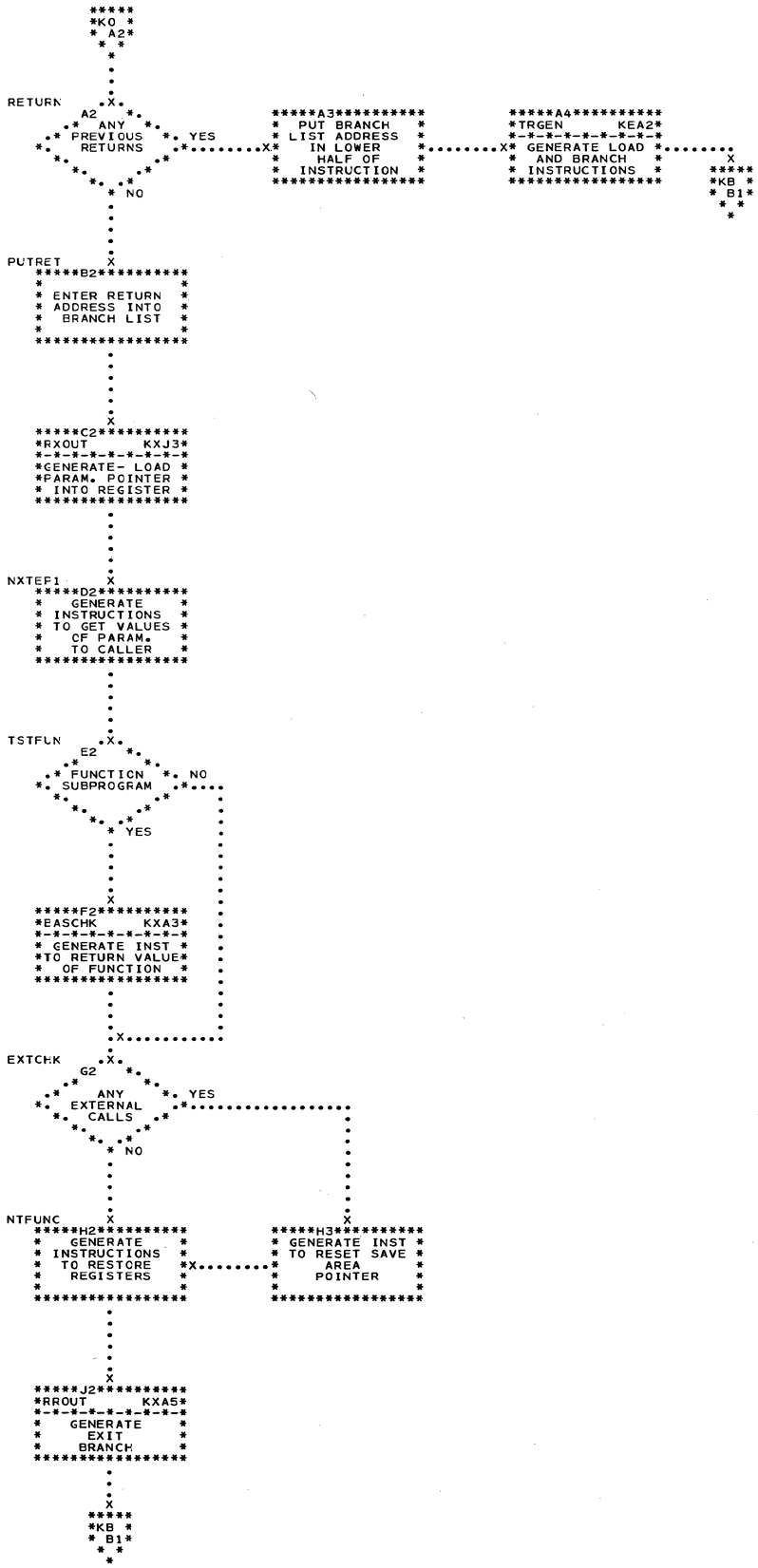


Chart KO. Subroutine RETURN

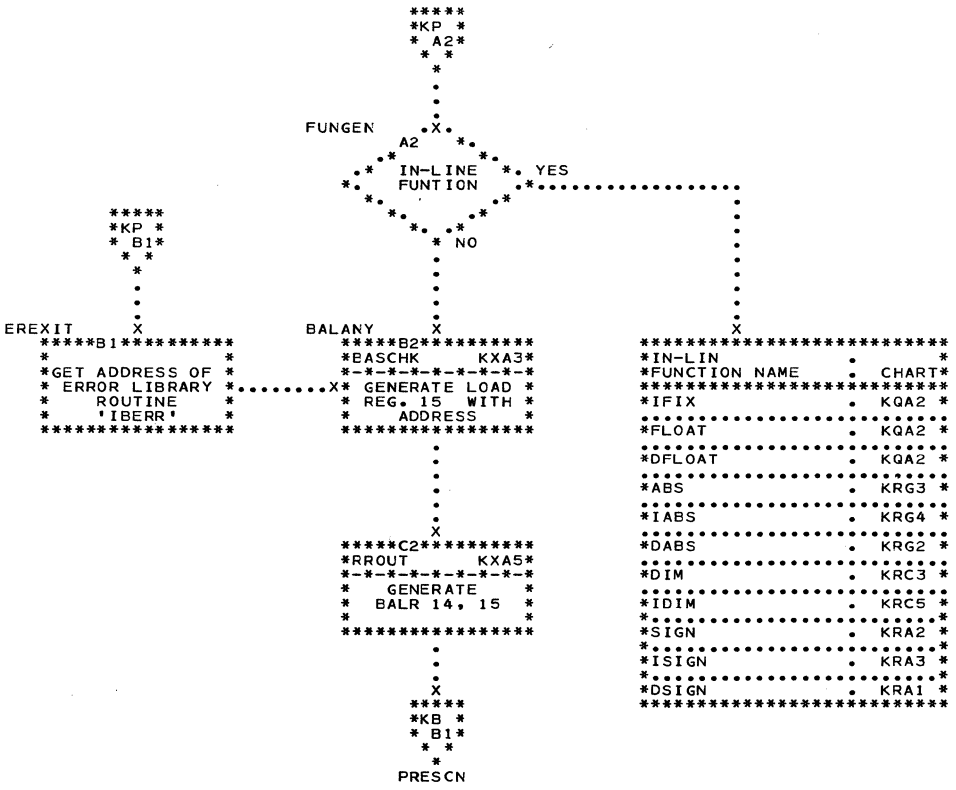


Chart KP. Subroutine FUNGEN/EREXIT

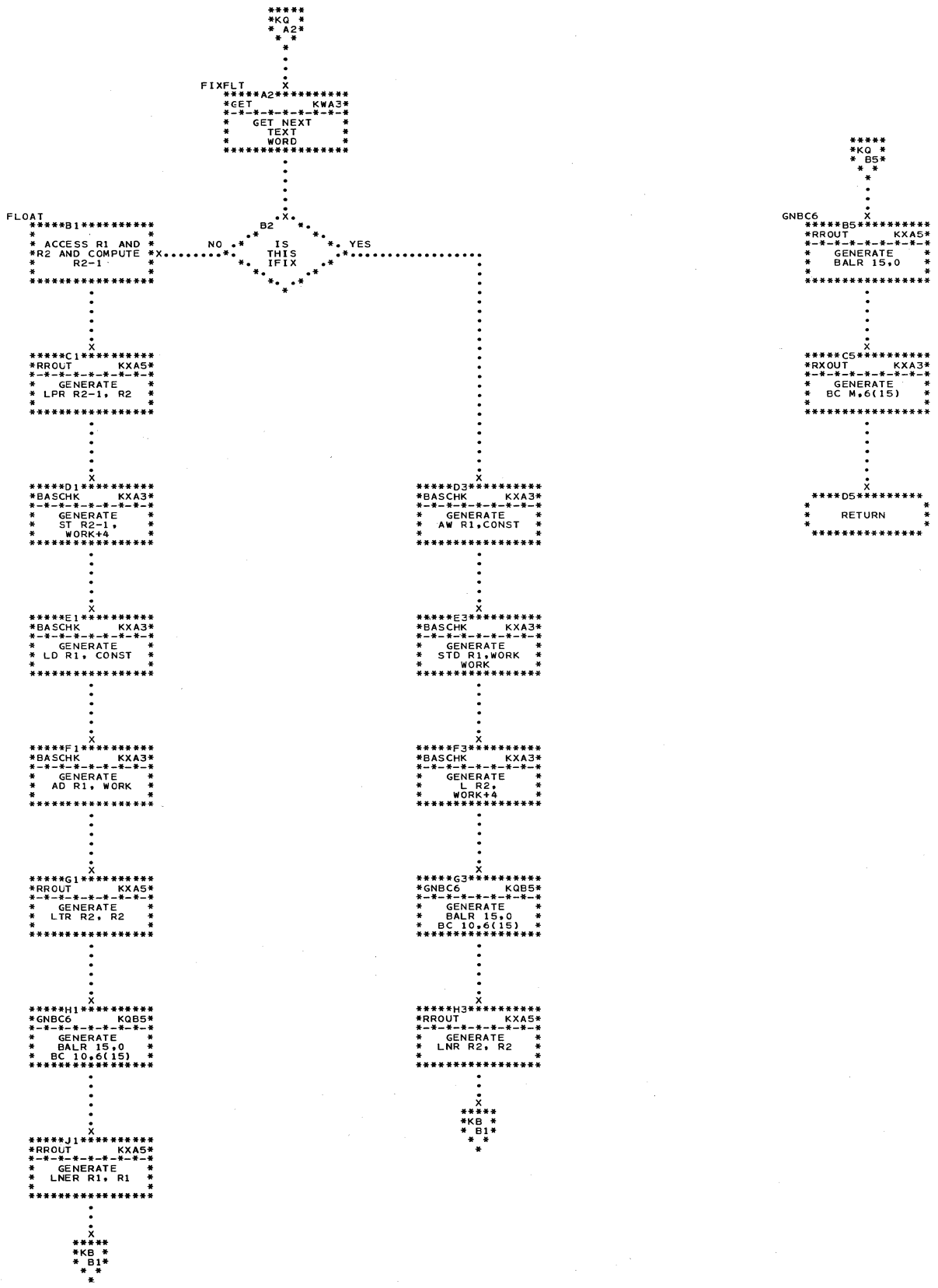


Chart KQ. Subroutine FIXFLT/GNBC6

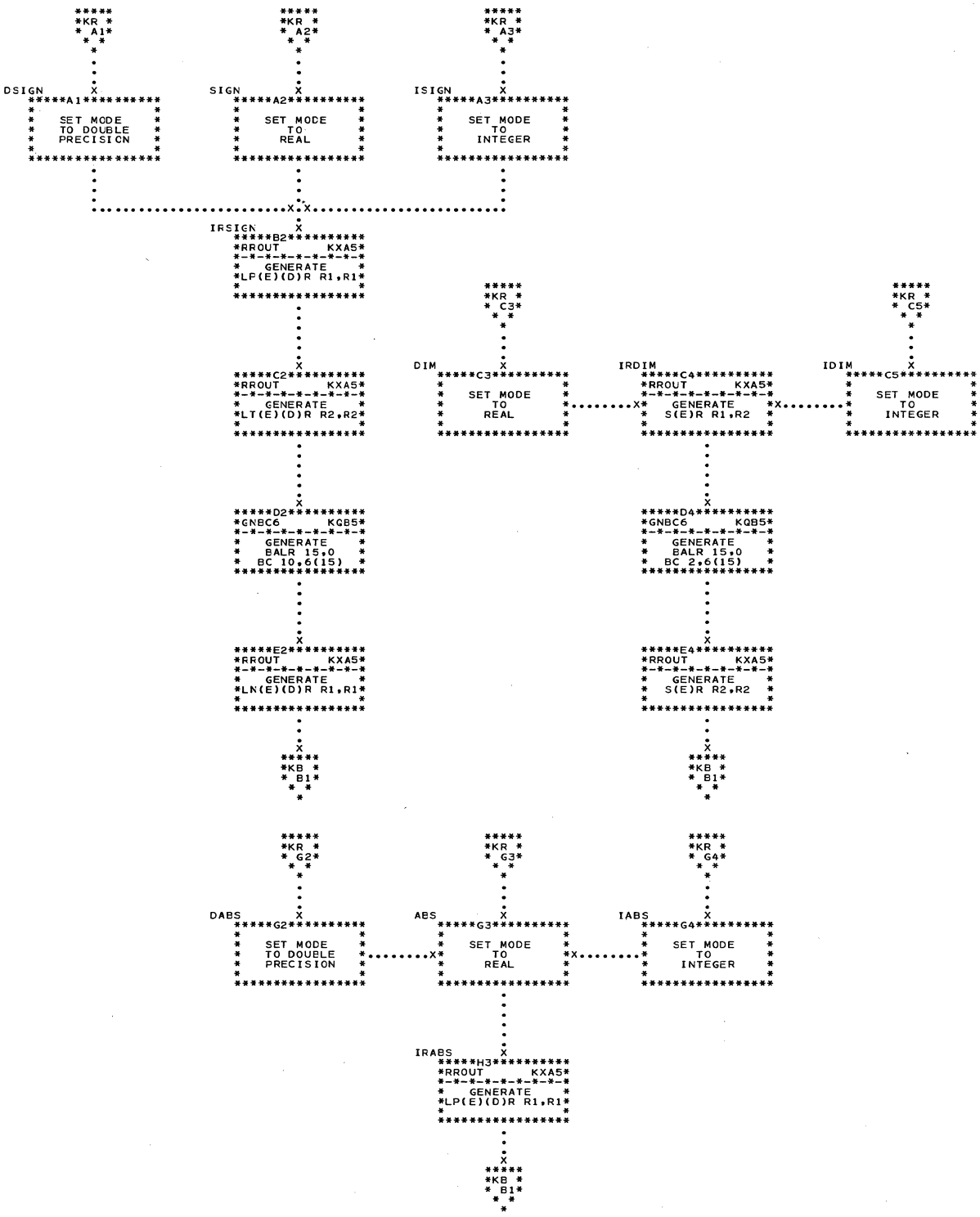


Chart KR. Subroutines SIGN, DIM, ABS

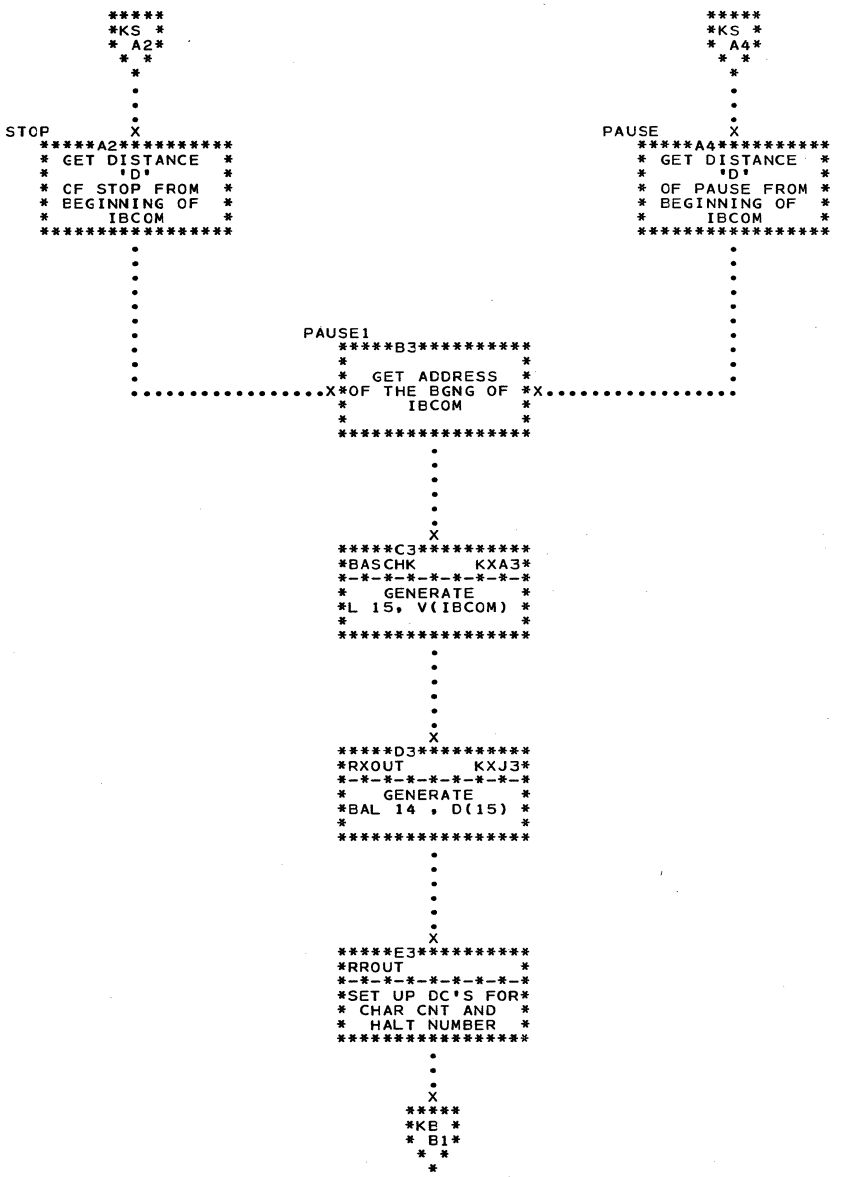


Chart KS. Subroutine STOP/PAUSE

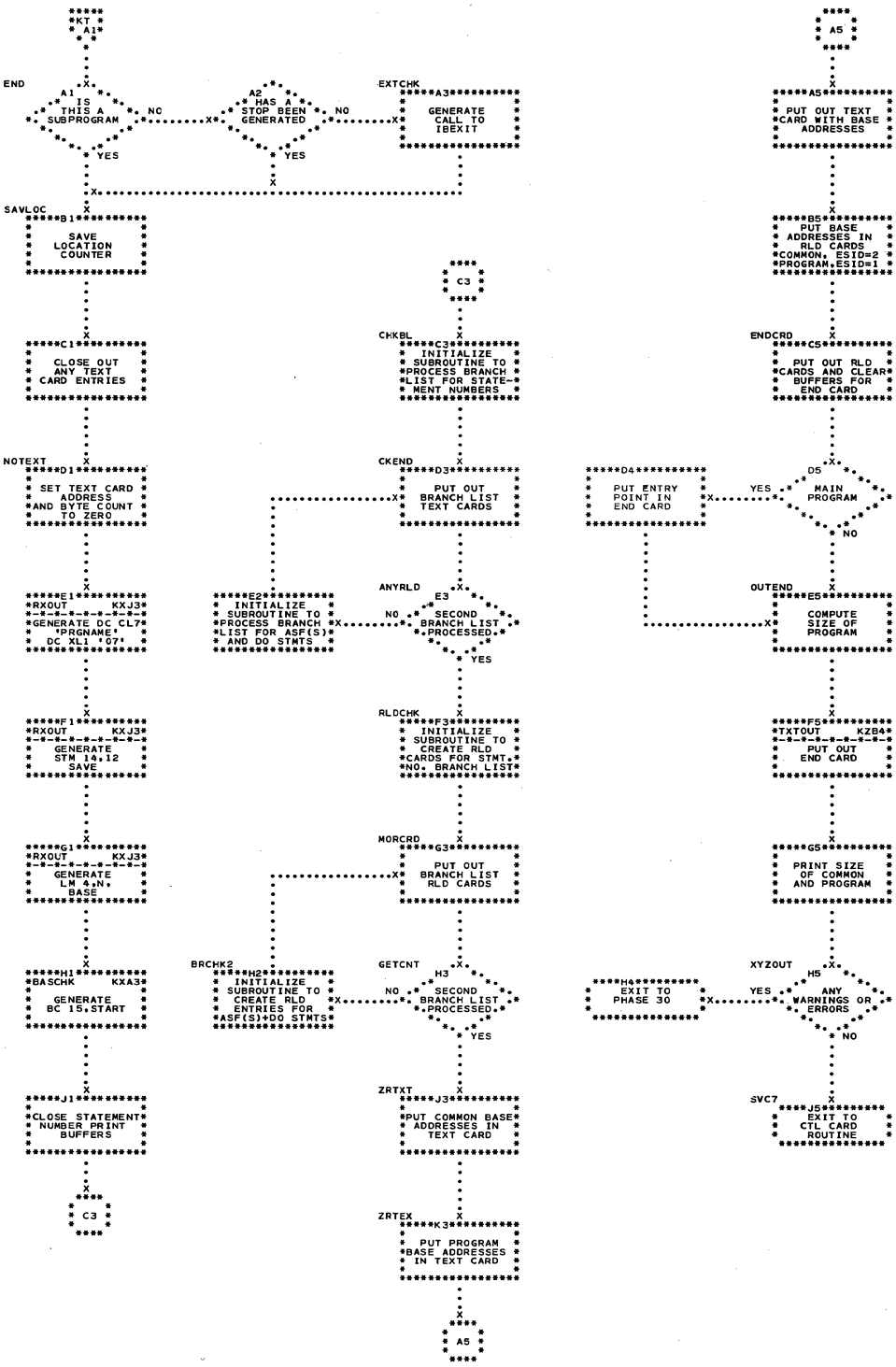


Chart KT. Subroutine END

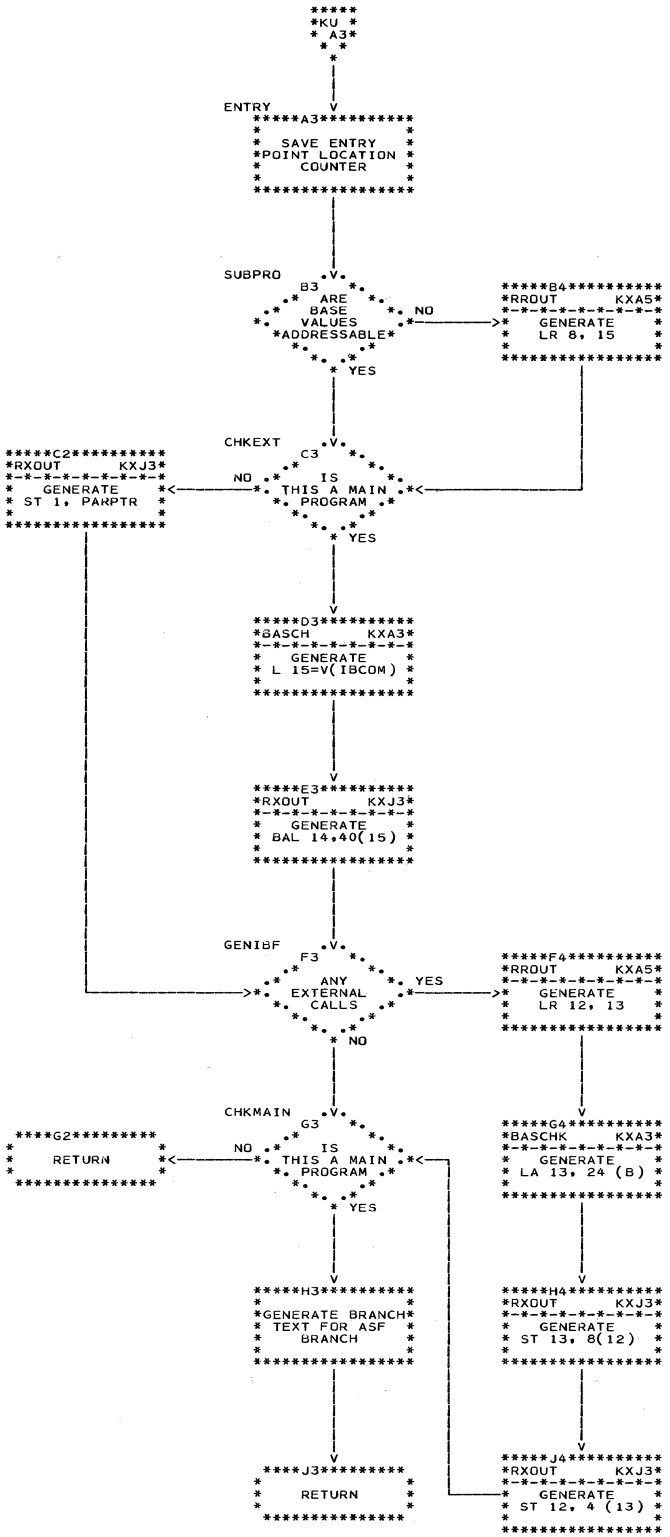


Chart KU. Entry Routine

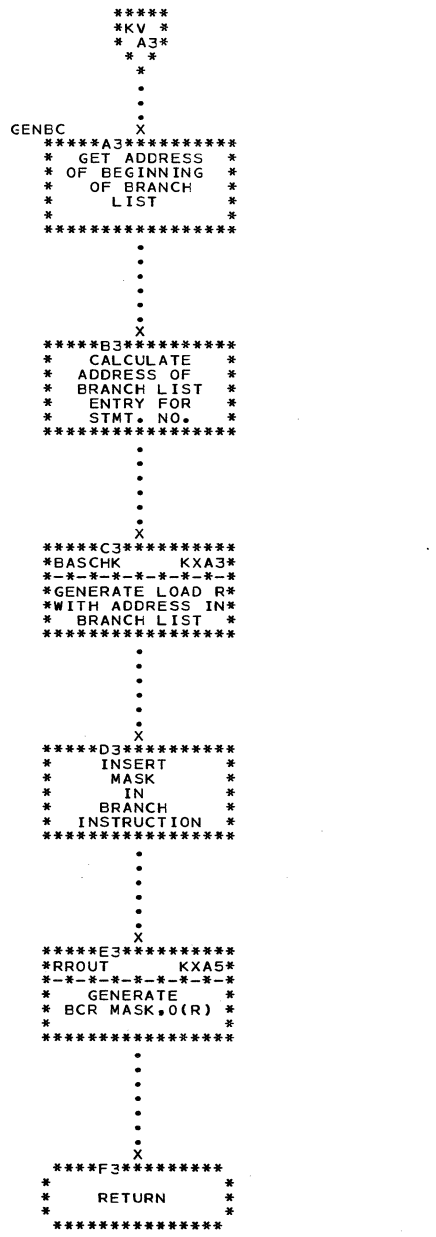


Chart KV. Subroutine GENBC

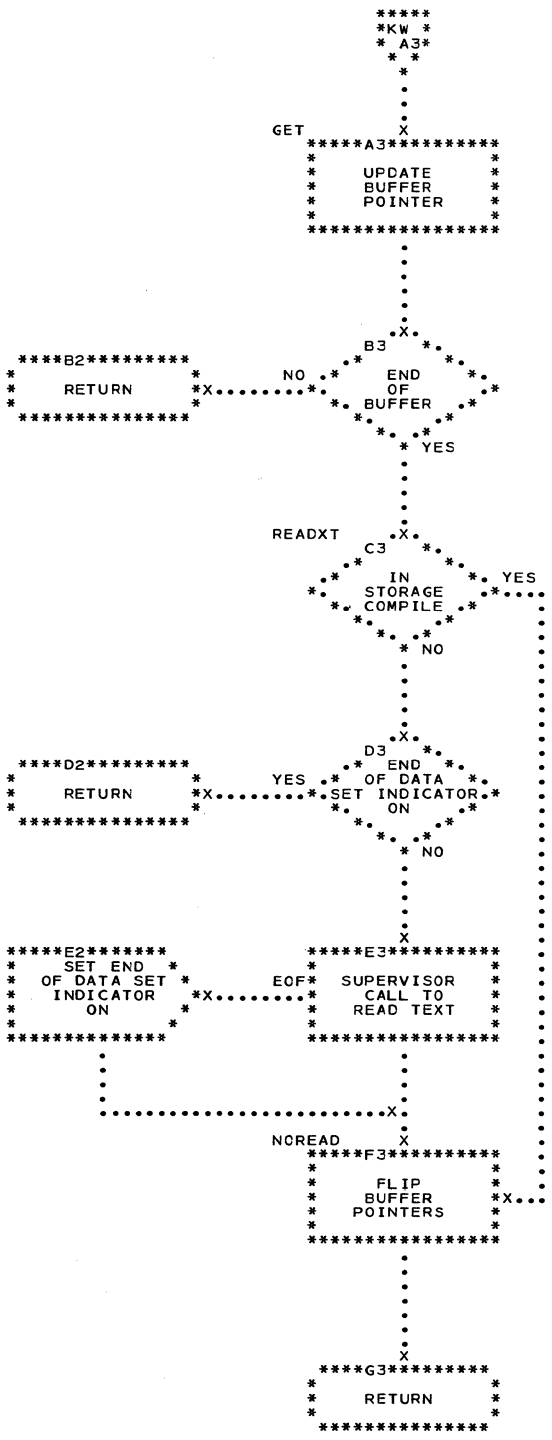


Chart KW. Subroutine GET

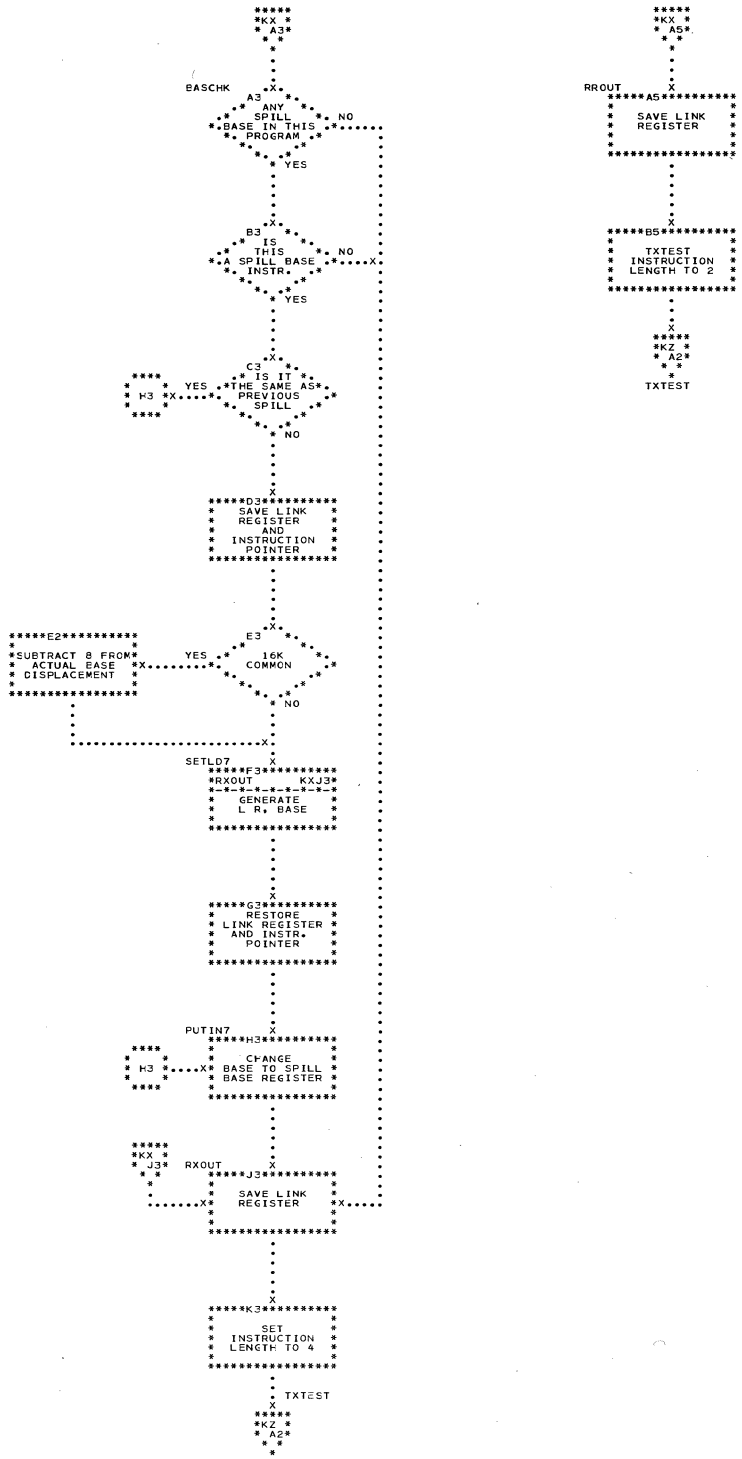


Chart KX. Subroutine BASCHK/RXOUT/RROUT

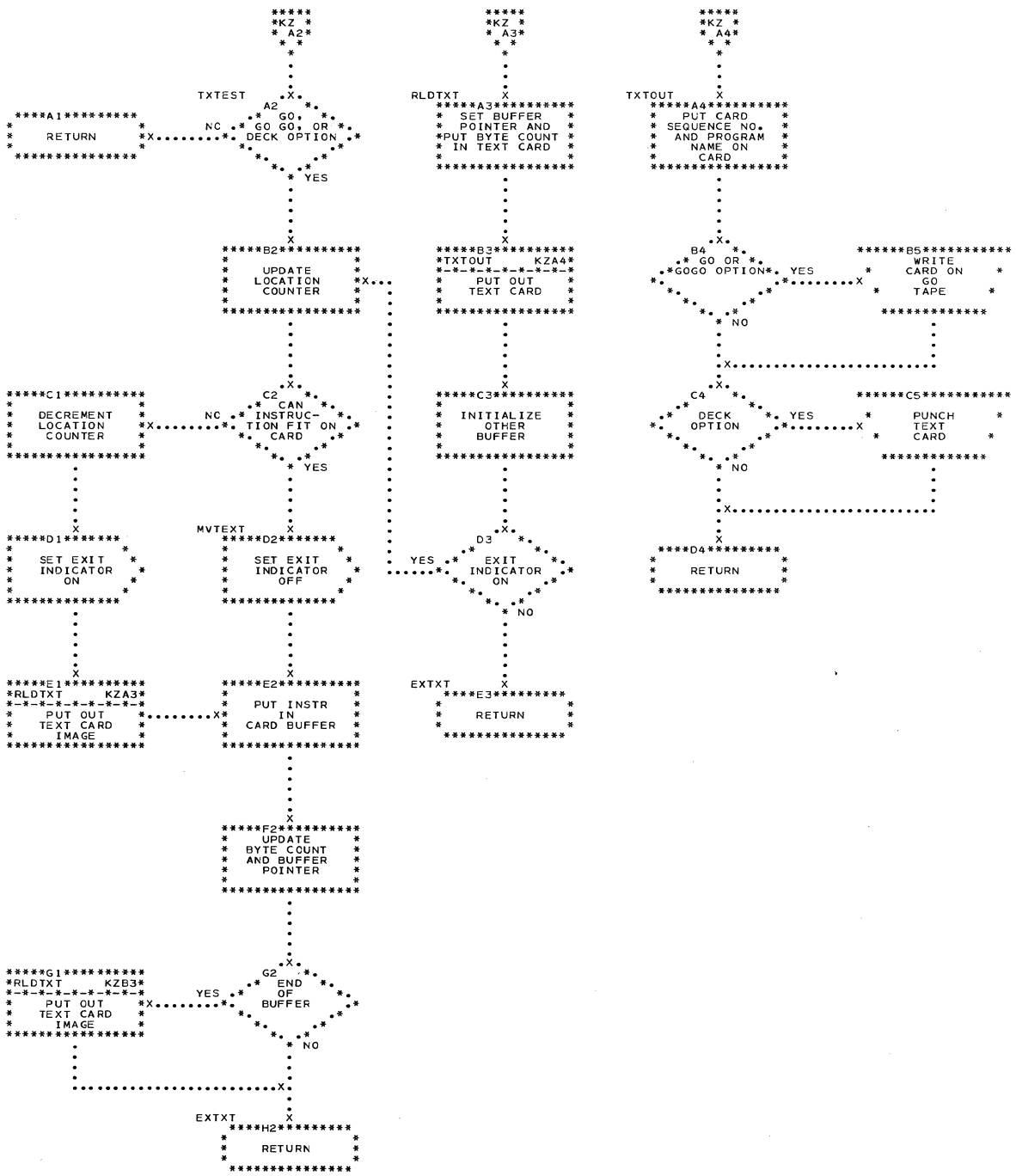


Chart KZ. Subroutines TXTEST, RLDTXT, TXTOUT

Phase 30 produces error and warning messages signalled by error/warning indicators set in the output text of any preceding phase.

Chart 09, the Phase 30 Overall Logic Diagram, indicates the entrance to and exit from Phase 30 and is a guide to the overall functions of the phase.

When an error condition is encountered during compilation, an error indicator is set in the communications area. Phase 20, at the completion of its processing, checks this indicator. When errors have been encountered in the preceding phases, Phase 20 determines if the GOGO option was specified by the user. If the GOGO option was not specified, entry is made into Phase 30; otherwise, Phase 20 exits to Phase 25.

When a warning condition is encountered during compilation, a warning indicator is set in the communications area. Phase 25, at the completion of its processing, determines if error or warning conditions have been encountered. When either condition is encountered, entry is made into Phase 30.

Phase 30 checks the adjective code of each text word for an error or warning condition. Phase 30 accesses the error/warning number (set up by the phase that found the error/warning condition) from the mode/type field of that text word. This number is compared to the numbers in the error/warning message table. (This table is illustrated in the System/360 Basic Programming Support FORTRAN Programmer's Guide, Diagnostic Messages.)

If that number is found in the table, Phase 30 prints the corresponding message. When a corresponding number is not found in the table, Phase 30 prints a message indicating a compiler error.

After the text is completely processed, Phase 30 returns control to the FORTRAN System Director to call in the Control Card routine.

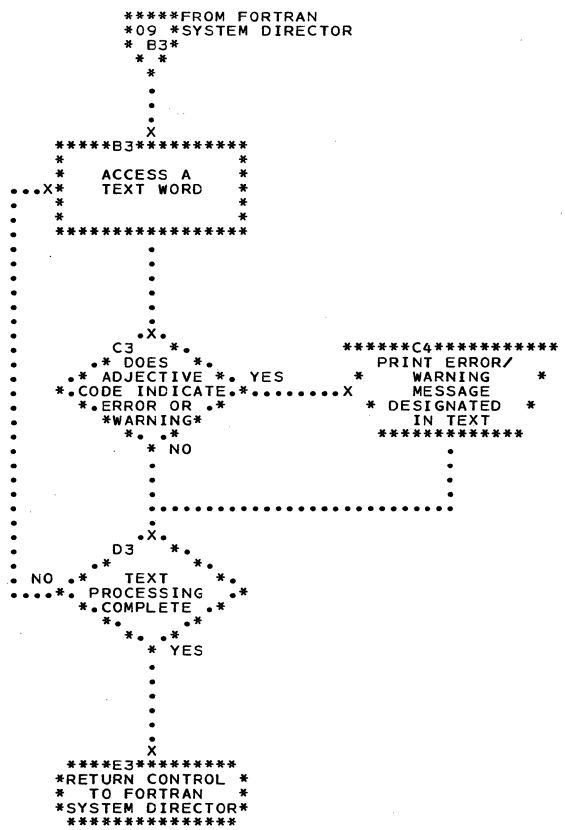


Chart 09. Phase 30 Overall Logic Diagram

An object program generated by the FORTRAN compiler is executed via four segments of the system: FORTRAN System Director (FSD), Control Card routine, FORTRAN loader, and IBCOM.

The FSD and Control Card routine are discussed in Part 2; the FORTRAN loader and IBCOM are to be discussed in this part of the manual.

The FORTRAN loader (BPS FORTRAN IV D Relocating Loader) places control segments into storage at locations other than those assigned by the compiler; that is, relocates them, completes the linkage between the various segments, and, at the end of the loading process, provides the FORTRAN System Director with the location to which control is to be transferred in the user

Chart 11, the Loader Overall Logic Diagram, indicates the entrance to and exit from the loader and is a guide to the overall functions of the loader.

LOADING PROCESS

The flow of operations of the FORTRAN loader can, in general, be defined as follows: the user program is read into storage, the library routines referenced by the user program are loaded into main storage, and relocation and linkage are effected.

More specifically, the loader reads the entire user program into main storage. The user program may consist of one or more control segments (a control segment is the output of a single compilation). The name, length, and starting address of a control segment are defined by an ESD Type 0 card; the end of the control segment is defined by an END card. (The segment length may also be indicated in the END card.) The end of the entire user program is defined by an LDT or DATA card. As the user program is read into storage, the control dictionary for segment relocation and linkage are built.

Once the program is in main storage, the loader begins to process the calls made by the program for library subprograms. Any reference to a library subprogram causes a search of the library on the system tape, provided the referenced subprogram is not already in main storage.

During the process of accessing and reading segments into storage, the loader writes on a work tape the information for relocation and linkage (contained in the ESD and RLD cards). When all library subprograms have been read in and the information for relocation and linkage is on the work tape, the loader rewinds the work tape.

The loader reads the work tape and carries out the relocation and linkage for the user program as instructed by the information on the loader cards.

When relocating control segments and establishing the linkage among them, the loader must calculate certain information during the loading process. The loader receives this information from loader cards encountered during the loading process and performs its calculations with a distinct routine for each type of load card.

Thus, for information on relocating, the loader analyzes and performs calculations on the Set Location Counter (SLC), Include Segment (ICS), External Symbol Dictionary (ESD), Text (TXT), and Replace (REP) cards. For information on linkage, the loader acts on External Symbol Dictionary (ESD types 0, 1, 2, and 5), Relocation List Dictionary (RLD), and Replace (REP) cards. The information for end-of-load transfer is provided by Load Terminate (LDT), Type/Data (/DATA), and Load End (END) cards.

CONTROL DICTIONARY ELEMENTS

The control dictionary consists of a reference table, an external symbol identification table (ESIDTB), and a location counter.

The reference table is a list of 12-byte entries built by the loader; it contains the names and entry points of a control segment, their present internal location, and the relocation factor. (The relocation factor is the difference between the compiler assigned address of a control segment and the address where the segment is actually loaded in storage.)

The ESID table contains pointers to the entries, in the reference table, for the current control segment.

The FORTRAN loader maintains its own location counter, which is used to determine where control segments are to be loaded. The counter is set to a constant value during the initial loading process and is subsequently incremented by the number of bytes indicated on an ESD Type 0 or END card. It may also be incremented by the length indicated on an ICS card or reset by an SLC card.

FORTRAN LOADER FUNCTIONS

As was pointed out, the FORTRAN loader must perform certain calculations on the information it receives from the loader cards. The routines that analyze and act upon the load cards are illustrated on Charts NC through NM.

The loader must also have routines to initialize itself and to read cards into storage: these routines are shown on charts NA and NB.

Apart from initializing itself, reading cards into storage, and acting upon the information contained in those cards, there are other necessary functions that must be provided for, such as handling error indications (Chart NS), providing a map of storage (Chart NT), obtaining or searching for frequently referenced data (Charts NO through NR), converting hexadecimal characters to binary (Chart NN), and routines to handle end of data set (EODS) indications.

CARD FORMATS

The relocating loader recognizes the card images specified under 'LOADING PROCESS.' The SLC, ICS, REP, LDT, and DATA cards are supplied by the user. The ESD, RLD, TXT, and END cards are generated by the compiler.

In the following paragraphs, the function of each card is stated briefly, the card format shown in tabular form, and each field of the card is explained.

SET LOCATION COUNTER CARD

The Set Location Counter (SLC) card is supplied by the programmer. The card sets the location counter of the loader to the address assigned to the name in the card. If there is also a hexadecimal address in the card, that address is added to the address assigned to the name in the card and the location counter is set to the resulting sum. The contents of the SLC card fields are explained in Figure 59.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	SLC. Identifies the type of load card.
5-6	Blank.
7-12	Address in hexadecimal (to be added to the value of the symbol in columns 17-22). The address must be right-justified in these columns, and unused leading columns filled in with zeros.
13-16	Blank.
17-22	Symbolic name, whose internal assigned location will be used by the loader. The symbol must be left-justified in these columns. If left blank, an error is indicated.
23-72	Blank.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 59. Set Location Counter (SLC) Card

INCLUDE SEGMENT CARD

The Include Segment (ICS) card is supplied by the programmer. The card defines a control segment by name and length and so enables one control segment to refer to another. The contents of the ICS card fields are explained in Figure 60.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	ICS. Identifies the type of load card.
5-16	Blank.
17-22	Name of segment, left-justified in these columns.
23-24	Blank.
25-28	Length (in bytes) in hexadecimal notation of the control segment. This must not be less than the actual length of the segment. The number must be right-justified in these columns, and unused leading columns filled in with zeros. Can be all zeros.
29-72	Blank.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 60. Include Segment (ICS) Card

EXTERNAL SYMBOL DICTIONARY TYPE 0 CARD

The External Symbol Dictionary (ESD) Type 0 card defines the name, starting address, and when necessary, the length of a control segment. There is only one ESD Type 0 card for each control segment. The contents of the ESD Type 0 card fields are explained in Figure 61.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	ESD. Identifies the type of load card.
5-10	Blank.
11-12	The number of bytes in the card. Extended card code 12-0-1-8-9 and 12-11-1-8-9 (hexadecimal value of 0010).
13-14	Blank.
15-16	External Symbol Identification (ESID). Number, in extended card code, assigned to the segment.
17-22	Program name.
23-24	Blank.
25	Extended card code 12-0-1-8-9 (hexadecimal value of 00), identifying this as a program name card.
26-28	Address, in extended card code, of the first byte of the segment as assigned by the compiler.
29	Blank.
30-32	Number, in extended card code, of bytes in the control segment. May be all zeros.
33-72	Blank.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 61. External Symbol Dictionary (ESD) Type 0 Card

EXTERNAL SYMBOL DICTIONARY TYPE 1 CARD

The External Symbol Dictionary (ESD) Type 1 card defines an entry point within the control segment to which another segment may refer. One card is produced for each entry point so defined.

The contents of the ESD Type 1 card fields are explained in Figure 62.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	ESD. Identifies the type of load card.
5-10	Blank.
11-12	The number of bytes in the card in extended card code. (decimal value of 16, 32; or 48 in hexadecimal)
13-16	Blank.
17-22	Name of entry point.
23-24	Blank.
25	Extended card code 12-1-9 (hexadecimal value of 01), identifying this as an entry point card.
26-28	Address, in extended card code, of the entry point as assigned by the compiler.
29-30	Blank.
31-32	External Symbol Identification (ESID). Number, in extended card code, assigned to control segment in which entry points occurred.
33-48	May be a repeat of columns 17-32.
49-64	May be a repeat of columns 33-48.
65-72	Blank.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 62. External Symbol Dictionary (ESD) Type 1 Card

EXTERNAL SYMBOL DICTIONARY (ESD) TYPE 2 CARD

The External Symbol Dictionary (ESD) Type 2 card points to a name within another control segment to which this control segment may refer. It is assigned an identifying number of 2 through 15, according to the order in which it is encountered by the compiler among the external symbols of the segment.

