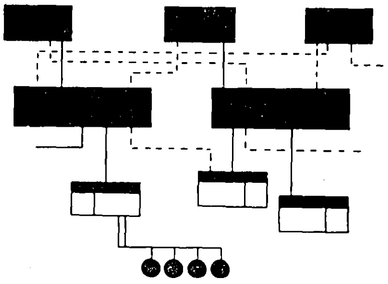
GENERAL ELECTRIC
COMPUTERS

GE-625/635
FORTRAN IV
Compiler

SYSTEM
SUPPORT
INFORMATION



ABSTRACT

This manual provides a description of the FORTRAN IV Compiler for the GE-625/635 computers. Details of the Compiler are discussed.

GENERAL ELECTRIC

CPB-1130

GE-625 / 635
FORTRAN IV
COMPILER

September 1965

GENERAL ELECTRIC
COMPUTER DEPARTMENT

PREFACE

The FORTRAN language--resembling the language of mathematics--is used primarily for writing scientific and engineering programs to be run on a computer. The GE-625/635 FORTRAN IV Compiler translates FORTRAN programs into machine language acceptable to a GE-625/635 computer.

This manual describes in detail the design and function of the FORTRAN IV Compiler. The reader should be experienced in programming and have a good working knowledge of FORTRAN IV.

CONTENTS

1. STRUCTURE OF THE COMPILER 1

2. THE EXECUTIVE PHASE 47

3. PHASE ONE--FORTRAN IV COMPILER 59

ILLUSTRATIONS

Figure

1. STRUCTURE OF THE COMPILER

The FORTRAN IV Compiler for the GE-625/635 computer is composed of three sections:

1. Executive Phase
2. Phase One
3. Phase Two

Each of the three phases consists of a group of relocatable subprograms written in the Macro Assembly Program (GMAP) assembly language.

When the General Comprehensive Operating Supervisor (GECOS) has determined that sufficient memory can be allocated for the FORTRAN IV Compiler, a Master Mode Entry (MME) is made to the System Loader (GECALL). The allocation parameters are set to ten minutes, 32k memory, and 10k print lines. Allocation is also made for the following files:

File Name	Purpose
S*	Source program input file
*1	Scratch file for POOL Table
P*	Printer listing output file
G*	Output; GMAP coding (used as input to assembly program)
B*	Output; object deck for loading (used by FORTRAN IV Compiler and GMAP assembly program)
C*	Output; object deck for punching
K*	Output; COMDECK output of FORTRAN IV source statements when requested

The entry to GECALL causes the System Loader to load the Executive Phase of the FORTRAN IV Compiler from the system library. When loading is completed, control is transferred to the Executive Phase and the compiler begins execution as a free-standing slave program.

THE EXECUTIVE PHASE

The Executive Phase of the FORTRAN IV Compiler is composed of six major segments. Since the Executive Phase remains in memory throughout the entire compilation, several routines common to Phase One and Phase Two are contained in this portion of the compiler. In addition, that part of the compiler responsible for overall control is a major component of the Executive Phase. Briefly, the Executive Phase performs the following functions:

1. Checks interfaces using the Switch Word.
2. Determines the size of a working storage and performs the necessary initializations.
3. Calls Phase One of the compiler and receives control when Phase One is complete.
4. Allocates additional working storage as allowed and calls for Phase Two loading.
5. Opens and closes files as needed by the compiler.
6. Calls for the GMAP assembler at the end of compilation.
7. Writes diagnostic messages.
8. Manipulates tables.
9. Converts binary integers to a BCD form.
10. Outputs GMAP coding for the FORTRAN IV source statements.
11. Dumps memory when serious compilation errors occur.

Chapter 2 of this manual will describe in detail the functions of the Executive Phase of the compiler.

PHASE ONE

The primary function of Phase One of the FORTRAN IV Compiler is the translation of the source program statements into a series of correlated table entries.