The contents of the ESD Type 2 card fields are explained in Figure 63.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	ESD. Identifies the type of load card.
5-10	Blank.
11-12	The number of bytes in the card in extended card code. (decimal value of 16, 32; or 48 in hexadecimal)
13-14	Blank.
15-16	External Symbol Identification (ESID). Sequential number, in extended card code, assigned to the first external symbol on this card.
17-22	Name of external symbol.
23-24	Blank.
25	Extended card code 12-2-9 (hexadecimal value of 02) identifying this as an external symbol card.
26-28	Extended card code 12-0-1-8-9, 12-0-1-8-9, and 12-0-1-8-9 (hexadecimal value of 000000). An address of 0 is always assigned to External Symbols by the compiler.
29-32	Blank.
33-48	May repeat columns 17-32 for a second entry.
49-64	For a third entry.
65-72	Blank.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 63. External Symbol Dictionary (ESD) Type 2 Card

EXTERNAL SYMBOL DICTIONARY TYPE 5 CARD

The External Symbol Dictionary (ESD) Type 5 card defines a given size for Blank COMMON.

The contents of the ESD Type 5 card fields are explained in Figure 64.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	ESD. Identifies the type of load card.
5-10	Blank.
11-12	The number of bytes in the card. Extended card code 12-0-1-8-9 and 12-11-1-8-9 (hexadecimal value of 0010).
13-14	Blank.
15-16	External Symbol Identification (ESID). Sequence number in extended card code, assigned to external symbol.
17-24	Blank.
25	Extended card code 12-5-9, identifying this as a Blank COMMON card.
26-28	Address of first byte of the program as assigned by the compiler. This field is always punched in extended card code with a hexadecimal value of 000000.
29	Blank.
30-32	Number, in extended card code, of bytes in BLANK COMMON.
33-72	Blank.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 64. External Symbol Dictionary (ESD) Type 5 Card

TEXT CARD

The Text (TXT) card contains the instructions and/or constants of the user program and the starting address at which the first byte of text is to be loaded.

The contents of the TXT card fields are explained in Figure 65.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	TXT. Identifies the type of load card.
5	Blank.
6-8	24-bit starting address (in extended card code) in storage where the information from the card is to be loaded.
9-10	Blank.
11-12	Number of bytes (in extended card code) of text to be loaded from the card.
13-14	Blank.
15-16	External Symbol Identification (ESID). Number, in extended card code, assigned to the control segment in which the text occurs.
17-72	A maximum of 56 bytes of instructions and/or constants assembled in extended card code.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 65. Text (TXT) Card

REPLACE CARD

The Replace (REP) card allows corrections and/or additions to be made to the user program at load time. The REP card is supplied by the programmer. It must be punched in hexadecimal.

If additions made by REP cards increase the length of a control segment, the pro-

grammer must place an ICS card (which defines the total length of the control segment) at the front of the control segment.

The constants of the REP card fields are explained in Figure 66.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	REP. Identifies the type of load card.
5-6	Blank.
7-12	Starting address, in hexadecimal, of the area to be replaced, as assigned by the compiler. It must be right-justified in these columns, and unused leading columns filled in with zeros.
13-14	Blank.
15-16	External Symbol Identification (ESID). Hexadecimal number assigned to the control segment in which replacement is to be made.
17-70	A maximum of 11 four-digit hexadecimal fields, separated by commas, each replacing one previously loaded half-word (two bytes). The last field must not be followed by a comma.
71-72	Blank.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 66. Replace (REP) Card

RELOCATION LIST DICTIONARY CARD

The Relocation List Dictionary (RLD) card is produced by the compiler when it encounters a DC instruction or the second operand of a CCW instruction, which defines an address as a relocatable symbol or expression. This may be the address of either an internal symbol, which occurs only within the control segment, or of an external symbol belonging to another control segment.

A control segment may contain more than one symbol or expression, definable in terms of one relocatable symbol. The RLD card lists addresses for as many as 13 expressions so defined. If there are more than 13 such expressions, other RLD cards associated with the symbol are produced.

The contents of the RLD card fields are explained in Figure 67.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	RLD. Identifies the type of load card.
5-10	Blank.
11-12	Number, in extended card code, of bytes of information in the variable field (card columns 17-72) of this card. The range is from 8 to a maximum of 56.
13-16	Blank.
17-72	Variable field (in extended card code). Consists of the following subfields: Relocation Header. (Two bytes.) An ESID with a value of from 01 through 15. Whether or not the value is 01 or from 02 through 15 depends on whether the symbol it points to is internal or external to the particular control segment. Position Header. (Two bytes.) The ESID assigned to this control segment. Flag Byte (bits 0 through 3 are not used). This byte contains three items: 1. Size. (Bits 4 and 5.) Two bits which indicate the length (in bytes) of the adjusted address cell (AA Cell). a. 00 - one-byte cell b. 01 - two-byte cell c. 10 - three-byte cell d. 11 - four-byte cell

Figure 67. Relocation List Dictionary (RLD) Card (continued)

Column	Contents
	2. Complement Flag. (Bit 6.) When this bit is a one, it means that the value (or address) of the symbol is to be subtracted from the contents of the AA Cell. When this bit is a zero, the value of the symbol is to be added to the contents of the AA Cell. 3. Continuation Flag. (Bit 7.) When this bit is a one, it means that this is one of a series of addresses to be adjusted. When this bit is a zero, this is the only AA Cell to be adjusted or the last in a series using the same Relocation and Position headers.
	Address. The three-byte address of the location of the AA Cell. The Flag Byte and Address may be repeated for AA Cells as long as the continuation flag bit is on in the current four-byte entry.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 67. Relocation List Dictionary (RLD) Card

LOAD END CARD

The Load End (END) card is produced by the compiler when it encounters the END statement instruction. This card ends the loading of a control segment and may specify a location within the control segment to which control is to be transferred at end-of-load.

The contents of the END card fields are explained in Figure 68.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	END. Identifies the type of load card.
5	Blank.
6-8	Address (may be blank), in extended card code, of the point in the control segment to which control may be transferred at the end of the loading process. See the conditions and priority discussed under Load Terminate card.
9-14	Blank.
15-16	External Symbol Identification (ESID). The number 0002, in extended card code is assigned to the program.
17-29	Blank.
30-32	Number, in extended card code, of bytes in the program.
33-72	Blank.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 68. Load End (END) Card

LOAD TERMINATE AND DATA CARDS

The Load Terminate (LDT) and Type/Data (DATA) cards which are provided by the programmer, have the same functions: they terminate the loading process and may provide the address, within one of the loaded control segments, to which control should be transferred.

The specific location to which control will be transferred is determined through the following order of priority:

1. Control is always transferred to a location specified on an LDT or DATA card.

2. If the LDT or DATA card does not specify a location, control is transferred to the location specified by the first END card containing an address encountered during the current loading process.

3. If neither the LDT, DATA, nor END card specifies a location, control is transferred to the first location into which the contents of a TXT card are loaded (or of a REP card, if one precedes the text cards).

The contents of the LDT and DATA card fields are explained in Figures 69 and 70, respectively.

Column	Contents
1	Load card identification (12-2-9). Identifies this as a card acceptable to the loader.
2-4	LDT. Identifies the type of load card.
5-16	Blank.
17-22	Name of entry point to the program segment, left-justified in these columns. Use of this field is optional.
23-72	Blank.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 69. Load Terminate (LDT) Card

Column	Contents
1	Load card identification (0-1). Identifies this as a card acceptable to the loader.
2-5	DATA. Identifies the type of load card.
6-16	Blank.
17-22	Name of entry point to the program segment, left-justified in these columns. Use of this field is optional.
23-72	Blank.
73-80	Not used by the loader. The programmer may leave blank or punch in program identification for his own convenience.

Figure 70. Type/Data (DATA) Card

IER Routine: Chart NA

The IER routine performs those initialization steps that enable the loader to begin the loading process.

ENTRANCE: The IER routine receives control from the FORTRAN System Director (FSD).

OPERATION: The IER routine sets an indicator in the communications area to notify FSD that the loader is in control and records the start time. IER obtains the beginning load address, the size of storage, and the logical input device from the communications area.

IER uses the highest available storage address to determine the starting address of Blank Common and sets the location counter to the begin-load address.

After setting up the reference table counter, loading the entry point for IBCOM into the reference table, and writing the storage map heading, the IER routine sets up: to read either cards or tape, rewind the work tape, and transfer control to the RD routine.

EXIT: The IER routine exits to the RD routine.

RD Routine: Chart NB

The RD routine reads one card image at a time into storage.

ENTRANCE: The RD routine is entered from the following routines: IER, CMPICS, CMPSLC, CMPESD, CESD0, CESD1, CESD2, CMPTXT, CMPRLD, CMPREP, CMPEND, MAP, and EODS.

OPERATION: The RD routine, after setting up to read, loads the input buffer address and the number of bytes to be read, and issues a read SVC. If the end of the data set has been reached, control is transferred to the EODS routine; otherwise, a set of constants, used throughout the loader, are loaded.

A series of tests, based on a set of switches, are performed; switches 1, 3, and 6 indicate, respectively: if the end of the control segment has been reached on the system tape; if the system tape is being searched for a given control segment; and if the control segment is needed from the system tape. When the system tape is not being read, switches 1, 3, and 6 are always off. If the end of the control segment has not been reached on the system tape (switch

1 off) and if the system tape is not being searched (switch 3 off), control is transferred to the CMPTXT routine.

However, when a search of the reference table indicates that all control segments have not yet been entered into storage, switch 3 is turned on, initiating a search for ESD Type 0 cards. When an ESD Type 0 card is found, the name in the card is compared to the entries in the reference table to determine if the segment is needed. If it is needed, switches 1, 3, and 6 are turned off and control is transferred to the CMPESD routine. If the segment is not needed, the name is placed in the reference table with a not-found indication, a 1 is added to the reference table count, switch 1 is turned on, and another card image is read.

If the card is an ESD card, other than ESD Type 0, a test is made to determine if the card is an ESD Type 1 card. When an ESD Type 1 card is encountered, the reference table is searched for the entry point name to further determine if this control segment is needed. If the entry point name is not in the reference table or if the card is not an ESD Type 1 card, another card is read. The name could be in the REFTBL and not needed.

If the card is an ESD Type 1 card and the search of the reference table indicates that the segment is needed, switch 6 is turned on, and another card is read.

When switch 1 is on and an END card is encountered, indicating the end of a library subprogram, a test is made of switch 6. Switch 6, being on, indicates that this control segment is needed, as requested through one of its entry points. Because this segment is already bypassed on the tape, another pass through the system library is needed. For this process, switches 1 and 6 are turned off. The ESD Type 0 name, that was placed in the reference table with a not-found indication, becomes a permanent entry in the table.

If switch 6 is off and an END card is encountered, switch 1 is turned off, indicating that the control segment was not needed and 1 is subtracted from the reference table count, effectively deleting the name contained in the ESD Type 0 card from the reference table.

EXITS: The RD routine exits to either the CMPTXT, EODS, or CMPESD routine, depending on the conditions discussed under "Operation."

ROUTINE CALLED: During execution, the RD routine references the EODS routine.

CMPSLC Routine: Chart NC

The CMPSLC routine sets the location counter to the address assigned to the symbolic name in the SLC card; if there is also a hexadecimal address in the card, that address is added to the address assigned to the symbolic name, and the location counter is set to the resulting sum.

ENTRANCE: The CMPSLC routine is entered from the CMPREP routine when that routine encounters a card that is not a REP card.

OPERATION: The CMPSLC routine determines if the card to be processed is an SLC card; if not, the routine transfers control to the CMPICS routine.

When an SLC card is encountered, a check is made for a name in the card. The absence of a name is an error. If the card has a name, a search for that name is made in the reference table. If the name is not in the table, an error exists.

When the table contains the name, the routine obtains the address currently assigned to the name and sets the location counter to this value.

If there is also a hexadecimal address in the card, however, that address is converted to binary and added to the address obtained for the name. The location counter is set to the resulting sum.

EXIT: This routine exits to the RD routine.

ROUTINES CALLED: During execution the CMPSLC routine references the SERCH and HEXB routines.

CMPICS Routine: Chart ND

The CMPICS routine establishes reference table entries for the control segment name on the ICS card.

ENTRANCE: The CMPICS routine is entered from the CMPSLC routine whenever that routine encounters a card that is not an SLC card.

OPERATION: When the card to be processed is not an ICS card, control is transferred to the CMPESD routine. When the CMPICS routine encounters an ICS card, it sets up for a possible error message and resets the card count counter.

After converting the indicated segment length to binary, the routine searches for the name in the reference table and exits to the READ routine if the name is found. If the name is not found, the CMPICS routine places the name into the reference table, adjusts the location counter to the next double-word boundary (if necessary), and assigns the current value of the location counter to the name.

The CMPICS routine then adds the segment length to the current value of the location counter and saves the highest address assigned to text.

EXIT: This routine exits to the RD routine.

ROUTINES CALLED: During execution, the CMPICS routine references the TBLREF, SERCH, and HEXB routines.

CMPESD Routine: Chart NE

The CMPESD routine determines if the card to be processed is an ESD card. If it is an ESD Type 5 card, it signifies Blank Common.

ENTRANCE: The CMPESD routine is entered from the Library Primary Search routine when that routine encounters an ESD Type 0 card for a segment that is needed; it is also entered from the CMPICS routine when that routine encounters a card that is not an ICS card.

OPERATION: If the card to be processed is not an ESD card, the CMPESD routine transfers control to the CMPRLD routine. If it is an ESD card and the work tape is not being read, the card image is written on the work tape for later processing. If it is an ESD card, but not Type 5, control is transferred to the CESD0 routine.

When the CMPESD routine encounters an ESD Type 5 card, it determines if the work tape is being read. Reading of the work tape causes the routine to determine if Blank Common has been established. If so, a test is made to determine if Blank Common overlays the text placed in storage. If it is, an error condition exists. Control is transferred to the RD routine if Blank Common is not overlaying text.

If the work tape is not being read, the size of Blank Common is subtracted from the size of storage. If the result is less than the saved Blank Common address, the CMPESD routine saves the size found in the ESD Type 5 card. Whether or not the routine saves the address, it exits to the RD routine.

EXIT: The CMPESD routine exits to the RD routine.

CESD0 Routine: Chart NF

The CESD0 routine determines the type of ESD card to be processed and makes reference table and ESID table entries for the control segment specified in the ESD Type 0 card.

ENTRANCE: The CESD0 routine is entered from the CMPESD routine when that routine encounters an ESD card that is not an ESD Type 5 card.

OPERATION: The CESD0 routine determines the type of ESD card to be processed. It transfers control to the appropriate routine for an ESD Type 1 or Type 2 card, retains control for an ESD Type 0 card; and exits to the ERROR routine for an ESD Type 3 or 4 card.

This routine resets the card count counter for possible error messages and maps the segment name and its assigned location.

The length of the control segment, as indicated on the ESD Type 0 card, is loaded and the reference table pointer to the control segment is placed in the ESID table.

The CESD0 routine calculates the relocation factor by subtracting the compiled address of the control segment from its current, relocated address. The routine places the relocation factor in the reference table.

EXIT: The CESD0 routine exits to the RD routine.

ROUTINES CALLED: During execution the CESD0 routine references the CMPICS and MAP routines.

CESD1 Routine: Chart NG

The CESD1 routine establishes a reference table entry for the entry point specified in the ESD Type 1 card, unless such an entry already exists.

ENTRANCE: This routine is entered from the CESD0 routine when that routine encounters an ESD Type 1 card.

OPERATION: The CESD1 routine obtains the relocation factor for the specified control segment through the external symbol iden-

tification (ESID) entry on the card. The address on the card is added to the relocation factor and the resulting sum is saved as the new, relocated address of the name.

The routine then searches for the name in the reference table; if it is not in the table, the name of the entry point and its relocated address are placed in the reference table and a map of the entry provided. If the name is in the table, the routine indicates that it was found and compares the new, relocated address to the address assigned to the name in the reference table; if they are not identical, an error exists.

EXIT: This routine exits to the RD routine.

ROUTINES CALLED: During execution, this routine references the MAP, TBLREF, REFTBL, LODREF, and SERCH routines.

CESD2 Routine: Chart NH

The CESD2 routine places the reference table pointer of the external name indicated on the ESD Type 2 card into the ESID table. The routine places the address assigned to the external name into the reference table as the relocation factor for the name.

ENTRANCE: The CESD2 routine is entered from the CESD0 routine when that routine encounters an ESD Type 2 card.

OPERATION: The routine searches for the name (indicated on the card) in the reference table. If the name is not found in the table, the reference table pointer to the name is placed in the ESID table. The ESID table is adjusted for the next entry, and control is transferred to location RDD to determine if more entries are in the card.

If the name is found in the reference table, a test is made to determine if the work tape is being read. If the tape is not being read, the CESD2 routine transfers control to the RD routine. If the work tape is being read, the address assigned to the external name is loaded into the reference table as the relocation factor of that name.

If further processing of the card is necessary, control is transferred to the CMPESD routine; otherwise, the CESD2 routine exits to the RD routine.

EXITS: The CESD2 routine exits to the RD routine.

ROUTINE CALLED: During execution, the CESD2 routine references the SERCH routine.

CMPTXT Routine: Chart NI

The CMPTXT routine makes address validity checks and places the contents of a TXT card into storage.

ENTRANCE: The CMPTXT routine is entered from the Library Primary Search routine; it is also entered from the CMPREP routine for address validity checks.

OPERATION: If the card to be processed is not a TXT card, this routine transfers control to the CMPREP routine; otherwise, the routine obtains the relocation factor for this entry from the reference table and adds it to the address on the card. If the result of this addition indicates that the relocated address is within the area of the loader or that it exceeds the size of storage, an error condition exists.

If the address is valid, the CMPTXT routine loads the text into storage, unless entry into this routine is from the CMPREP routine. In the latter case, control is returned to the CMPREP routine.

EXITS: The CMPTXT routine exits to the READ or CMPREP routines.

ROUTINES CALLED: During execution the CMPTXT routine references the TBLREF, REFTBL, and LODREF routines.

CMPREP Routine: Chart NJ

The CMPREP routine places corrections to text into storage.

ENTRANCE: The CMPREP routine is entered from the CMPTXT routine when that routine encounters a card that is not a TXT card.

OPERATION: The encounter of any card type, other than a REP card, causes the CMPREP routine to transfer control to the CMPSLC routine.

When a REP card is encountered, the routine converts the address and ESID on the card to binary. The routine branches to CMPTXT routine to determine the validity of that address.

The routine converts the correction on the card to binary and places the correction into storage. If there is another entry on the REP card, it adjusts the

address for the next two bytes and branches back to that part of the routine that checks address validity.

EXITS: The CMPREP routine exits to the RD routine.

ROUTINES CALLED: During execution the CMPREP routine references the CMPTXT and HEXB routines.

CMPRLD Routine: Chart NK

The CMPRLD routine processes RLD cards, which are produced by the compiler when it encounters address constants within the program being compiled. The CMPRLD routine places the relocated storage address of a given symbol or expression into the address indicated by the compiler. The routine must calculate the proper value of a given symbol or expression and the proper address for adjustment of that value.

ENTRANCE: The CMPRLD routine is entered from the CMPESD routine when that routine encounters a card that is not an ESD card.

OPERATION: If the card to be processed is not an RLD card, control is transferred to the CMPEND routine. If it is an RLD card, but has not been written on the work tape, the CMPRLD routine writes it on the work tape and transfers control to the RD routine.

An RLD card that has already been written on the work tape will be tested to determine if the external ESID (Relocation Header ESID) is valid; if it is not valid, an error condition exists. If the ESID is valid, it is used to obtain the relocated address of the symbol referred to by the RLD card. (This address is found in the relocation factor position of the proper reference table entry.)

The internal ESID (Position Header ESID) is then tested to determine if it is valid; if it is not, an error condition exists. A valid internal ESID is used to obtain the relocation factor of the control segment in which the "Define Constant" instruction occurred.

The CMPRLD routine decrements the card-specified byte count by four; if the result is zero, the routine transfers control to the RD routine. Otherwise, the routine obtains the length, in bytes, of the symbol referred to in the RLD card and sets up to place the specified address value in storage at the specified address.

The relocation factor of the control segment in which the current address of the symbol must be stored is added to the card-specified address. The sum is the current address of the location at which the symbol address must be stored. The symbol address is then calculated and placed at the indicated address.

If there are no more entries on the RLD card, the CMPRLD routine exits; otherwise, a test is made of the continuation flag. If the flag is on, the CMPRLD routine branches back to process data for a new symbol (i.e., to that part of the CMPRLD routine that tests for a valid external ESID). If the flag is off, the routine returns to process data for the same symbol (i.e., to that part of the CMPRLD routine that determines the length, in bytes, of the symbol being processed).

EXIT: This routine exits to the RD routine.

ROUTINES CALLED: During execution, the CMPRLD routine references the REFTBL, TBLREF, and LODREF routines.

CMPEND Routine: Chart NL

The CMPEND routine saves the address in the first END card encountered as a possible end-of-load transfer address.

ENTRANCE: The CMPEND routine is entered from the CMPRLD routine when that routine encounters a card that is not an RLD card.

OPERATION: If the card to be processed is not an END card, the CMPEND routine transfers control to the CMPLDT routine. A test is made to determine if there is an end address in the card. If there is, and an end address has not already been saved from a previously processed END card, the relocation factor for this control segment is added to the address in the card. The resulting value is saved for a possible end-of-load transfer address. If there is an end address in the card, but one has already been saved from another END card, or if there is not an end address in the card, a test is made to determine if there is a segment size indicated in the END card. When a segment size is present, it is added to the current value of the location counter. The result is saved for the Blank Common overlay test, only if the result is greater than the highest address assigned to text.

If there is no segment size in the card, or after the result of adding the segment size to the location counter has been

saved, or after the test and processing of an end address, the ESID table is cleared. A test is made to determine if the work tape is being read; if it is, control is transferred to the EODS routine. If the work tape is not being read, the CMPEND routine exits to the RD routine.

EXITS: The CMPEND routine exits to the RD routine.

ROUTINES CALLED: During execution the CMPEND routine references the TBLREF, REFTBL, and LODREF routines.

CMPLDT, WARN Routines: Chart NM

CMPLDT Routine

The CMPLDT routine saves the address of the name on an LDT or DATA card for end-of-load transfer.

ENTRANCE: The CMPLDT routine is entered from the CMPEND routine when that routine encounters a card that is not an END card.

OPERATION: If the CMPLDT routine encounters a card that is not an LDT or DATA card, a warning message is issued (only if a GOGO job is indicated, otherwise an error is indicated) to indicate that the card is not recognized by the loader. For the LDT or DATA card, the routine determines if the card has a name and, if so, saves the address of that name. If there is no name in the card, or after the address name has been found and saved, the routine clears the ESID table and transfers control to the EODS routine.

EXIT: The CMPLDT routine exits to the EODS routine.

ROUTINES CALLED: During execution the CMPLDT routine references the SERCH and TBLREF routines.

WARN Routine

The WARN routine writes a warning message when the loader encounters a card that it does not recognize; it also determines whether to continue processing.

ENTRANCE: The WARN routine is entered from the CMPLDT routine when that routine encounters a card that is not recognized by the loader.

OPERATION: The routine converts the card count number accumulated since the last ESD Type 0 or ICS card was read to hexadecimal, places the converted address into the warning message with the name of the last control segment, and writes the warning message.

The WARN routine determines whether the current load is a GOGO load (i.e., a load that is to proceed regardless of this error). A GOGO load causes a transfer of control to the RD routine. For other than a GOGO load, the routine issues a Terminate Load SVC to the FORTRAN System Director.

EXITS: The WARN routine issues an SVC to the FORTRAN System Director, unless a GOGO load is indicated, then the WARN routine exits to the RD routine.

HEXB Routine: Chart NN

The HEXB routine converts hexadecimal numbers to binary.

ENTRANCE: The HEXB routine is entered from the CMPREP, CMPSLC, and CMPICS routines.

OPERATION: The HEXB routine determines if the character to be converted is either a valid alphabetic or numeric character; if it is not, an error exists.

Valid numeric characters are converted by clearing the four high-order bits of the character; alphabetic characters are converted by subtracting a constant from the character.

After the character has been converted, the HEXB routine shifts the general register (in which the entire converted number is returned) four bits to the left, and places the converted character in the four low-order bits.

If there is another character to be converted, the process is repeated.

EXITS: The HEXB routine exits to the routine that called it.

TBLREF Routine: Chart NO

The TBLREF routine loads three standard constants.

ENTRANCE: The TBLREF routine is entered from the following routines: IER, EODS, RD, CMPTXT, CMPSLC, CMPICS, CESD1, CESD2, CMPRLD, CMPEND, and CMPLDT.

OPERATION: The TBLREF routine loads the following constants into three sequential registers: the address of the first location above the reference table, the size of the reference table entries (12 bytes each), and the number of entries in the reference table.

EXITS: The TBLREF routine exits to the routine that called it.

REFTBL Routine: Chart NP

The REFTBL routine calculates the storage address of a given entry in the reference table.

ENTRANCE: REFTBL is entered from the CESD1, CMPRLD, and CMPEND routines.

OPERATION: The REFTBL routine loads the entry position in the reference table from the ESID table, by using the ESID on the card currently being processed. It multiplies the entry position by the reference table entry size (12), and subtracts the product from the highest address (plus 1) of the reference table. The result is the location of the reference table entry.

EXITS: The REFTBL routine exits to the routine that called it.

LODREF Routine: Chart NQ

The LODREF routine obtains the relocation factor of a control segment and returns it to the calling routine.

ENTRANCE: LODREF is entered from the CMPTXT, CESD1, CMPRLD, and CMPEND routines.

OPERATION: The LODREF routine uses the address obtained from the REFTBL routine as an index to the reference table. It places the relocation factor contained in the indicated reference table entry into a general register.

EXITS: The LODREF routine exits to the routine that called it.

SERCH Routine: Chart NR

The SERCH routine searches for a given name in the reference table.

ENTRANCE: The SERCH routine is entered from the CMPSLC, CMPICS, CESD1, CESD2, and CMPLDT routines.

OPERATION: After clearing the specified register and loading the address of the name to be searched for, the SERCH routine sets up for a possible mapping of this entry.

The routine compares the given name with each name in the reference table. When found, the SERCH returns with the position at which the name is located in the reference table. If the name is not found, the SERCH routine places the name in the table, increments the reference table count by one, and checks for loader overlay. If the loader has been overlaid, there will be an error.

EXITS: This routine exits to the routine that called it.

ERROR Routine: Chart NS

The ERROR routine writes an error message for the routine in which the error has occurred.

ENTRANCE: The ERROR routine is entered from the CMPTXT, XMPSLC, SERCH, CMPICS, CMPESD, CESD0, CESD1, CMPRLD, CMPLDT, WARN and HEXB routines.

OPERATION: The ERROR routine loads the address of the error message (determined by the routine in which the error occurred) and writes that message.

EXIT: The ERROR routine exits to the FORTRAN System Director.

MAP Routine: Chart NT

The MAP routine provides a map of storage for a given entry.

ENTRANCE: The MAP routine is entered from the IER, RELCTL, CESD0, and CESD1 routines.

OPERATION: The MAP routine unpacks a given address, converts the binary address to hexadecimal, loads the mapping address, and writes the map for the given entry (e.g., IBCOM at location 000FA0).

EXITS: The MAP routine exits to the calling routine.

RELCTL Routine: Chart NU

The RELCTL routine performs all the final functions, preparatory to releasing control from the loader.

ENTRANCE: This routine is entered from the loader when the end of data set is encountered on the work tape.

OPERATION: The RELCTL routine spaces a data set on the system tape, maps Blank Common if called for, records the time from the timer, and loads the end-of-load transfer address into a general register.

EXIT: The RELCTL routine exits to FSD.

ROUTINE CALLED: During execution the RELCTL routine references the MAP routine.

EODS Routine: Chart NV

On EODS indications, this routine determines where the loader is in the loading process and, depending on the results of the tests made within the routine, either continues processing or releases control.

ENTRANCE: This routine is entered on EODS indications.

OPERATIONS: If EODS has been reached on the work tape (switch 4 on), control is transferred to the RELCTL routine; otherwise, a test is made to determine if the system tape is being read (switch 2 on). If the system tape has not been read, the EODS routine sets up to read the system tape and turns switch 2 on. If the system tape is being read, the library data set is backspaced to its beginning. In either case, processing continues to determine if all necessary library routines have been loaded.

If all entries have not been loaded, a retry will be made. For this action, switch 3 is turned on; switches 1, 6, and 8 are turned off. If all entries have been loaded, a tape mark is written on the work tape and the tape is rewound.

The EODS routine sets up to read the work tape, turns switches 1 and 3 off, and turns switch 4 on. The routine then transfers control to the RD routine.

EXITS: If further processing is necessary, the EODS routine exits to the RD routine: otherwise, it exits to the RELCTL routine.