Each statement of the source program is processed using one or more closed subroutines. In most cases, specific subroutines exist for each type of statement. Additional subroutines are available to process features common to several types of statements, such as conversions, etc. Frequently, the statement processor subroutine may call on one or more other subroutines to accomplish the translation of the statement. This hierarchy of subroutines is characterized by the return of control through the same linkage path from which control was originally obtained.

Initially, the first source statement is checked to determine its type. If the first statement is a comment (* or C character in column 1), then columns 2 through 9 are used as a label for the object program deck and columns 13 through 72 are used as a title for the printed output listing. If the second statement is also a comment, columns 13 through 72 will be used as a subtitle for the printed output listing. Non-comment statements, whether they occur initially or following the first or second statements, are scanned to determine if they are arithmetic or nonarithmetic. It should be noted that the entire statement is scanned including all continuation cards. If it is determined that the statement is arithmetic, the Arithmetic Processor will be called. Additional tests are made on non-ARITHMETIC statements to determine which specific subroutine must be used. Each statement is translated in turn until the END statement is processed, at which time the control is returned to the Processor Executive Controller of the Executive Phase.

Chapter 3 of this document describes in detail the functions of Phase One of the FORTRAN IV Compiler.

PHASE TWO

The second phase of the FORTRAN IV Compiler is composed of two sections; the first is the indexer routines and the second is the main compiler. Through the use of a series of closed subroutines, the following functions are performed:

1. The DATA statements of the source program receive their final processing and the required GMAP code is generated.

2. The source program is checked to ensure that every statement can be reached at execution.

3. The indexer routines are called to generate the necessary indexing instructions.

4. The main compiler portion of Phase Two processes all source statements except DO, DO-ENDING and END statements.

5. Assembly language coding which has been temporarily stored in tables is merged.

6. The prologue logic of necessary save and initialization instructions is compiled including the erasable storage to be used at execution.

Upon completion of the Phase Two functions, control is returned to the Executive Phase of the compiler. Chapter 4 of this manual describes in detail the functions of Phase Two of the FORTRAN IV Compiler.

STORAGE MAP AND OVERLAYS

Figures 1, 2 and 3 illustrate storage maps for the three phases.