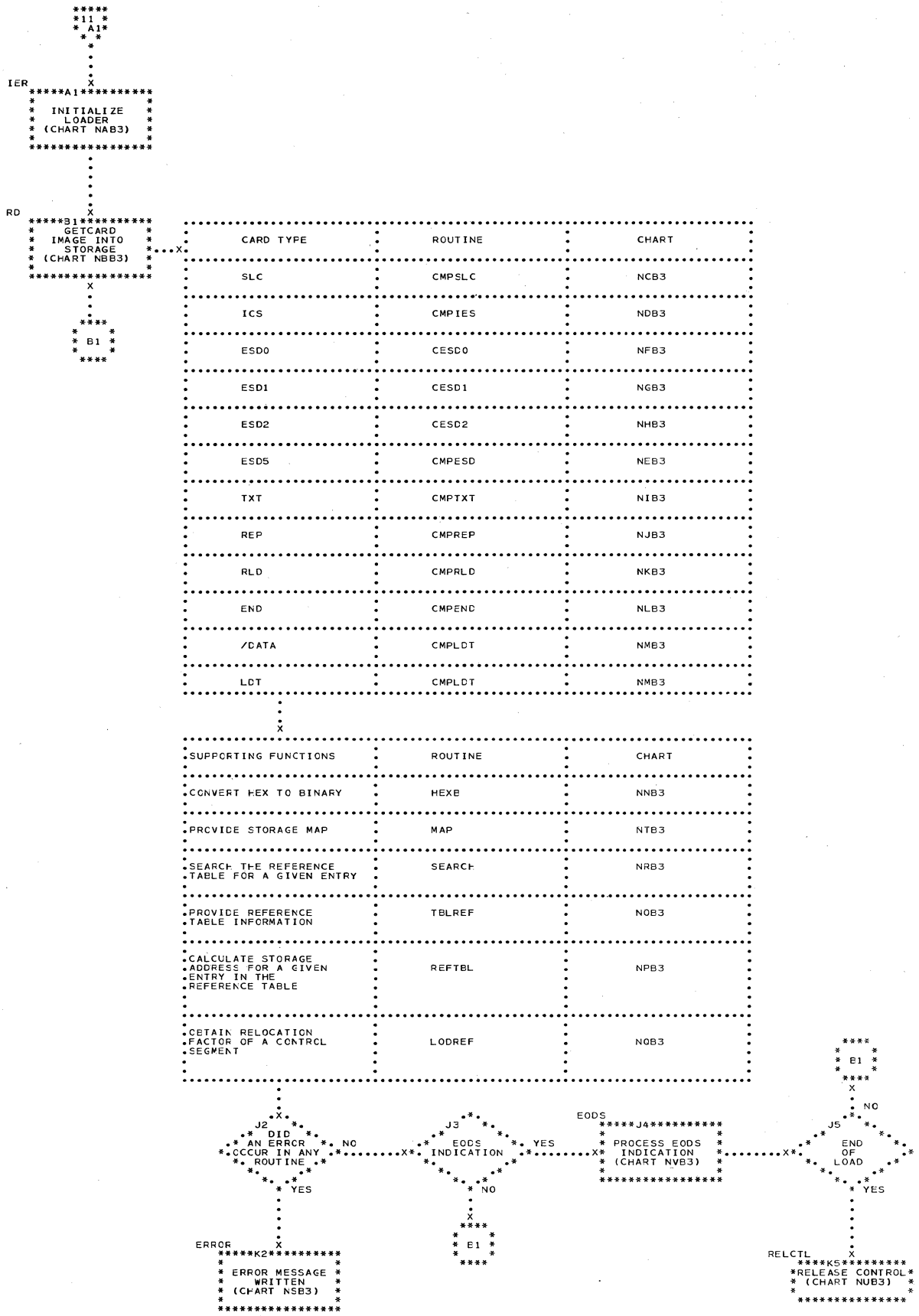


Chart 11. Relocating Loader Overall Logic Diagram

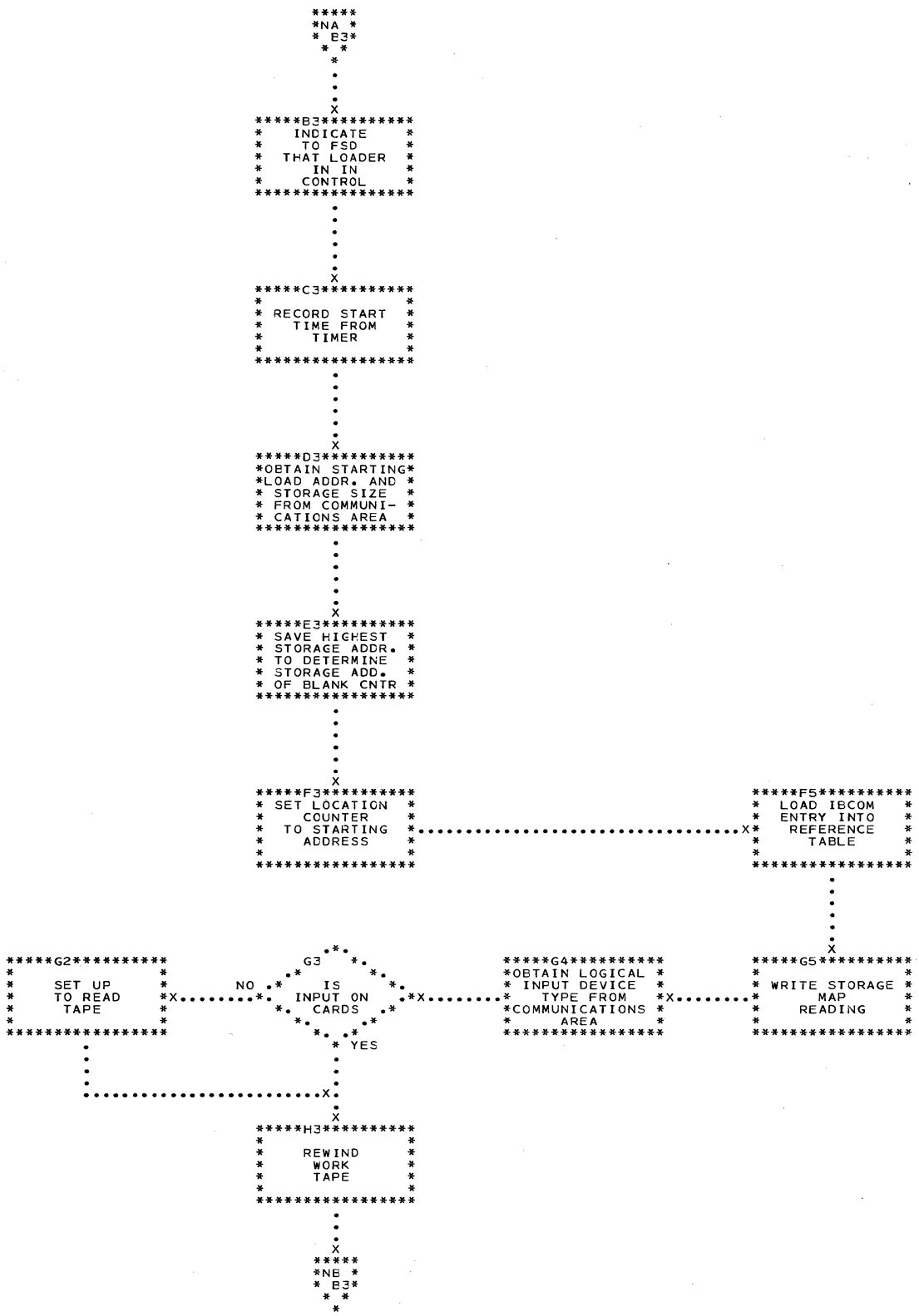


Chart NA. IER Routine

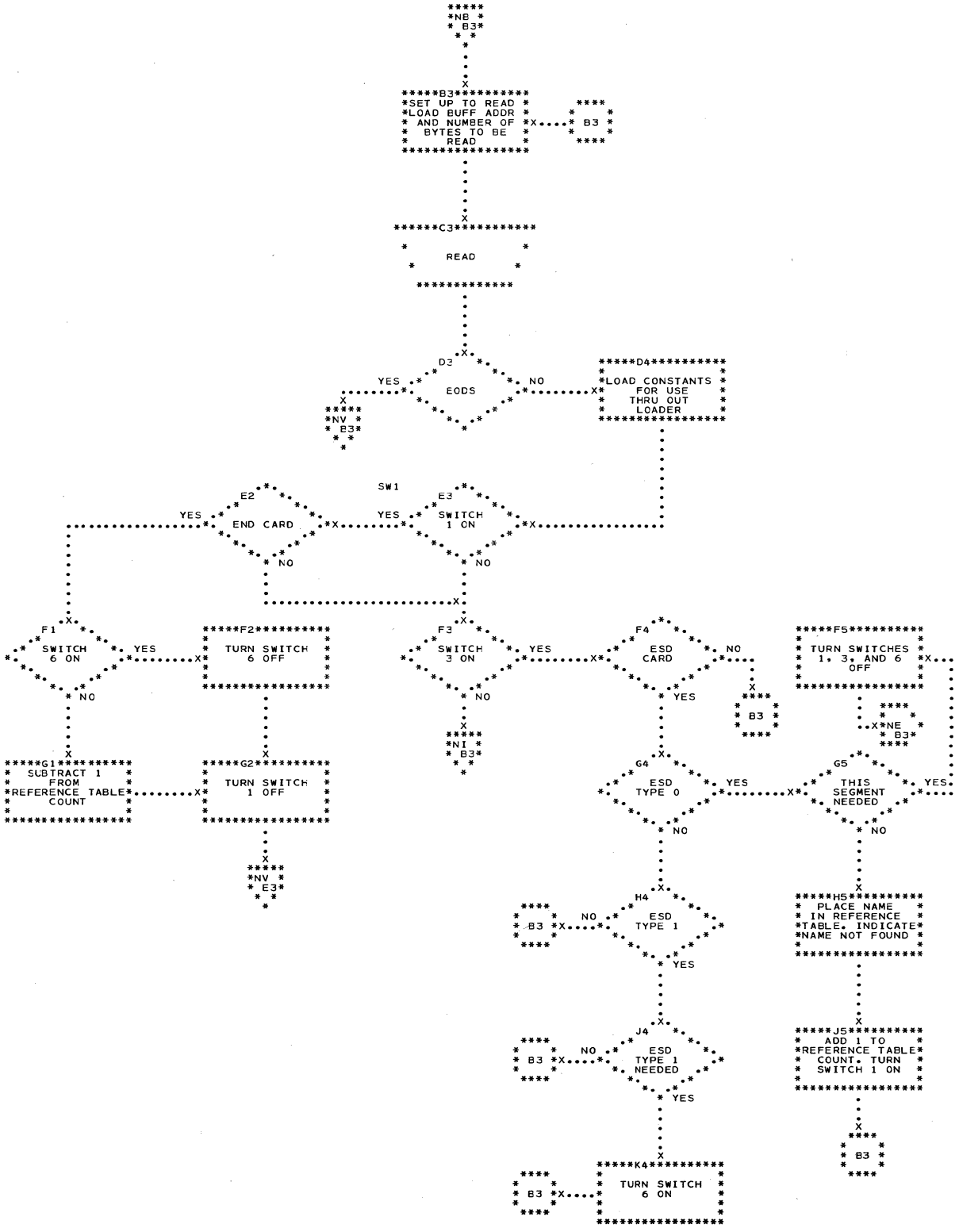


Chart NB. RD Routine

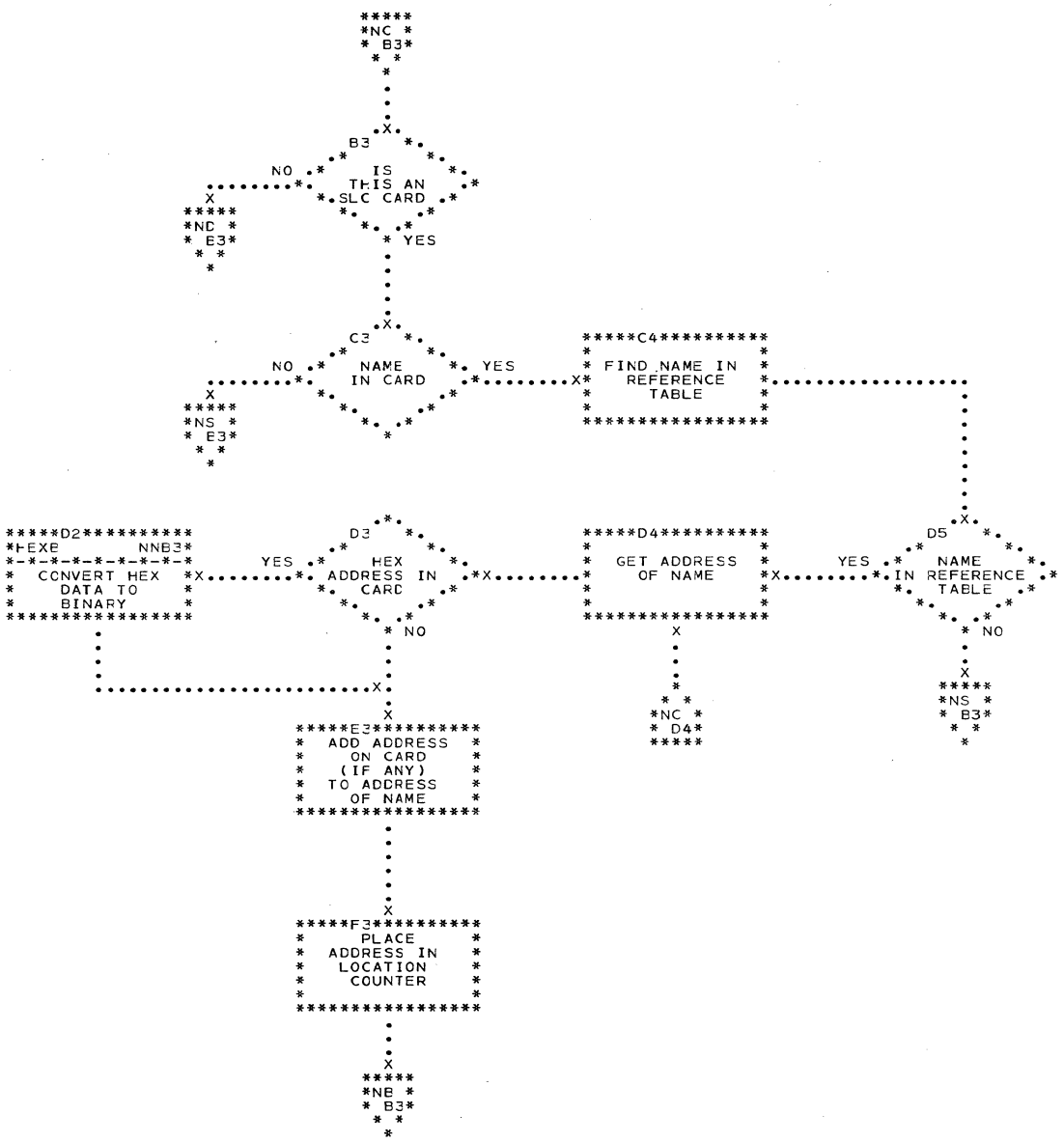


Chart NC. CMPSLC Routine

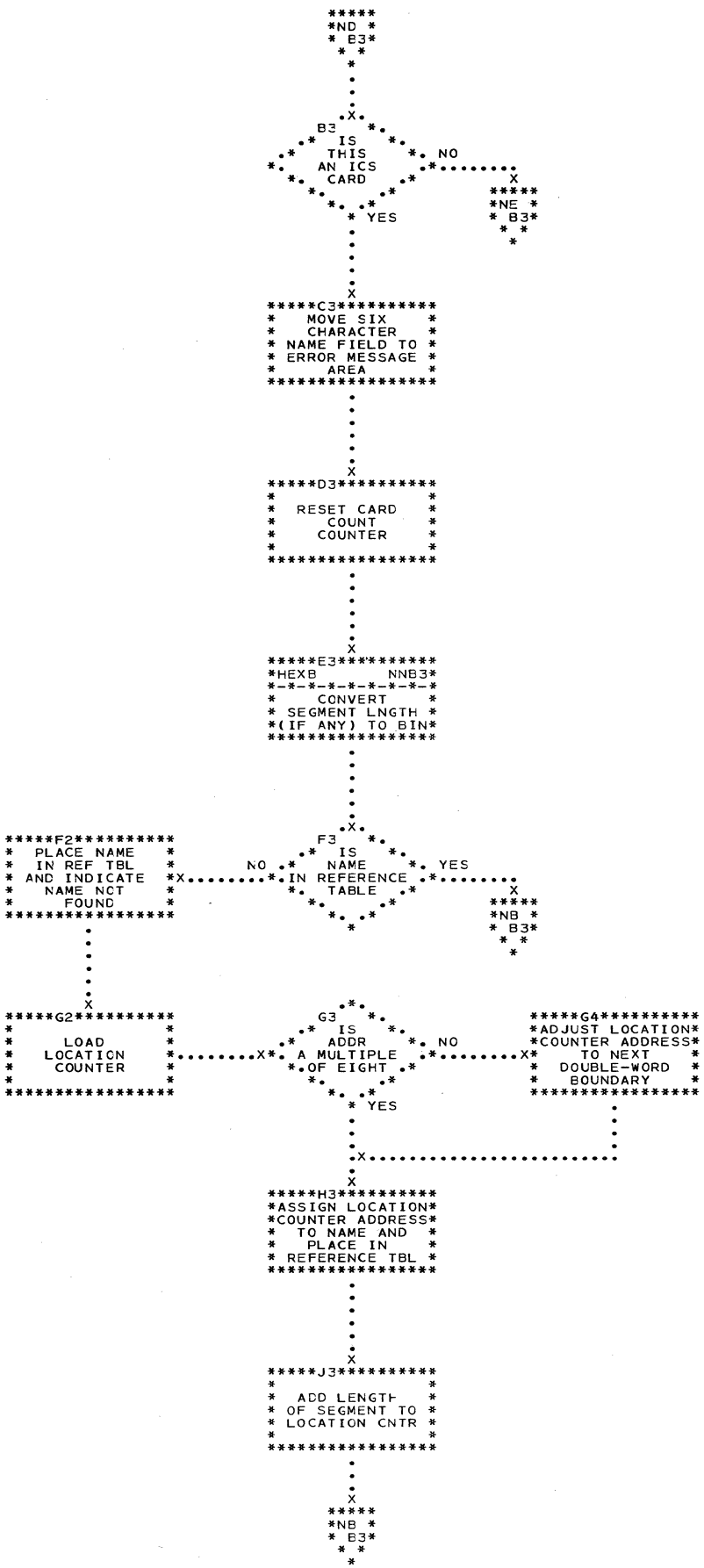


Chart ND. CMPICS Routine

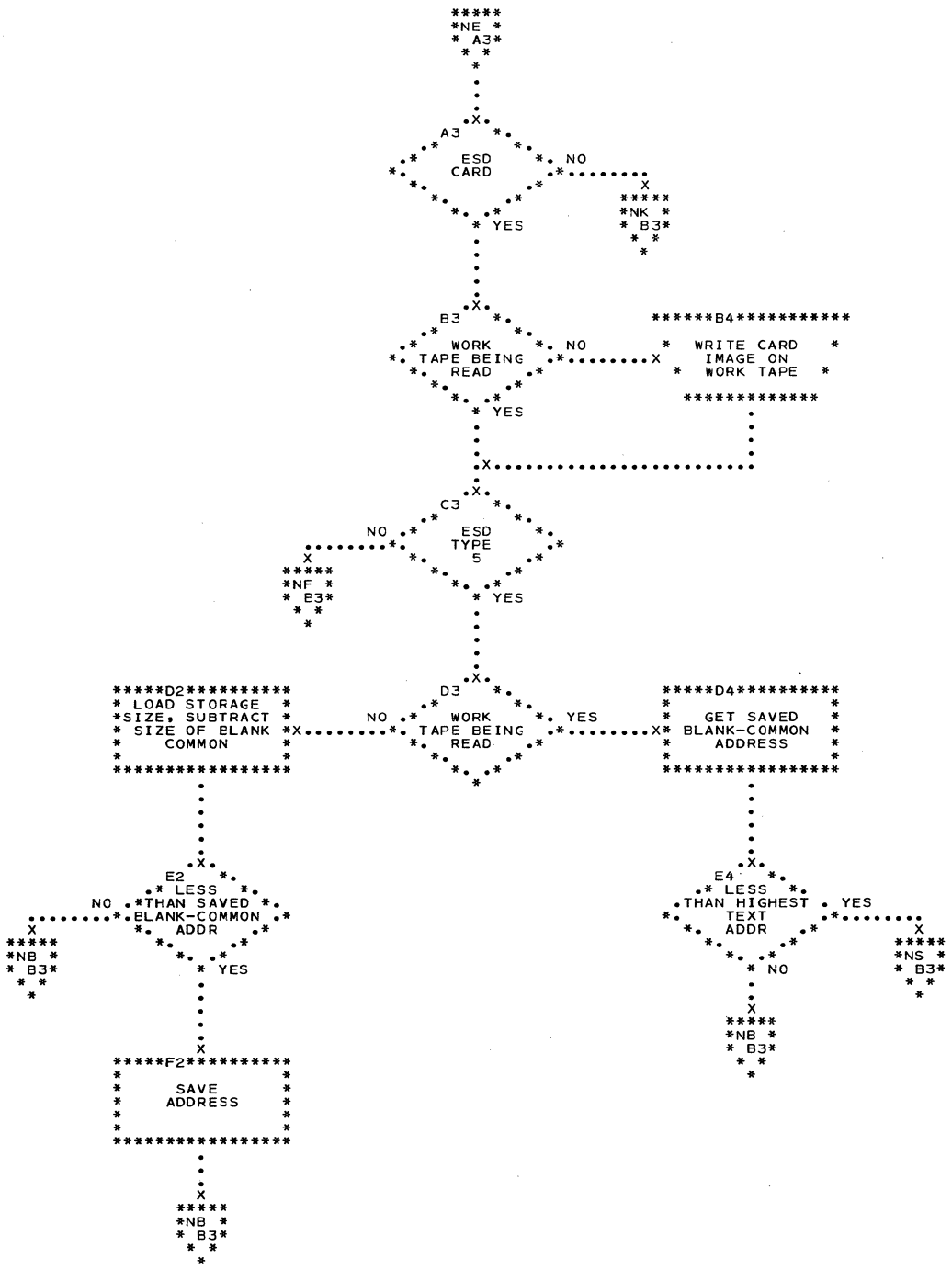


Chart NE. CMPESD Routine

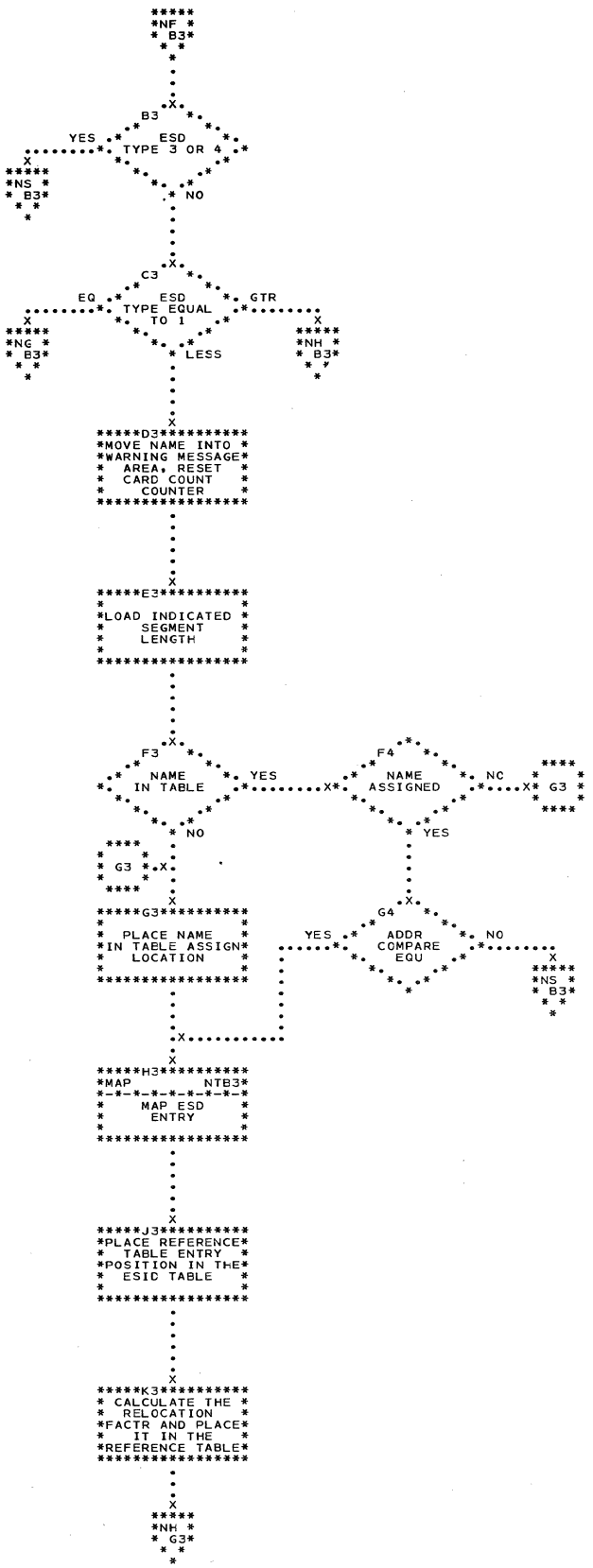


Chart NF. CESDo Routine

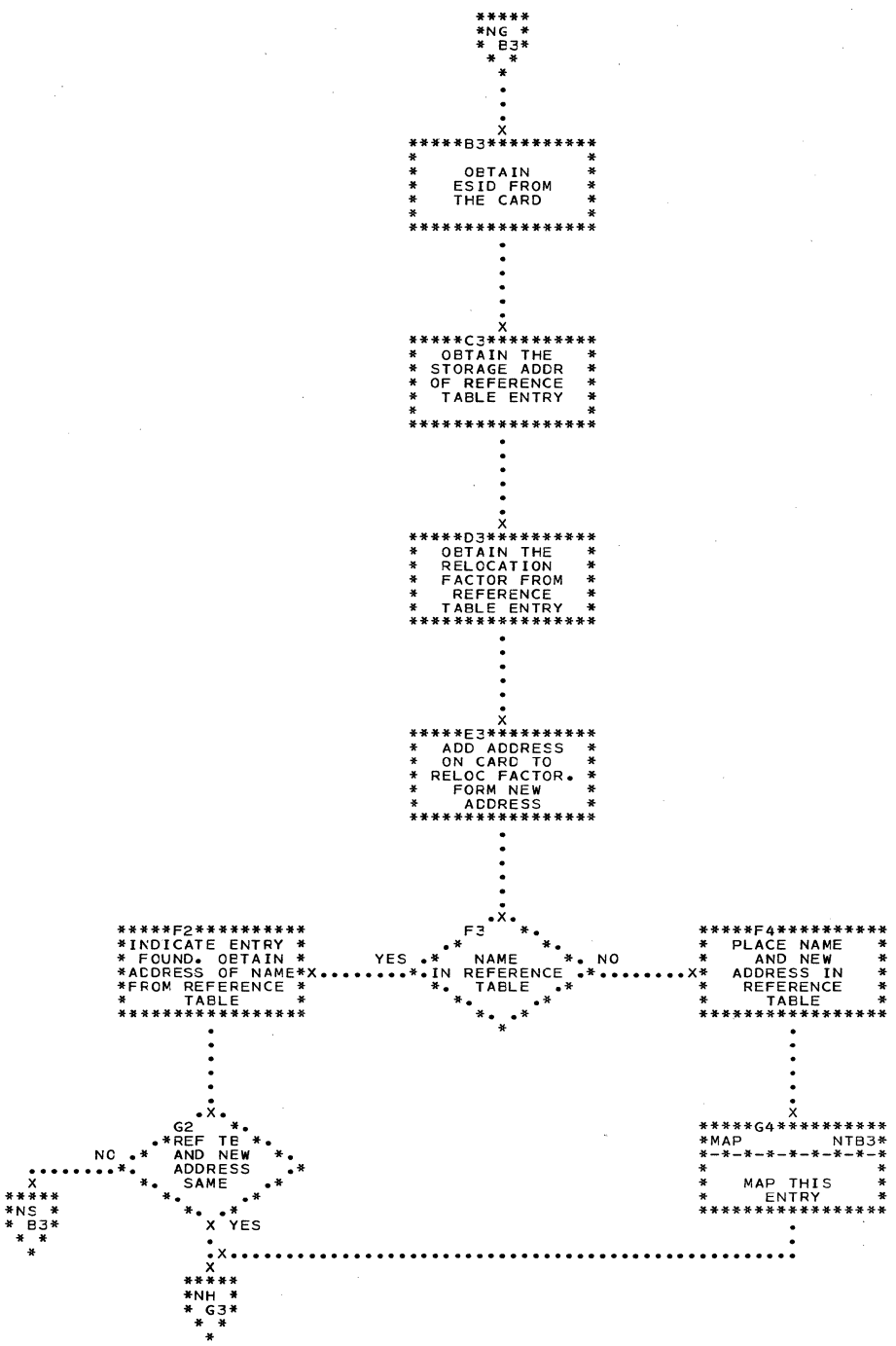


Chart NG. CESD1 Routine

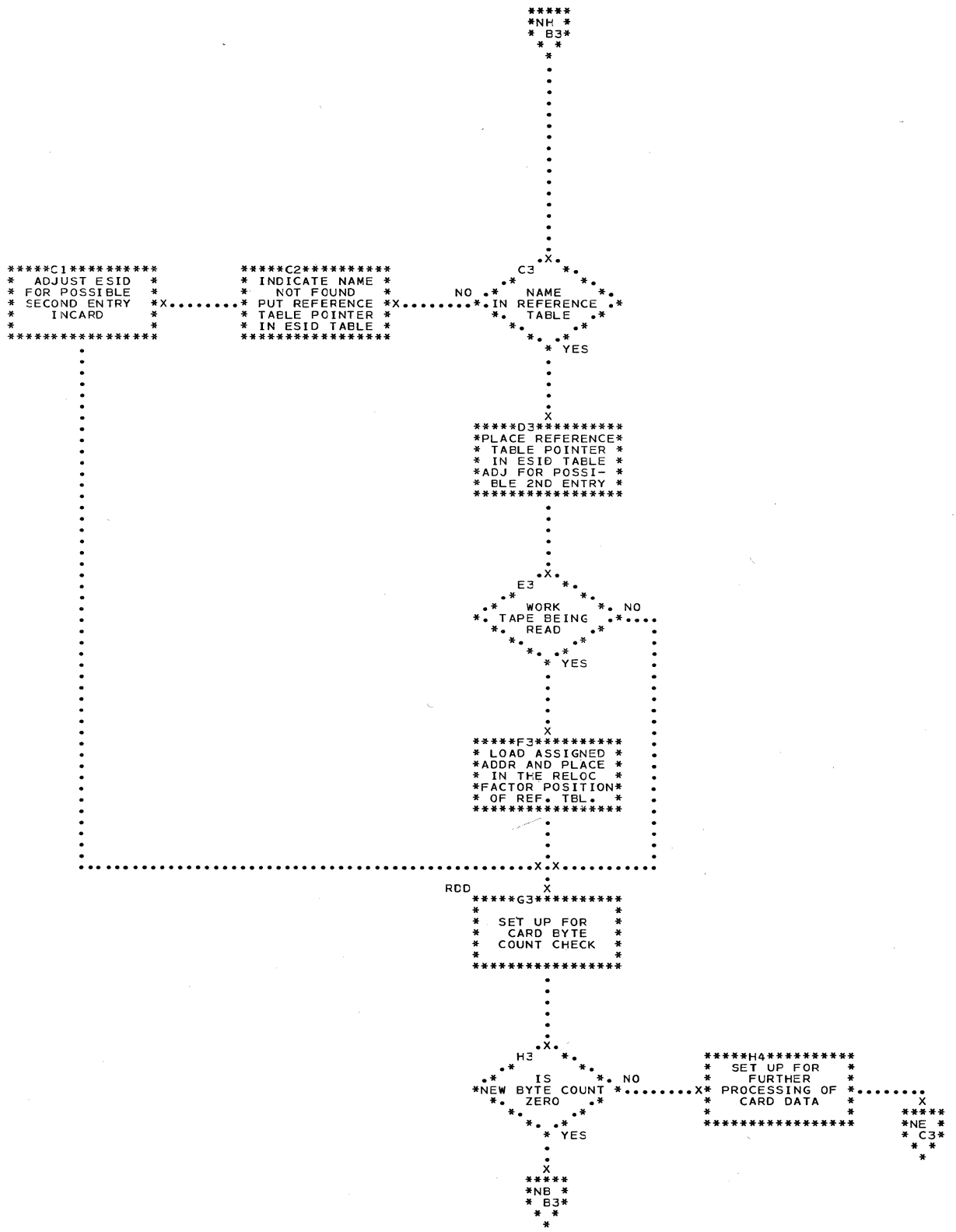


Chart NH. CESD2 Routine

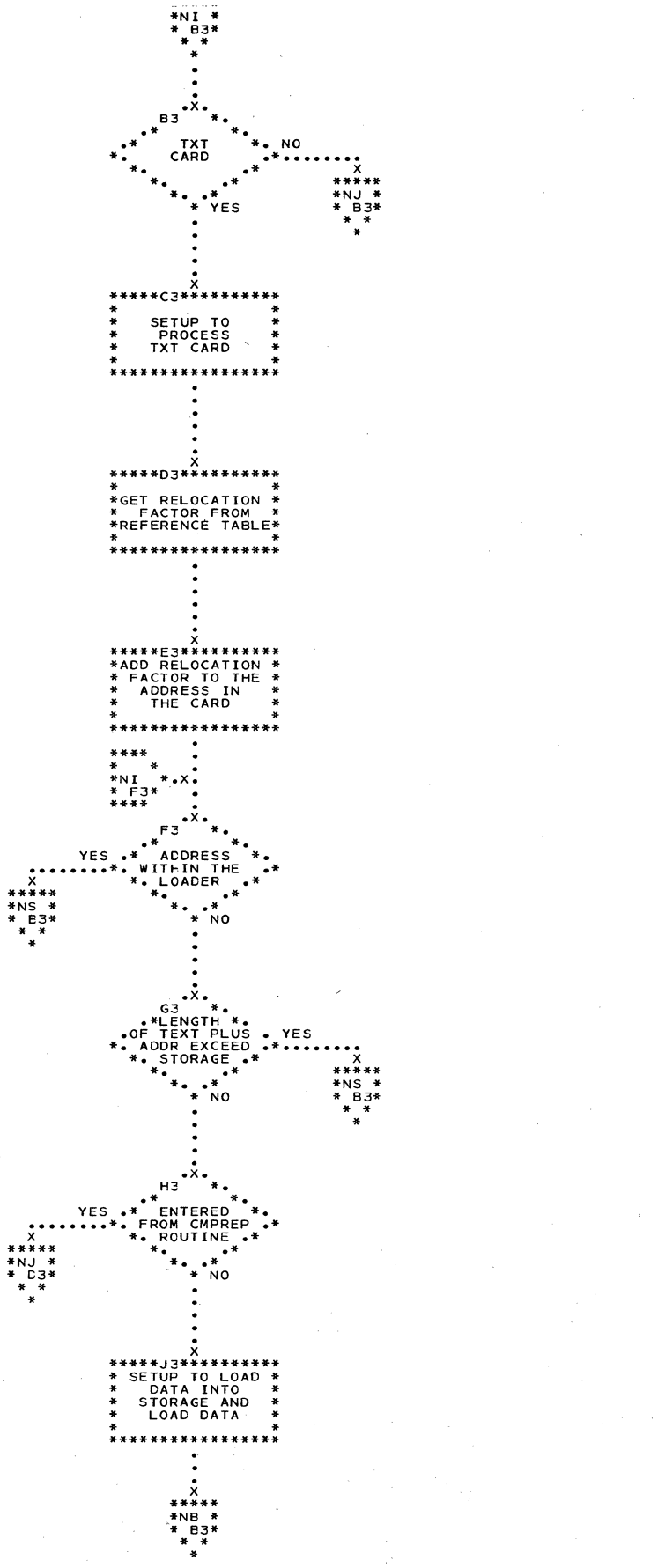


Chart NI. CMPTXT Routine

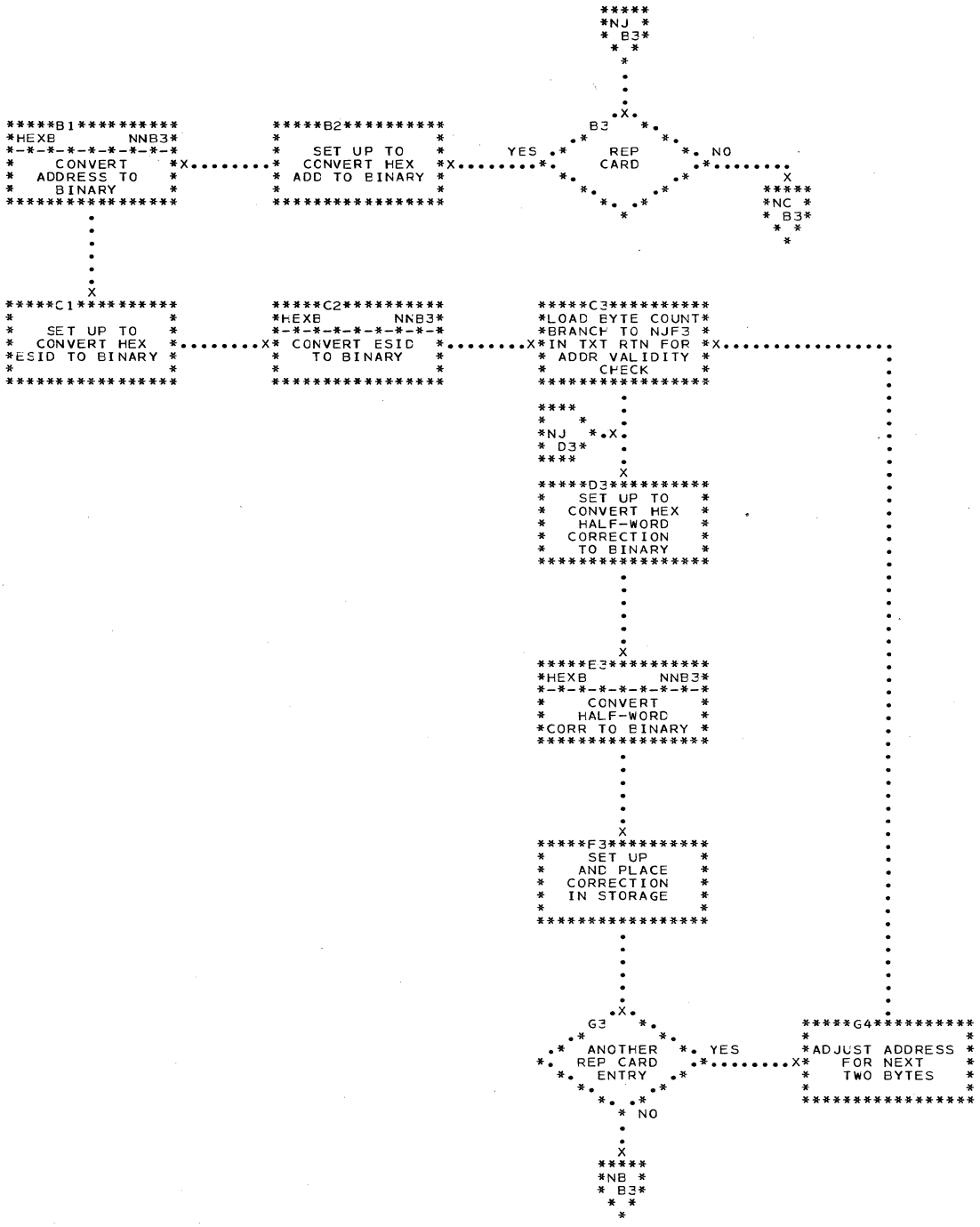


Chart NJ. CMPREP Routine

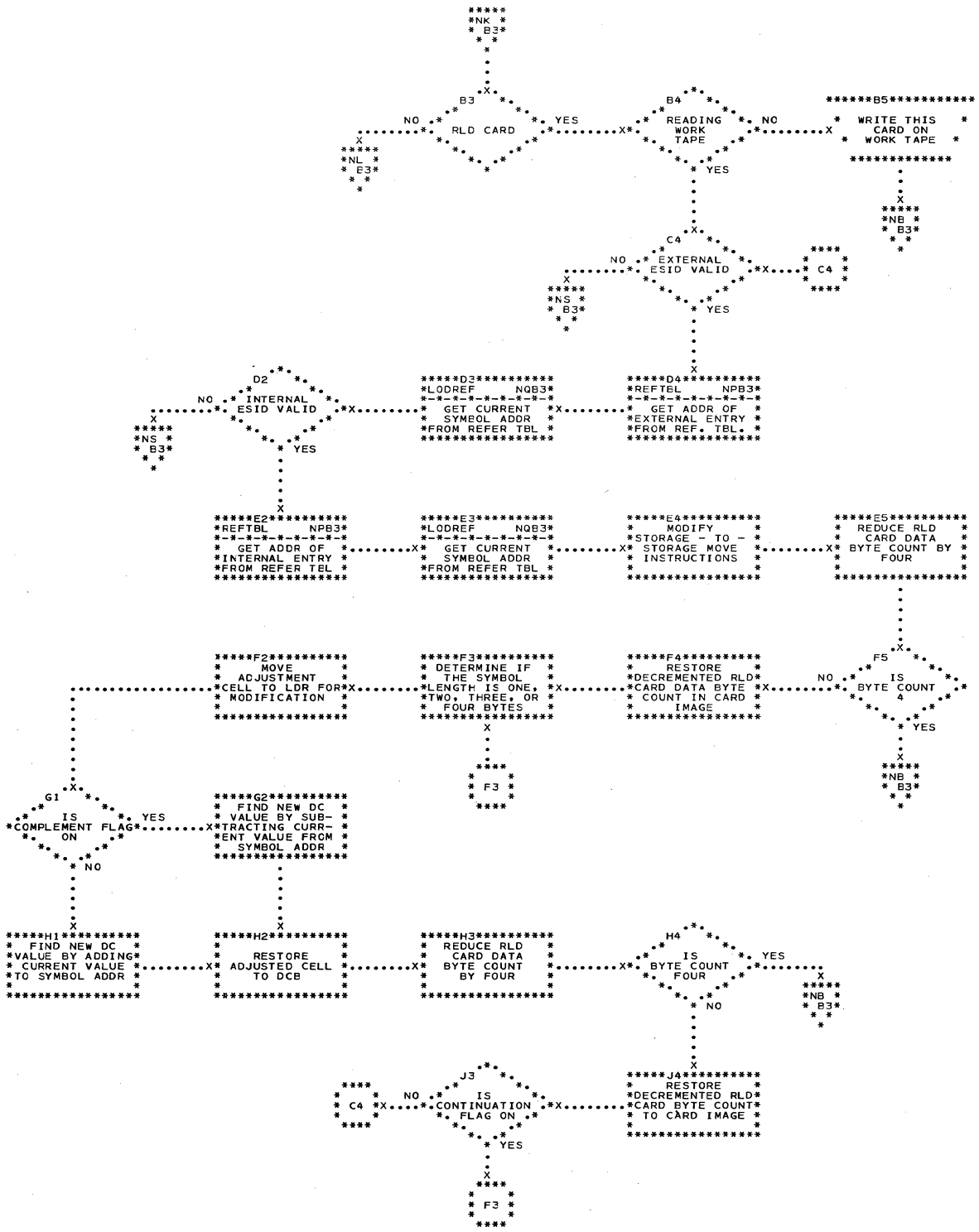


Chart NK. CMPRLD Routine

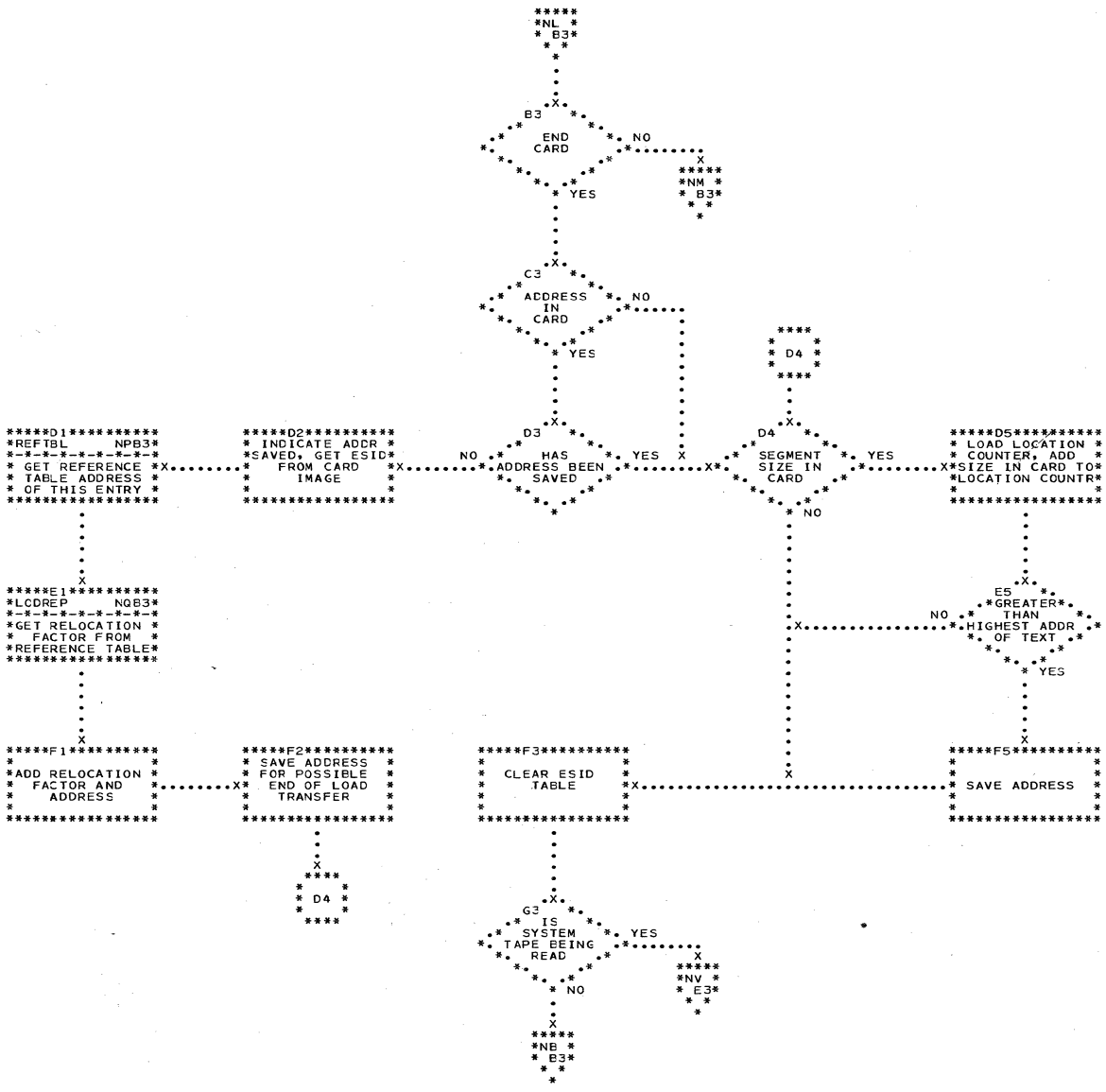


Chart NL. CMPEND Routine

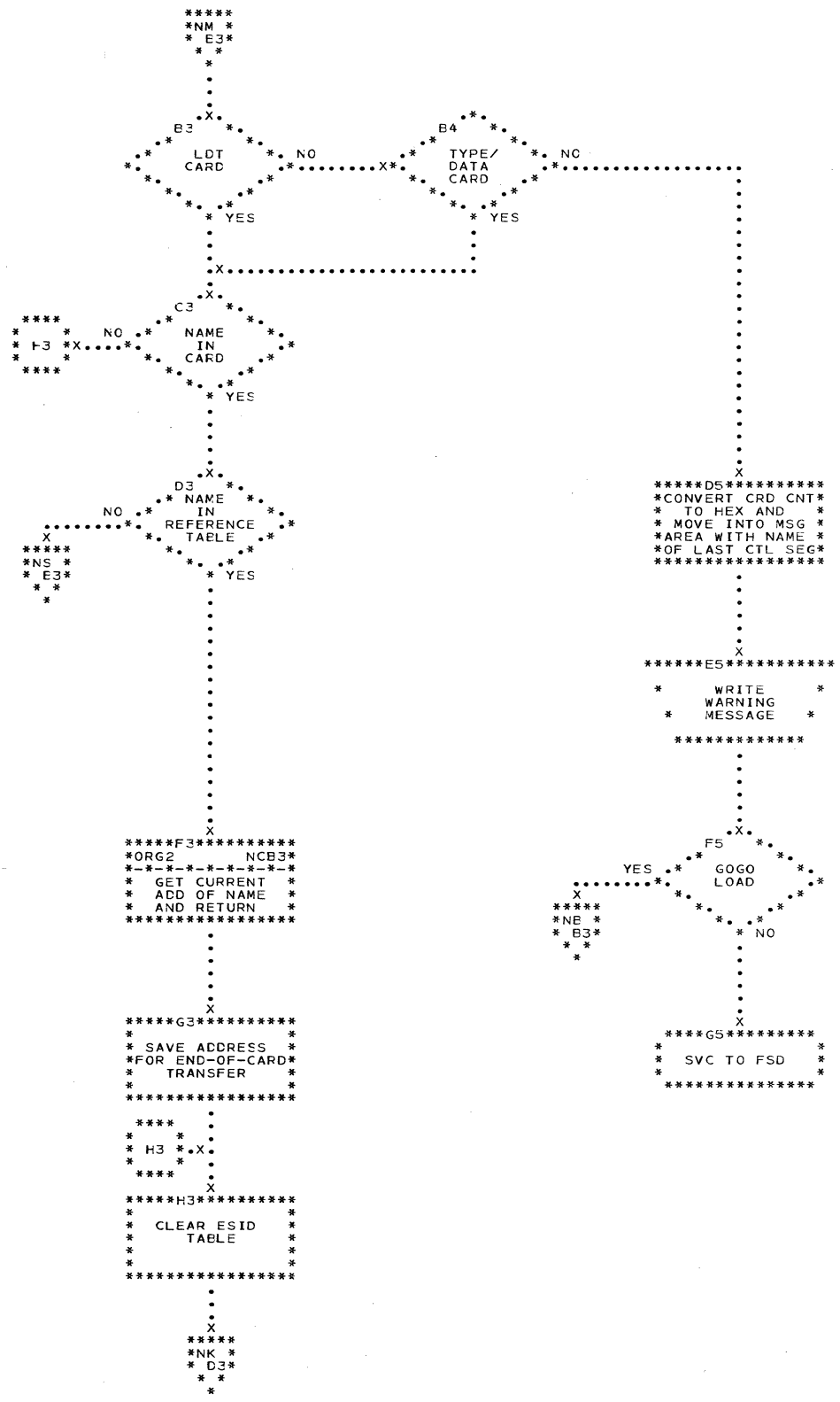


Chart NM. CMPLDT, WARN Routines

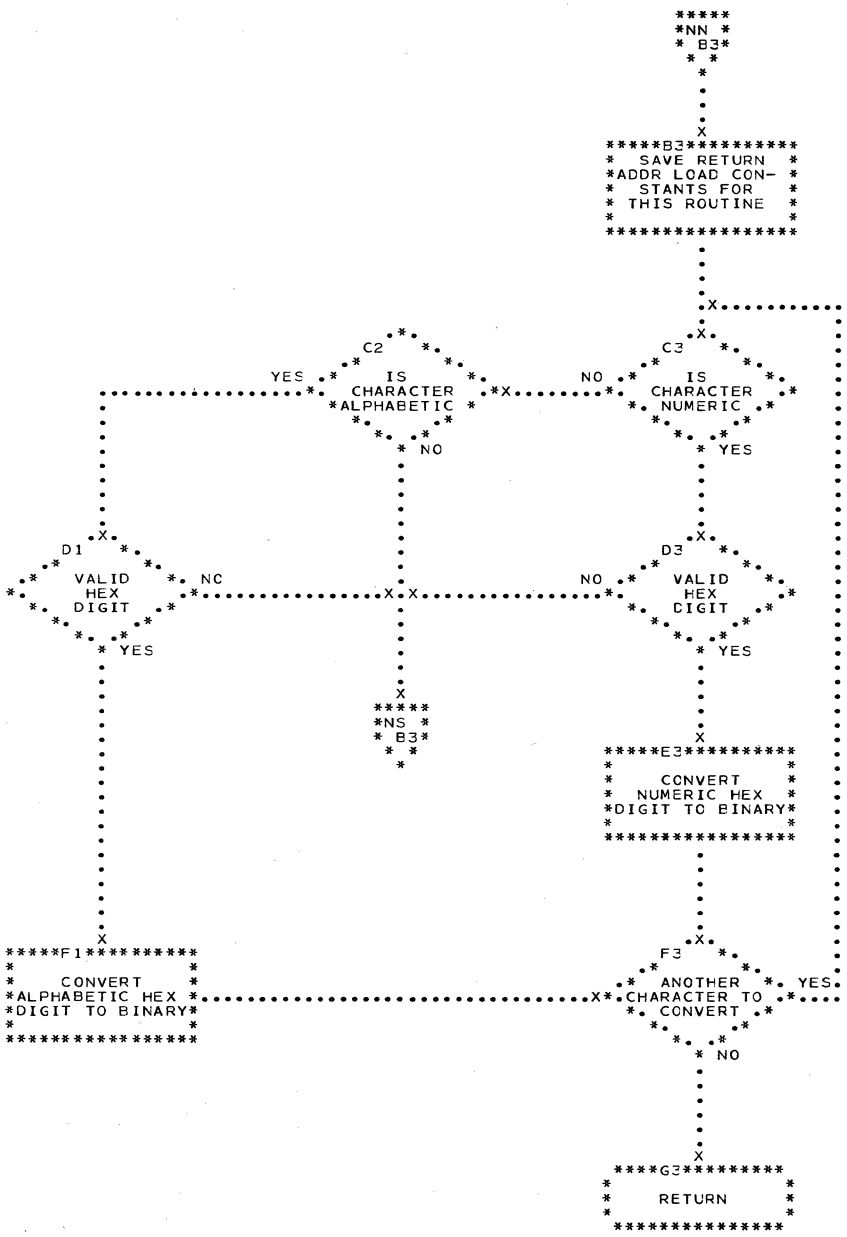


Chart NN. HEX Routine

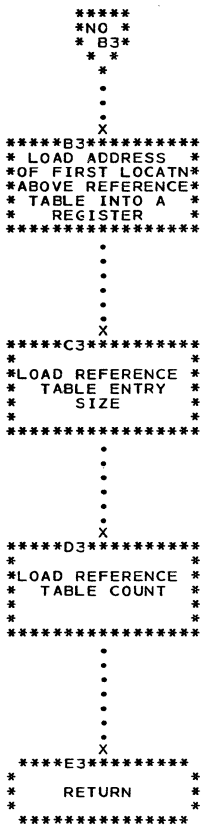


Chart NO. TBLREF Routine

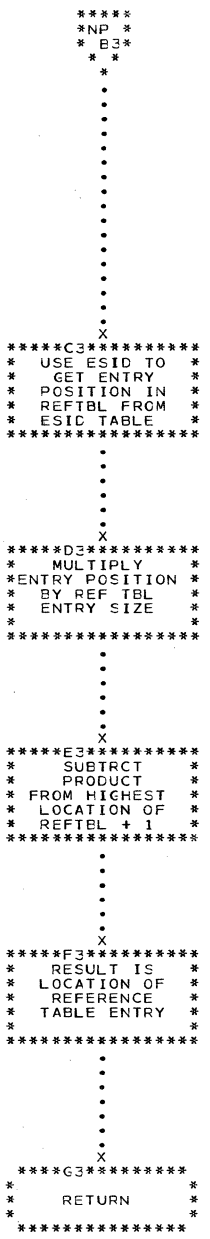


Chart NP. REFTBL Routine

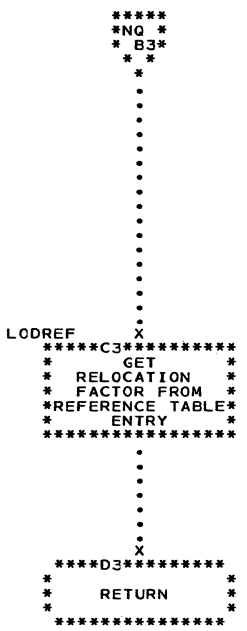


Chart NQ. LODREF Routine

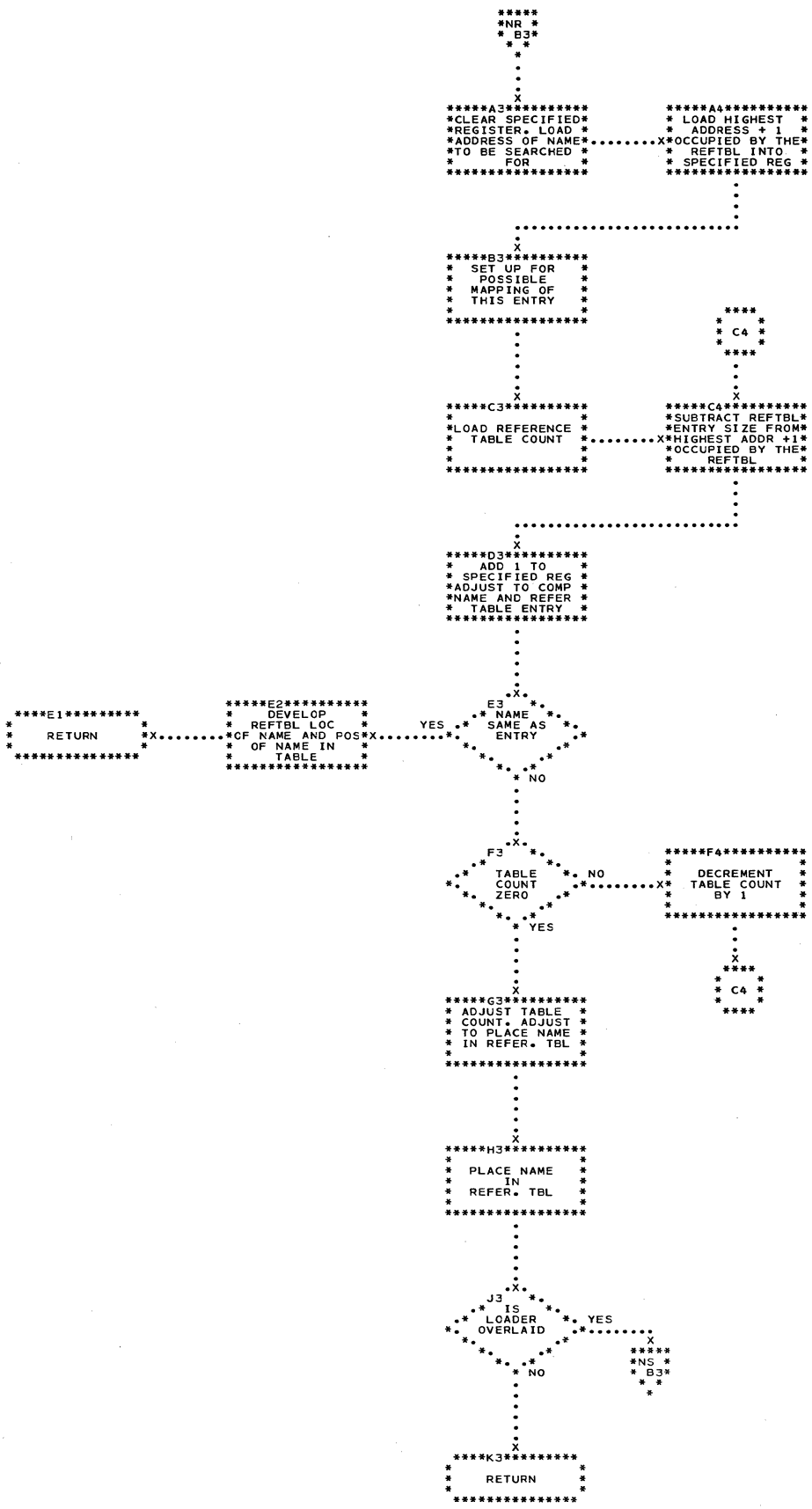


Chart NR. SERCH Routine

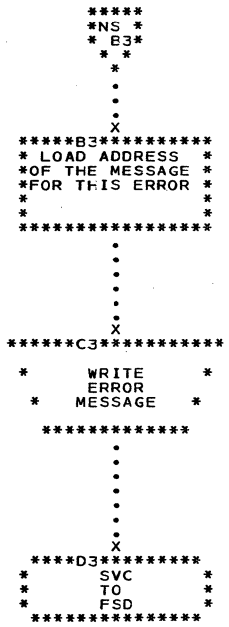


Chart NS. ERROR Routine

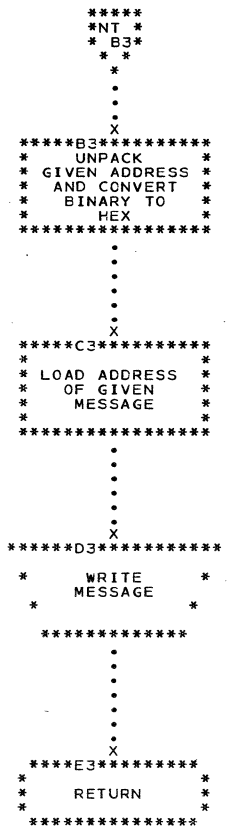


Chart NT. MAP Routine

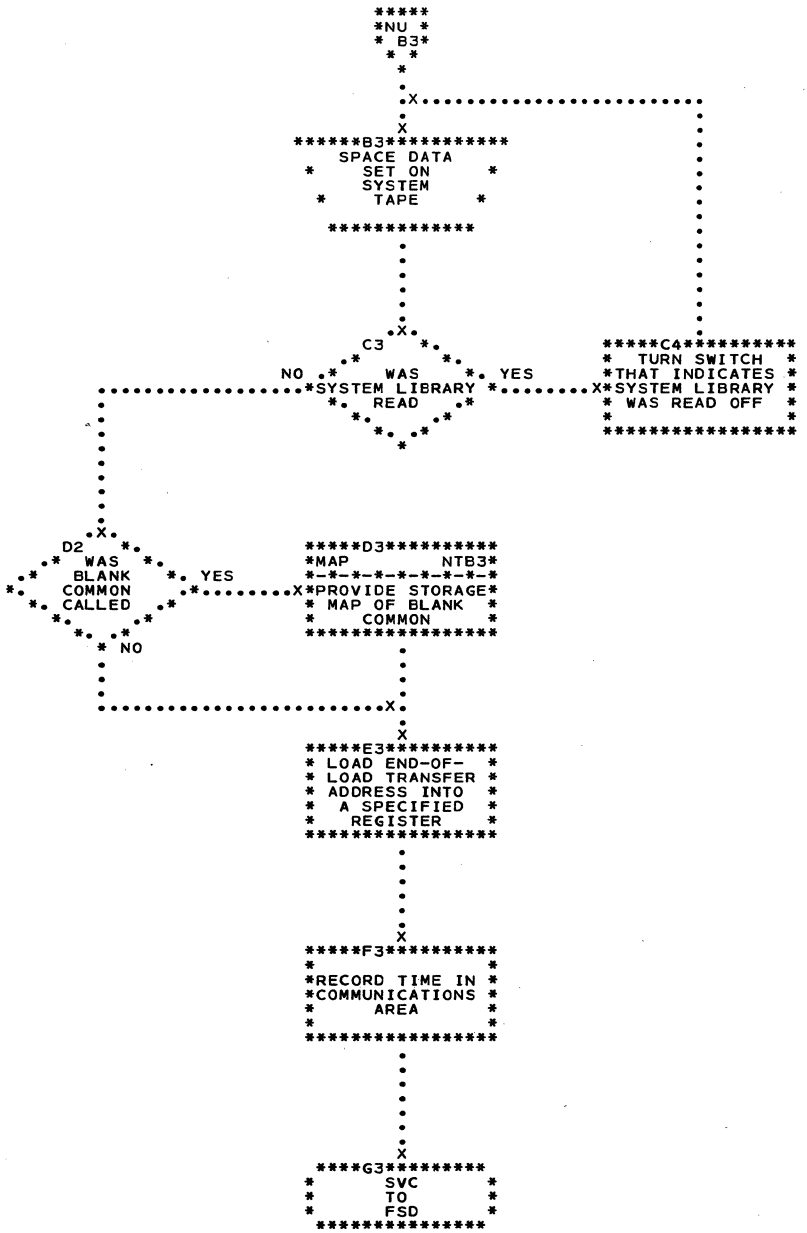


Chart NU. RELCTL Routine

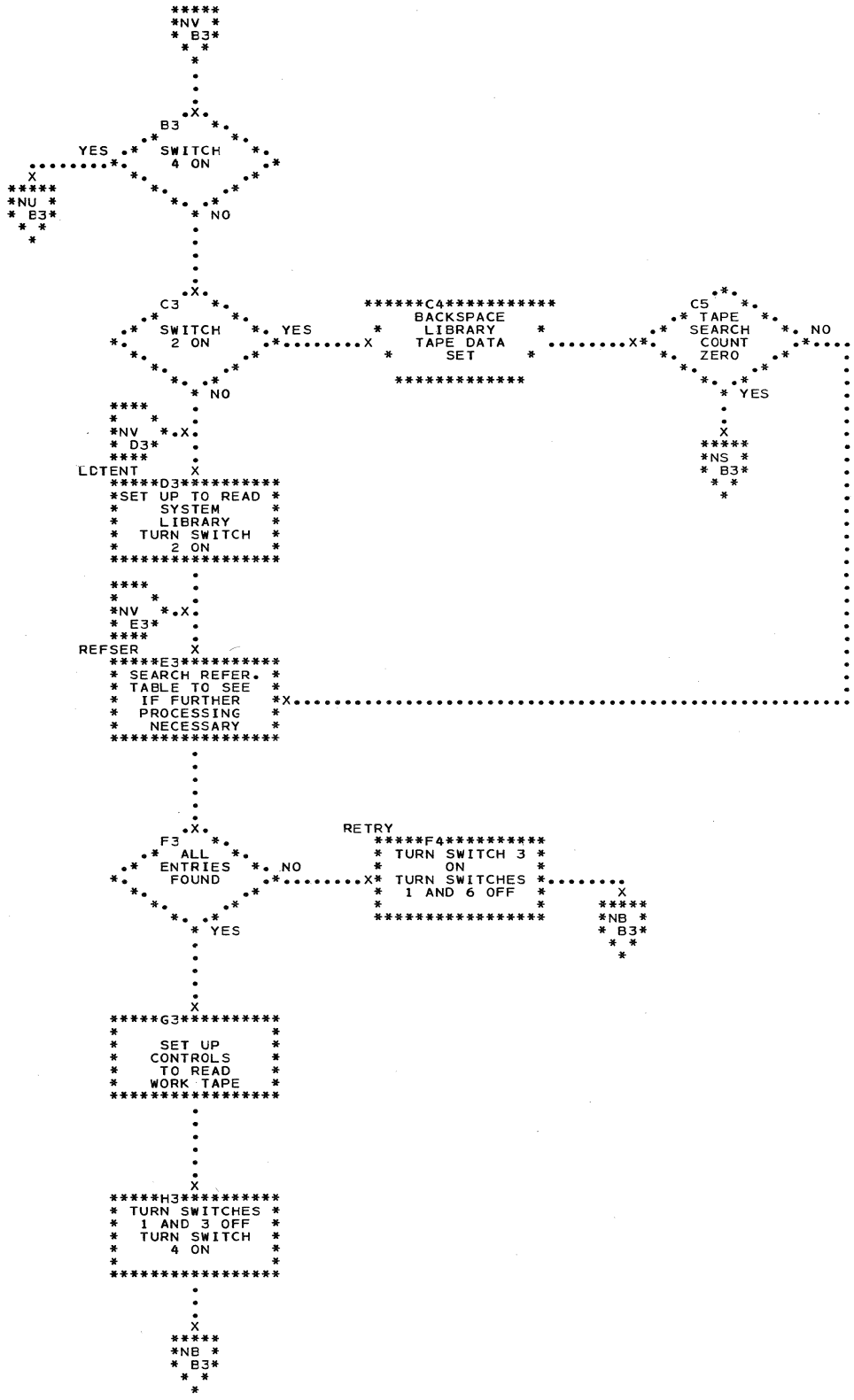


Chart NV. EODS Routine

IBCOM, a segment of the FORTRAN system, performs object time implementation of the following FORTRAN I/O source statements:

1. READ and WRITE (both requiring and not requiring a format).
2. BACKSPACE, REWIND, and END FILE (tape manipulation).
3. STOP and PAUSE (write to operator).

In addition, IBCOM processes object time errors detected by the various FORTRAN library subroutines, processes arithmetic type program interrupts, and terminates object program execution.

Chart 12 the IBCOM-Object Program Overall Logic Diagram indicates the entrance to and exit from IBCOM as the guide to the overall functions of IBCOM.

All linkages to IBCOM are compiler-generated. Each time one of the above-mentioned source statements is encountered during compilation, an appropriate calling sequence to IBCOM is generated and included as part of the object program. At object time, these calls are executed, passing control to IBCOM to perform the indicated I/O operation.

Except for READ/WRITE implementation, the operation of IBCOM, is straightforward. Therefore, only READ/WRITE implementation is discussed in this introduction. The other operations are discussed as part of the subroutine descriptions.

For the implementation of READ and WRITE statements, IBCOM consists of the opening, I/O list, and closing sections.

OPENING SECTION

The compiler generates a linkage to the opening section of IBCOM when it detects a READ or WRITE statement in the FORTRAN source program.

The opening section determines the nature of the operation (READ or WRITE, requiring or not requiring a format) and the address of the device upon which the operation is to be performed. The device address is saved for future operations.

READ Requiring a Format

If the determined operation is that of a READ requiring a format, a record is read into an I/O buffer. The location and size of the I/O buffer are saved, a pointer to the I/O buffer is initialized to the first location in that buffer, and the address of the FORMAT statement associated with the READ is saved. (The address of the FORMAT statement is passed as an argument to the opening section.) Control is passed to a portion of IBCOM that scans the FORMAT statement. The first format code (either a control or conversion type) of the FORMAT statement is then accessed.

For the control type code (e.g., an H format code or a group count), a list item is not required. Control passes to the control routine associated with the format code under consideration to perform the indicated operation. (See Subroutines FRDWF, FWRWF, FIOLF, FIOAF, and FENDF, Table E.1, Format Codes.) Control then passes to the scan portion, which obtains the next format code. The above operation is repeated for all control type codes, until either the end of the FORMAT statement or the first conversion code is encountered.

A conversion type code (e.g., an I format code) requires a list item in a READ statement. Upon the first encounter of a conversion type code in the scan of the FORMAT statement, the opening section of IBCOM completes its processing of a READ requiring a format and returns control to the next sequential instruction of the object program. The object program obtains the list item associated with the conversion code and calls the I/O list section of IBCOM.

WRITE Requiring a Format

If the opening section determines that the desired operation is that of a WRITE requiring a format, it proceeds in a manner similar to a READ requiring a format.

READ Not Requiring a Format

If the desired operation is that of a READ not requiring a format, the opening section of IBCOM reads a record into the I/O buffer. This section saves the location and size of the I/O buffer, initializes the buffer pointer to the first location in the I/O buffer beyond the control word, and returns control to the next sequential instruction of the object program. The object program obtains a list item and calls the I/O list section of IBCOM.

WRITE Not Requiring a Format

If the operation to be performed is a WRITE not requiring a format, the opening section proceeds in a fashion similar to a READ not requiring a format.

I/O LIST SECTION

The compiler generates a linkage to the I/O list section of IBCOM when it encounters an I/O list item in the FORTRAN source program.

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

In processing list items for any READ or WRITE requiring a FORMAT, the I/O list section passes control to the conversion routine that puts the list item in a format according to its associated conversion type format code. (This conversion routine has been pre-determined by the scan portion of IBCOM, and its address is made available to the I/O list section.) For input, the conversion routine accesses data from the I/O buffer and converts the data to the form dictated by the format code. The converted data is then moved into the list item. For output, the conversion routine accesses the list item, converts it to the form dictated by the format code, and moves the result to the I/O buffer.

After the conversion routine has processed the list item, the I/O list section determines if the format code applied to the list item just processed is to be repeated for the next list item. It looks for a field count (the number of times a conversion is to be repeated for an I/O list item) associated with the format code. If the format code is to be repeated and the list item just processed was a variable, control returns to the object program to obtain the next item. The object program again links to the I/O list section, and the conversion routine which processed the previous list item is given control. This action applies the same format code to the next list item.

If the format code is to be repeated and the list item just processed was an array element, the next element of the array is obtained. The format code is repeated for this element. There is no return to the object program until all array elements have been satisfied. If the format code is not to be repeated, control is passed to the scan portion of IBCOM to continue the scan of the FORMAT statement.

If the scan portion determines that a group of format codes is to be repeated, the format statement pointer is adjusted to the first format code in the group. The codes of the group are then repeated. If a group of codes is not to be repeated, the scan portion of IBCOM accesses the next format code. For the control type code, control is passed to its associated control routine. For the conversion type code,

control is returned to the object program which obtains the list item associated with the conversion code. The object program again links to the I/O list section to process the list item.

In processing list items for READ and WRITE statements not requiring a format, the I/O list section determines the size of the list item (i.e., the number of bytes reserved for the list item). The list item may be either a variable or an array. In either case, the number of bytes specified by the size of the list item is moved from the I/O buffer to the list item on input and reversed on output. Control is then returned to the object program to obtain the next list item.

Conversion Routines: The conversion routines, an integral part of the I/O list section for any READ or WRITE requiring a format, have an associated, conversion type format code. Each conversion routine converts the list item presented to it into the form dictated by its associated format code. The conversion routine moves the converted result to the address assigned to the list item if a READ statement is being implemented, or to the I/O buffer if a WRITE statement is being implemented.

CLOSING SECTION

The compiler generates a linkage to the closing section of IBCOM after all list items associated with the READ or WRITE statement have been processed. The closing section closes input/output operations for I/O statements irrespective of format requirements.

IBCOM SUBROUTINES

The IBCOM subroutines are broken down in the following categories:

Charts PA through PH

+0 1. FRDWF--Opening section for a READ requiring a format
+4 2. FWRWF--Opening section for a WRITE requiring a format
+8 3. FIOLF--I/O list section for the list variable of a READ or WRITE requiring a format
+12 4. FIOAF--I/O list section for list array of a READ or WRITE requiring a format
+16 5. FENDF--Closing section for a READ or WRITE requiring a format

Charts PI through PN

6. D, E, F, I, A conversion routines (both input and output)

Charts PO through PT

+20 7. FRDNF--Opening section for a READ not requiring a format
+24 8. FWRNF--Opening section for a WRITE not requiring a format
+28 9. FIOLN--I/O list section for list variable of a READ or WRITE not requiring a format
+32 10. FIOAN--I/O list section for list array of a READ or WRITE not requiring a format
+36 11. FENDN--Closing section for a READ or WRITE not requiring a format

Charts PU through PW

+40 12. FBKSP
+44 13. FRWND Tape manipulation
+48 14. FEOFM

Chart PX

+52 15. FSTOP Write to operator
+56 16. FPAUS

Chart PY

+60 17. IBFERR Execution error monitor

Chart PZ

+64 18. IBFINT Interrupt processor

Charts QA, QB

19. FIOCS I/O interface

Chart QC

+68 20. IBEXIT Job terminator

Subroutines FRDWF, FWRWF, FIOLF, FIOAF, and FENDF: Charts PA through PH

These five subroutines transfer data between external storage and main storage under control of a FORMAT statement. The format code specifies the type of conversion to be performed between the internal and external representations of the data. These subroutines constitute that portion of IBCOM which implements any READ or WRITE requiring a format.

ENTRANCE: The five subroutines are entered at object time under the following conditions:

1. Subroutine FRDWF when a READ statement requiring a format is to undergo opening section operations.
2. Subroutine FWRWF when a WRITE statement requiring a format is to undergo opening section operations.
3. Subroutine FIOLF when a list variable of a READ or WRITE requiring a format is to undergo I/O list section processing.
4. Subroutine FIOAF when a list array of a READ or WRITE requiring a format is to undergo I/O list section processing.
5. Subroutine FENDF when a READ or WRITE operation requiring a format is to undergo closing section operations.

OPERATION: Subroutines FRDWF and FWRWF, the opening section subroutines, call subroutine FIOCS to select the actual channel and device that corresponds to the data set reference number. FIOCS also initializes the data set for input or output.

Upon return from FIOCS, FORMAT statement processing is initiated. The FORMAT statement has been analyzed, translated, and packed during compilation to a format recognizable by IBCOM (see introduction to Phase 14). The processing performed for the various format codes is explained in Table 2.

Table 2. Format Codes

FORMAT Code	Description	Type	Corresponding Action Upon Code
	beginning of statement 02	control	Save location for possible repetition of the format codes; clear counters.
n(	group count 04nn	control	Save n and location of left parenthesis for possible repetition of the format codes in the group.
n	field count 06nn	control	Save n for repetition of format code which follows.
nP	scaling factor 08nn	control	Save n for use by F, E, and D conversions. (left-most bit if nn=1 if nn is negative)
Tn	column reset 12nn	control	Reset current position within record to nth column or byte.
nX	skip or blank 18nn	control	Skip n characters of an input record or insert n blanks in an output record.
'text' or nH	literal data 1Annmm bytes	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 F 0Awwdd E 0Cwwdd D 0Ewwdd I 10ww A 14ww	conversion	Exit to the object program to return control to subroutine FIOLF or FIOAF. Using information passed to the I/O list section, the address and length of the current list item are obtained and passed to the proper conversion routine together with 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 object program to obtain another list item.
)	group end 1C	control	Test group count. If greater than 1, repeat format codes in group; otherwise continue to process FORMAT statement from current position.
/	record end 1E	control	Input or output one record using subroutine FIOCS.
	end of statement 22	control	If no I/O list items remain to be transmitted, return control to the object program to link to subroutine FENDF, the closing section; if list items remain, input or output one record using subroutine FIOCS. Repeat format codes from last left parenthesis.

Note: The internal representation of the above format specification codes is discussed in the introduction to Phase 14.

When any conversion code is encountered in the FORMAT statement and no I/O list items remain to be transmitted, control is passed to subroutine FENDF, the closing section. If a WRITE operation is being implemented, the current record is put out using subroutine FIOCS. General housekeeping is performed, and control is returned to the object program.

EXIT: Each IBCOM section (opening, I/O list, and closing) returns control to the object program after execution. However, the exit is to subroutine IBFERR for an error.

Subroutines FCVII and FCVIO: Charts PI, PJ

Subroutine FCVII

Subroutine FCVII reads integer data (integer input conversion) according to an Iw format code.

ENTRANCE: Subroutine FCVII receives control from subroutines FIOLF or FIOAF.

OPERATION: The number of bytes, (specified by w, in the format code) is scanned from left to right, starting at the appropriate I/O buffer location. The characters contained in these bytes are converted to a signed binary integer and stored in the list item.

EXIT: After execution subroutine FCVII exits to subroutines FIOLF or FIOAF.

Subroutine FCVIO

Subroutine FCVIO writes integer data (output integer conversion) according to an Iw format code.

ENTRANCE: Subroutine FCVIO receives control from subroutines FIOLF or FIOAF.

OPERATION: The contents of the list item are converted from a binary integer to decimal digits. These characters, preceded by leading blanks sufficient to fill the number of bytes, specified by w, are stored from left to right in the I/O buffer, starting at the appropriate buffer location.

EXIT: After execution subroutine FCVIO exits to subroutines FIOLF or FIOAF.

Subroutines FCVEI/FCVDI and FCVEO/FCVDO: Charts PK, PL

Subroutines FCVEI/FCVDI

Subroutine FCVEI/FCVDI reads real data with an external exponent (real/double input conversion, exponent) according to an Ew.d or Dw.d format code.

ENTRANCE: Subroutine FCVEI/FCVDI receives control from subroutines FIOLF or FIOAF.

OPERATION: The number of bytes, (specified by w, in the format code) is scanned from left to right, starting at the appropriate buffer location. The characters in these bytes are converted to a binary integer and scaled according to the value of d and the exponent field. The result is stored in the list item.

EXIT: After execution, subroutine FCVEI/FCVDI exits to subroutines FIOLF or FIOAF.

Subroutine FCVEO/FCVDO

Subroutine FCVED/FCVDO writes real data with an external exponent (real/double output conversion, exponent) according to an Ew.d or Dw.d format code.

ENTRANCE: Subroutine FCVED/FCVDO receives control from subroutines FIOLF or FIOAF.

OPERATION: The contents of the list item are scaled according to its characteristic and the scale factor. The result is segmented into integer and fractional portions, and then converted to properly signed decimal digits, separated by a decimal point. An exponent field is placed to the right of the fraction indicating a power of 10 to which the preceding number must be raised to obtain the proper value. All of these characters, preceded by sufficient blanks to fill w bytes, are stored from left to right in the I/O buffer, starting from the appropriate buffer position.

EXIT: After execution subroutine FCVEO/FCVDO exits to subroutines FIOLF or FIOAF.

Subroutines FCVFI and FCVFO: Charts PK, PL

Subroutine FCVFI

Subroutine FCVFI reads real data without an external exponent (real-double input conversion, no exponent) according to a Fw.d format code.

ENTRANCE: Subroutine FCVFI receives control from subroutines FIOLF or FIOAF.

OPERATION: A number of bytes (specified by w in the format code) is scanned from left to right starting from the appropriate I/O buffer position. The characters contained in these bytes are converted to a binary integer and scaled according to the value of d and the scale factor. The result is stored in the list item.

EXIT: After execution subroutine FCVFI exits to subroutine FIOLF or FIOAF.

Subroutine FCVFO

Subroutine FCVFO writes real data without an external exponent (real-double output conversion, no exponent) according to a Fw.d format code.

ENTRANCE: Subroutine FCVFO receives control from subroutine FIOLF and FIOAF.

OPERATION: The contents of the list items are scaled according to its characteristic and scale factor. The result is segmented into integer and fractional portions and then converted to properly signed decimal digits separated by a decimal point. These characters, preceded by sufficient blanks to fill w bytes, are stored from left to right in the I/O buffer, starting from the appropriate buffer location.

EXIT: After execution subroutine FCVFO exits to subroutines FIOLF or FIOAF.

Subroutines FCVAI and FCVAO: Charts PM, PN

Subroutine FCVAI

Subroutine FCVAI reads alphameric data (alphameric input conversion) according to an Aw format code.

ENTRANCE: Subroutine FCVAI receives control from subroutines FIOLF or FIOAF.

OPERATION: If the value of w (in the format code) is greater than or equal to L (the length of the list item), the L rightmost characters in the I/O buffer are used to fill the list item. If the value of w is less than L, w characters from the I/O buffer are left-justified in the list item with L minus w trailing blanks.

EXIT: After execution subroutine FCVAI exits to subroutine FIOLF or FIOAF.

Subroutine FCVAO

Subroutine FCVAO writes alphameric data (alphameric output conversion) according to an Aw format code.

ENTRANCE: Subroutine FCVAO receives control from subroutine FIOLF or FIOAF.

OPERATION: If the value of w is less than or equal to L the w leftmost characters in the list item are placed into the I/O buffer. If the value of w is greater than L, L characters from the list item are right justified in the I/O buffer with w minus L leading blanks.

EXIT: After execution subroutine FCVAO exits to subroutines FIOLF or FIOAF.

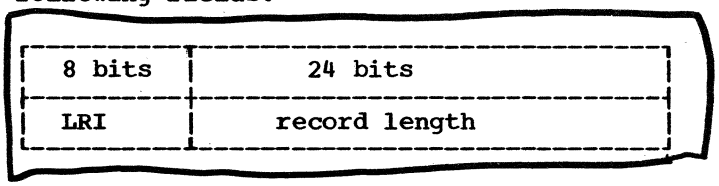
Subroutines FRDNF, FWRNF, FIOLN, FIOAN, and FENDN: Charts PO through PT

These five subroutines transfer data between external storage and main storage with no intermediate conversion. These subroutines constitute that portion of IBCOM which implements any READ or WRITE not requiring a format.

ENTRANCE: The five subroutines are entered at object time under the following conditions:

1. Subroutine FRDNF when a READ statement not requiring a format is to undergo opening section operations.
2. Subroutine FWRNF when a WRITE statement not requiring a format is to undergo opening section operations.
3. Subroutine FIOLN when a list variable for a READ or WRITE not requiring a format is to undergo I/O list section processing.
4. Subroutine FIOAN when a list array for a READ or WRITE not requiring a format is to undergo I/O list section processing.
5. Subroutine FENDN when a READ or WRITE not requiring a format is to undergo closing section operations.

OPERATION: Each record read that does not require a format is expected to contain a control word of the type prefixed to every record written not requiring a format. This control word occupies the first four bytes of the record and consists of the following fields:



A FORTRAN logical record consists of the total number of records necessary to contain all I/O list items within a single WRITE statement. For all but the last record within the logical record, LRI (Last Record Indicator) equals zero; for the last record, LRI equals the total number of records within this logical record. The

"record length" is always present and equals the number of bytes within the record, excluding the control word. Subroutines FRDNF and FWRNF, the opening section subroutines, call subroutine FIOCS to select the actual channel and device corresponding to the data set reference number and to initialize the data set for input or output.

Upon return from subroutine FIOCS, control returns to the object program to obtain an I/O list item and to call the I/O list section, either subroutine FIOLN or FIOAN. Using the information passed to the I/O list section, the address of the current list item and its length are obtained. They are used to transfer bytes from the I/O buffer to the list item on input or from the list item to the I/O buffer on output.

If end of record is reached and I/O list items remain to be transmitted, one record is read or written using subroutine FIOCS. Then processing of the I/O list resumes.

When no I/O list items remain to be transmitted, control is passed to subroutine FENDN, the closing section. If a READ operation is being implemented, successive records are read using FIOCS until the end of the logical record is reached. If a WRITE operation is being implemented, the current record (containing an end-of-logical record indicator) is written using FIOCS. General housekeeping is performed, and control is returned to the object program.

EXIT: Each section of IBCOM (opening, I/O list and closing) returns control to the object program after execution. However, if an error exists, the exit is to subroutine IBFERR.

Subroutine FBKSP: Chart PU

Subroutine FBKSP implements the BACKSPACE source statement. This subroutine backspaces the specified tape unit one physical record for a data set requiring a format, and one logical record for a data set not requiring a format.

ENTRANCE: Subroutine FBKSP receives control from the object time execution of a compiler-generated linkage. This linkage is generated during compilation when a BACKSPACE statement is encountered.

CONSIDERATION: If the tape is at load point and a backspace operation is requested, the backspace operation is not performed.

OPERATION: Subroutine FBKSP calls subroutine FIOCS to select the actual channel and device corresponding to the data set reference number and issue a backspace record control to that channel and device. If the data set required a format, subroutine FBKSP returns control to the object program. If the data set did not require a format, subroutine FBKSP reads the record backspaced over, using subroutine FIOCS, and obtains the control word specifying the number of records within this logical record (see control word format under subroutine FRDNF/FWRNF). Subroutine FBKSP issues an equal number of backspaces (using subroutine FIOCS) before returning to the object program.

EXIT: After execution subroutine FBKSP returns control to the object program.

Subroutine FRWND: Chart PV

Subroutine FRWND implements the REWIND source statement.

ENTRANCE: Subroutine FRWND receives control from the object time execution of a compiler generated linkage. This linkage is generated during compilation when a REWIND source statement is encountered.

OPERATION: Subroutine FRWND calls subroutine FIOCS to select the actual channel and device corresponding to the data set reference number and to issue a rewind control to that channel and device.

EXIT: After execution subroutine FRWND returns control to the object program.

Subroutine FEOFM: Chart PW

Subroutine FEOFM implements the END FILE source statement.

ENTRANCE: Subroutine FEOFM receives control from the object time execution of a compiler generated linkage. This linkage is generated during compilation when the END FILE source statement is encountered.

OPERATION: Subroutine FEOFM calls subroutine FIOCS to select the actual channel and device corresponding to the data set reference number and to issue a write end-of-data set control to that channel and device.

EXIT: After execution subroutine FEOFM returns control to the object program.

Subroutine FSTOP: Chart PX

Subroutine FSTOP implements the STOP source statement.

ENTRANCE: Subroutine FSTOP receives control from the object time execution of a compiler-generated linkage. This linkage is generated during compilation when a STOP statement is encountered.

OPERATION: Subroutine FIOCS is called twice: to initialize the data set for a write; and, to write on the typewriter the message associated with the STOP statement.

EXIT: After execution subroutine FSTOP passes control to subroutine IBEXIT to terminate program execution.

Subroutine FPAUS: Chart PX

Subroutine FPAUS implements the PAUSE source statement.

ENTRANCE: Subroutine FPAUS receives control from the object time execution of a compiler-generated linkage. This linkage is generated during compilation when a PAUSE statement is encountered.

OPERATION: Subroutine FIOCS is called twice: to initialize the data set for a write; and, to write on the typewriter the message associated with the PAUSE statement. Subroutine FPAUS places the system in a "wait" state, until a reply is received from the operator.

EXIT: When the operator's reply is received, subroutine FPAUS returns control to the next sequential instruction of the object program.

Subroutine IBFERR: Chart PY

Subroutine IBFERR (Execution Error Monitor) processes object-time errors (e.g., a permanent tape redundancy).

ENTRANCE: Subroutine IBFERR receives control from the various FORTRAN library subroutines, when they detect object-time errors.

OPERATION: Subroutine IBFERR, using subroutine FIOCS, prints out a message to indicate the type of error detected and the point in the object program where the error occurred.

EXIT: After execution subroutine IBFERR passes control to subroutine IBEXIT to terminate the execution of the object program.

Subroutine IBFINT: Chart PZ

Subroutine IBFINT (Interrupt Processor) processes arithmetic type program errors (e.g., overflow, underflow, divide check).

ENTRANCE: Subroutine IBFINT initially receives control from a compiler generated linkage, which is included in as the initial object program coding to be executed. Subsequent entries into subroutine IBFINT are effected whenever program interrupts occur.

OPERATION: The first time subroutine IBFINT receives control, it initializes the object time 2540 punch error recovery mechanism in the FSD error recovery routine (SERP). In addition IBFINT saves the new program PSW and substitutes its own. This is done to insure that subroutine IBFINT receives control each time a program interrupt occurs. In handling interrupts, subroutine IBFINT determines if the interrupt is the arithmetic type. If not, it loads the saved new program PSW, thereby giving control to the program interrupt routine of the FORTRAN System Director to process non-arithmetic program interrupts. If the interrupt is arithmetic, subroutine IBFINT writes out the old program PSW, using subroutine FIOCS. This PSW can be examined to determine the nature of the interrupt.

If overflow or underflow has caused the interrupt, subroutine IBFINT sets the appropriate indicators, which are referenced by subroutine OVERFL (a library subroutine). If any type of divide check has caused the interrupt, subroutine IBFINT sets the indicator referenced by subroutine DVCHK (a library subroutine).

EXIT: After processing an arithmetic program error, subroutine IBFINT returns control to the point in the object program at which the interrupt occurred. If the program interrupt is not arithmetic, subroutine IBFINT exits to the program interrupt routine of the FORTRAN System Director.

Subroutine FIOCS: Charts QA, QB

Subroutine FIOCS (I/O Interface) handles all I/O requests from other FORTRAN library subprograms.

ENTRANCE: Subroutine FIOCS receives control from the various FORTRAN library subroutines when they request I/O operations.

CONSIDERATIONS: Subroutine FIOCS does not perform the actual I/O operations. It acts as an interface between the subprogram requesting the I/O operation and the FORTRAN System Director (FSD), which actually fulfills the I/O request of the subprogram. Subroutine FIOCS receives and passes the I/O request on to the FSD by means of a Supervisor Call. The FSD performs the requested I/O operation and returns control to subroutine FIOCS.

OPERATION: The operation of FIOCS is comprised of initialization, read, write, and control.

Initialization: Data set initialization is considered as part of the opening section operation. The requested data set reference number is saved and used, until another initialization occurs, for all subsequent read or write requests. If the data set is to be read, a record is read into an I/O buffer, using the FSD. The beginning address of the I/O buffer and the size of the record read are returned to the calling subprogram. If the data set is to be written, an I/O buffer area is located and initialized. The beginning address of this area and its maximum length (in bytes) is returned to the calling subprogram.

Read: A record is read, using the FSD, from the current data set into an I/O buffer. The beginning address of the buffer and the size of the record read are returned to the calling subprogram.

Write: The contents of the last located I/O buffer are written, using the FSD, onto the current data set. If the write was a successful 2540 punch operation, the information just punched and the device parameters are saved for possible punch equipment check recovery procedures. A new I/O buffer area is located and initialized. The beginning address of this area and its maximum length (in bytes) are returned to the calling subprogram.

Control: The qualifying argument passed to subroutine FIOCS is examined and its corresponding operation (backspace, rewind, or end of data set) is performed on the requested data set, using the FSD. If the operation was a backspace, the previous qualifier set during data set initialization for this data set reference is returned to the calling subprogram (to indicate the format requirement). Subroutine FBKSP needs this information to complete the backspacing of a data set that does not require a format.

The other types of control operations (rewind and end of data set) do not require output from subroutine FIOCS.

EXIT: After execution subroutine FIOCS returns control to the calling subprogram.

Subroutine IBEXIT: Chart QC

Subroutine IBEXIT terminates the execution of an object program.

ENTRANCE: Subroutine IBEXIT receives control from subroutines FSTOP, DUMP, EXIT, and IBFERR after it prints an error message; and from the object time execution of a compiler generated linkage for the RETURN statement appearing in the main program.

OPERATION: Subroutine IBEXIT closes all FORTRAN data sets that are open.

EXIT: After execution subroutine IBEXIT passes control to the FORTRAN System Director.

Subroutine IB2540: Chart QD

Subroutine IB2540 handles 2540 punch equipment check retries during object time.

ENTRANCE: IB2540 receives control from the external interrupt after FSD has printed the error message.

OPERATION: IB2540 routine sets up the error device UCB and CSW to indicate a successful punch operation, and changes the FSD return address to return to the IB2540 retry address (IORTRY). It then gives control to the FSD interrupt routine (SNTPIN). Routine SNTPIN will find a successful punch, restore IBCOM registers and I/O device indicators, and then return to the retry entry (IORTRY).

IB2540 will punch the first card and then give control back to the IBCOM I/O Interface Subroutine (FIOCS) to punch the second card.

EXIT: After execution of subroutine IB2540 control is returned to the IBCOM I/O Interface Subroutine (FIOCS).