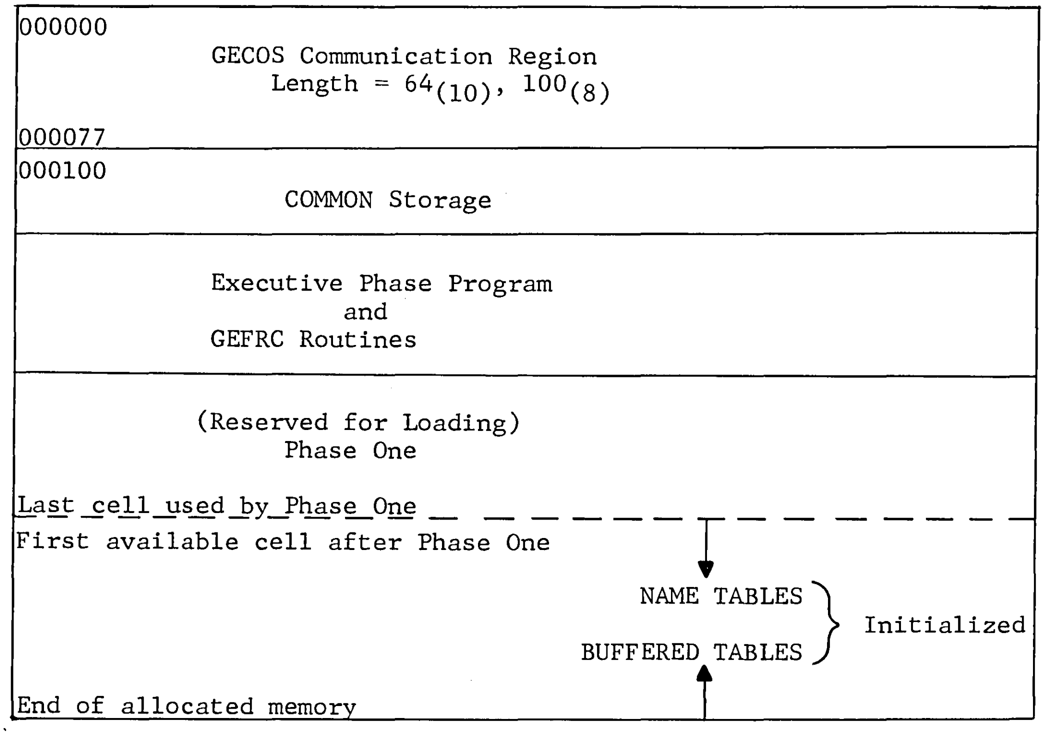


Figure 1. Executive Phase Storage Map

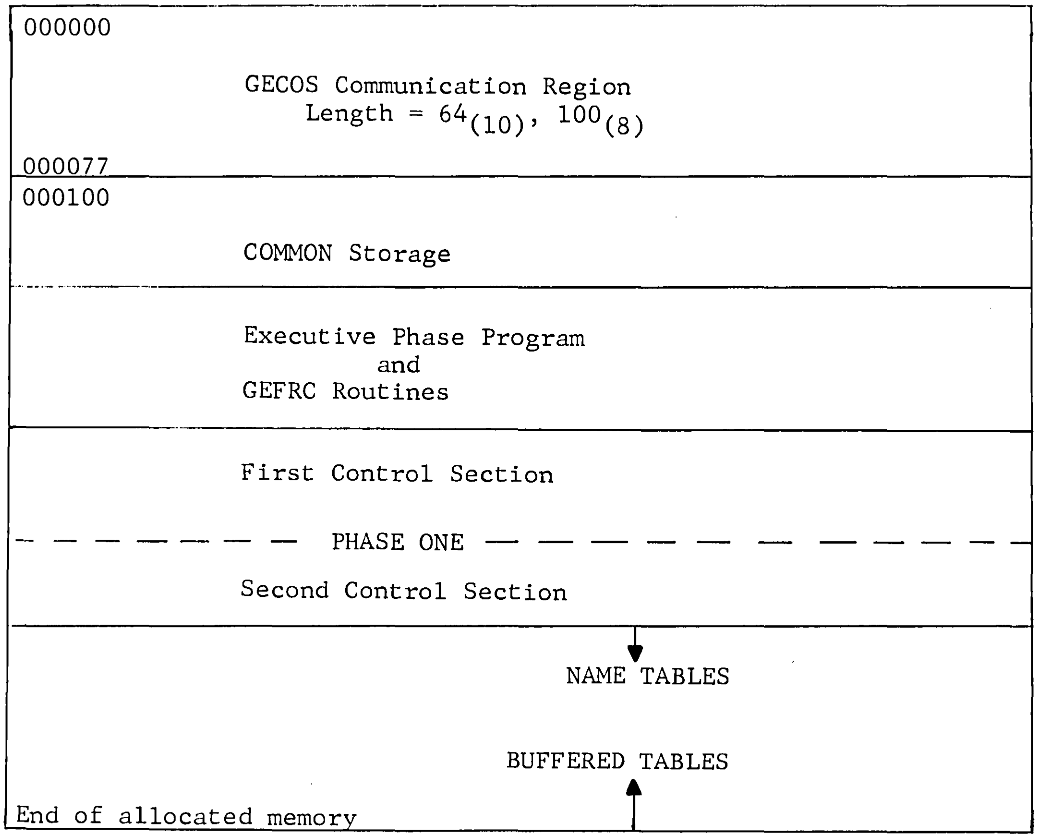


Figure 2. Phase One Storage Map

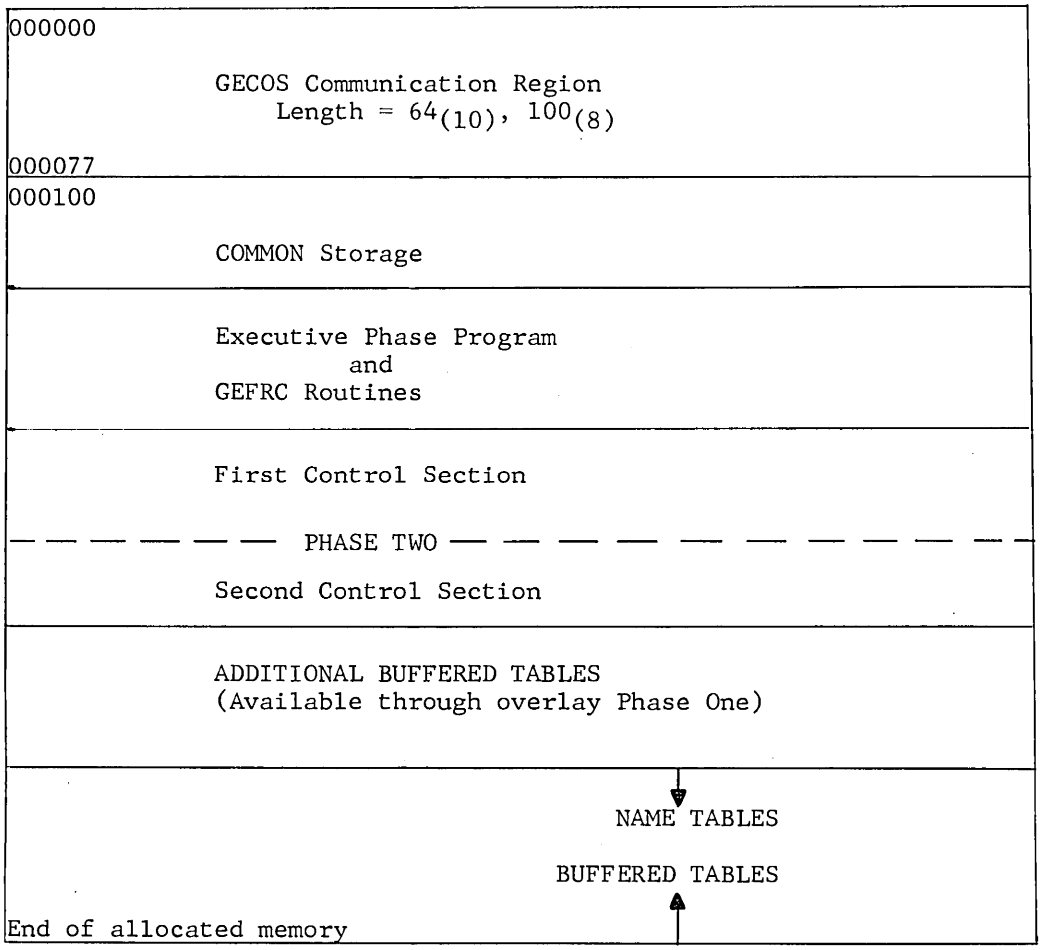


Figure 3. Phase Two Storage Map

INPUT/OUTPUT RELATIONSHIPS

The input and output functions of the FORTRAN IV Compiler are handled by the Generalized File and Record Control (GEFRC) program.

When the System Library is being created, the three phases of the FORTRAN IV Compiler are combined with the required GEFRC routines which are obtained from the Subroutine Library tape. The FORTRAN IV Compiler becomes a three link program in the System Library. For a complete description of the GEFRC system, the reader should refer to the manual GE-635 File and Record Control, CPB-1003.

GECOS allocates the following files for use by the FORTRAN IV Compiler:

File	Purpose
S*	Source input file
*1	Scratch file for POOL entries
P*	Printer output file
G*	Output; GMAP coding
B*	Output; object deck
K*	Output; COMDECK of FORTRAN IV source statements

Throughout the execution of the FORTRAN IV Compiler, GEFRC is used for the required manipulation of these files.

TABLE DESCRIPTIONS

There are three types of tables generated and used by the FORTRAN IV Compiler. Each of these tables and the specific formats are described below.

The POOL Table

Introduction. The POOL Table is generated and written on the scratch file *1 during Phase One. Each entry in the POOL Table consists of one or more words. The first word (entry) is of the general form:

0 2	3 17	18 20	21 35
7	Type Number(octal)	0	IFN (binary)

where:

IFN = the Internal Formula Number assigned to the statement generating the POOL Table entry.

TYPE NUMBER = a unique numeric code assigned to each different type of executable statement.

The table shown in Figure 4 lists the Type Numbers assigned to executable statements in the POOL Table.

STATEMENT TYPE	SYMBOL NAME	CODE$_{(8)}$
ARITHMETIC	Y.ARIT	01
ARITHMETIC STATEMENT FUNCTION	Y.ASFD	02
NULL	Y.NULL	03
BCD COMMENT	Y.BCDC	04
ARITHMETIC IF	Y.ARIF	05
LOGICAL IF	Y.LGIF	06
GO TO (Unconditional)	Y.UNGO	07
GO TO (Assigned)	Y.ASGO	10
GO TO (Computed)	Y.COGO	11
PUNCH	Y.PRPN	12
PRINT	Y.PRPR	13
READ CARDS	Y.PRRC	14
PAUSE	Y.PAUS	15
DEBUG ARITHMETIC IF	Y.DIFA	16
DEBUG LOGICAL IF	Y.DIFL	17
ASSIGN	Y.ASSN	20
DO	Y.IODO	21
CONTINUE	Y.CONT	22
STOP	Y.STOP	23
WRITE TAPE	Y.PRWT	24
END	Y.END.	25
CALL	Y.CALL	26
RETURN	Y.RETN	27
DEBUG FOR	Y.DBUG	30
READ TAPE	Y.PRRT	31
END FILE	Y.PREN	32
REWIND	Y.PRRW	33
BACKSPACE	Y.PRBK	34
Not used	------	35
SETIN	Y.STIN	36
END READ	Y.ENDR	37
END WRITE	Y.ENDW	40
SETOUT	Y.STOT	41
DO Ending	Y.DODO	42

Figure 4. Type Numbers

POOL Table Formats. The format of each of the possible entries into the POOL Table is described below for all of the executable statements in the FORTRAN IV Compiler. In the following descriptions, the notation "T.ARIT Information" signifies POOL Table entries generated by the Arithmetic Statement Processor and the notation "NAME" is the relative pointer to the flag word of a NAME Table entry.

1. The ARITHMETIC Statement

0 2	3 17	18 20	21 35
7	Y.ARIT (Type $1)_8$	0	IFN
T.ARIT Information ⋮			

2. Arithmetic Statement Function Definition

0 2	3 17	18 20	21 35
7	Y.ASFD (Type $2)_8$	0	IFN
T.ARIT Information ⋮			

3. NULL Statement

0 2	3 17	18 20	21 35
7	(Type $3)_8$	0	1

4. BCD Comment

0 2	3 17	18 20	21 35
7	Y.BCDC (Type $4)_8$	0	$14_{(8)}$
Comment card text, $12_{(10)}$ words ⋮			

5. ARITHMETIC IF Statement

0 2	3 17	18 20	21 35
7	Y.ARIF (Type 5)$_8$	0	IFN
T.ARIT Information ⋮			
0	COUNT = 3	0	P(T.JUMP)$_1$
0	P(T.JUMP)$_2$	0	P(T.JUMP)$_3$

The parenthesized expression is processed by the Arithmetic Processor, page 93.