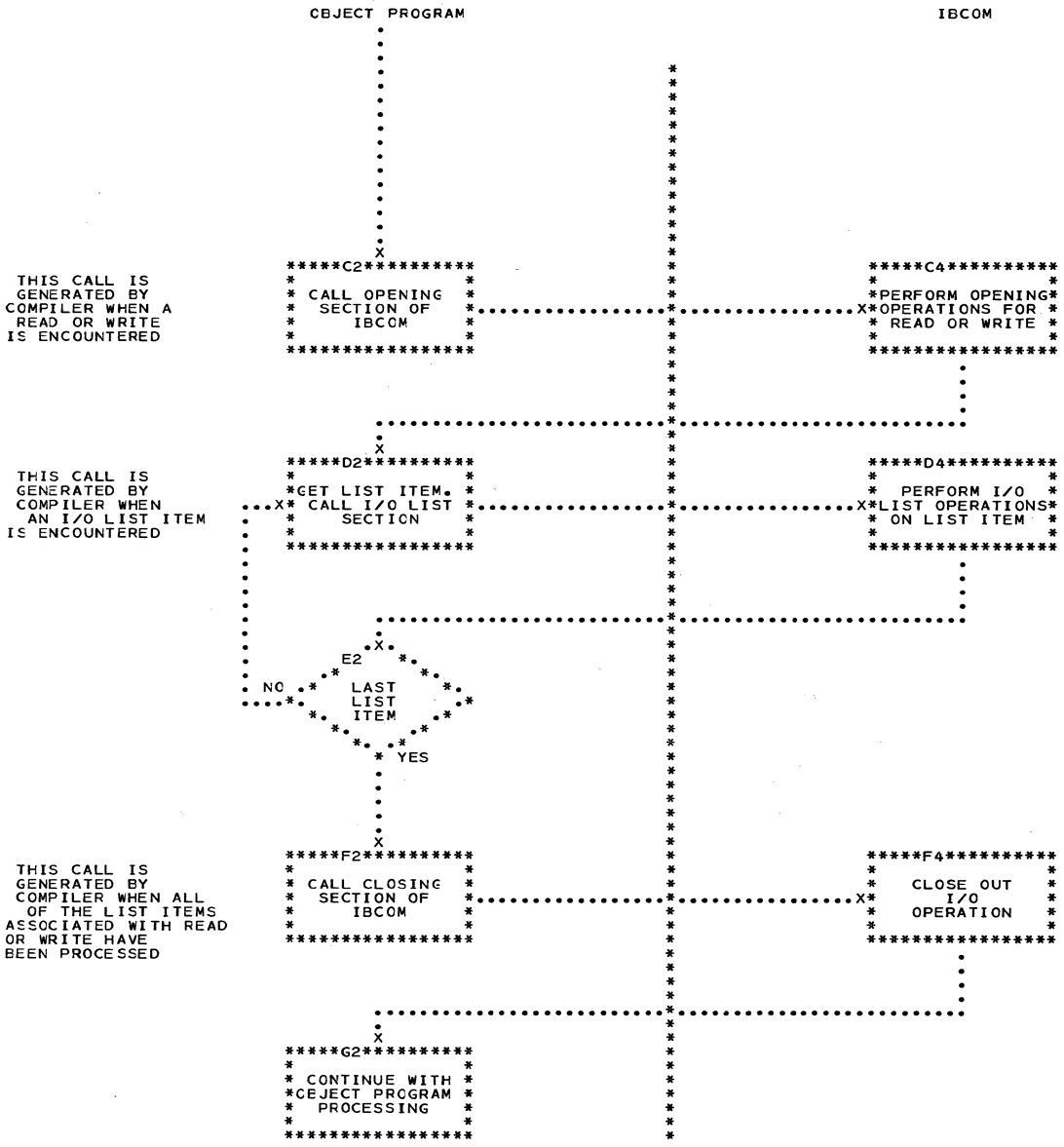


Chart 12. IBCOM-Object Program Logic Diagram

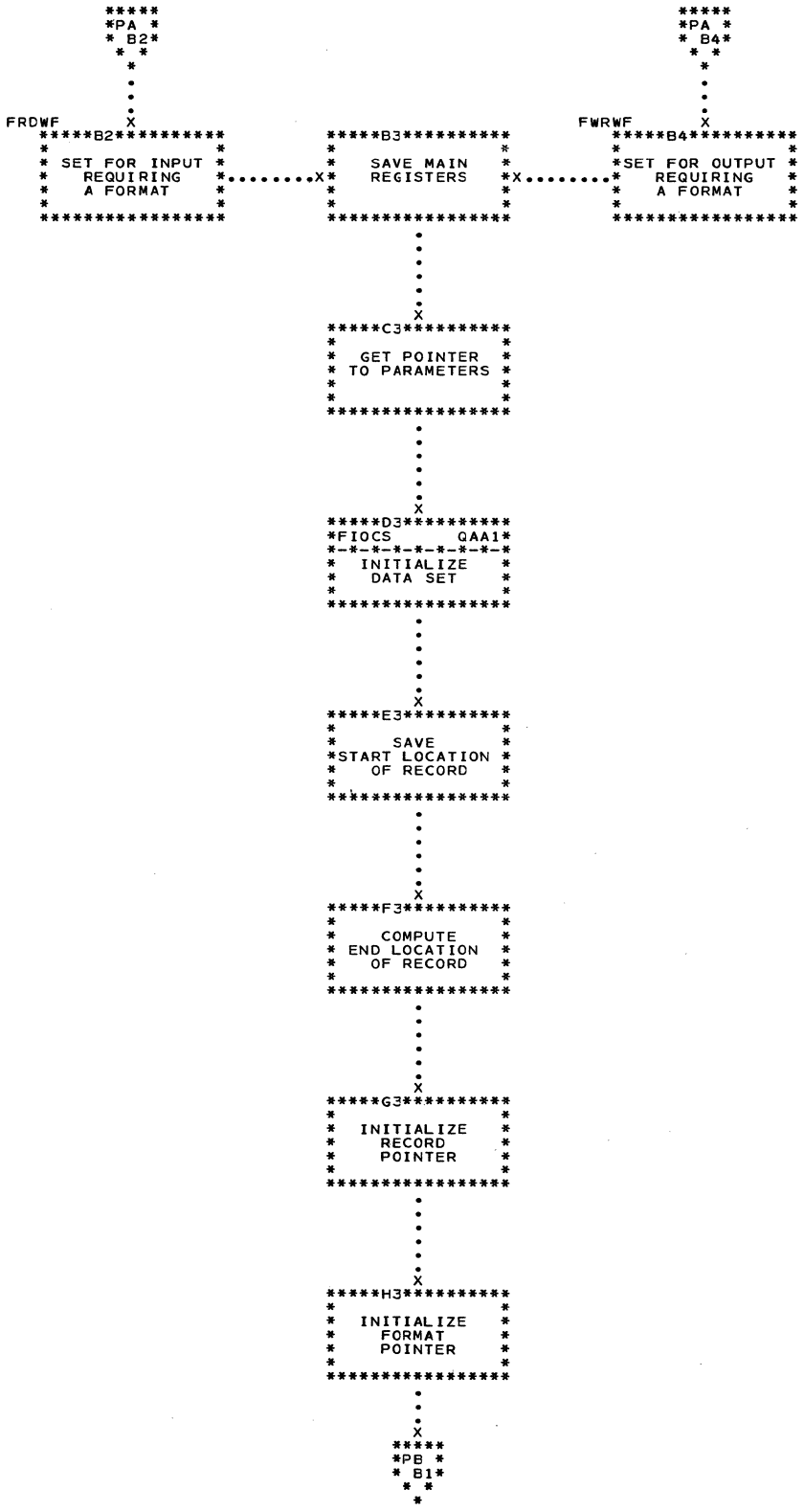


Chart PA. Subroutines FRDWF, FWRWF

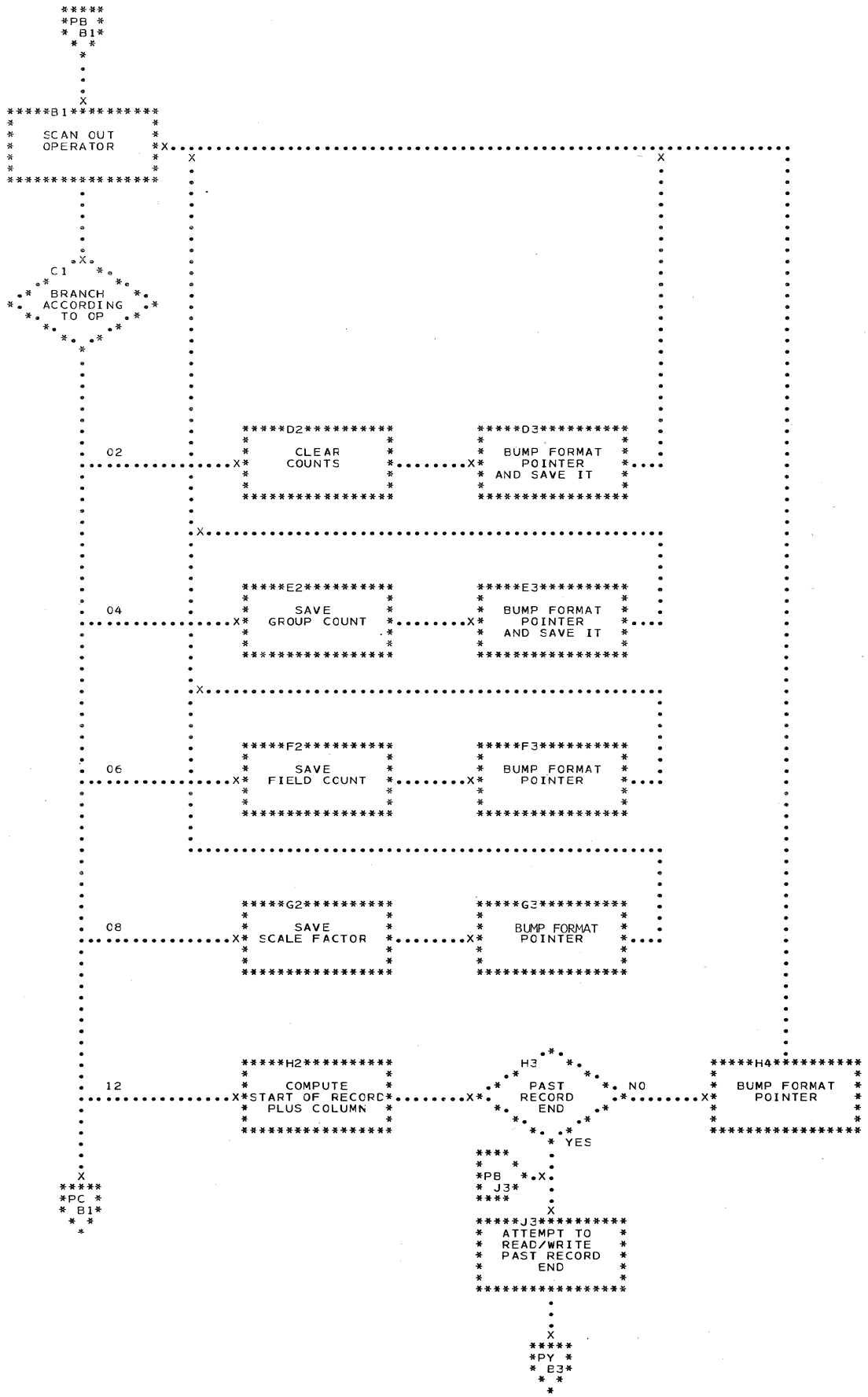


Chart PB. Subroutines FRDWF, FWRWF

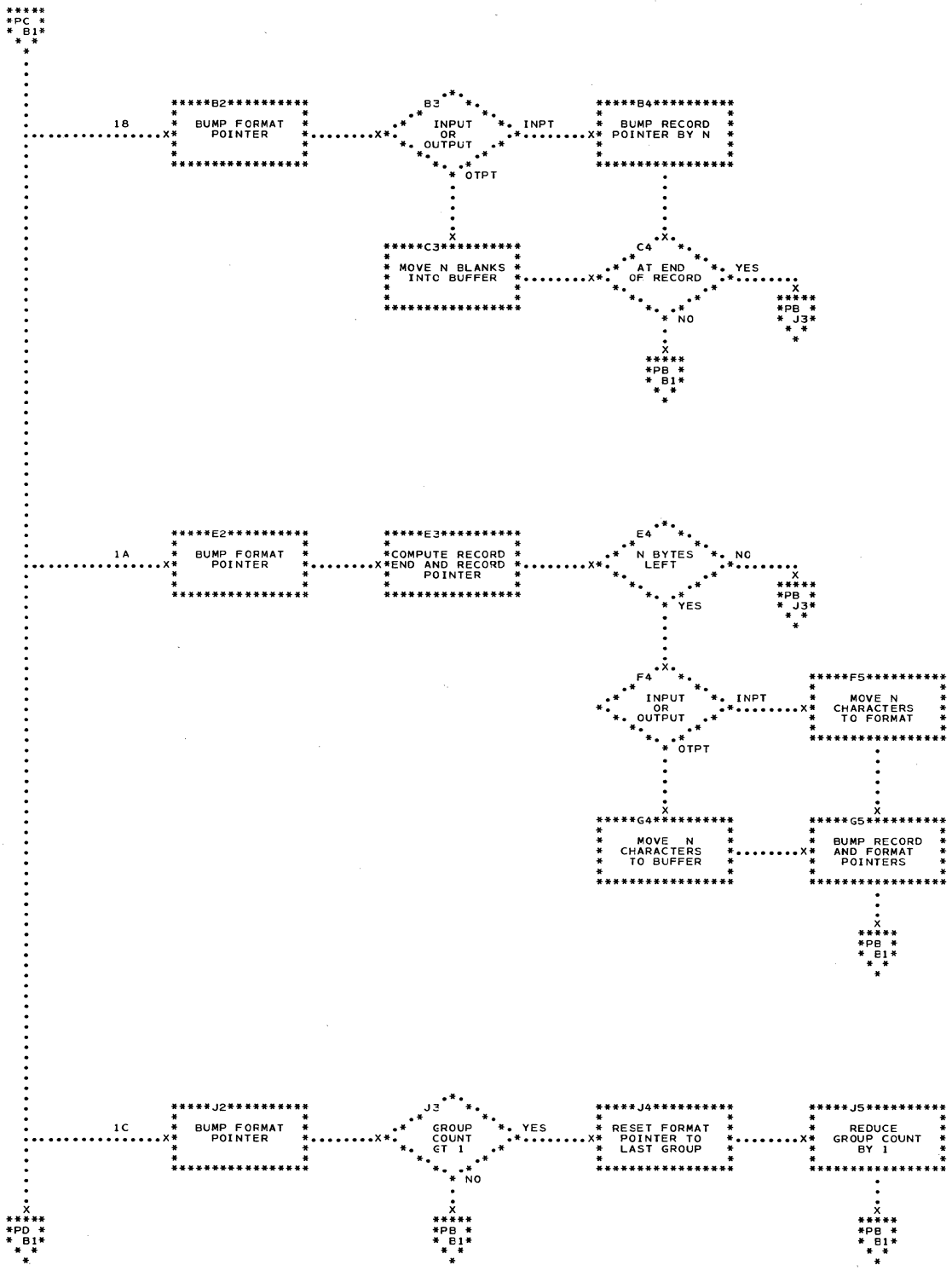


Chart PC. Subroutines FRDWF, FWRWF

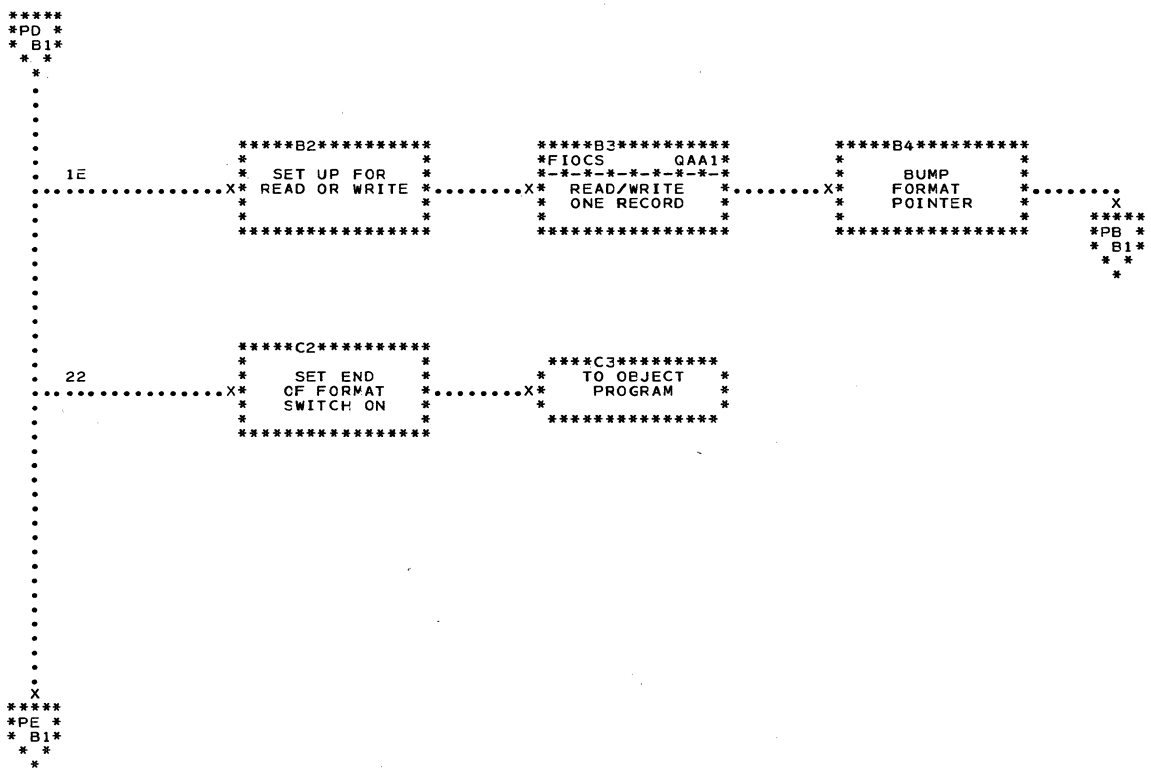


Chart PD. Subroutines FRDWF, FWRWF

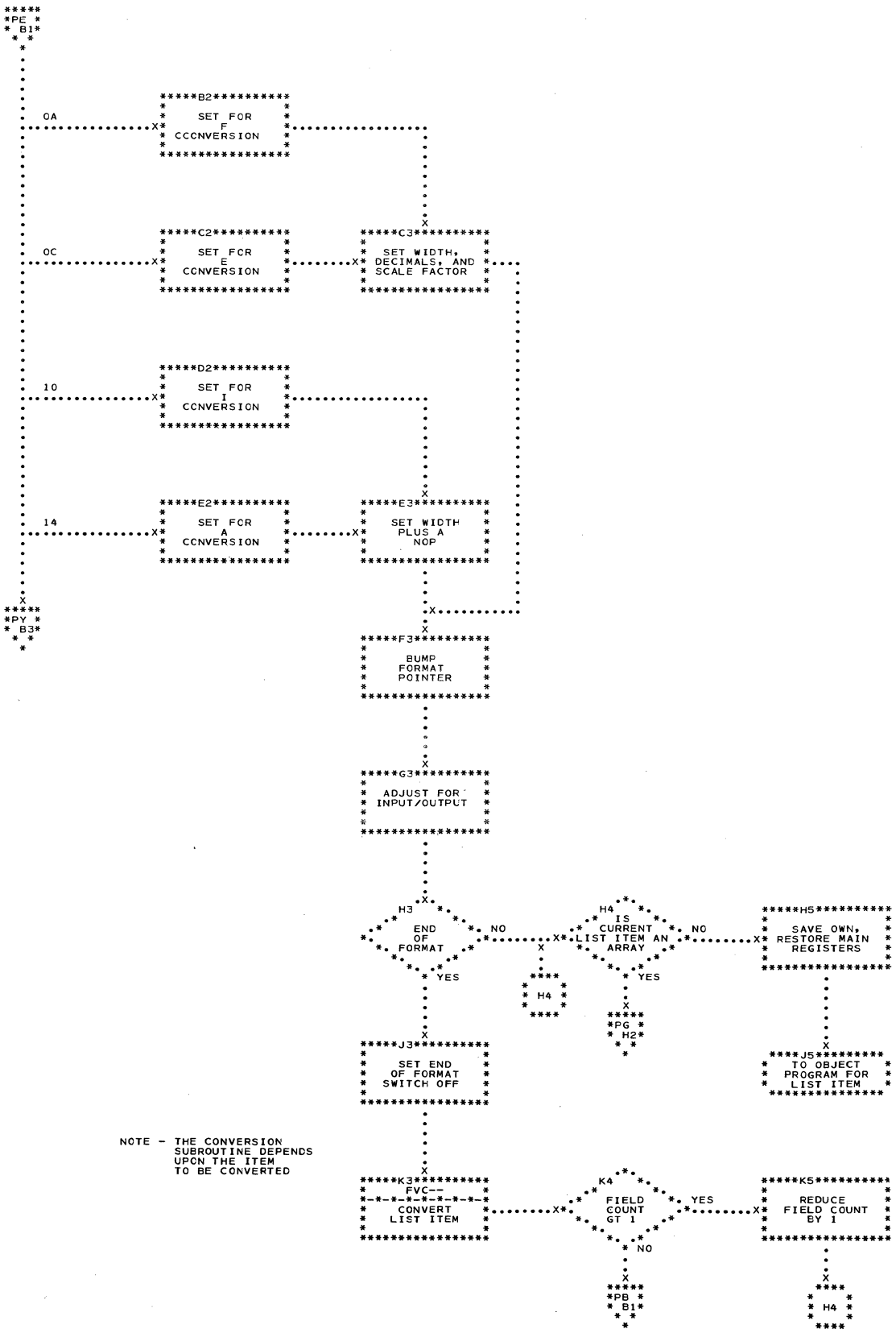


Chart PE. Subroutines FRDWF, FWRWF

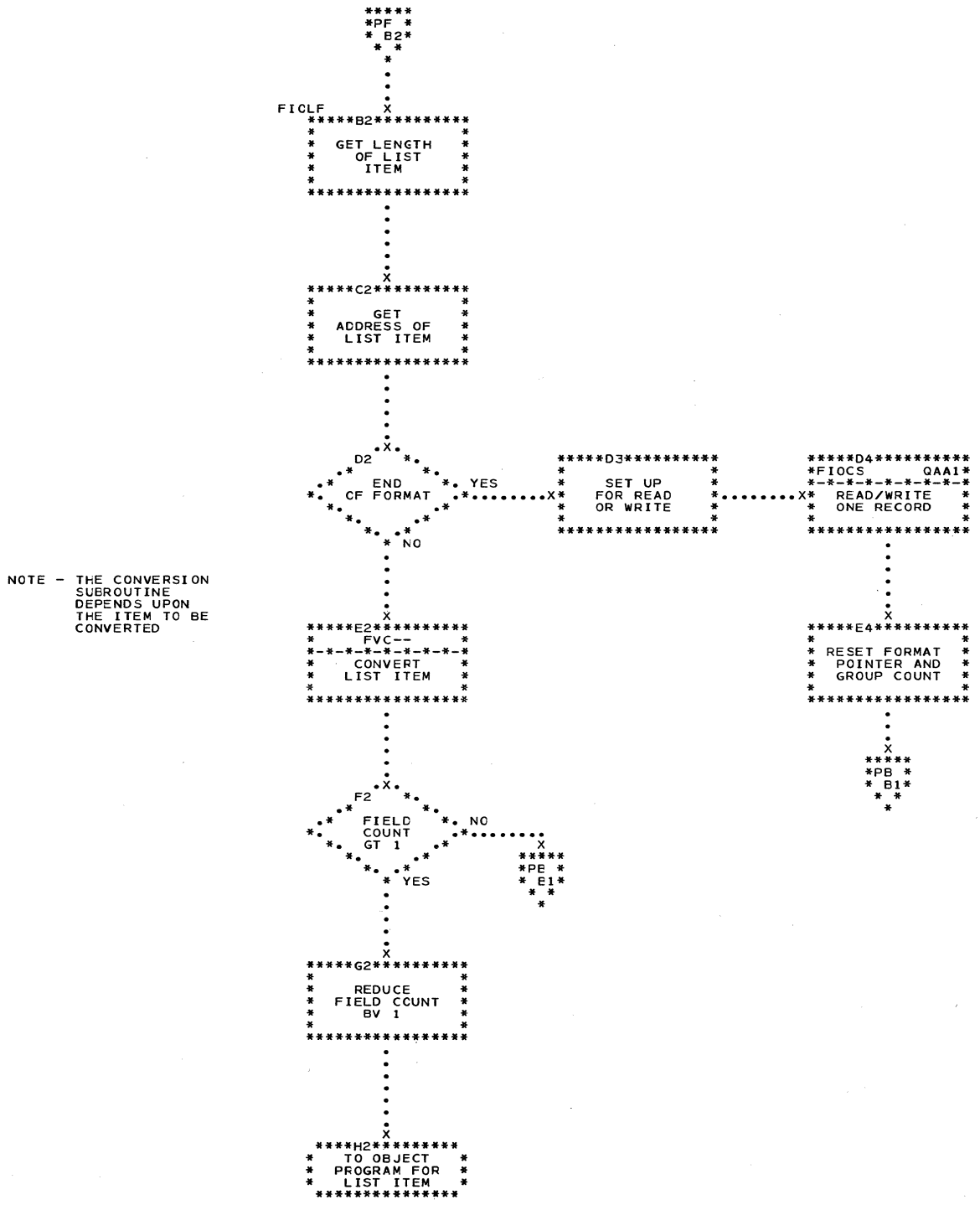


Chart PF. Subroutine FIOLF

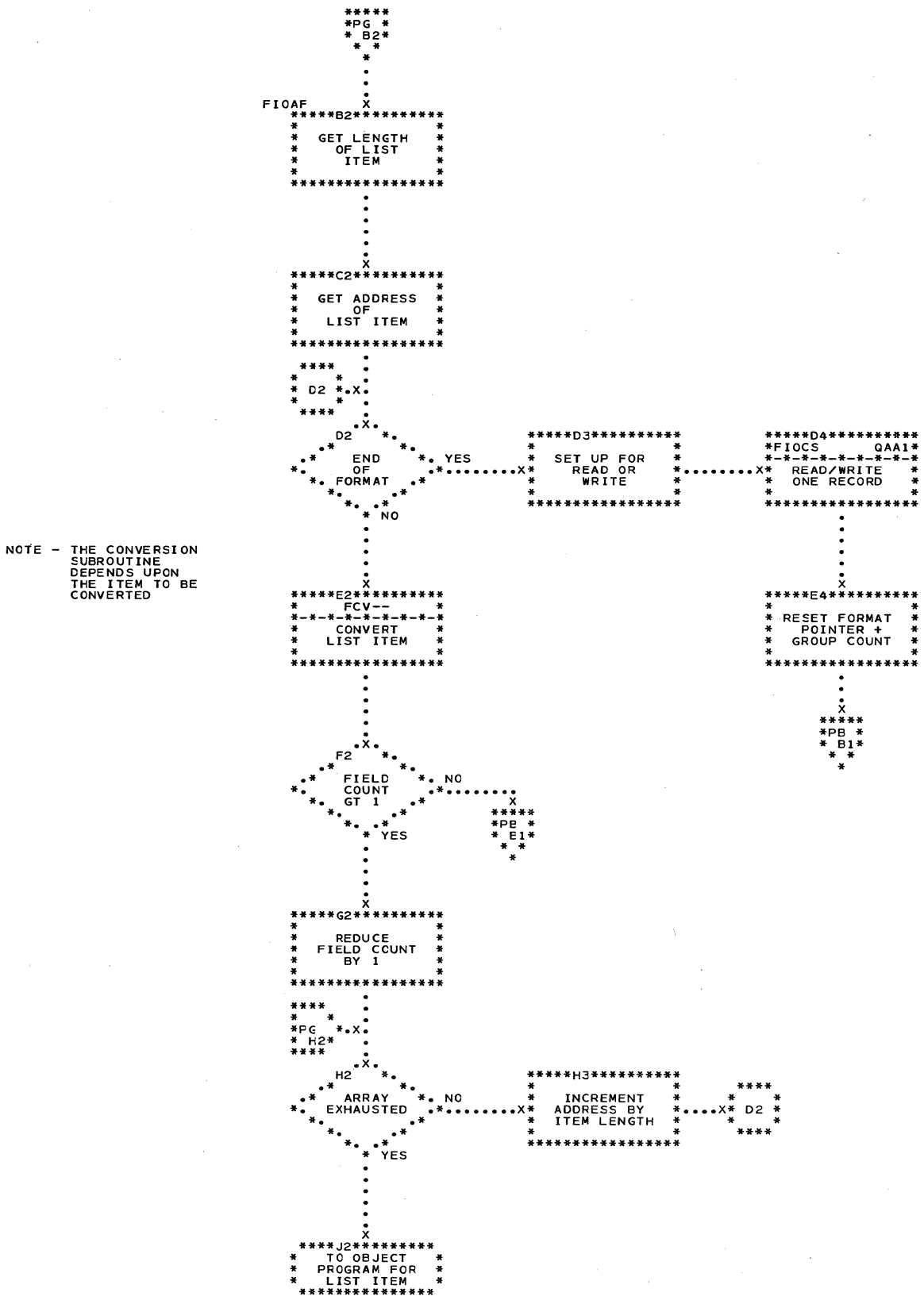


Chart PG. Subroutine FIOAF

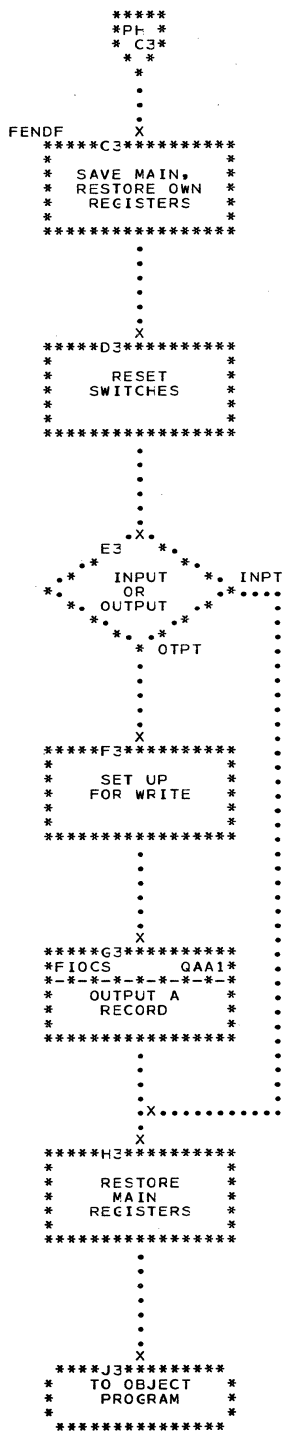


Chart PH. Subroutine FENDF

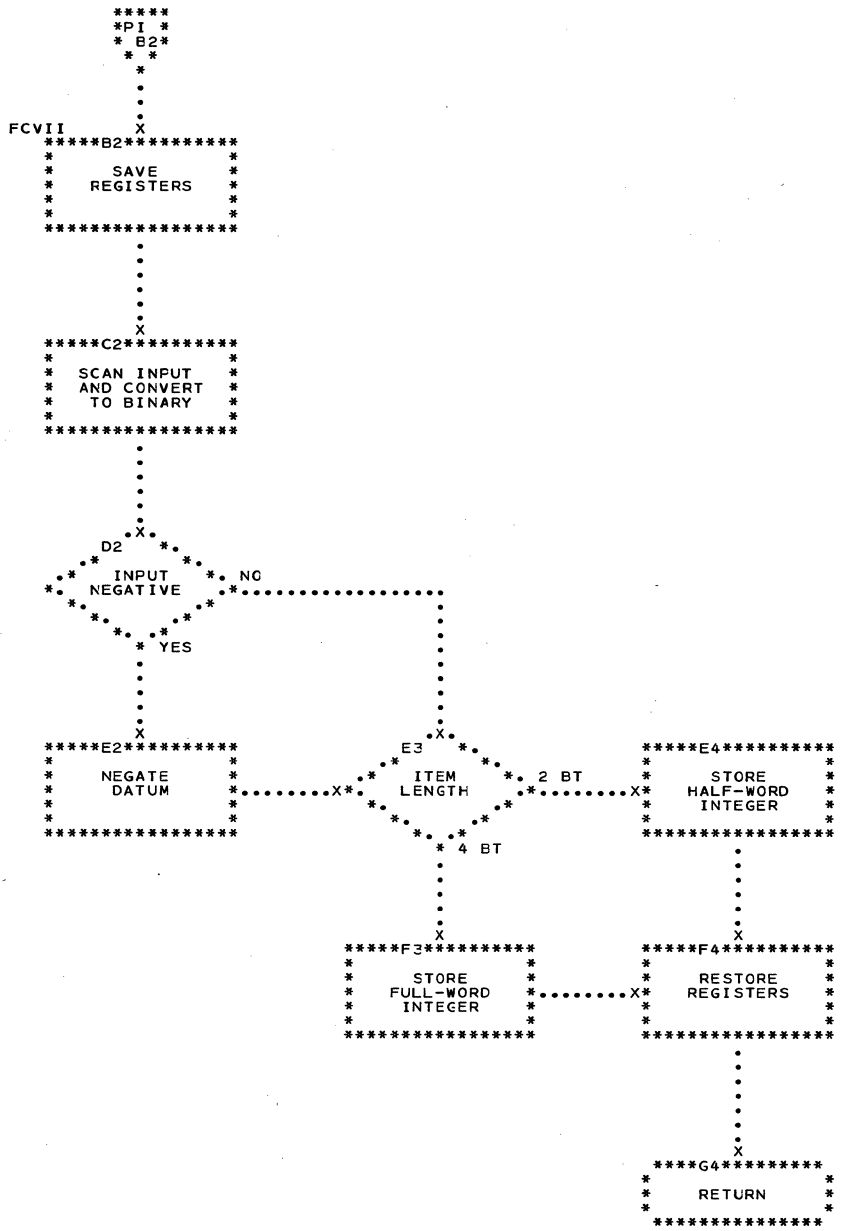


Chart PI. Subroutine FCVII

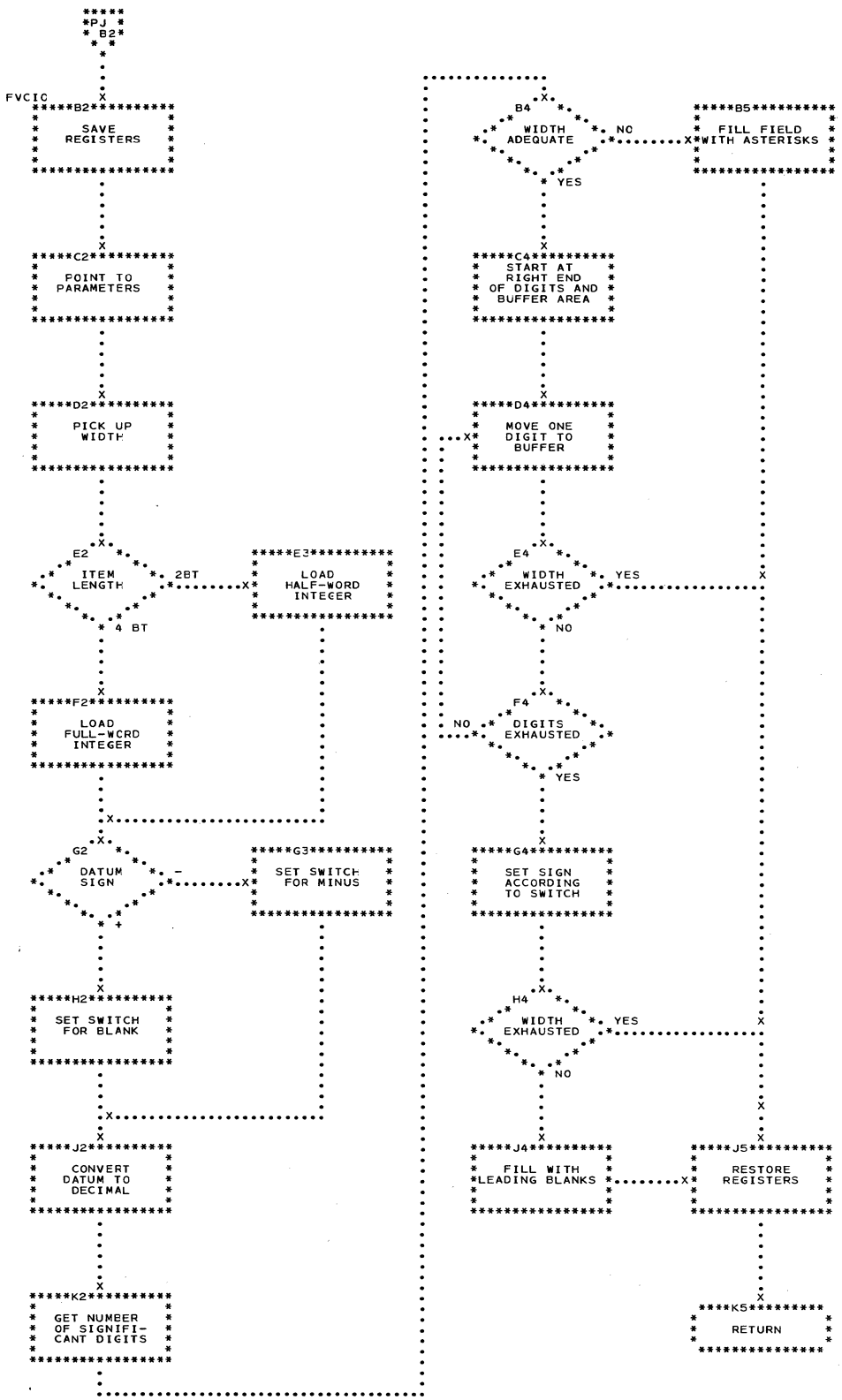


Chart PJ. Subroutine FCVIO

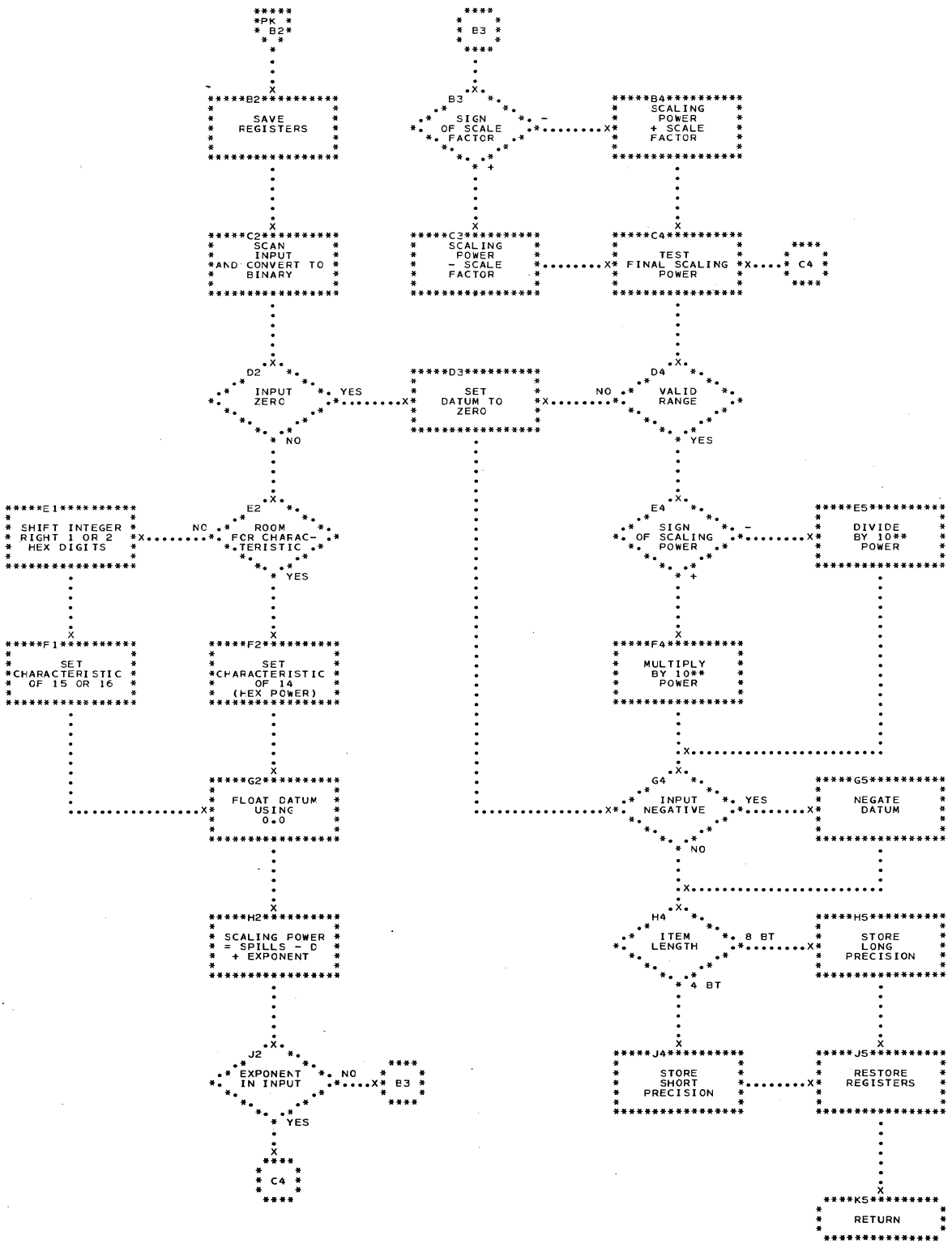


Chart PK. Subroutine FCVFI/FCVEI/FCVDI

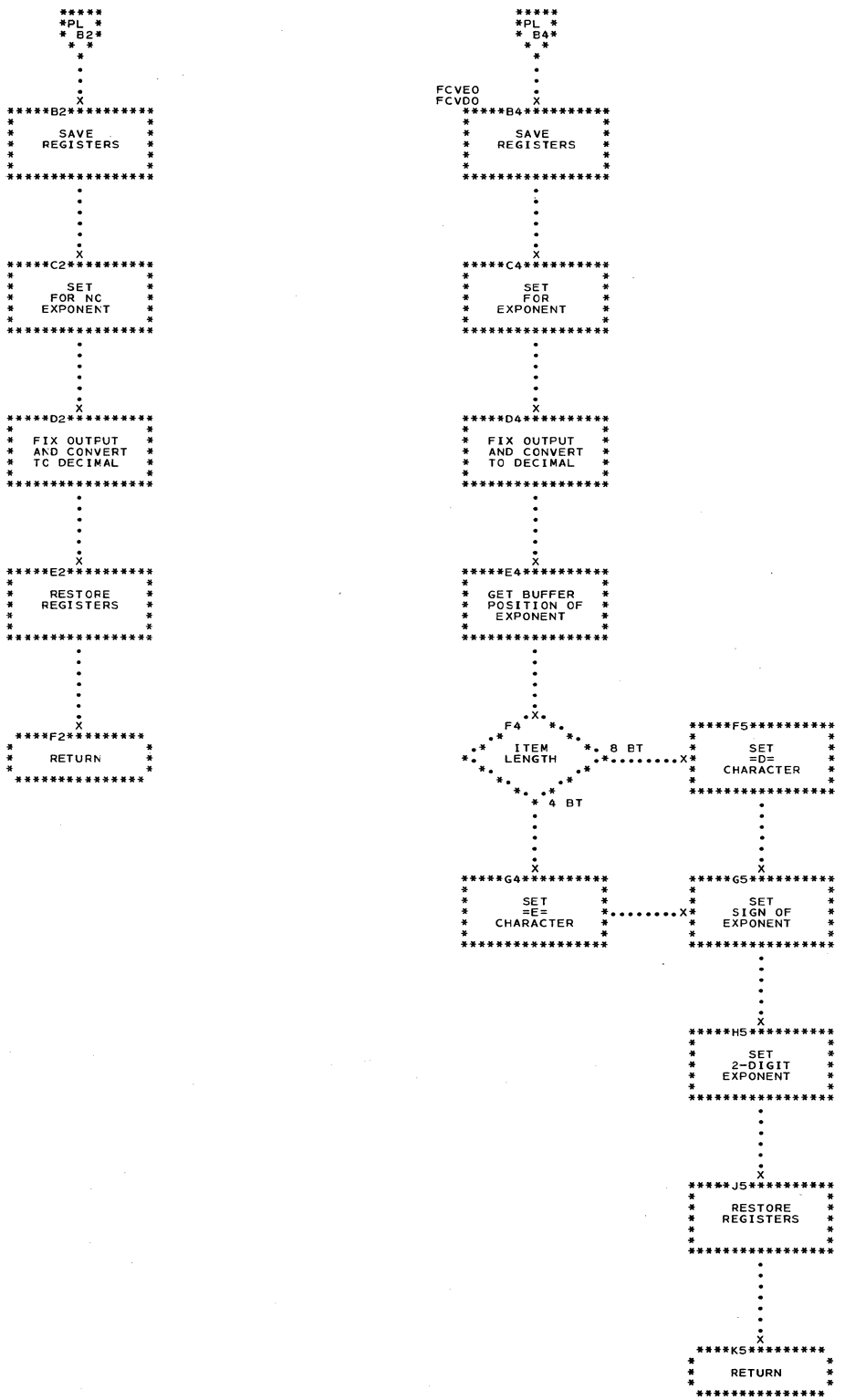


Chart PL. Subroutine FCVFO/FCVEO/FCVDO

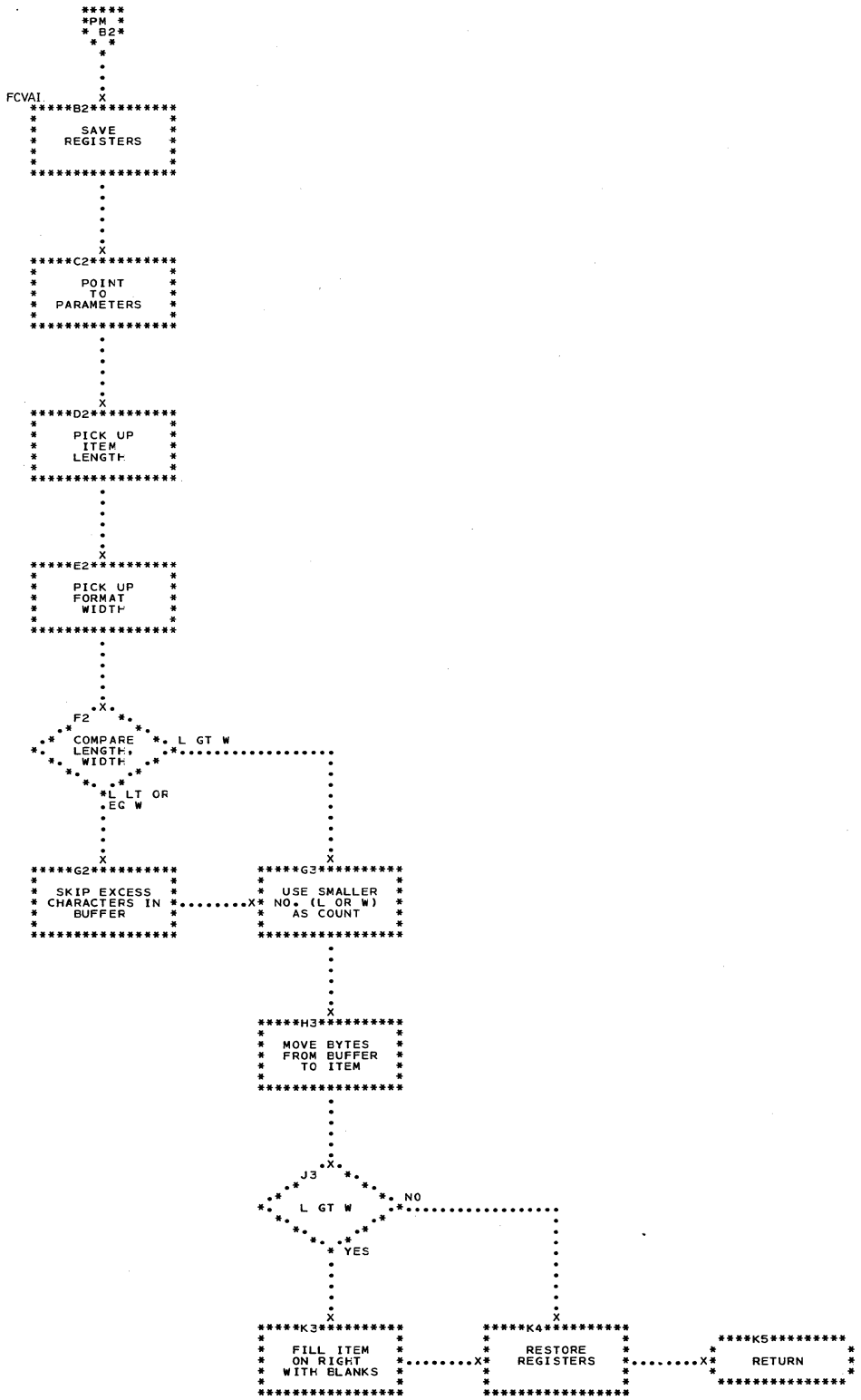


Chart PM. Subroutine FCVAI

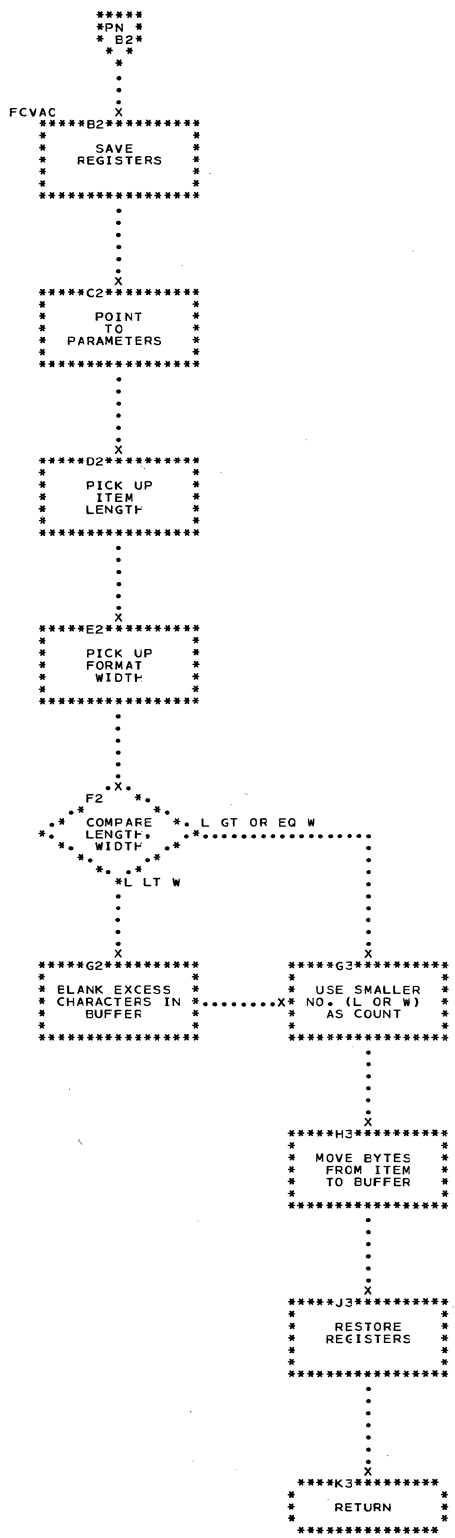


Chart PN. Subroutine FCVAO

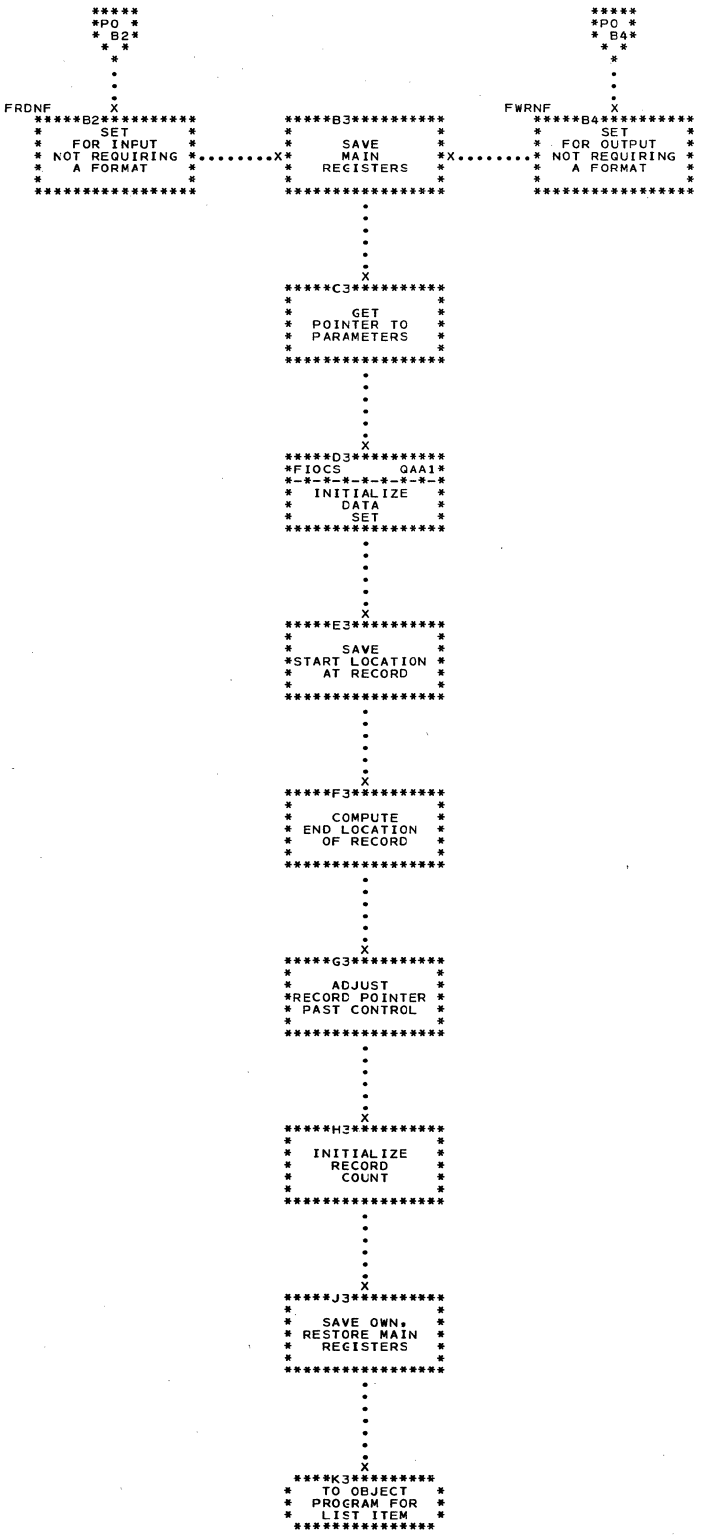


Chart PO. Subroutines FRDNF, FWRNF

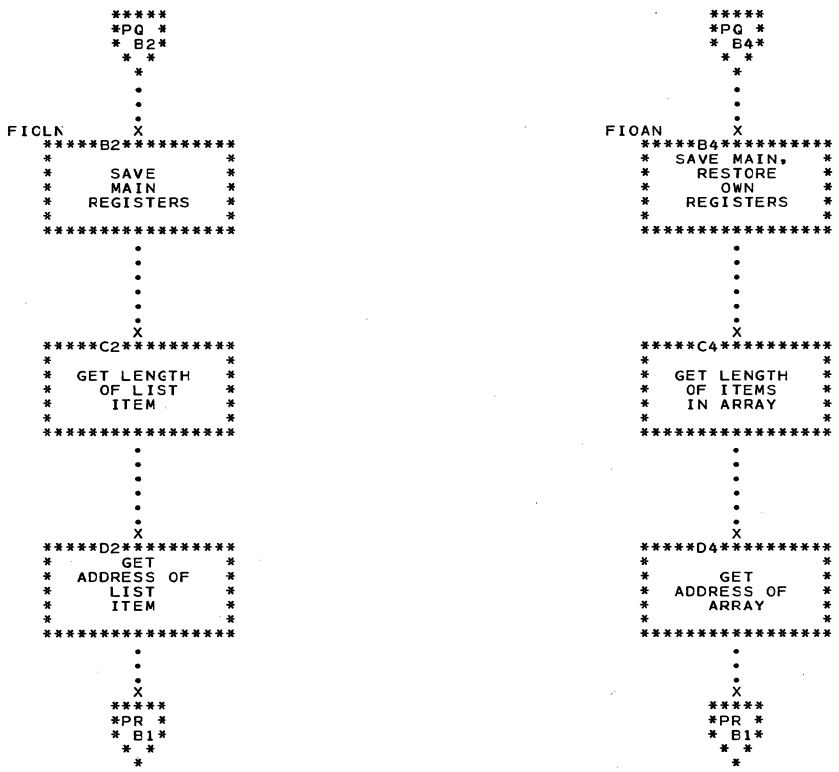


Chart PQ. Subroutines FIOLN, FIOAN

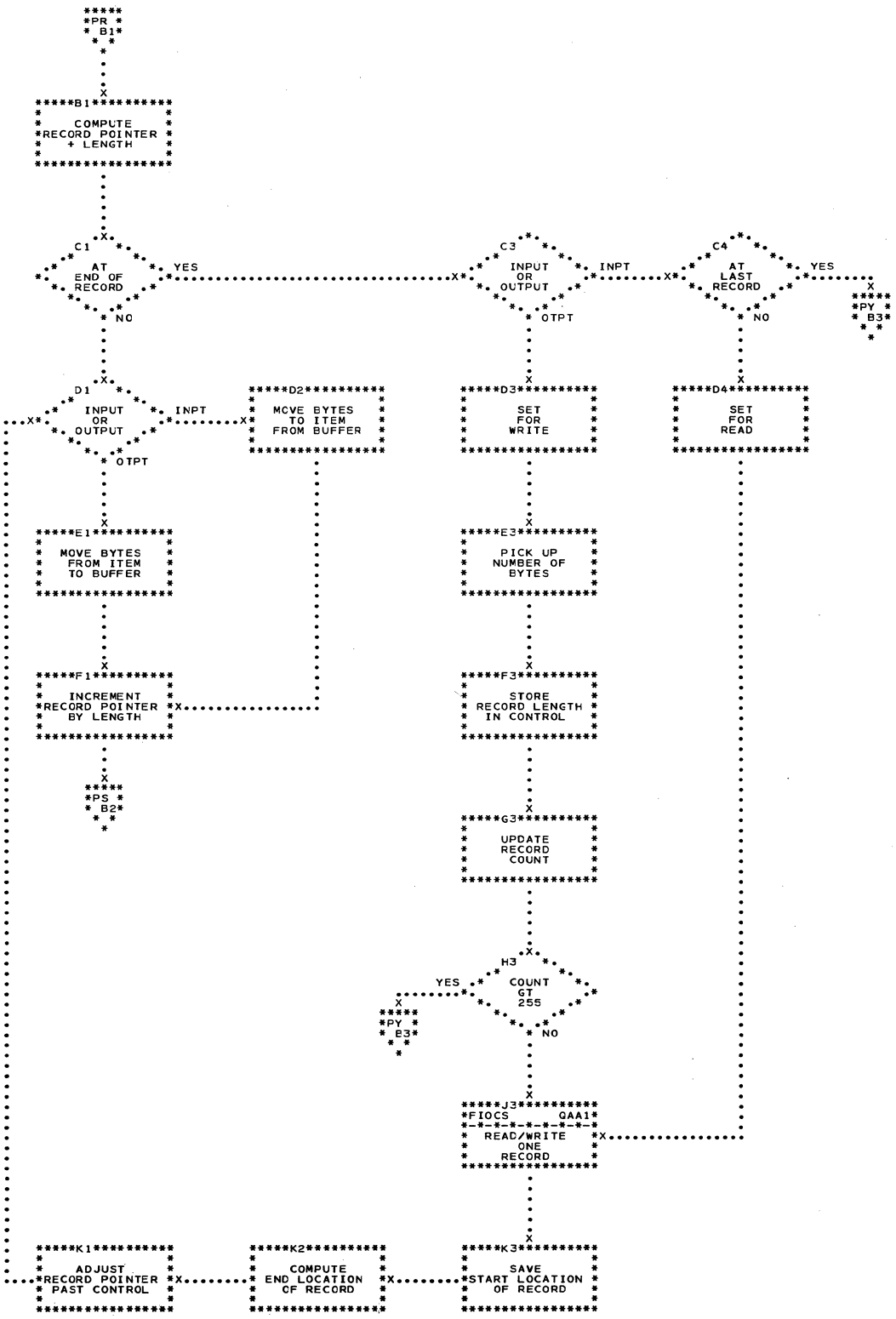


Chart PR. Subroutines FIOLN, FIOAN

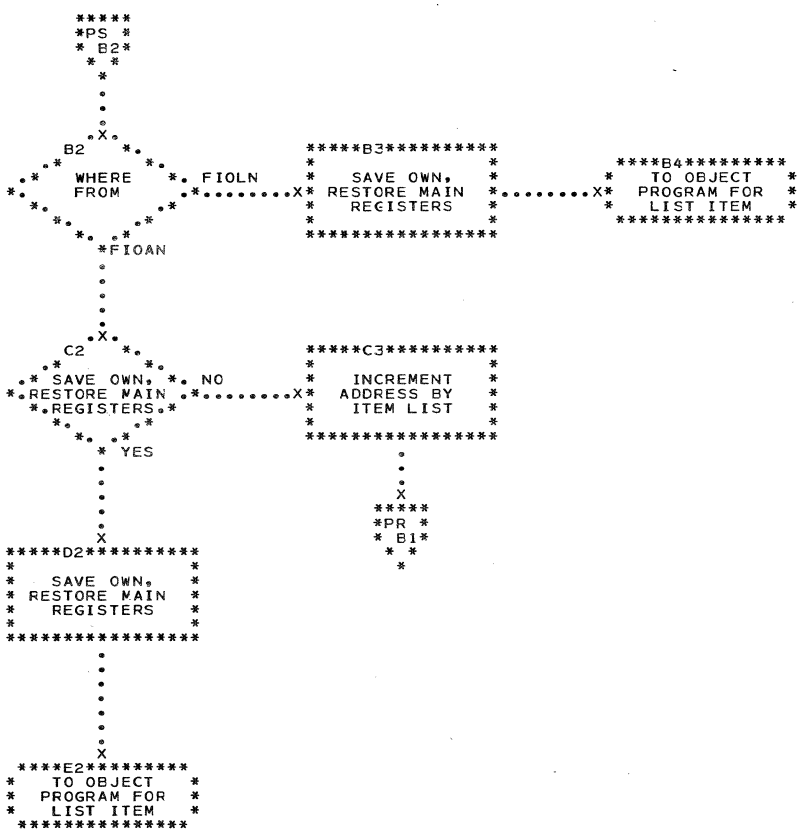


Chart PS. Subroutines FIOLN, FIOAN

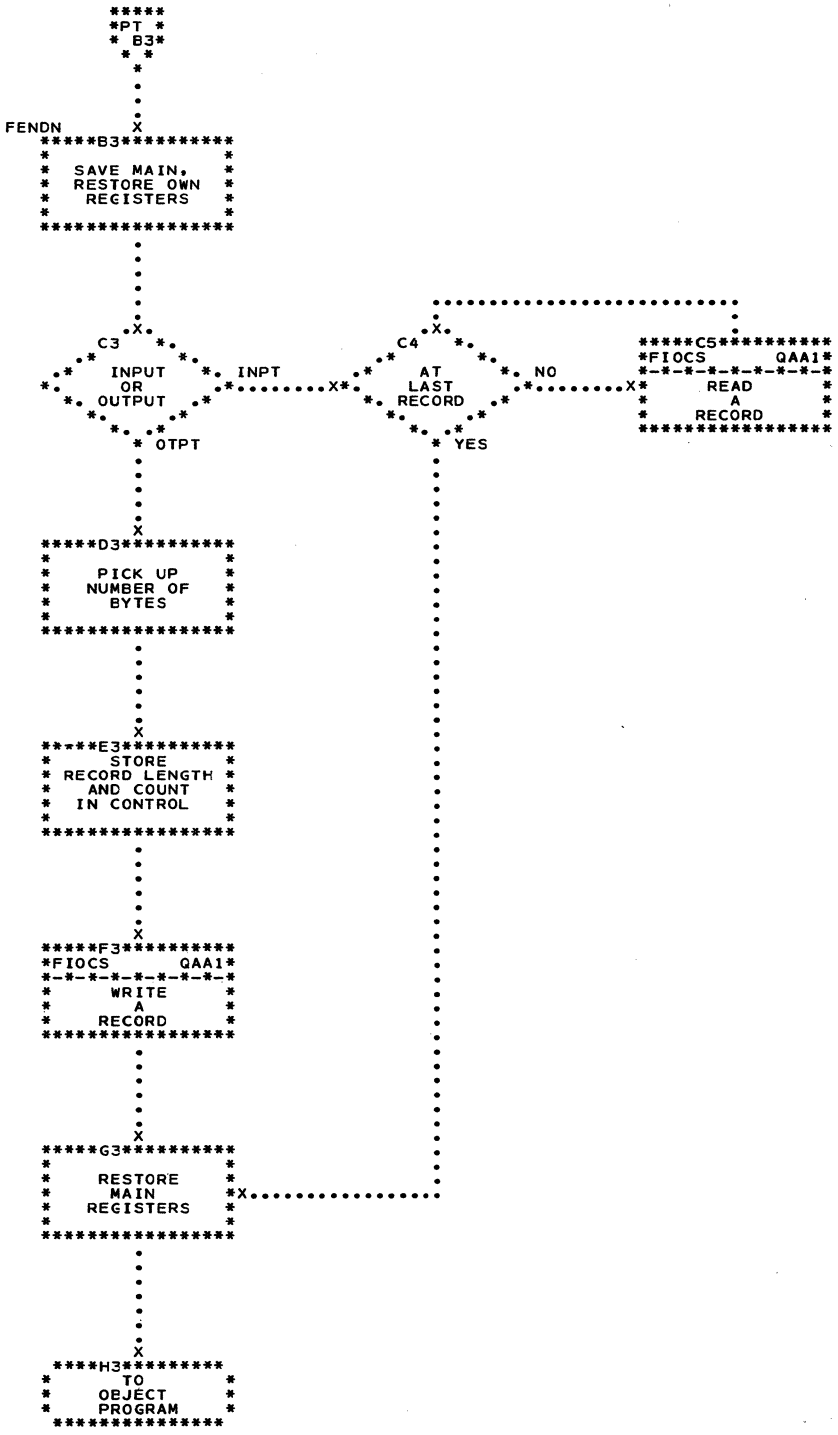


Chart PT. Subroutine FENDN

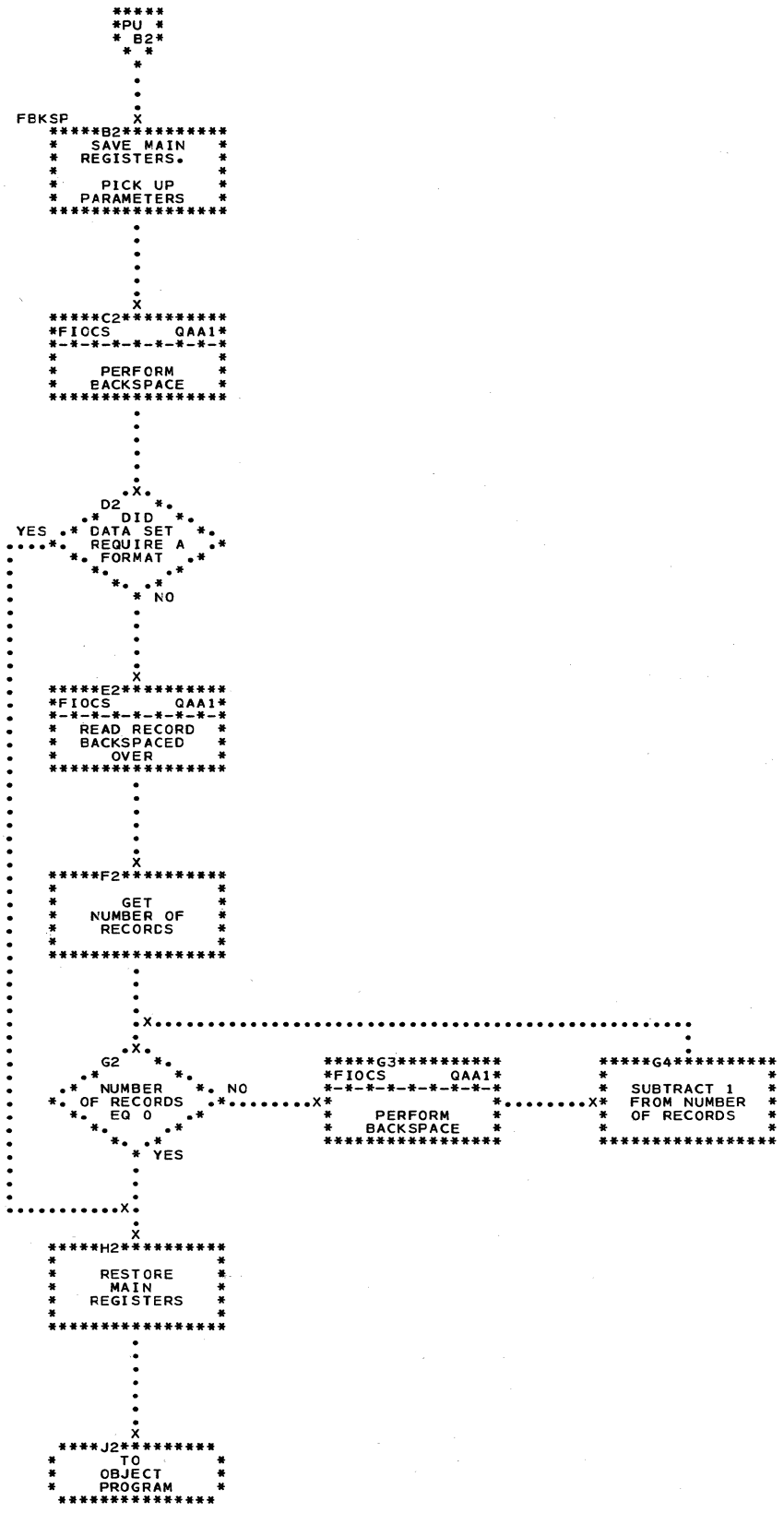


Chart PU. Subroutine FBKSP

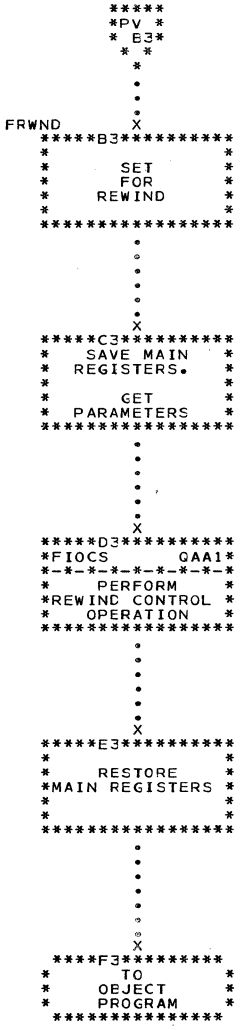


Chart PV. Subroutine FRWND

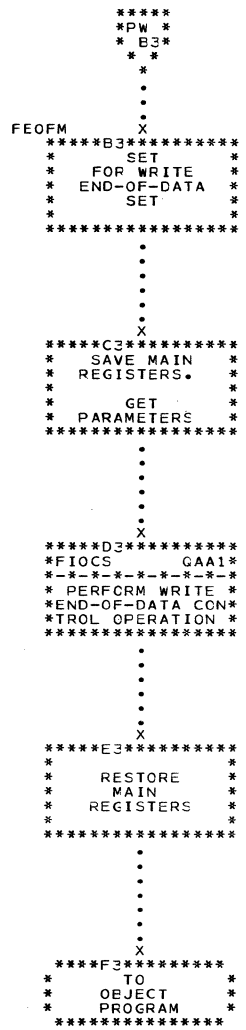


Chart PW. Subroutine FEOFM

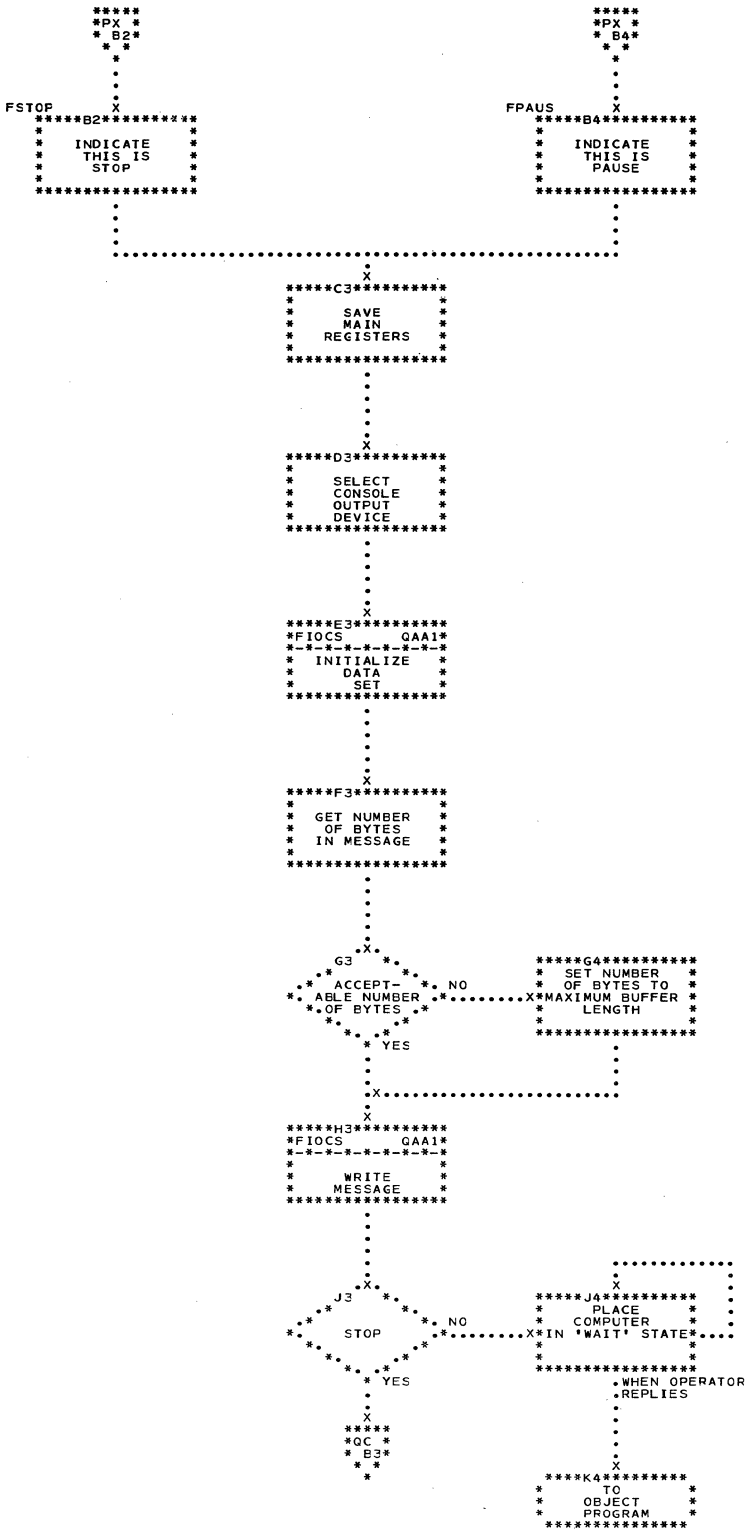


Chart PX. Subroutines FSTOP, FPAUSE

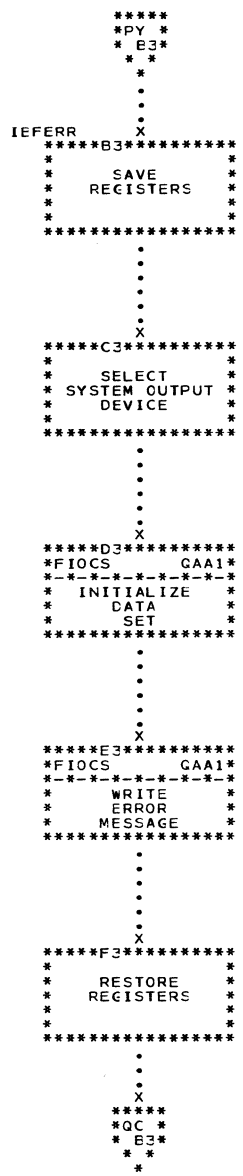
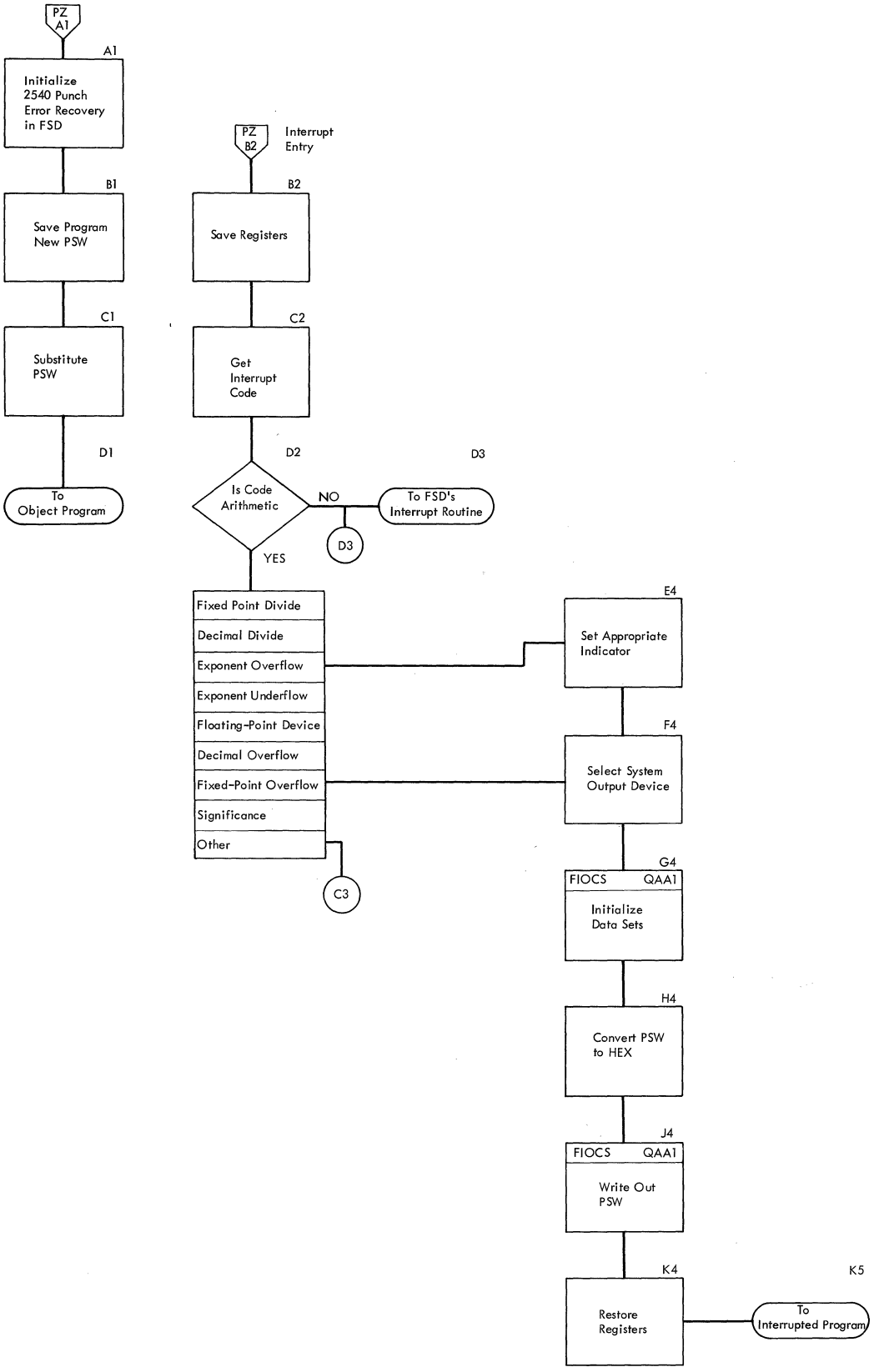


Chart PY. Subroutine IBFERR



● Chart PZ. Subroutine IBFINT

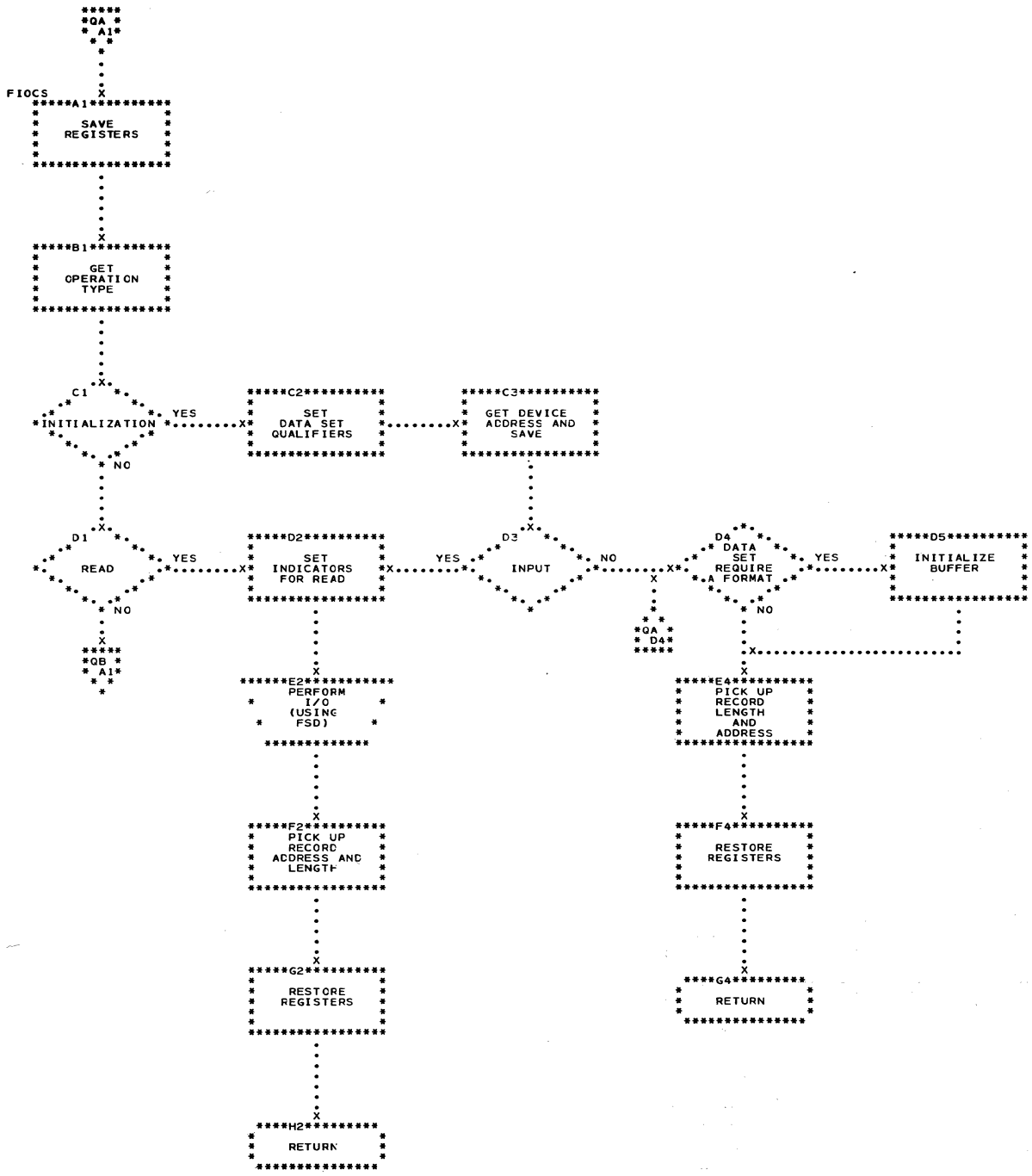
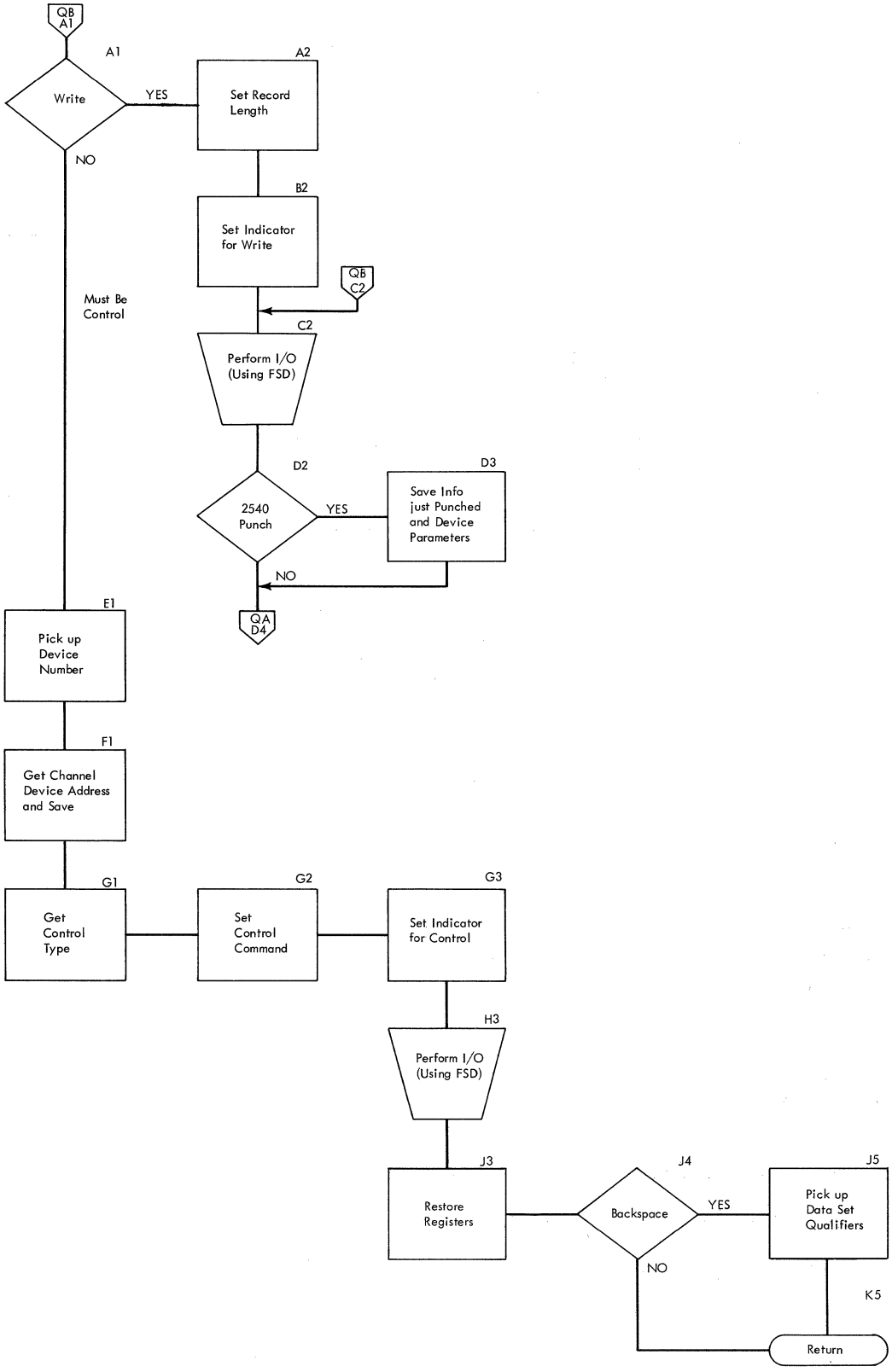


Chart QA. Subroutine FIOCS I/O Interface



● Chart QB. Subroutine FIOCS I/O Interface

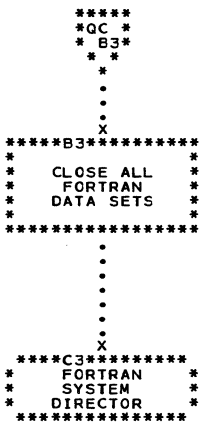
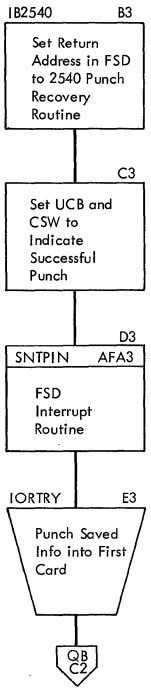


Chart QC. Subroutine IBEXIT



● Chart QD. Subroutine IB2540

PART 5: SYSTEM MODIFICATION

The Basic Programming Support FORTRAN system may be tailored to fit the programming requirements of a particular installation. Any editing of the system is performed via three segments of the system: FORTRAN System Director (FSD), Control Card routine, and editor.

The FSD and Control Card routine are discussed in Part 2; the editor is to be discussed in this part of the manual.

The editor, with its associated routines, makes it possible for the user to revise one or more portions of the system tape by adding, replacing, or deleting features to meet the requirements of his installation. For reference purposes in the subsequent discussions of the various editor routines, a generalized layout of the system tape is shown in Figure 71.

Chart 10, the Editor Overall Logic Diagram, indicates the entrance to and exit from the editor and is a guide to the overall functions of the editor.

An editing process begins with the recognition of an EDIT control card by the CCLASS routine. The FORTRAN System Director loads the editor from the system tape and transfers control to the editor.

The logic of the editor is developed on the basis of the modifications that can be incorporated in the various portions of the system tape. These modifications are indicated on editor control cards. The cards and the affected portions of the system tape are:

Card	System Tape Portion
REPLACE (REP)	Initial Program Load FORTRAN System Director Control Card Routine Compiler Phases Loader Library
SET	FSD - Device Assignment Table - Line Length
AFTER	Library Subprograms
DELETE	Library Subprograms
EDR	Editor IBCOM
ASTERISK	Signals end of processing

As revisions, additions, and deletions are made as a result of information on the control cards, the revised old system tape is copied onto the new system tape or tapes.

The editing process ends when the editor recognizes the end of data set on the input device or when it encounters a card which indicates that no more editing is to be done. The editor then gives control to the FORTRAN System Director.

ROUTINES

Initialization for the editor is represented in Chart MA. Classification of the current control card to be processed by the editor is represented in Chart MB. The resultant processing, depending upon the current control card, is represented in Charts MC through MP.

START Routine: Chart MA

The START (EDIT Control Card) routine forms a device assignment table for the new system tapes, places the object machine size in the communications area; rewinds the old and new system tapes; and makes the Initial Program Load available for processing.

ENTRANCE: The START routine receives control from the FORTRAN System Director, after the CCLASS routine has recognized an EDIT control card and called the FORTRAN System Director to find and load the editor from the system tape.

CONSIDERATIONS: The EDIT control card is read by the CCLASS routine and the information on the card is stored in the FORTRAN System Director buffer, where it is retrieved by the START routine.

Unlike the standard control cards, the format of the EDIT card is relatively fixed. The first field is either blank or contains the specification of machine size in bytes. The data set reference number(s) are specified after the first field.

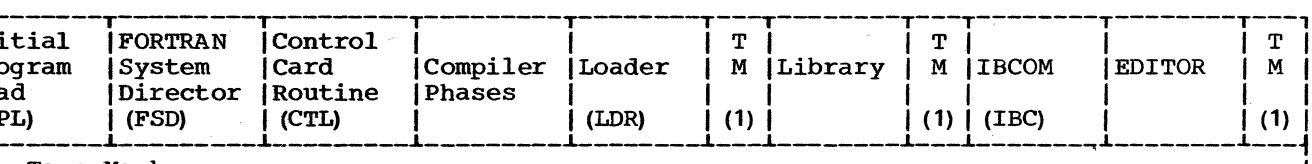


Figure 71. System Tape Layout

If the size field is blank, the routine retains the old machine size in the communications area and starts to search for the data set reference numbers. When the first blank is encountered, the routine will finish any other processing and exit to read another card.

OPERATION: Following initialization, the routine moves the information on the EDIT control card from the FORTRAN System Director buffer area to the editor buffer area and rewinds the old system tape. If the edit control card specifies a GO option, an indication of this edit and go condition is set for the ASTRSK routine. The START routine then lists the data set reference numbers in packed form, and tests the size field for a blank.

If the size field is not blank, a check is made to determine that the specified size is a valid machine size. If the size is valid, it is placed in the communications area. An error message is written for an invalid size specification, and the job is aborted. Also, if the size field is blank, the old machine size in the communications area is not disturbed.

The data set reference numbers are loaded into the device assignment table, the tapes to be used for new systems tapes are rewound, and the Initial Program Load is read in from the old system tape.

EXIT: After the Initial Program Load is read in, control is transferred to the RDACRD routine.

ROUTINE CALLED: During execution, the START routine calls the RDOSYS routine.

RDACRD Routine: Chart MB

The RDACRD (Classification) routine reads a system maintenance control card, determines if it is a valid control card, and interprets the type of control card for transfer to the appropriate processing routine.

ENTRANCE: The RDACRD routine receives control initially from the START routine. However, control is also returned to it from every card processing routine which it calls, with the exception of the ASTRSK routine.

CONSIDERATIONS: The RDACRD routine has few functions of its own that directly affect the system. It merely determines, on the basis of the type of card it reads in and the setting of various switches, the proper processing routine and transfers control to that routine. Under normal conditions, control is returned to the RDACRD routine from the called routine so that the next card may be read in.

OPERATION: Initially, the RDACRD routine reads a card and prints its contents. If an end of data set is encountered, the ASTRSK routine is called.

If an end of data set is not encountered, the routine determines if the card is an ASTERISK control card. If it is, the ASTRSK routine is called. If the card is not an ASTERISK control card, the RDACRD routine determines if the old system tape is at editor.

If the tape is at editor, it is an indication that no more editing is required. Thus, the RDACRD routine seeks only an ASTERISK control card. Any other control card is considered invalid. If a card other than an ASTERISK control card is read, the routine prints an error message for an invalid control card and then returns to FSD as end of job.

If the tape is not at editor, the routine checks for a loader control card. A loader control card, if present, is examined for a phase name. The presence of a phase name requires that one or more parts of the compiler be revised, and control is transferred to the T92CMP routine. If no phase name exists, the loader control card is for revision of one or more library subroutines, and control is transferred to the T92LB1 routine.

A card other than a loader control card, causes the RDACRD routine to check for a SET control card, which is used for revision of the FORTRAN System Director on the system tape. A SET control card causes the routine to transfer control to the SET routine.

If the card is a SET control card and the FORTRAN System Director has not been copied from the old system tape, the routine calls the COPYC routine. This routine uses FSD as the phase name, to allow the FORTRAN System Director to be revised on the system tape. Control is then transferred from the RDACRD routine to the SET routine to accomplish the revisions. The RDACRD routine then returns to read another card.

If the card is not a SET control card, a check is made for a DELETE control card. A DELETE card causes transfer of control to the DELET routine. When control is returned to the RDACRD routine, another card is read.

If the card is not a DELETE control card, a check is made for an AFTER control card. If the card is an AFTER control card, control is transferred to the AFTER routine, and then returned to the RDACRD routine to read in another card.

If the card is not an AFTER control card, the possible legitimate control cards (ASTERISK, LOADER, SET, DELETE, and AFTER) have been checked. An error message is, therefore, printed to indicate an invalid control card, and the routine returns to FSD to end the job.

EXIT: Following the end of data set and/or processing of an ASTERISK control card, the ASTRSK routine calls the FORTRAN System Director to terminate the job. If no more editing is required, but the card read is not an ASTERISK control card, an error message is printed and the FORTRAN System Director is called to terminate the job.

ROUTINES CALLED: During execution, the RDACRD routine references any one of the following processing routines:

1. ASTERSK Control Card routine.
2. T92CMP routine.
3. T92LB1 routine.
4. SET routine.
5. DELET routine.
6. AFTER routine.

AFTER Routine: Chart MC

The AFTER (AFTER Control Card) routine processes the insertion of library subroutines in the compiler library, following the subroutine specified on the AFTER control card. The system tape is copied on a new system tape, up to and including the specified subroutine. Text cards, containing the additional subroutine or subroutines are copied onto the new system tape. The RDACRD routine is then called.

ENTRANCE: The AFTER routine receives control from the RDACRD routine when an AFTER control card is encountered.

OPERATION: The AFTER routine assumes that the AFTER control card is followed by cards concerned with a library subroutine. Initially, the routine examines the AFTER control card and lists the names of the subroutines to be inserted, which are on that card. The AFTER control card also contains the name of the library subroutine after which new subroutines are to be inserted. The routine checks this name against a directory of subroutine names which have already been copied into the new library.

If the subroutine name on the AFTER card is in the directory, the routine further checks whether this name is the last entry in the directory. If it was the last entry up to this time, the routine turns on a switch to indicate new subroutines are to be inserted and returns to the RDACRD routine to begin reading new subroutines from the card reader. If the subroutine name is in the directory, but not the latest entry, the routine name is out of sequence; an error message is given. A return to the RDACRD routine is made to read another card. The listed subroutines are inserted at the end of the library.

If the AFTER name is not in the directory, the routine transfers control to the COPYCL routine to copy the old system tape up to, but not including, the AFTER-named subroutine.

Control returns to the AFTER routine, which checks to see if the entire library has been copied. If it has, the AFTER-named subroutine, which would have stopped the copying, was not found. The routine prints an error message to this effect and exits to the RDACRD routine to read another card. The listed subroutines are inserted at the end of the library.

If the entire library has not been copied, the next subroutine on the old system tape (the AFTER-named subroutine) is copied onto the new system tape(s). A switch is set to indicate new subroutines are to be inserted. Then, the AFTER routine exits to the RDACRD routine.

EXIT: The AFTER routine exits to the RDACRD routine.

ROUTINE CALLED: During execution the AFTER routine calls the COPYCL routine.

ASTRSK Routine: Chart MD

The ASTRSK (ASTERISK Control Card) routine terminates editing of the system tape and copies the remainder of the old system tape onto the new system tape or tapes. Under certain conditions, the ASTRSK routine merely makes new copies of the old system tape.

ENTRANCE: The ASTRSK routine receives control from the RDACRD routine if the RDACRD routine recognizes either an ASTERISK control card or an end of data set.

CONSIDERATIONS: The only data on the ASTERISK control card is /*, which is necessary for recognition. An ASTERISK control card is not necessarily required

for copying tapes. For example, if an EDIT control card has been recognized, and if the START routine encounters an end of data set, the old system tape is copied. Normally, the conditions for copying a system tape are an EDIT control card, followed by an ASTERISK control card. If there is an EDIT control card and an end of data set, there is nothing to replace or delete on the system tape; therefore, the tapes are simply copied.

OPERATION: The ASTRSK routine checks first whether the read position of the old system tape is at the editor or IBCOM. If it is not, the routine sets a fictitious library subroutine name so that copying is not halted by any recognizable subroutine name. The COPYCL routine is called to copy the old system tape, to the end of the library, onto the new system tapes. Following this, control returns to the ASTRSK routine, which writes end of data set on the new system tapes.

If the read position of the old system tape is presently at the editor or IBCOM, the routine determines if IBCOM has already been copied onto the new system tapes. If it has not, it is copied.

Next, the routine reads in the editor from old system tape and copies it onto the new system tapes. The routine then writes another end of data set.

The ASTRSK routine then rewinds the old and new system tapes, prints an END OF EDIT message, and checks to see if the GO option is specified. If not, the FORTRAN System Director is called to terminate the job.

When the GO option is specified, the routine reads in, from the new system tape, the new FORTRAN System Director, and overlays the old FORTRAN System Director in main storage. Following this, the Control Card routine is read in.

The ASTRSK routine revises the device assignment table to permit access to the new FORTRAN system tape. The routine then exits to the FORTRAN System Director.

EXIT: Control is passed to the FORTRAN System Director under either of two conditions:

1. If the GO option is not specified, the FORTRAN System Director is called to terminate the job.
2. If the GO option is specified, the routine calls the FORTRAN System Director to return control to the CCLASS routine to read in a card.

ROUTINE CALLED: During execution the ASTRSK routine calls the COPYCL routine.

COPYC Routine: Chart ME

The COPYC (Copy Compiler) routine facilitates revision of compiler phases by copying a portion of the compiler from the old system tape onto new system tapes.

ENTRANCE: The COPYC routine receives control from the RDACRD, T92CMP, or COPYEC routine.

OPERATION: The COPYC routine obtains the name of the phase to be revised from the loader control card and determines if the phase name is valid. If not, an error message is printed and an end-of-job return is made to FSD.

If the phase name is valid, the routine computes and stores the number of phases to be copied. The number, reduced by one each time a phase is copied, is checked for zero. At zero, an exit to the calling routine is made. If the count becomes minus, an error message is printed to indicate a phase name that is out of sequence. The routine then makes an end-of-job return to FSD.

For each phase to be written on the new system tape, a message is printed to identify the phase just copied, and then the RDOSYS routine is called.

EXIT: The conditions under which exit is made from the COPYC routine are:

1. When an invalid phase name is found on the loader control card, control is returned to the FSD.
2. When the phase count is found to be less than zero, control is again returned to the FSD.
3. When the phase count is found to be zero, indicating that all phases have been copied from the old to the new system tapes, control is returned to the calling routine.

COPYCL Routine: Chart MF

The COPYCL (Copy Compiler and Library) routine copies the remaining records from the old system tape, up to the desired point in the library, onto the new system tapes.

ENTRANCE: The COPYCL routine receives control from the T92CMP routine.

OPERATION: The routine checks to see if the library is now being copied and/or modified. If it is not, the routine calls the COPYEC routine. In either case, the

name of a specified library subroutine is used to halt copying at that routine. The routine then calls the COPYL routine. When the COPYL routine is finished, the COPYCL routine receives control and exits to the routine which initiated the call.

EXIT: The COPYCL routine returns control to the T92CMP routine.

ROUTINE CALLED: During execution the COPYCL routine calls the COPYEC and COPYL routines.

COPYL Routine: Chart MG

The COPYL (Copy Library) routine copies the library from the old system tape onto the new system tapes. Any subroutines specified for deletion on the delete list are not copied.

ENTRANCE: The COPYL routine is called from either the COPYCL or AFTER routine.

OPERATION: The COPYL routine initially determines whether or not the library has already been copied. If the library is already copied onto the new system tapes, the routine exits to the calling routine.

If the library has not been completely copied, a record is read in from the old system tape. The routine checks to determine if the entire library has now been read. If the record is the last one in the library, an indicator is set and the routine exits to the calling routine. The routine checks the record to see if it is an ESD card, the first record of a subroutine. If it is not an ESD card, a subroutine is in the process of being copied and the card is written on the new system tape. This process continues until the last record of the library has been read or until an ESD card is read, indicating the start of another subroutine.

If the entire library has not yet been read, the name on the ESD card, which is the name of the subroutine, is compared with the name on the fter card, which indicates the subroutine after which new subroutines are to be inserted in the library. If the names match, insertions are to be made here, and the routine exits to the calling routine.

If the name on the ESD card is not the AFTER name, the name of this subroutine is checked against the list of subroutines to be deleted and not copied on the new system tapes. If the subroutine is to be deleted, the routine skips to the END card of the subroutine, prints a message to identify the deleted subroutine. The name of the next subroutine is then checked against a stored subroutine name, representing a subroutine in the library before which a stop is to be made for revisions to the library. If the names match, a switch is set to indicate to other routines that a replacement deck is needed. The routine then exits to the calling routine.

If an ESD card is found, but the subroutine is not to be deleted, the name of the subroutine is placed on the list or directory of subroutines in the new library. The COPYL routine checks the subroutine name to determine whether or not a stop is to be made here for revisions to the library. If the names match, the routine exits to the calling routine. If not, the subroutine is copied on the new system tape. The routine returns to read another record.

EXIT: Control passes from the COPYL routine to the calling routine under any one of the following conditions:

1. A match exists between the name on the ESD card and the name on the AFTER card.
2. The library has already been copied.
3. The last record in the library is copied.
4. A match between the ESD name and the stored stopping name indicates that a library subroutine is to be replaced.

COPYEC Routine: Chart MH

The COPYEC (Copy to End of Compiler) routine controls the copying of the old system tape onto the new system tape up to, but not including, the library.

ENTRANCE: The COPYEC routine receives control from:

1. The COPYCL routine
2. The T92LB1 routine

OPERATION: The COPYEC routine determines if the library is currently being copied and/or modified. If the library is being processed at this time, the tape is already past the compiler on the old system tape, and the routine exits to the calling routine.

If the tape is not in the library portion of the old system tape, the loader name (LDR) is inserted as the phase name for terminating the compiler copying process. The reason for this is that, if nothing halts the process before the loader is reached on the tape, the compiler will

have been copied to the end. Control is then given to the COPYC routine to perform the copying.

When the compiler is copied, control returns to the COPYEC routine, which writes the loader on the new system tapes and spaces over the end of data set indicator. End of data set is then written on the new system tape. The switch checked at the beginning of this routine to determine if the library is currently being copied and/or modified is turned on. That switch, being set on, indicates that, if the copying process continued, the library would be copied next. The routine then exits to the calling routine.

EXIT: The COPYEC routine exits to the calling routine under either of the following conditions:

1. The library on the old system tape is currently being copied and/or modified.
2. The compiler and loader have been copied from the old system tape to the new system tape.

ROUTINE CALLED: During execution the COPYEC routine calls the COPYC routine.

DELET Routine: Chart MJ

The DELET (Delete) routine produces a list of library subroutines to be deleted from the library on the system tape. The routine also determines if the subroutine to be deleted from the library in the new version has already been copied on the new system tape.

ENTRANCE: The DELET routine receives control from the RDACRD routine when a DELETE control card is encountered.

OPERATION: The DELET routine examines the DELETE control card and makes a list, or table, of the names of the subroutines to be deleted. The capacity of the DELETE table is limited. If the number of subroutines on the DELETE card exceeds the capacity of the table, the routine prints an error message that the DELETE table is full and exits to the calling routine.

As each subroutine name is entered in the DELETE table, the routine checks to see if the name is in the directory of subroutines which have already been copied onto the new system tapes. When this occurs, an error message is printed that the subroutine name is out of sequence. The routine returns to the FSD to end the job.

If no more subroutine names remain to be examined, the routine exits to the RDACRD routine to read another card.

EXIT: Exit from the routine and passage of control to the RDACRD routine occurs in one of two ways:

1. A subroutine name on the DELETE control card is found to be out of sequence.
2. All DELETE table names are checked and none found in the directory.

REDCRD Routine: Chart MK

The REDCRD (Read New Phase) routine revises a phase of the compiler by reading in and processing REP and/or TXT cards to replace some or all of the information in the specified phase on the old system tape.

ENTRANCE: Entrance to the REDCRD routine is from the T92CMP routine. If it is established that the editor or IBCOM is not to be modified or replaced, the compiler is copied up to the point of the phase to be revised. The REDCRD routine is then called.

OPERATION: The REDCRD routine reads a card from the card reader. It then makes a check for end of data set in the card reader. An end of data set prior to the END card means that insufficient cards were supplied to the card reader and/or the END card was not furnished. An error message is printed and the FORTRAN System Director is called to abort editing.