6. LOGICAL IF Statement

0 2	3 17	18 20	21 35
7	Y.LGIF (Type 6)$_8$	0	IFN
T.ARIT Information ⋮			
0	COUNT = 1	0	P(T.JUMP)-False

7. Unconditional GO TO Statement

0 2	3 17	18 20	21 35
7	Y.UNGO (Type 7)$_8$	0	IFN
0	P(T.JUMP)	0	0

8. Assigned GO TO Statement

0 2	3 17	18 20	21 35
7	Y.ASGO $(\text{Type } 10)_8$	0	IFN
0	BRANCH COUNT	0	$P(T.JUMP)_1$
	⋮		
0	$P(T.JUMP)_{n-1}$	0	$P(T.JUMP)_n$
0	(NAMEP)Ø	0	00000

where Ø is the GO TO variable.

9. Computed GO TO Statement

0 2	3 17	18 20	21 35
7	Y.COGO $(\text{Type } 11)_8$	0	IFN
0	BRANCH COUNT	0	$P(T.JUMP)_1$
	⋮		
0	$P(T.JUMP)_{n-1}$	0	$P(T.JUMP)_n$
0	(NAMEP)Ø	0	0

where Ø is the GO TO variable.

10. PUNCH (cards), PRINT, and READ (cards) Statements

ØL0000 Pg 139

0 2	3 17	18 20	21 35
7	Y.PRRC/Y.PRPR/ Y.PRPN	0	IFN
0	0	f	n

Begin I/O

where Y.PRRC = 14 octal for READ
Y.PRPR = 13 octal for PRINT
Y.PRPN = 12 octal for PUNCH
and f = 4 if n is a constant FORMAT reference
= 0 if n is a variable FORMAT reference

11. PAUSE Statement

0 2	3 17	18 20	21 35
7	Y.PAUS (Type $15)_8$	0	IFN
0	0	0	N

12. DEBUG ARITHMETIC IF Statement

0 2	3 17	18 20	21 35
7	Y.DIFA (Type $16)_8$	0	IFN

T.ARIT Information
⋮

0 11	12 23	24 35
Transfer code	Transfer code	Transfer code

13. DEBUG LOGICAL IF Statement

0 2	3 17	18 20	21 35
7	Y.DIFL (Type 17)$_8$	0	IFN
	T.ARIT Information		
0	0	0	False Transfer IFN

14. ASSIGN Statement

0 2	3 17	18 20	21 35
7	Y.ASSN (Type 20)$_8$	0	IFN
0	(NAMEP)Ø	0	EFN

where Ø is the variable appearing in an Assigned GO TO statement.

15. DO Statement

0 2	3 17	18 20	21 35
7	Y.IODO (Type 21)$_8$	0	IFN
0	P(T.IODO)	0	0

16. CONTINUE Statement

0 2	3 17	18 20	21 35
7	Y.CONT (Type 22)$_8$	0	IFN

17. STOP Statement

0 2	3 17	18 20	21 35
7	Y.STOP (Type 23)$_8$	0	IFN

18. READ or WRITE Tape Statement

0 2	3 17	18 20	21 35
7	Y.PRRT/Y.PRWT	0	IFN
f1	u	f2	n

Begin I/O

where Y.PRRT = $31_{(8)}$ for READ tape
Y.PRWT = $24_{(8)}$ for WRITE tape
and f1 = 4 if u is a constant
= 0 if u is a variable
f2 = 4 if n is a constant
= 0 if n is a variable or zero (binary read-write)

19. END Statement

0 2	3 17	18 20	21 35
7	Y.END (Type $25)_8$	0	IFN

20. CALL Statement

With Arguments

0 2	3 17	18 20	21 35
7	Y.CALL (Type $26)_8$	0	IFN
T.ARIT Information ⋮			

The Subprogram Name as well as the argument list is processed by the Arithmetic Processor.

Without Arguments

0 2	3 17	18 20	21 35
7	Y.CALL (Type $26)_8$	0	IFN
0	00000	0	00000
0	CODE	0	(NAMEP)Ø
0	00000	0	00000

where Ø is the Subprogram Name and CODE is the Function Operator Code needed by the Arithmetic Processor in Phase 2.

21. RETURN Statement

0 2	3 17	18 20	21 35
7	Y.RETN (Type $27)_8$	0	IFN

A second entry will follow depending on the type of RETURN statement used in the source program.