If no end of data set exists, the routine checks for an END card. The presence of an END card means that all of the cards necessary to the revision of this phase have been read. The routine calculates the length of the phase and exits to the calling routine.

If the card read is not an END card, it is examined to determine whether or not it is a REP card. The presence of a REP card causes the routine to check to determine if the entire phase is to be replaced. Replacement of a whole phase implies that TXT cards follow the REP card with sufficient additional data for replacement of the entire phase. Another card is read and all of the above checks are made. If the entire phase is not to be replaced, the routine processes the REP card and goes back to read another card.

Whenever a card is read by the routine, the checks for end of data set, END card, and REP card are followed by checks for RLD

and ESD cards. If an RLD or ESD card is found, the routine ignores it and goes back to read another card. If the card is still unidentified, a check is made for a TXT card. If the card is not a TXT card, the routine makes a final check for an EDR card. If the card is an EDR card, it is processed like a REP card and the routine then goes back to read another card. If the card is not an EDR card, the routine assumes that the card is an invalid control card. An error message is printed, and an exit is made to FSD to abort the edit.

If the card is a TXT card, it is applied to revise the specified portion of the compiler phase. The routine then reads another card. TXT cards are read and applied in this fashion until there is an illegal end of data set (as noted above), or until an END card is found

EXIT: Under any one of three conditions, control passes from the REDCRD routine:

1. An end of data set signal is on. An error message is written and the FORTRAN System Director is called to abort the edition.
2. The card read is not a valid control card for this routine. An error message is written and FSD is called to abort the edit.
3. An END card is encountered. The routine exits to the calling routine.

RDOSYS Routine: Chart ML

The RDOSYS (Read Old System Tape) routine reads a record from the old system tape, then calculates and stores the length of the record. If the phase just read is FSD, the machine size is changed to the current machine size.

ENTRANCE: The RDOSYS routine receives control from the RDACRD routine to read in the Initial Program Load. Control is also received from the AFTER routine when one or more library subroutines are to be read from the old system tape and copied on the new tapes. The routine is called in the COPYC routine when the compiler on the old system tape is copied on the new tapes, up to a specified phase or to the end of the compiler. The routine is also called by the ASTRSK routine when the Initial Program Load and the FORTRAN System Director are to be read from the old system tape.

OPERATION: The RDOSYS routine sets a pointer at the name of the phase to be read. It then reads a record from the old system tape and a check is made for end of data set. When end of data set is detected, there is an exit from the routine to write an illegal end of data set message. Absence of the end of data set causes the length of the record to be computed and stored. If the phase read is FSD, the current machine size is moved from the communications area to the communications area in the FSD just read. This move is performed because the machine size may have been changed by the EDIT card.

EXIT: In addition to an illegal end of data set, control is passed from the RDOSYS routine to the calling routine.

T92CMP Routine: Chart MM

The T92CMP (Compiler) routine checks all 12-9-2 cards that are concerned with phase modification. The processing consists of replacing or modifying the specified phase and copying the old system tape onto the new system tapes as far as the specified phase.

ENTRANCE: The T92CMP routine receives control from the RDACRD routine, when any loader control card, containing a phase name is encountered.

OPERATION: Initially the routine gets the phase name from the loader control card and stores it. It then determines if the editor or IBCOM is to be either modified or replaced. If the editor or IBCOM is not involved, the routine calls the COPYC routine in order to copy the old system tape up to the phase to be modified. Control then returns to the T92CMP routine which branches to read in the new phase or modifications from the card reader.

If the editor or IBCOM is to be modified or replaced, the routine checks to see if IBCOM has been copied onto the new system tape. If IBCOM has not been copied, the routine calls the COPYCL routine to copy the old system tape as far as IBCOM. Control returns to the T92CMP routine, which calls the RDOSYS routine to read IBCOM from the old system tape.

If an IBCOM replacement is to occur, the routine passes control to the REDCRD routine to read in replacement cards from the card reader. If there is no IBCOM replacement, the routine writes IBCOM on the new system tape(s) and reads in the editor. The routine then branches to read in replacement cards, so the editor can be modified.

EXIT: The T92CMP routine passes control to the REDCRD routine, under any one of three conditions:

1. Some phase, other than the editor, is to be modified or replaced and the system tape has been copied up to the phase to be modified.
2. IBCOM is to be modified or replaced.
3. IBCOM has been copied and the editor has been read in from the old system tape for modification.

ROUTINES CALLED: During execution the T92CMP routine calls the COPYC, COPYCL, and RDOSYS routines.

T92LB1 Routine: Chart MN

The T92LB1 (Library) routine copies the compiler and library up to a specified library subroutine which is to be revised. The routine adds object decks from the card reader to the library.

ENTRANCE: The T92LB1 routine receives control from the RDACRD routine when a loader control card without a phase name is encountered.

OPERATION: The T92LB1 routine checks a switch, which is turned on whenever an AFTER card is encountered, indicating that there are subroutines to be added to the new library. If the switch is off, no library subroutines are to be inserted and the routine calls the COPYEC routine.

When library subroutines are to be inserted, or at the end of the COPYEC routine, T92LB1 routine searches the directory of copied subroutines to see if the directory contains the name of the subroutine to be revised. If it does, the revision cards for this subroutine are out of sequence, causing an Out of Sequence message to be printed. The routine then exits to the RDACRD routine to read the next card.

If the name of the subroutine to be revised is not in the directory and if the switch, indicating that subroutines are to be inserted, is still on, the routine copies, from the old to the new system tape, the subroutine after which insertions are to be made. It then reads a new subroutine from the card reader and adds it to the library on the new system tape. Subroutines are read from the card reader and copied on tape until the list of subroutine names is exhausted. At that time, the name of the subroutine after which insertions are to be made is blanked out.

EXIT: When the revision cards for a specified library subroutine are out of sequence, the T92LB1 routine exits to the RDACRD routine to read the next card. The T92LB1 routine also exits to the T92LB2 routine when a subroutine is to be revised.

Editor T92LB2 Library Routine #2: Chart MO

The T92LB2 (Library) routine replaces a library subroutine.

ENTRANCE: The T92LB2 routine receives control from the T92LB1 routine, after the COPYL routine has copied the library up to the subroutine specified for revision.

OPERATION: The operation performed by this routine depends upon the last card read from the card reader. An ESD card indicates that an entire subroutine replacement exists in the card reader. Thus, the subroutine, on the old system tape, that corresponds to the subroutine replacement in the card reader is bypassed. If, however, that same subroutine on the old system tape has already been copied onto the new system tape and cannot be bypassed, an Out of Sequence message is printed. In either case, the subroutine replacement in the card reader is copied onto the new system tapes.

EXIT: The T92LB2 routine passes control to the RDACRD routine to read the next card.

ROUTINE CALLED: During execution the T92LB2 routine calls the COPYL routine.

SET Routine: Chart MP

The SET (Editor SET Control Card) routine uses information supplied on the editor SET control card to modify the device assignment table and line length in the FORTRAN System Director on the new system tape.

ENTRANCE: The SET routine receives control from the RDACRD routine when an editor SET control card is encountered.

CONSIDERATIONS: The information on the editor SET control card specifies:

1. The data set reference number of the device specified.
2. The physical address of the device.
3. The designation of the actual type of input/output device.
4. Where applicable, certain characteristics of the tape and data set to be used.
5. Length of print line if this is being changed.

OPERATION: A check is made to determine if the FSD has already been copied from the old system tape to the new system tape. If the FSD has been copied, it is too late to revise it for this job. An error message is written, and a return is made to the RDACRD routine to read another card. The SET routine calls the COPYC routine, if the FSD has not been copied from the old system tape.

When control is returned to the SET routine, it examines each field on the SET control card to determine if a line length change or a data set change is involved. A line length change causes the new line length to be inserted in the communications area of the new system tape(s). Data set changes are checked for validity, and the physical address and device type are stored in the device assignment table. The device type code is stored in the DSCB for the associated data set. If the device is a 7-track tape device, the mode set code to be used with this device is stored in the DSCB.

When a blank field is encountered, control is returned to the RDACRD routine. An invalid field causes an error message to be printed and the job to be terminated.

EXIT: The SET routine exits to the RDACRD routine, upon encountering the first blank field.