0 2	3 17	18 35
4	Return Number	0

or, variable RETURN statements will be of the form:

0 2	3 17	18 35
0	P(T.NAME)	0

where P(T.NAME) is the pointer to the RETURN variable name.

22. DEBUG FOR Statement

0 2	3 17	18 20	21 35
7	Y.DBUG (Type $30)_8$	0	IFN

0 2	3 17	18 35
0	Parameter Count (1, 2 or 3)	Parameter 1 (m_1)

0 2	3 17	18 35
0	Parameter 2 (m_2)	Parameter 3 (m_3)

Note that the third word will not be present if the Parameter Count is equal to one.

23. END FILE, REWIND and BACKSPACE Statements

0 2	3 17	18 20	21 35
7	Y.PRBK/Y.PRRW/ Y.PREN	0	IFN
f	u	0	0

where Y.PRBK = 34 octal for BACKSPACE
Y.PRRW = 33 octal for REWIND
Y.PREN = 32 octal for END FILE
and f = 4 for a constant unit reference
= 0 for a symbolic unit reference

24. END READ and END WRITE Statements

0 2	3 17	18 20	21 35
7	Y.ENDR/Y.ENDW	0	IFN
f1	u	f2	n

END I/O

where Y.ENDR = 37 octal for END READ
Y.ENDW = 40 octal for END WRITE
f1 = 4 if u is a constant unit reference
= 0 if u is a variable unit reference
f2 = 4 if n is a constant FORMAT reference
= 0 if n is a variable or zero (binary read-write)

25. DO Ending Statement

0 2	3 17	18 20	21 35
7	Y.DODO (Type 42)$_8$	0	IFN

The format of the general Input/Output POOL Table String is shown in Figure 5.

	0 2	3 17	18 20	21 35	
	7	TYPE	0	IFN	Begin I/O
	f1	u	f2	n	
Variable String	7	SETIN/SETOUT	0	IFN	I/O List if any
	f3	NAMEP (var 1)	0	$P(T.SUBS)_1$ *	
	⋮	⋮	⋮	⋮	
	f3	NAMEP (var n)	0	$P(T.SUBS)_n$ *	
Implied DO	7	Y.IODO (Type 21)	0	IFN	
	0	P(T.IODO)	0	0	
DO Ending	7	Y.DODO (Type 42)	0	IFN	
	⋮	⋮	⋮	⋮	
	7	Y.ENDR (Type 37)/ Y.ENDIN (Type 40)	0	IFN	End I/O
	f1	u	f2	n	

where Type = 14 octal for READ
= 13 octal for PRINT
= 12 octal for PUNCH
= 31 octal for READ TAPE
= 24 octal for WRITE TAPE
f1 = 4 if u is a constant
= 0 if u is a variable
f2 = 4 if n is a constant
= 0 if n is a variable or zero (binary read-write)
f3 = 3 if the variable is a dimensioned variable
= 0 if the variable is nondimensioned
SETIN = 36 octal } set depending whether I/O is input or
SETOUT = 41 octal } output

*The pointer to the T.SUBS Table is entered when the variable name being processed is dimensioned and the dimension exists in the list. For a "short list" notation usage of a dimensioned variable the P(T.SUBS) field is zero.

Figure 5. I/O POOL Table String Format

The POOL Table Routine--S.TP00

Purpose. The FORTRAN IV Compiler contains two routines named S.TP00. The routine loaded and used in Phase One performs output functions for building the POOL Table. The S.TP00 routine loaded and used in Phase Two performs an input function and pulls entries from the POOL Table. These routines either make a POOL Table entry or pull an entry from the POOL Table, depending on whether the call is made in Phase One or Phase Two, respectively.

Method. These routines use the PUTBK and GET entry points of the I/O Editor. The buffer size is initialized at 318 words. During Phase One, it is the responsibility of this routine to append a label word to each record written. (This label is checked for sequencing in Phase Two when retrieving records to ensure no lost or out-of-order information.)