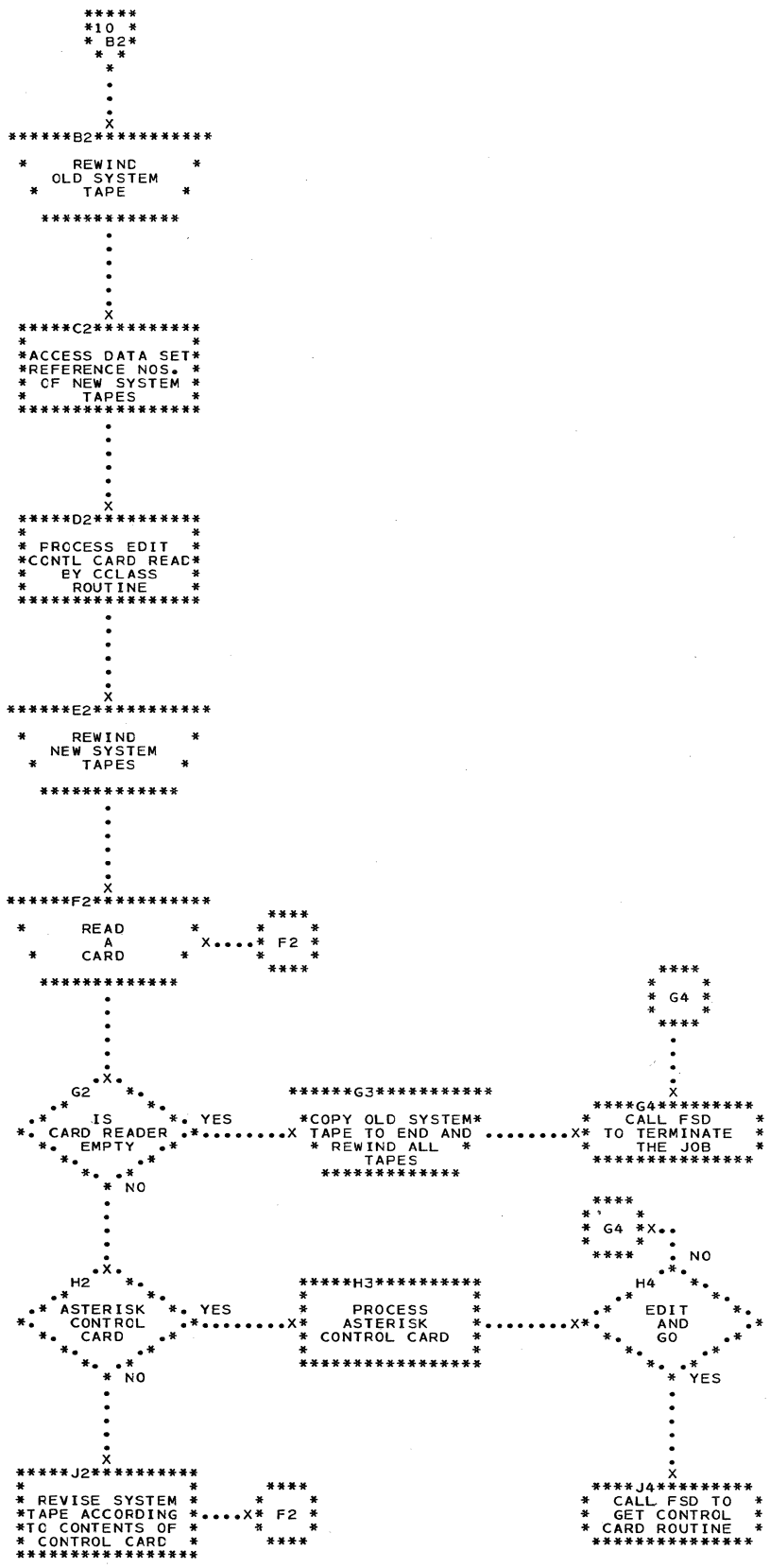


Chart 10. Editor Overall Logic Diagram

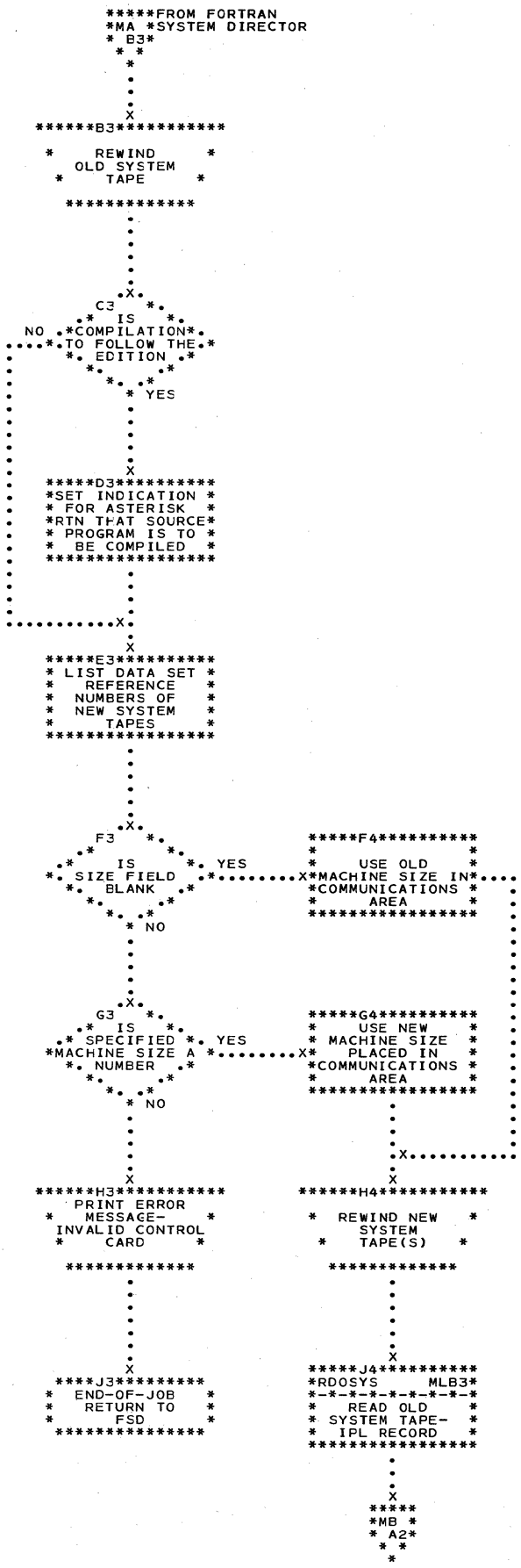


Chart MA. START Routine

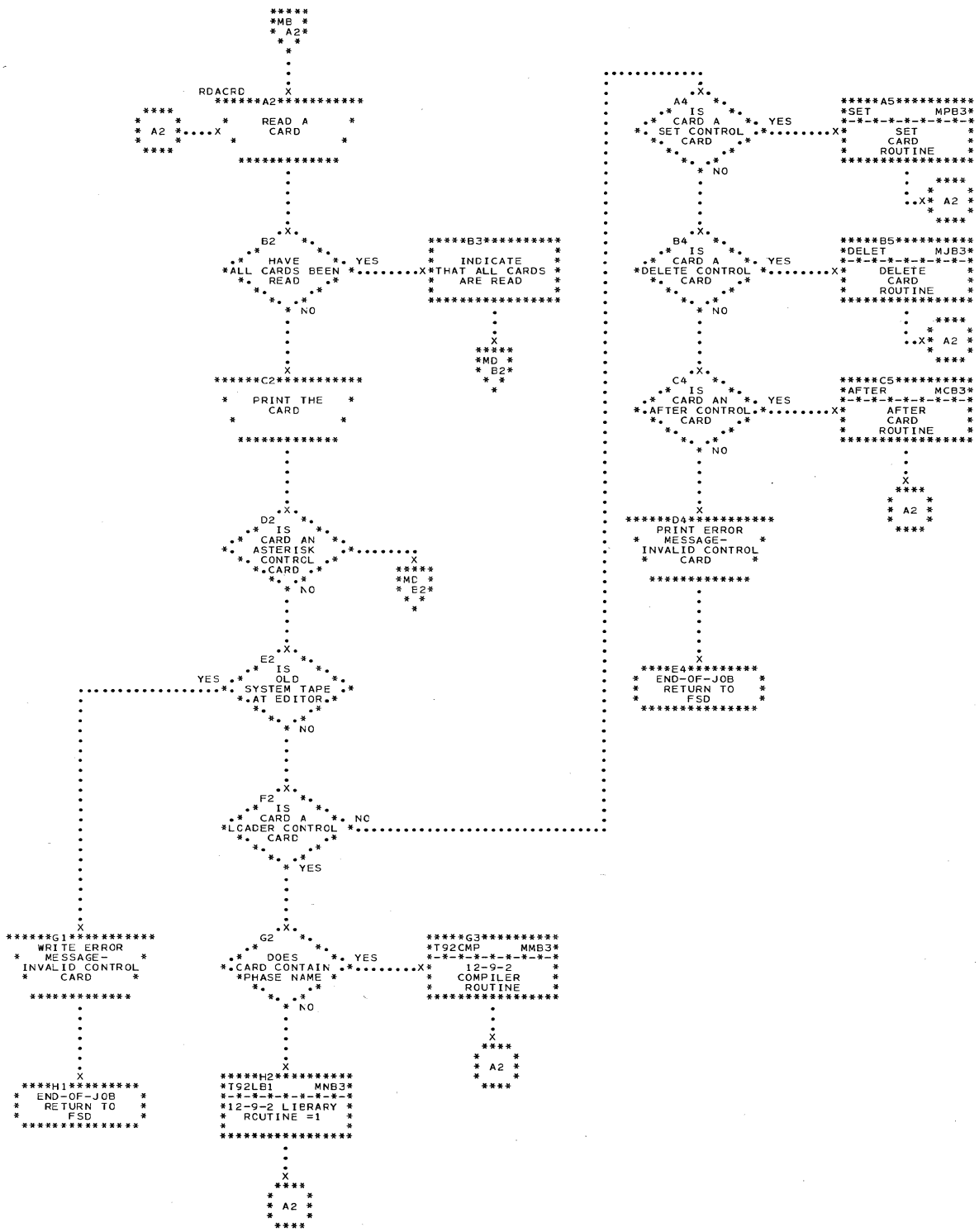


Chart MB. RDACRD Routine

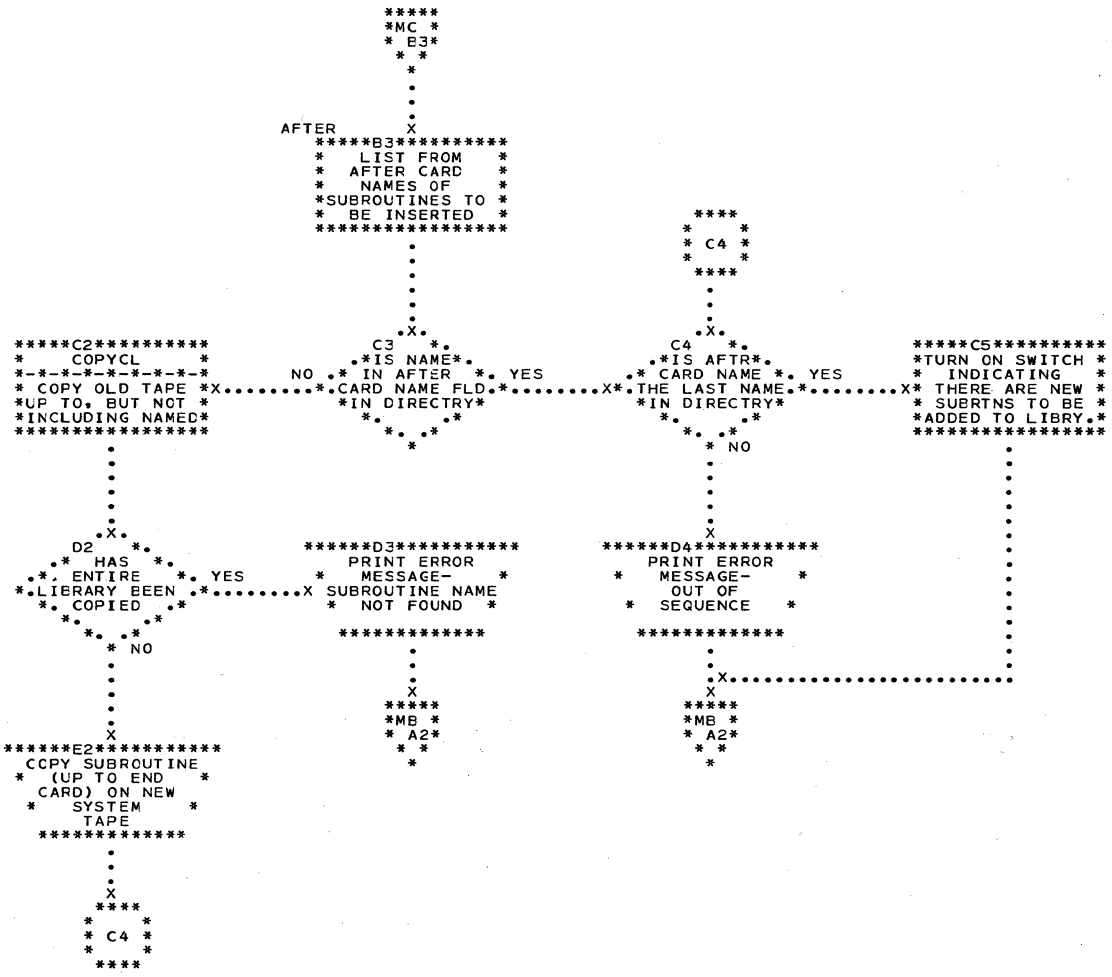


Chart MC. AFTER Routine

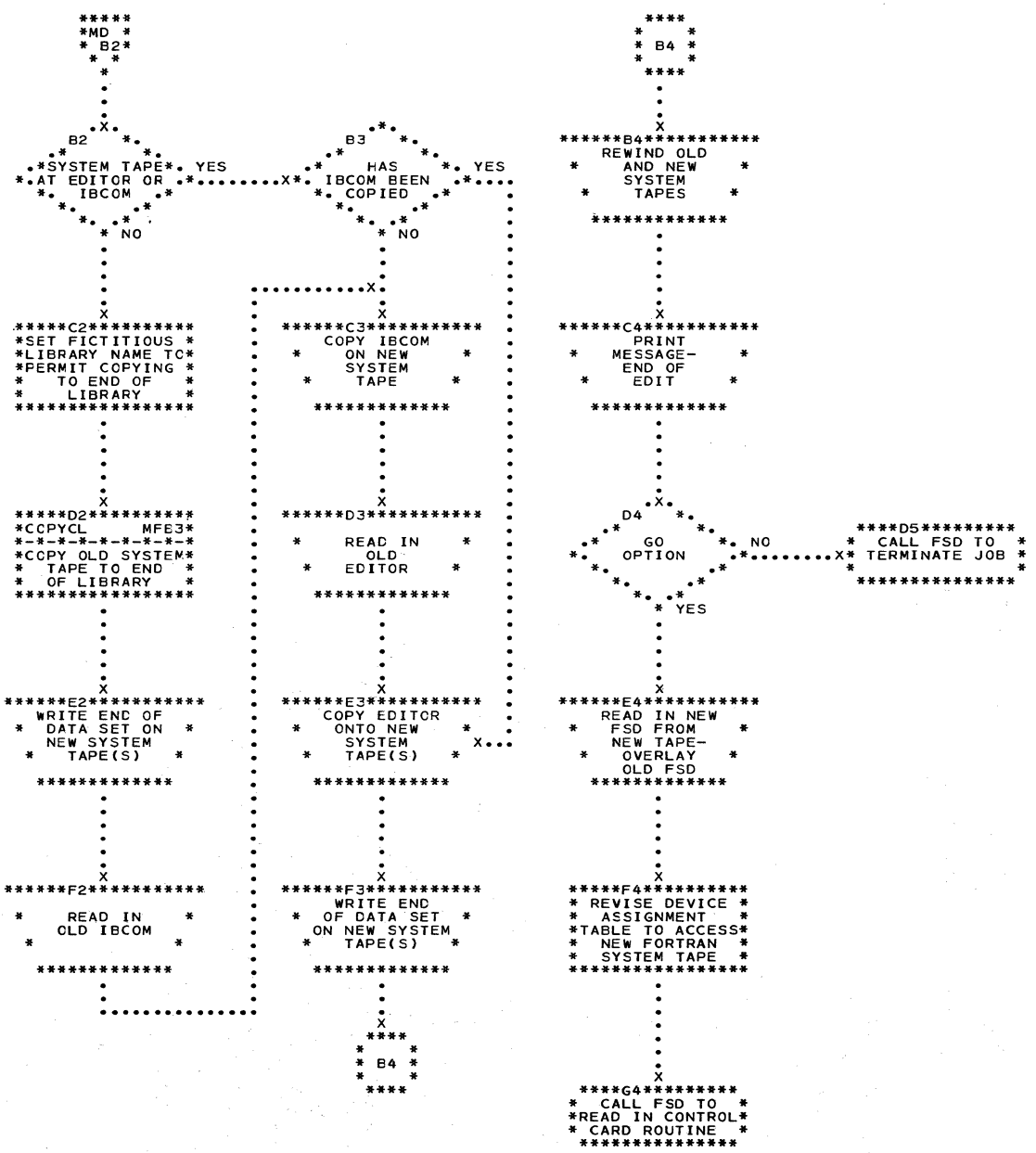


Chart MD. ASTRSK Routine

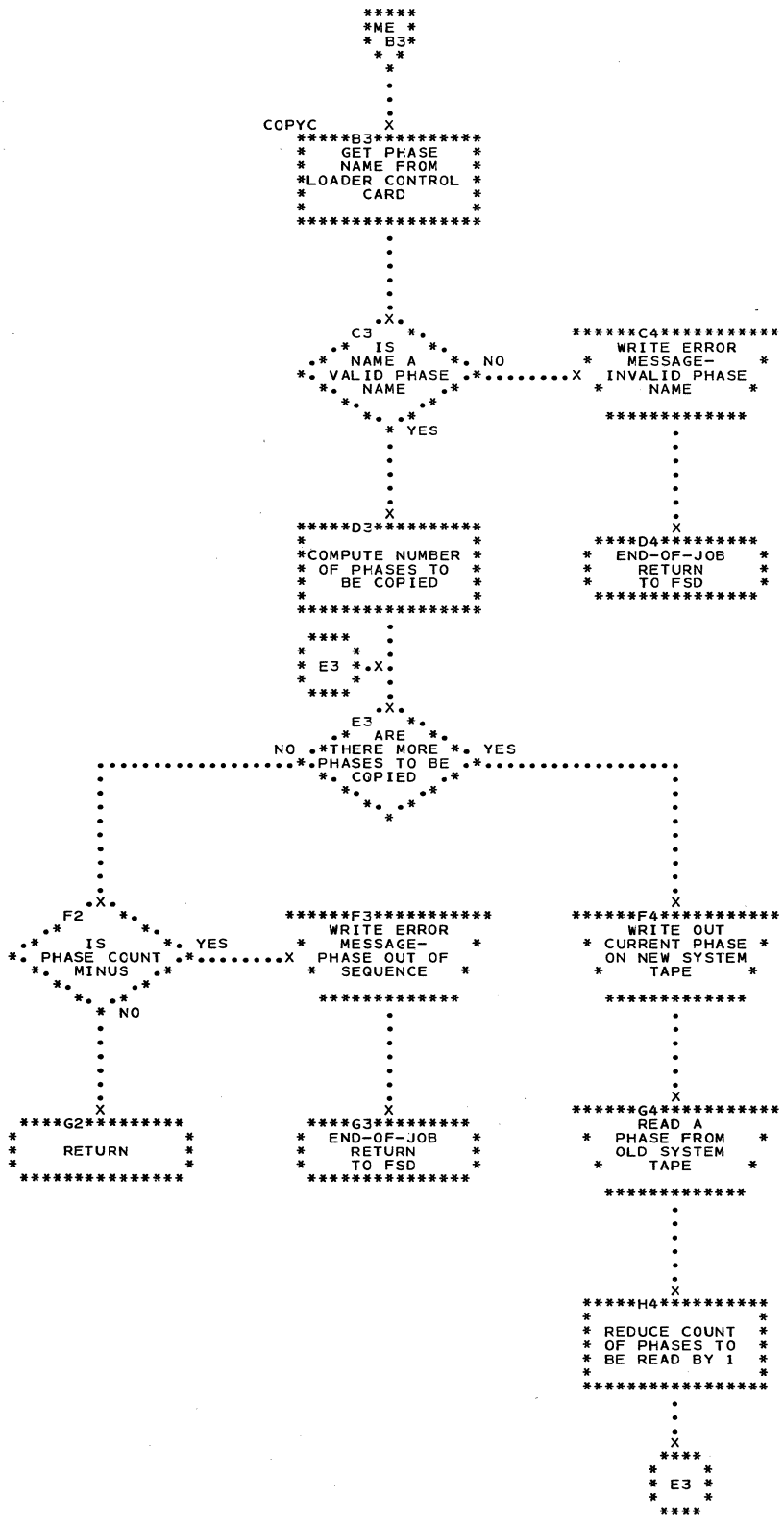


Chart ME. COPYC Routine

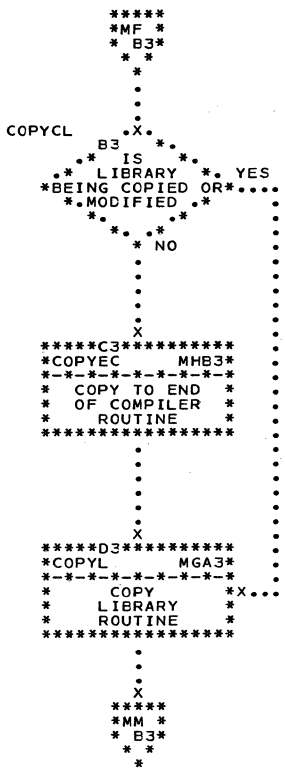


Chart MF. COPYCL Routine

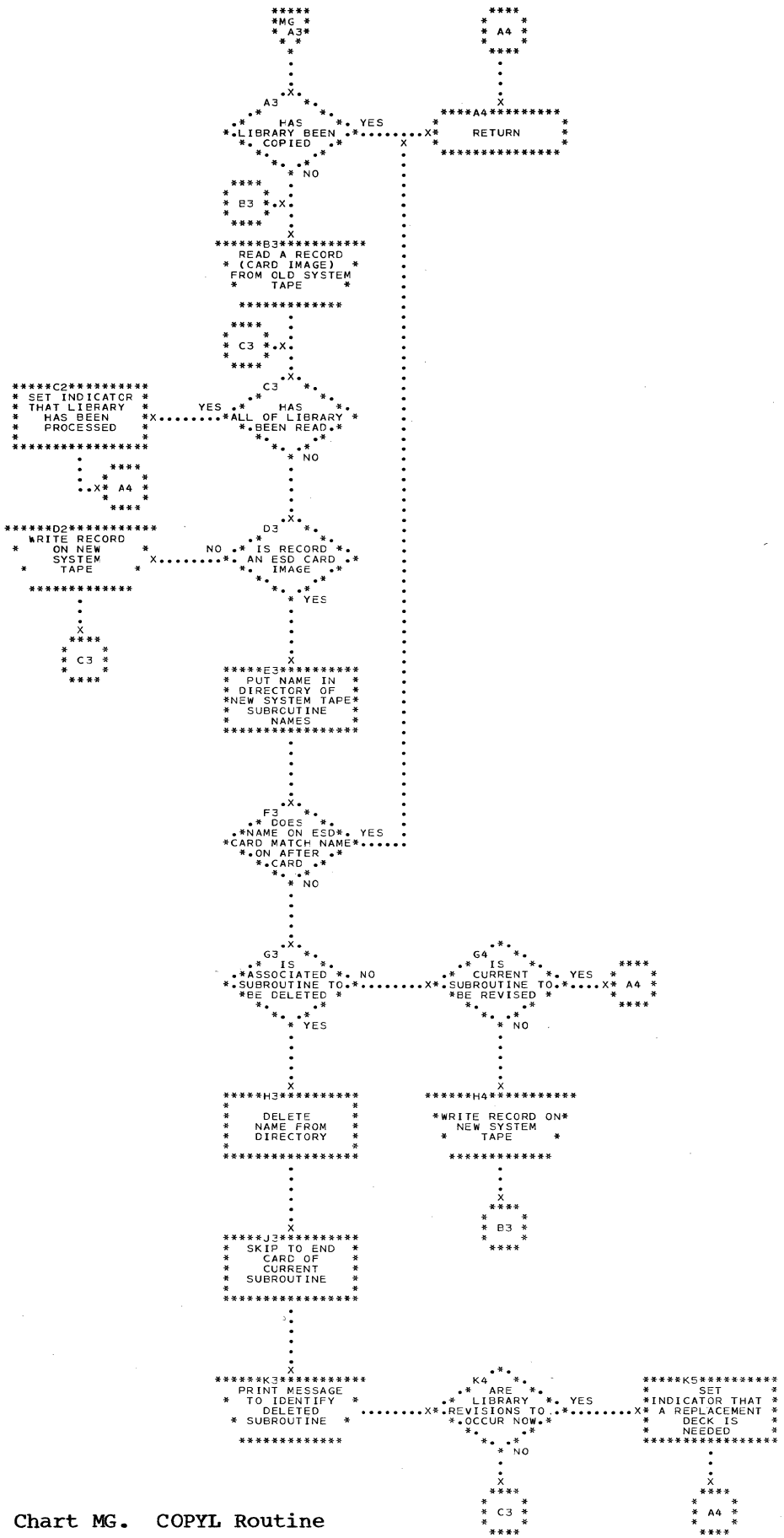


Chart MG. COPYL Routine

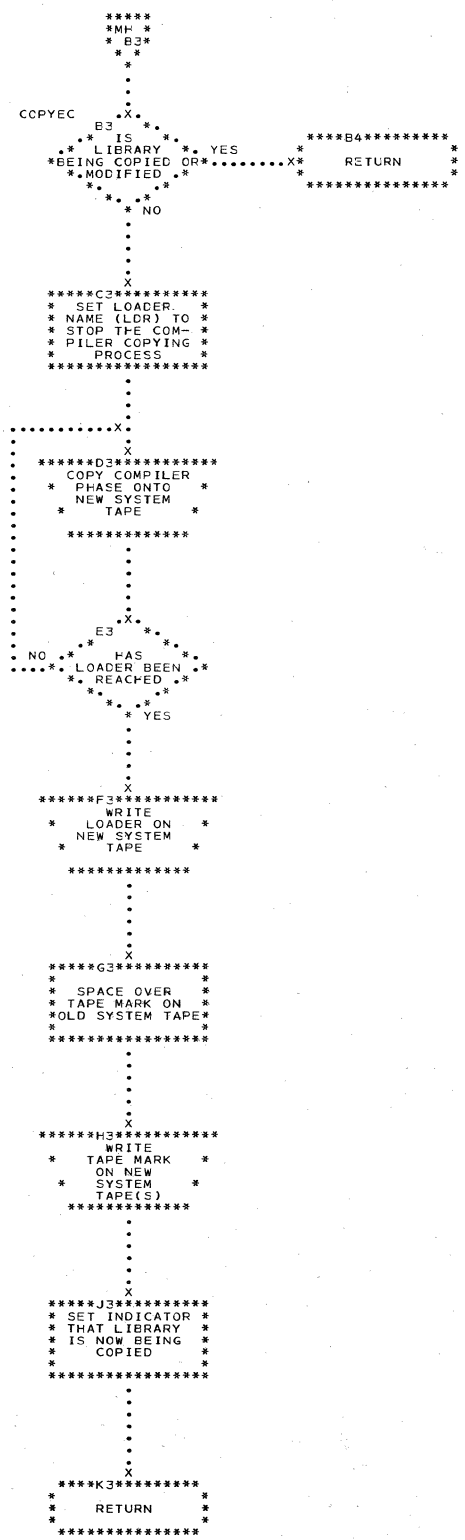


Chart MH. COPYEC Routine

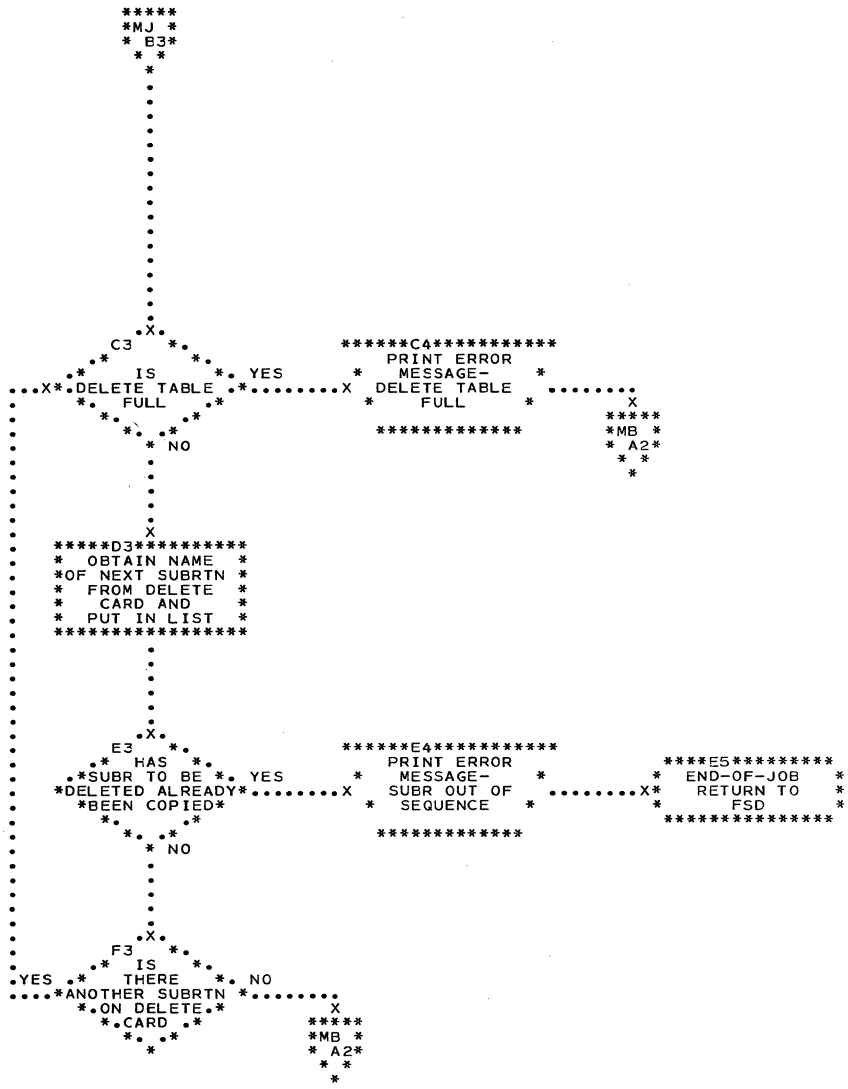


Chart MJ. DELET Routine

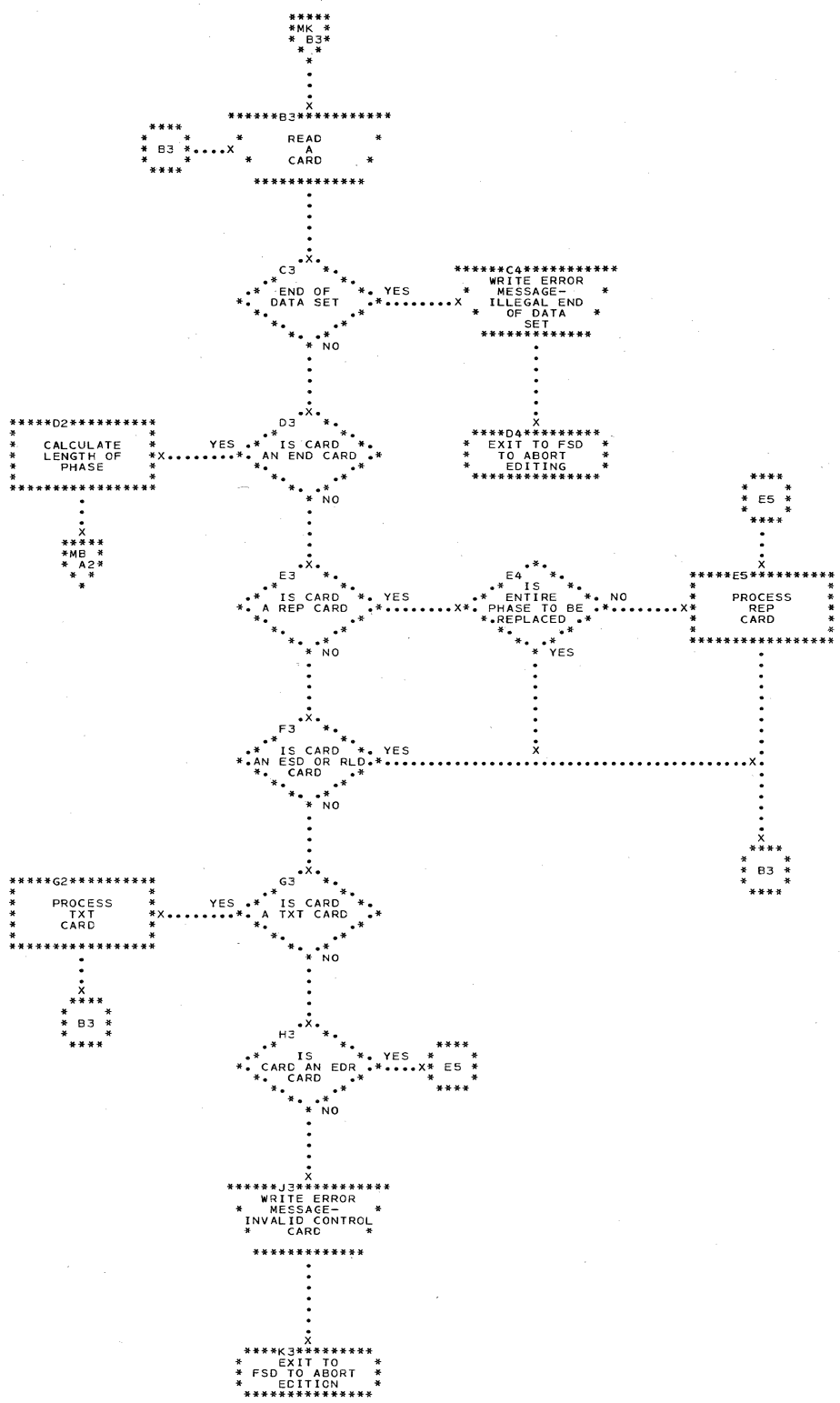


Chart MK. REDCRD Routine

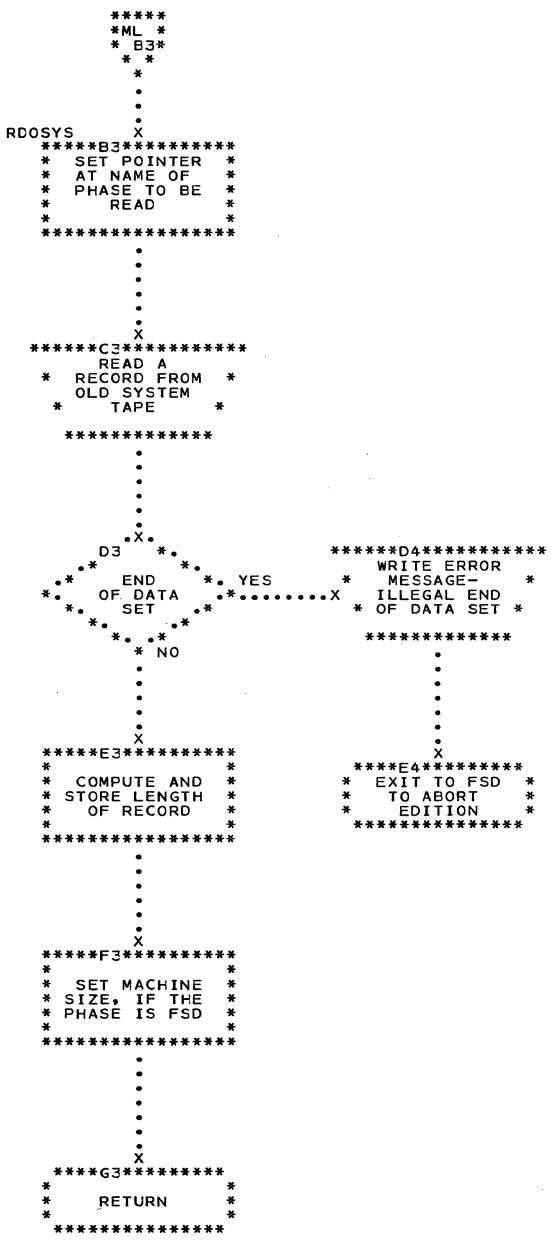


Chart ML. RDOSYS Routine

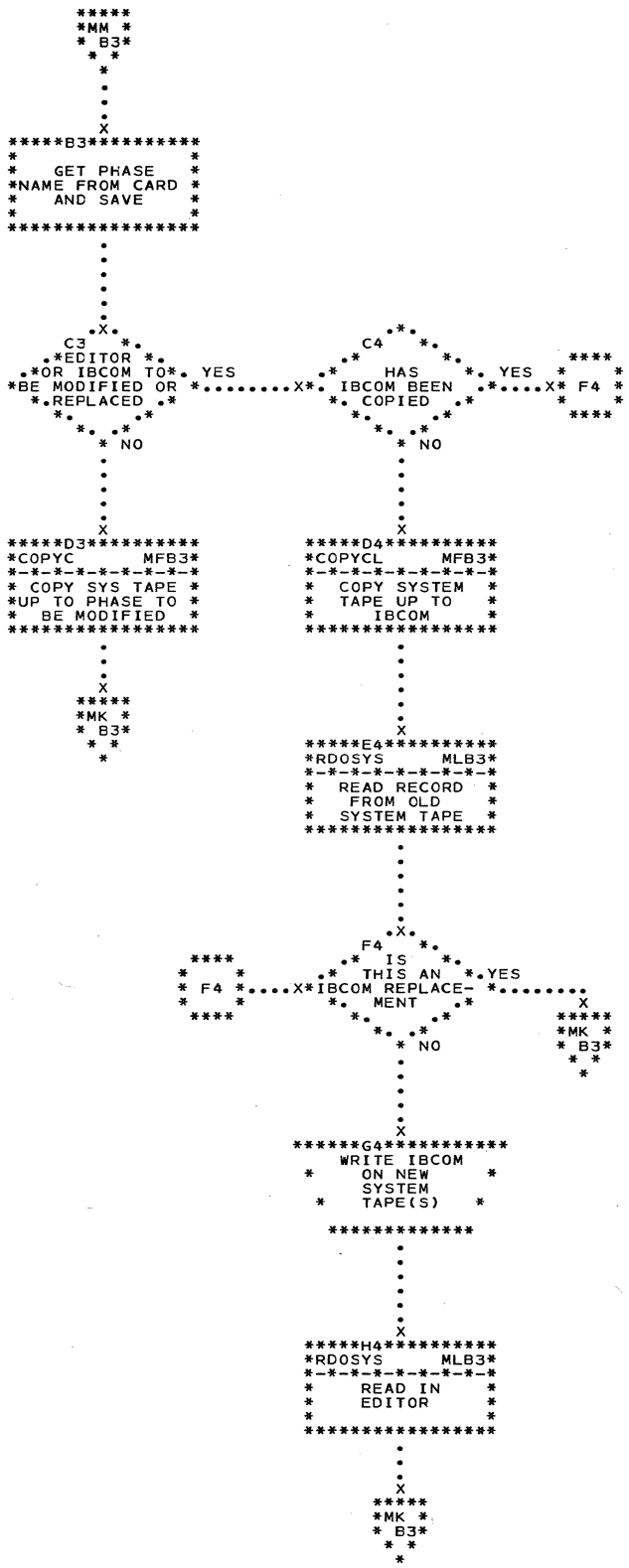


Chart MM. T92CMP Routine

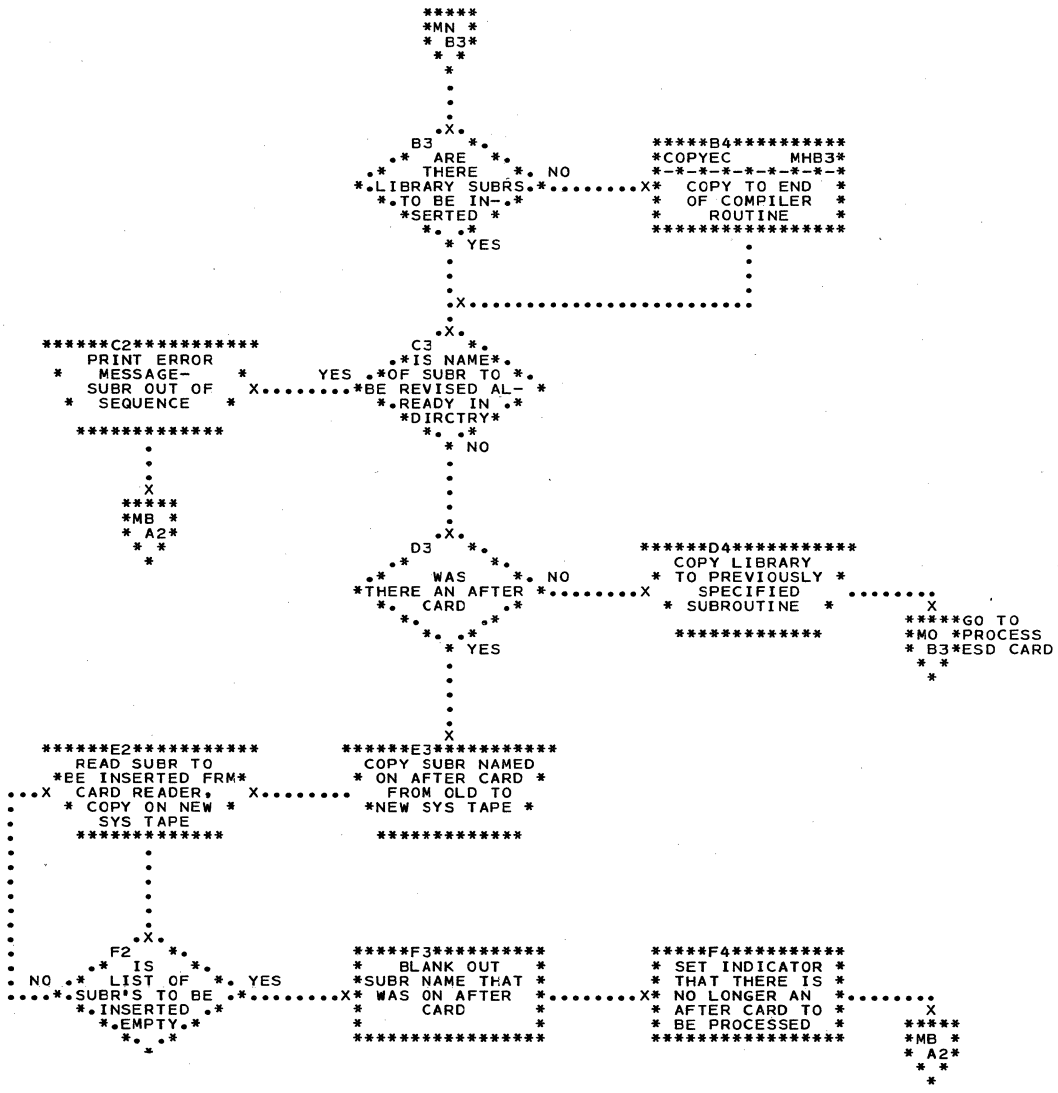


Chart MN. T92LB1 Routine

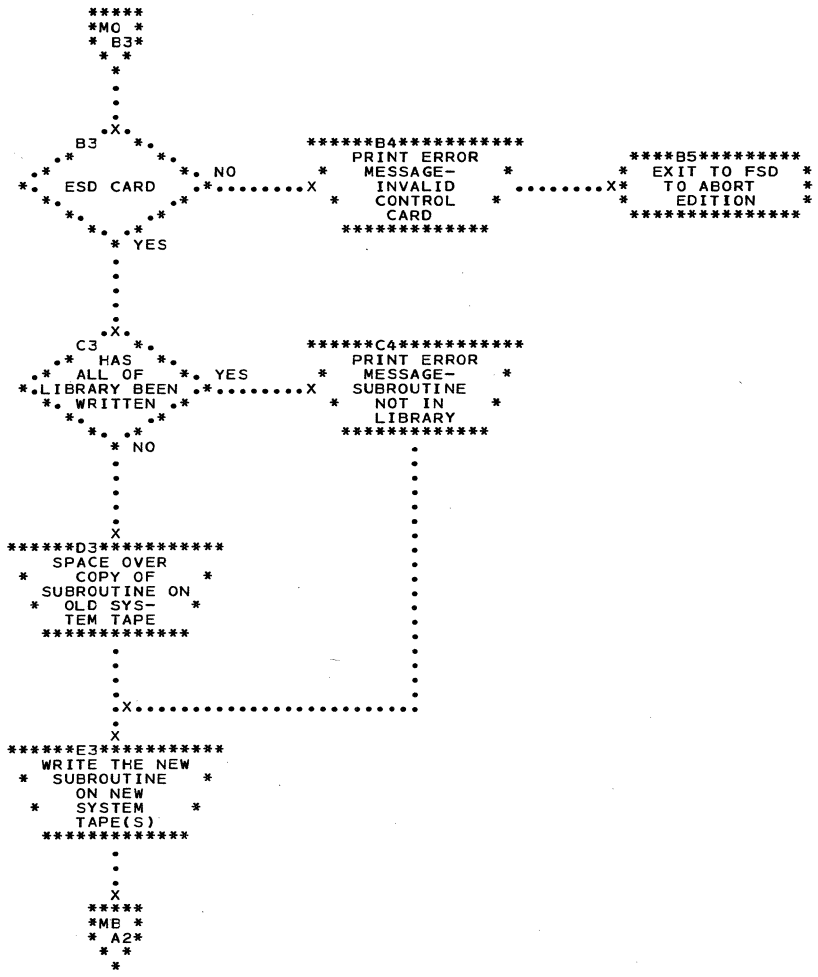


Chart MO. T92LB2 Routine

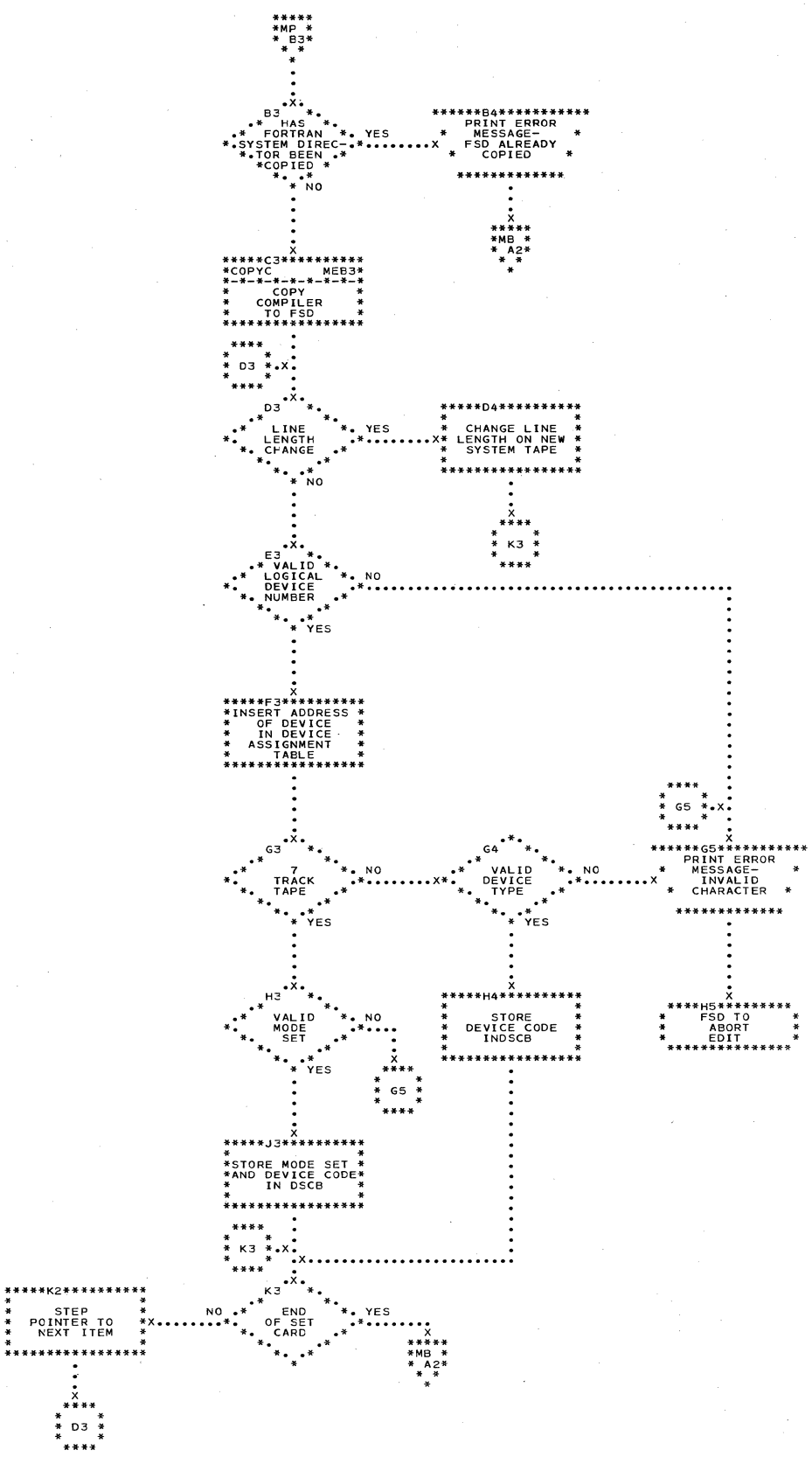


Chart MP. SET Routine

MESSAGES

The messages produced by the FORTRAN system are explained in the IBM System/360 Basic Programming Support Programmer's Guide. Each message is identified by an associated number.

Table 3 associates a message number with the particular routine/subroutine in which the corresponding message is generated.

Table 3. Error and Warning Messages

Message Number	Phase	Subroutine or Routine
029	10	DIMSUB
030	10	COMMON, EQUIVALENCE
031	10	COMMON, EQUIVALENCE
032	10	LITCON
033W	10	GETWD
034	10	FUNCTION/SUBRTN
035	10	FUNCTION/SUBRTN
036	10	ARITH
037	10	CLASSIFICATION, ARITH, ASF, SUBIF
038	10	INTEGER/REAL/DOUBLE, EXTERNAL, COMMON, EQUIVALENCE, DIMENSION
039	10	SYMTLU
041	10	ASF, EXTERNAL, DIMENSION
043	10	INTEGER/REAL/DOUBLE, GOTO
	12	ALOC
044	10	LITCON
045	10	LITCON
046	10	LITCON
047	10	CLASSIFICATION, DIMENSION
048	10	DIMSUB
049	10	DIMENSION, DIM90
050	10	EQUIVALENCE
051W	10	EQUIVALENCE, DIMENSION
052	10	SUBS, EQUIVALENCE

(continued)

Table 3. Error and Warning Messages (continued)

Message Number	Phase	Subroutine or Routine
053	10	SUBS
054	10	ASF
055	10	FUNCTION/SUBRTN
056	10	GOTO
057	10	READ/WRITE
058	10	READ/WRITE
060	10	EQUIVALENCE
061W	10	END MARK CHECK
063	10	EQUIVALENCE
064	10	LABTLU, SYMTLU
065W	10	CLASSIFICATION, LABLU, PAKNUM
066	10	DO
068	10	LITCON
069	10	ASF
070	10	FUNCTION/SUBRTN
071	10	CALL
072	10	ARITH
073	10	PUTX
074	10	COMMON
075	14	FORMAT, LINECK
076	14	READ/WRITE, FORMAT, RPAREN
077	10	ASF, READ/WRITE, END MARK CHECK, DO, SUBS, EQUIVALENCE, FUNCTION/SUBRTN, DIMSUB, DIMENSION, SKPBLK
	14	READ/WRITE, DO, FILLEG, SKPBLK
078	14	CKENDO
079	10	GO TO
	14	READ/WRITE, DO
080W	10	GOTO
	14	READ/WRITE
081W	10	ARITH, EQUIVALENCE
	14	READ/WRITE, D/E/F/I/A
082	10	LITCON

(continued)

Table 3. Error and Warning Messages (continued)

Message Number	Phase	Subroutine or Routine
	14	NOFDCT, INTCON
083	10	CSORN, INTCON
	14	INTCON
084	10	ERRET/WARNING
	14	ERROR/WARNING
085	12	DPALOC, SALO
	14	PRESCN
086	14	BLANKZ
087	14	D/E/F/I/A, T
088	14	LPAREN
089	14	UNITCK/UNIT1
090	14	QUOTE/H
091	14	+/-/P
092	14	FCOMMA
093	14	GETWDA
094	14	D/E/F/I/A
095	14	READ/WRITE
096	14	READ/WRITE
097	14	READ/WRITE
098	14	QUOTE/H
099	14	QUOTE/H
100	14	DO
123	15	MOPUP
124	15	COMMA
125	15	DO, BEGIO
126	15	CKARG
127	12	COMAL, ALOC
	15	PRESCN, UMINUS, UPLUS, FOSCAN
128	15	LFTPRN
129	15	TYPE
130	15	COMMA

(continued)

Table 3. Error and Warning Messages (continued)

Message Number	Phase	Subroutine or Routine
131	15	INLIN1
132	15	LABEL
133	15	EQUALS
134	15	ERROR/WARNING
135	15	COMMA, TYPE
136	15	LAB
137	15	COMMA, TYPE, RTPRN
139	15	COMMA
140	15	FOSCAN
141	15	COMMA
142	15	DO, BEGIO
143	15	EQUALS
144	15	ARTHIF
145	12	EXTCOM
	20	PHEND
146	12	COMAL, RENTER/ENTER, SWROOT
147	12	EQUIVALENCE
148	12	RENTER/ENTER, SWROOT
149	12	COMAL
150	12	ALOC
160W	10	PUTX
	14	INTCON
	15	COMMA
161W	12	EXTCOM
162W	10	CLASSIFICATION
163W	10	LITCON
164W	10	CONTINUE/RETURN
	14	PAUSE/STOP/SKIP, FORMAT
166W	10	END MARK CHECK, DO, FUNCTION/SUBRTN
	14	READ/WRITE
167W	14	LINECK

(continued)

Table 3. Error and Warning Messages (continued)

Message Number	Phase	Subroutine or Routine
168W	10	END MARK CHECK
169W	10	DIMSUB
	15	COMMA
170W	14	X
171W	10	END MARK CHECK
	14	RPAREN
172W	10	ASF
173W	10	ARITH
174	15	EQUALS, LFTPRN, INARG, TYPE
175W	14	LABEL

STATEMENT PROCESSING

Table 4 indicates, within each compiler phase, the routine/subroutine responsible for the processing of the statement under consideration.

Table 4. Processing Subroutines

Statement Condition or Keyword	Phase 10	Phase 12	Phase 14	Phase 15	Phase 20	Phase 25
Arithmetic Expression or Statement	ARITH		PASSON[1]	FOSCAN	ARITH	RXGEN
FUNCTION Call	ARITH	LDCN	PASSON[1]	FOSCAN	HANDLE/ CALSEQ	FUNGEN/ EREXIT
Subscripted Variable	SUBS	SSCK	PASSON[1]	MVSBXX/ MVSBRX	OPTMIZ	SAOP, AOP
ASF definition and expansion	ASF	LDCN	ASF[1]	FOSCAN	ARITH	ASFDEF, ASFEXP
Statement Number Definitions	CLASSIF- ICATION	ASSNBL	LABEL	LABELDEF	LABEL	LABEL
ASF Call	ARITH	LDCN	PASSON[1]	FOSCAN	CALSEQ	ASFUSE
BACKSPACE	BKSP/REWIND/ END/ENDFILE		BSPREF[1]	SKIP	ESDRLD	RDWRT
CALL	CALL	LDCN	PASSON[1]	FOSCAN	CALL/ CALSEQ	FUNGEN/ EREXIT
COMMON	COMMON	COMAL				
Computed GOTO	GOTO		PASSON[1]	COMPGOTO	CDGOTO	CGOTO
CONTINUE	CONTINUE/ RETURN		SKIP[1]	SKIP		

(continued)

Table 4. Processing Subroutines (continued)

DIMENSION	DIMENSION					
DO	DO		DO	DO	DO	DO1, ENDDO
DOUBLE PRECISION	INTEGER/ REAL/DOUBLE	DPALOC				
END	BKSP/REWIND/ END/ENDFILE		END	MOPUP	PHEND	END
END FILE	BKSP/REWIND/ END/ENDFILE		BSPREF[1]	SKIP	ESDRLD	RDWRT
EQUIVALENCE	EQUIVALENCE	EQUIVALENCE				
EXTERNAL	EXTERNAL	LDCN				
FORMAT	FORMAT		FORMAT			
FUNCTION	FUCTION/ SUBRTN	LDCN	SUBFUN[1]			SUBRUT
GOTO	GOTO		PASSON[1]	GOTO		TRGEN
IF	SUBIF		PASSON[1]	FOSCAN	IF	ARITHI
INTEGER	INTEGER/ REAL/DOUBLE	SALO				
PAUSE	STOP/PAUSE		PAUSE[1]	SKIP		STOP/PAUSE
READ	READ/WRITE		READ/WRITE	BEGI/O	READ, LIST	RDWRT/ IOLIST
REAL	INTEGER/ REAL/DOUBLE	SALO				
RETURN	CONTINUE/ RETURN		RETURN[1]	SKIP		RETURN
REWIND	BKSP/REWIND/ END/ENDFILE		BSPREF[1]	SKIP	ESDRLD	RDWRT
STOP	STOP/PAUSE		STOP[1]	SKIP		STOP/PAUSE
SUBROUTINE	FUNCTION/ SUBRTN	LDCN	SUBFUN[1]	FOSCAN		SUBRUT
WRITE	READ/WRITE		READ/WRITE	BEGI/O	LIST	RDWRT/ IOLIST
In-line Functions	ARITH	LOCN	PASSON[1]	FOSCAN	FIXFLO	FUNGEN/ EREXIT

[1]Described in Phase 14 adjective code subroutine.

The exponential subprograms are elements of the FORTRAN Library. Their function is to compute, at object time, the value of exponential terms appearing in arithmetic statements.

All linkages to the exponential subprograms are compiler generated. Each time an exponential term (e.g., X**Y) is encountered during compilation, the compiler selects the exponential subprogram which is to perform the computation. The selection of the subprogram is dependent upon the modes of the base and exponent. A calling sequence to the selected subprogram is then generated and included as part of the object program. At object time, the call is executed, thereby giving control to the exponential subprogram to compute the value of the exponential term.

The five exponential subprograms (FIXPI, FRXPI, FDXPI, FRXPR, AND FDXPD) compute the value of an exponential term according to the modes of the base and exponent.

FIXPI Subprogram

The function of the FIXPI subprogram is to compute the value of an exponential term of the form X**Y where both X and Y are of integer mode.

ENTRANCE: The FIXPI subprogram receives control from the object time execution of a compiler generated linkage. This linkage is generated during compilation when an exponential term, in which both the base and exponent are of integer mode, is encountered.

CONSIDERATION: The following considerations apply to the execution of the FIXPI subprogram:

1. An error exists in the term X**Y where X = 0 and Y ≤ 0
2. For Y ≠ 0 and X = 1, X**Y = 1
3. For Y ≠ 0 and X = -1
   a. X**Y = +1 if Y is even
   b. X**Y = -1 if Y is odd
4. For X = 0 and Y > 0, X**Y = 0
5. For X ≠ 0 and Y = 0, Y**Y = 1

OPERATION: For positive values of Y,X**Y= (X**K(31))*((X**2)**K(30))*((X**4)**K(29)) ((X**8)**K(28))... where K(I) is either 1 or 0 and I represents the bit position in the register containing Y. (The sign bit position does not enter into the computation.) For negative values of Y, X**Y=0.

EXIT: After computing the value of the exponential term, the FIXPI subprogram returns control to the object program. However, if an error exists, exit is to subroutine IBFERR which is contained in IBCOM.

FRXPI Subprogram

The FRXPI subprogram computes the value of an exponential term of the form X**Y where X is of real mode and Y of integer mode.

ENTRANCE: The FRXPI subprogram receives control from the object time execution of a compiler-generated linkage. This linkage is generated during compilation when an exponential operation, in which the base is of real mode and the exponent of integer mode, is encountered.

CONSIDERATION: The following considerations apply to the execution of the FRXPI subprogram:

1. An error exists in the term X**Y where X = 0.0 and Y ≤ 0
2. For X = 0.0 and Y >0, X**Y = 0.0
3. For X ≠ 0.0 and Y = 0, X**Y = 1.0

OPERATION: For positive values of Y, implementation of the exponential term X**Y is the same as that described in subroutine FIXPI. For negative values of Y, the 2's complement of Y is taken and the exponential operation is implemented in the same manner as that described in subroutine FIXPI for positive values of Y; the reciprocal of the result is then taken.

EXIT: After computing the value of the term X**Y, the FRXPI subprogram returns control to the object program; however, if an error exists, exit is to subroutine IBFERR.

FDXPI Subprogram

The function of the FDXPI subprogram is to compute the value of an exponential term of the form X**Y where X is of double precision mode and Y of integer mode.

ENTRANCE: The FDXPI subprogram receives control from the object time execution of a compiler-generated linkage. This linkage is generated during compilation when exponential term, in which the base is of double precision mode and the exponent of integer mode, is encountered.

CONSIDERATION: The following considerations apply to the execution of the FDXPI subprogram:

1. An error exists in the term X**Y where X = 0.0 and Y ≤ 0
2. For X = 0.0 and Y > 0, X**Y = 0.0
3. For X ≠ 0.0 and Y = 0, X**Y = 1.0

OPERATION: For positive values of Y, implementation of the exponential term X**Y is the same as that described in subroutine FIXPI. For negative values of Y, the 2's complement of Y is taken and the exponential operation is implemented in the same manner as that described in subroutine FIXPI for positive values of Y; the reciprocal of the result is then taken.

EXIT: After computing the value of term X**Y, the FDXPI subprogram returns control to the object program; however, if an error exists, exit is to subroutine IBFERR.

FRXPR Subprogram

The function of the FRXPR subprogram is to compute the value of an exponential term of the form X**Y where both X and Y are of real mode.

ENTRANCE: The FRXPR Subprogram receives control from the object time execution of a compiler generated linkage. This linkage is generated during compilation when an exponential operation, in which both the base and exponent are of real mode, is encountered.

CONSIDERATION: Errors exist in the term X**Y where:

1. X = 0.0 and Y ≤ 0.0
2. X < 0.0

OPERATION: The computation is based on the identity:

X**Y=e**(Y*log e X)

Log e X is calculated by using the ALOG subprogram; the result is multiplied by Y and e is then raised to the power Y*log e X by using the EXP subprogram.

EXIT: After computing the value of the exponential term, the FRXPR subprogram returns control to the object program; however, if an error exists, exit is to subroutine IBFERR.

FDXPD Subprogram

The function of the FDXPD subprogram is to compute the value of an exponential term of the form X**Y where both X and Y are double precision mode.

ENTRANCE: The FDXPD subprogram receives control from the object time execution of a compiler generated linkage. This linkage is generated during compilation when an exponential operation, in which the base is of double precision mode and the exponent is of either real or double precision mode, is encountered.

CONSIDERATION: The following considerations apply to the execution of the FDXPD subprogram:

1. An error exists in the term X**Y where:
   a. X = 0.0 and Y ≥ 0.0
   b. X < 0.0
2. If an exponential term, of the form X**Y where X is double precision mode and Y is real mode, is encountered during compilation, the exponent Y is converted to double precision. This action permits such an exponential term to be processed by the FDXPD subprogram.

OPERATION: Implementation of the exponential operation is similar to that described in subroutine FRXPR; however, because the operation is double precision, subprograms DLOG and DEXP are used instead of subprograms ALOG and EXP.

EXIT: After computing the value of the exponential term, the FDXPD subprogram returns control to the object program; however, if an error exists, exit is to subroutine IBFERR.

Array displacement is the distance between the first element in an array and a specified element to be accessed 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.

ACCESS

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

ONE DIMENSION

Assume a 1-dimensional array of five elements, expressed as A(5). To access 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 purposes, 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 (e.g., 4 for a real array). 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 accessing a specific element in a 2-dimensional array.

Prior to discussing how a specified element is accessed, the hexadecimal number scheme used to address an array element must be taken into consideration. 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 accessed from the array dimensioned as A(3,2). Observation shows one dimension part must be bypassed in order to access the specified element. The computation to access 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 bypass 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 access the specified element.

THREE DIMENSIONS

The same reasoning can be projected into a 3-dimensional array. For a three-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 purposes, therefore, an element in a 3-dimensional array will be called a dimension section. For example, in the array of 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 accessed 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 access the specified element. The computation to access 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 accessed 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 access the specified element. The preceding example is illustrated in Figure 72.

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

GENERAL SUBSCRIPT FORM

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
```

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
```

```
A(2,2,3)
   |
   | Zero-addressing adjustment
   |
   V
A(1,1,2)
  | | |
  | | |
  | | └---> 2 dimension sections ┐
  | |                            | Must be bypassed to
  | └-----> 1 dimension part     |
  |                              | access specified element
  └-------> 1 array element      ┘
```

Figure 72. Access of Specified Element in Array

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 program. 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 CDL components are calculated during Phase 20.

The second portion of the array displacement:

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

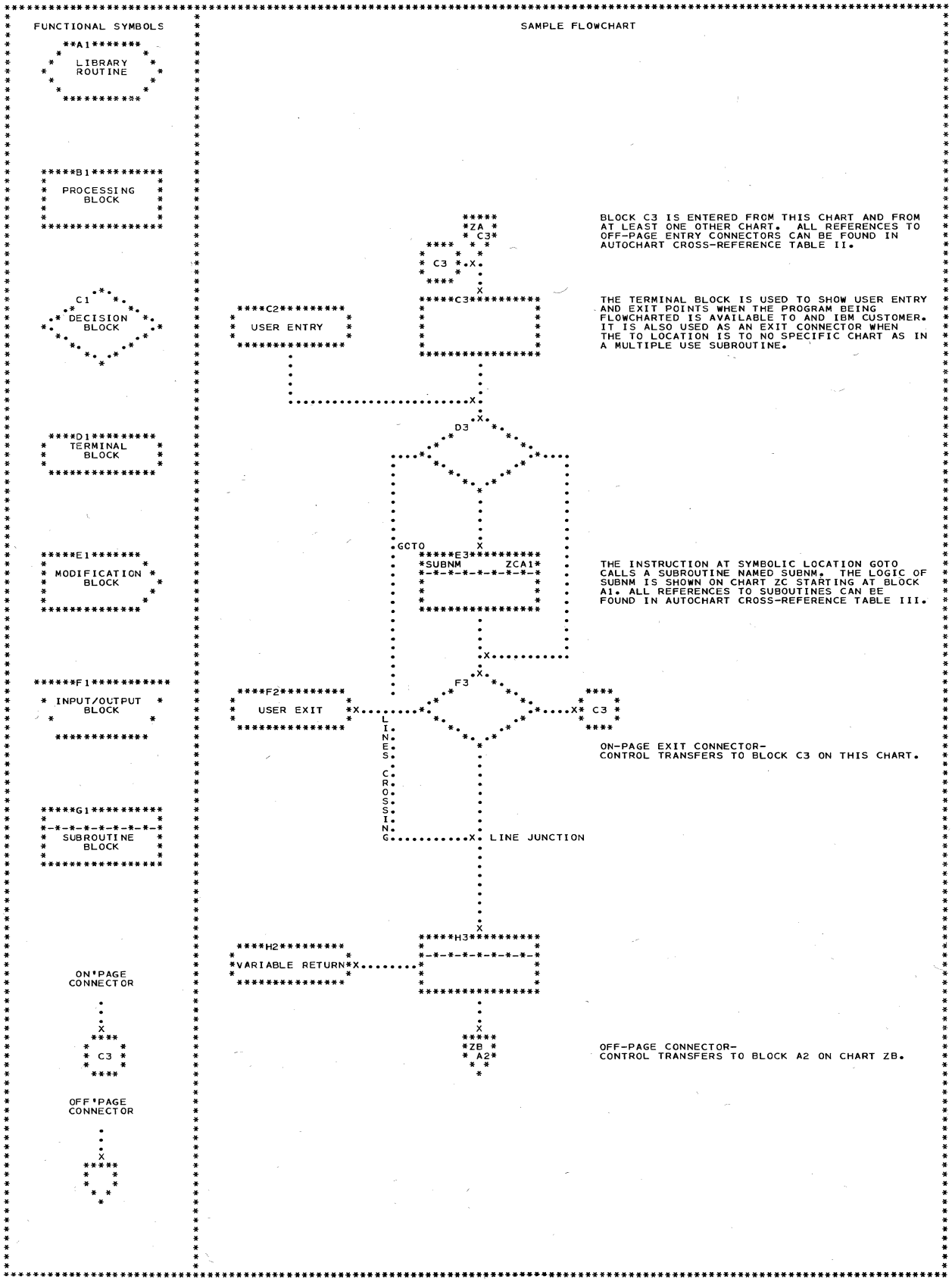
is known as the offset portion and is calculated by Phase 10.

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.
```

APPENDIX D: LIST OF ABBREVIATIONS

ADDR	Address
ADJ	Adjective
ARG	Argument
ARGLST	Argument List
ARITH	Arithmetic
ASF	Arithmetic Statement Function
ASFDEF	Arithmetic Statement Function Definition
ASGND	Assigned
ASSOC	Associated
AVAIL	Available
BALR	Branch and Link Register
BCD	Binary Coded Decimal
BGNG	Beginning
BKSP	Backspace
BND VAR	Bound Variable
BUFF	Buffer
CHAR	Character
COMM	Common
COND	Condition
CORR	Correct
CNT	Count
CTRL	Control
CUR	Current
DEC	Decimal
DICT	Dictionary
DIM	Dimension
DIRCTRY	Directory
D.P.	Double Precision
DR	Drive
DRCTOR	Director
DSPL	Display
EBCDIC	Extended Binary Coded Decimal Interchange Code
ENT	Entry
EODS	End of Data Set
EOF	End of File
EQU	Equivalence
EQUTAB	Equivalence Table
ESD	External Symbol Dictionary
ESID	External Symbol Identification Number
EXP	Exponent
EXT	External
FLDCNT	Field Count
FLT	Floating
FSD	FORTRAN System Director
FUNC	Function
GENRTED	Generated
GT	Greater Than
HEX	Hexadecimal
INCRMT	Increment
IDENT	Identification
IDX	Index
INFO	Information
INSTR	Instruction
INTERMED	Intermediate
I/O	Input/Output
IPL	Initial Program Load
ISN	Internal Statement Number
LFTPRN	Left Parenthesis
LNGTH	Length
LOCATN	Location
LOC CTR	Location Counter
LT	Less Than
NEC	Necessary
NO	Number
OPRND	Operand
OP TBL	Operations Table
PARAM	Parameter
PAREN	Parenthesis
PHSE	Phase
PNTR	Pointer
POS	Position
PREV	Previous
PRGNAME	Program Name
PROG	Program
REF	Reference
REFRNCD	Referenced
REG	Register
REGUL	Regular
REP	Represent
RLD	Relocatable Dictionary
ROUT	Routine
RR	Register to Register
RTN	Return
RTPRN	Right Parenthesis
RX	Register to Storage
SBCRPT	Subscript
SEQ	Sequence
SPEC	Specification
STD	Standard
STMNT	Statement
SUBRTN	Subroutine
SUBS	Subscript
SVC	Service
SW	Switch
SYM	Symbol
SYS	System
TXT	Text
UNASS	Unassigned
VARS	Variations
WARN	Warning
WRK	Work


FUNCTIONAL SYMBOLS
LIBRARY ROUTINE
PROCESSING BLOCK
DECISION BLOCK
TERMINAL BLOCK
MODIFICATION BLOCK
INPUT/OUTPUT BLOCK
SUBROUTINE BLOCK
ON-PAGE CONNECTOR
OFF-PAGE CONNECTOR
SAMPLE FLOWCHART
USER ENTRY
VARIABLE RETURN
USER EXIT
LINES CROSSING
LINE JUNCTION
BLOCK C3 IS ENTERED FROM THIS CHART AND FROM AT LEAST ONE OTHER CHART. ALL REFERENCES TO OFF-PAGE ENTRY CONNECTORS CAN BE FOUND IN AUTOCHART CROSS-REFERENCE TABLE II.
THE TERMINAL BLOCK IS USED TO SHOW USER ENTRY AND EXIT POINTS WHEN THE PROGRAM BEING FLOWCHARTED IS AVAILABLE TO AND IBM CUSTOMER. IT IS ALSO USED AS AN EXIT CONNECTOR WHEN THE TO LOCATION IS TO NO SPECIFIC CHART AS IN A MULTIPLE USE SUBROUTINE.
THE INSTRUCTION AT SYMBOLIC LOCATION GOTO CALLS A SUBROUTINE NAMED SUBNM. THE LOGIC OF SUBNM IS SHOWN ON CHART ZC STARTING AT BLOCK A1. ALL REFERENCES TO SUBOUTINES CAN BE FOUND IN AUTOCHART CROSS-REFERENCE TABLE III.
ON-PAGE EXIT CONNECTOR-
CONTROL TRANSFERS TO BLOCK C3 ON THIS CHART.
OFF-PAGE CONNECTOR-
CONTROL TRANSFERS TO BLOCK A2 ON CHART ZB.

address constant: Area into which the address of a respective routine, external function, or symbol is to be relocated by the FORTRAN loader. It may be used to calculate storage addresses.

argument: A variable that is given a constant value for a specific purpose or process. An independent variable.

argument list: List containing the addresses of arguments constructed when an adjective code indicating a call to an external or arithmetic statement function is detected.

array displacement: The distance in bytes between the first element in an array and a specified element to be accessed from the array.

backward DO: Condition occurring when the statement ending the DO loop is sequentially in front of the statement that defines the DO.

base displacement address: A 2-byte address in hexadecimal representing the base register and the displacement in a machine language instruction.

bound variable: An integer variable that is used in a subscript expression and redefined.

branch table: A table compiled by the FORTRAN compiler: resident in the object program: a list of statement number addresses that control branching.

CDL: A portion of the array displacement for a subscript expression; calculated by utilizing Constant, Dimension, and Length information.

chain: A series of items linked together by addresses.

chaining: Technique used by the FORTRAN compiler to arrange and retrieve items entered in the dictionary and overflow table.

COMMON text: Table of variables assigned to the COMMON area by COMMON statements in the source program.

communications area: Central gathering area for information common to all phases and communications between phases.

control segment: The output of a single compilation within the user program.

DECK option: An option that indicates that the object program is to be punched on cards during compilation.

data parameter: The address of the first byte of data to be processed, and the number of bytes to be processed.

data set: A named collection of data in one of several prescribed arrangements.

data set control block (DSCB): A block that varies in size from 22 to 44 bytes. Describes the physical device identified in DSTAB and the extent of operations to be performed on that device.

data set table (DSTAB): A list referenced from the I/O routines, and composed of one 6-byte block for each data set.

decimal length: Number of bytes reserved for decimal places within the field length in a FORMAT statement.

dictionary: A reference area that contains all names, constants, and data set reference numbers used in the program.

displacement: Distance in bytes between a variable and its root in an EQUIVALENCE class or group.

DO list: A list of subscript expressions within the DO loop.

dummy: Something characteristic of a specified item, but not having the capacity to function as that item.

dummy variable: A dummy used to define operations performed on arguments in statement function or subprogram definitions.

element count: The number of entries in an EQUIVALENCE group or class.

end DO: The statement that ends a DO loop.

end of data set: A signal that the last record of a data set has been read or written.

error: Incorrect usage of the FORTRAN language that can force the end of compilation.

epilog table: Area containing information necessary to return the value of variables used as parameters to the calling program.

EQUIVALENCE class: A number of EQUIVALENCE groups linked together by names common to two or more groups.

EQUIVALENCE group: Names between a left and right parenthesis in an EQUIVALENCE statement.

EQUIVALENCE root: A member of an EQUIVALENCE group or class to which all other variables are equated.

EQUIVALENCE table: A table used by the subroutines that assign addresses for EQUIVALENCE entries.

EQUIVALENCE text: Table of EQUIVALENCE groups assigned by EQUIVALENCE statements.

ESD cards: Cards containing segment names and external and internal entries to the segments in the program being compiled.

ESD table: A table which contains the address of each external symbol and an address constant.

executable statement: A statement that causes the compiler to generate machine instructions.

explicit specification statement: Statement which declares the mode of a particular variable or array by its name.

field count: The number of times a conversion is to be repeated for an I/O list.

field length: Number of bytes reserved in the input/output record for the variable in the record.

forcing scan: Directs the ordering of text words of a statement by comparing the forcing values of the respective adjective codes.

FORMAT specification group: FORMAT specifications that appear within a set of parentheses.

GO (Compile and Go) option: Option indicating that an entire job is to be compiled and executed if there are no serious source program errors.

GOGO option: Entire job is to be compiled and executed irrespective of any source program errors.

halt number: A number identifying a STOP or PAUSE.

image: Refers to the BCD card image of a symbol in the dictionary.

immediate DO parameter: A constant, less than 4096 bytes, used as a parameter in a DO or implied DO statement.

implied DO: A method of indexing arrays in input/output lists.

index mapping table: A table which maintains a record of all registers used at object time as index registers in subscript calculation, and a record of the unique subscript expression associated with each register.

in-line function: Function that generates code whenever it occurs in the source program.

intermediate text: An internal representation of the source program that can be easily converted to machine language instructions.

internal statement number: a number assigned to each FORTRAN statement before it is processed.

I/O list: A list of variables and arrays in READ/WRITE statements.

IPL: T he act of initial program load on an IBM System/360 computer.

job: One or more source or object programs in many combinations along with any associated input data.

keyword: A FORTRAN reserved word which indicates the specific FORTRAN statement to be compiled.

LIST option: An option which indicates that the source program is to be printed (listed).

literal: Data which is defined in the source program as opposed to being read by I/O commands.

location counter: A counter used to assign addresses.

main program: A program to which control is transferred upon completion of the relocatable loading of a set of programs.

main storage: All addressable storage from which instructions can be executed or from which data can be loaded directly into registers.

MAP option: Indicates the storage map of an entry is to be printed on-line.

mode: A code used in the dictionary and intermediate text denoting whether a symbol or literal represents real, integer, or double precision. In I/O, mode indicates

set mode and whether the function is multiplex or burst mode.

multiple job: More than one compilation with no regard to the number of object programs within a job.

name: A string of alphabetic and numeric characters the first of which must be alphabetic.

nested implied DO: An implied DO within another implied DO.

offset: A calculated indexing factor used to find the correct element in an array for a particular subscript expression or EQUIVALENCE element.

operations table: Temporary storage area used during the ordering of operations within a statement for any text words referring to the operation.

ored: An inclusive OR machine operation.

overflow table: A table which contains all dimension, subscript, and statement number information within the program.

overlay: Loading a program or data into a portion of storage where the current resident program or data is no longer required.

parameters: Variables given a constant value for a specific process or purpose.

parenthesis count: Determines whether an implied DO is nested within another implied DO; determines hierarchy of arithmetic operations.

pointer: An address indicated in text by p that denotes the location of data or another address.

point of definition: Point at which a statement number is referenced by some statement other than a DO or FORMAT statement.

redefinition: Point at which the value of an integer variable changes.

RLD cards: Contain addresses of items to be relocated by the FORTRAN loader at LOAD time.

save register technique: When a register is required but one is not available, the contents of required registers are placed in the first available work area.

sequence error: FORTRAN statements that are out of sequence.

single job: Single compilation of a source program or subprogram, with no regard to the number of object programs within a job.

skeleton instructions: Instructions generated for use as constants or literals to generate instructions.

spill base register: When registers 4, 5, 6, and 7 are used as base registers, register 7 accepts all overflow.

spill technique: A method of using the spill base register as a temporary base register by inserting the proper base value into the register before use, information spills into the next register with the exception of register 7 which repeats.

string: Contiguous group of characters with no embedded blanks.

subprogram: A program that is a FUNCTION or SUBROUTINE.

subscript table: Temporary storage area for subscript text.

thumb index: A storage area which contains the addresses of the first entry for each chain.

type: A code used in the dictionary and intermediate text denoting whether a symbol represents a variable, array, function, or constant.

unassigned register: Condition existing when a register is available for use as an index register at object time.

warning: An error that is not serious enough to abort object program execution.

zero addressing scheme: A numerical scheme with zero as the lowest value.

IBM S/360 Printed in U.S.A. Z28-6620-0

IBM®
International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N. Y. 10601

READER'S COMMENT FORM

IBM System/360 Form Z28-6620-0
Basic Programming Support
FORTRAN IV
Program Logic Manual

- **Your comments, accompanied by answers to the following questions, help us produce better publications for your use. If your answer to a question is "No" or requires qualification, please explain in the space provided below. All comments will be handled on a non-confidential basis. Copies of this and other IBM publications can be obtained through IBM Branch Offices.**

	Yes	No
• Does this publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

- What is your occupation? _____
- How do you use this publication?

As an introduction to the subject?	☐	As an instructor in a class?	☐
For advanced knowledge of the subject?	☐	As a student in a class?	☐
For information about operating procedures?	☐	As a reference manual?	☐

Other _____

- Please give specific page and line references with your comments when appropriate. If you wish a reply, be sure to include your name and address.

COMMENTS:

- **Thank you for your cooperation. No postage necessary if mailed in the U. S. A.**

IBM **Technical Newsletter**

File Number	S360-25
Re: Form No.	Z28-6620-0 (formerly C28-6584-0)
This Newsletter No.	Z28-2117
Date	January 11, 1966
Previous Newsletter Nos.	None

IBM System/360 Basic Programming Support
FORTRAN IV, 360P-FO-031
Program Logic Manual

This technical newsletter amends the publication IBM System/360 Basic Programming Support FORTRAN IV, Program Logic Manual, 360P-FO-031, Form Z28-6620-0 (formerly C28-6584-0). The attached pages replace and supplement pages in the original publication. Corrections, additions, and deletions to the text are noted by vertical bars to the left of the change. The attached pages are:

21-22	51-52	101-102
23-24	53-54	111-112
27-28	55-56	123-124
33-34	59-60	229-230
35-36	61-62	368A
43-44	67-68	485-486
49-50		

Summary of Amendments

The capabilities of the error processor of the FORTRAN System Director has been expanded. An additional message (FIW) has been added. See pages 22, 33-36, 43, 44, 51-56.

A CCunit routine has been added to the Control Card routine. See pages 59, 62, 67.

Chart KU, Phase 25, has been added. See page 368A.

Detailed changes have been made on pages 23, 27, 101-102, 111, 123, 229, and 486.

Note: Please file this cover letter at the back of the publication. Cover letters provide a quick reference to changes and a means to check the receipt of all amendments to the manual.

RESTRICTED DISTRIBUTION

International Business Machines Corp., Programming Systems Publications, P. O. Box 390, Poughkeepsie, N. Y.

PRINTED IN U.S.A.

IBM **Technical Newsletter**

File No. S360-25

Re: Form No. Z28-6620-0
(formerly C28-6584-0)
This Newsletter No. Z31-5008

Date: March 15, 1966

Previous Newsletter Nos. Z28-2117

Replacement pages for IBM System/360 Basic Programming Support, FORTRAN IV, Program Logic Manual, Form Z28-6620-0 (formerly Form C28-6584-0).

To bring your publication up to date, please replace the following pages with the corresponding pages attached to this Newsletter. Changes are indicated by a vertical line at the left of the affected text, and by a dot (•) at the left of an affected figure.

Replace pages

1-2
35-36
55-56
421-422
447-448
449-450
451-452
499-(500, new back cover)
(501-502, new reader's comment form)

Please insert this page to indicate that your publication now includes the modified pages issued with Technical Newsletters:

Form	Modified Pages	Date
Z28-2117	22, 23, 27, 28, 33, 34, (35, 36 - replaced by following TNL) 43, 44, 49, 51-56, 59, 62, 67, 101, 102, 111, 123, 229, 368A, 486	January 11, 1966
Z31-5008	1, 2, 35, 36, 55, 56, 421, 422, 448, 450, 452, back cover (500). reader's comment form (501, 502)	March 15, 1966

IBM Corp., Programming Publications Dept., Rochester, Minn. 55901

PRINTED IN U.S.A.