Usage.

Phase One--The calling sequence for Phase One includes parameters which indicate the location and number of words to be accumulated in making the POOL Table entry. There may be more than one parameter and the routine will continue to pick up parameters until one is encountered which does not contain a bit in position 18. Thus, the information to be placed in the POOL Table may be contained in as many randomly located buffers within memory as necessary.

```
TSX1        S.TP00

ZERO        LOC,N   (Location of entry,
                     number of words)
 .
 .
 .
 .
 .
(Normal return)
```

Phase Two--In Phase Two, a call to S.TP00 will return with the A-register (0-17) containing the initial address of the entry and the tally position (18-29) containing the length of the entry in words.

```
TSX1        S.TP00
(Normal return)
```

Error Returns. The following errors will cause the processing to be discontinued and return made directly to the Processor Executive Controller of the Executive Phase of the compiler.

The Buffered Tables (BU Tables)

Introduction. The BU Tables are used extensively in Phase One and Phase Two of the compiler. The buffer size is initially set to sixty words.

Memory space is allocated for the BU Tables and for the NAME Tables (page 43) by the Executive Phase of the compiler in the following way.

The Base Address Register (BAR) is checked to determine the upper bound of memory as allocated by GECOS. The location TOP. is set to the value of the highest available memory location allocated and the cell BOT. is initialized from the location EPH1. The latter specifies the first available location following Phase One. The area specified by the contents of BOT. and TOP. will be used for the BU Tables and for the NAME Table during the execution of Phase One. In Phase Two some additional space is available from the smaller size of Phase Two after it overlays the larger Phase One. The BU Tables are assigned from the top of available memory downward; the NAME Table is assigned upward in available storage.

If an overlap occurs, an immediate exit is made to the control routine of the Executive Phase.

Buffers of one table are connected via the label word containing the location of the next buffer and location of the last buffer. When there is no longer any use for all or part of a table, buffers may be released for reuse by another table. A table of indexes which contains the location of the initial buffer and location of the final buffer for each table is maintained. When this index = 0, the table has not yet been allocated. An index of tally words for each table is also maintained. The tally words are pointers to next available buffer word. When the tally word = 0, the table has not been located.

For table label and index nomenclature the following definitions will hold:

IB - Initial Buffer
FB - Final Buffer
NB - Next Buffer
LB - Last Buffer
R - Reuse
PTR - Pointer to Buffer
Indexes consist of (IB, FB)
Labels consist of (NB, LB)

Table of Indexes

```
S.TL98

             +1 (IB1,FB1)          table #1
             +2 (IB2,FB2)          table #2
                   .
                   .
                   .
             +n (IBn,FBn)          table #n
```

Table of TALLY Words

```
T.TABL                     **,ID

             +1 TALLY    A,n       table #1
             +2 TALLY    A,n       table #2
                   .
                   .
                   .
             +n TALLY    A,n       table #n
```

where: A is address of next available buffer word (buffer + 1 if buffer is full)
n is number of words remaining in the buffer
A=n=0 means table not located yet.

An example of how buffers for a table are strung by labels is as follows:

Buffer #1 Label = (NB2,0)
Buffer #2 Label = (NB3,LB1)
Buffer #3 Label = (NB4,LB2)
Buffer #n Label = (0,LB(n-1))
Table Index = (IB1,FBn)

When an entry is made in a table, the last word of the entry is followed by an end mark (all bits) unless the last word of the entry is made in the last location of a buffer. Therefore, all buffers that are partially filled will have an end mark flag. Note: only one end-mark flag will appear in any one buffer.

Some table entries are constant in length (1, 2, or 3 words) and the entire entry will always be complete within a buffer. Other entries are variable in length and may be allowed to overlap the end of a buffer and continue in the next buffer. The table entries that may continue across buffers are entries made to the T.COMO, T.ARGS, T.DATA, T.LITR, T.IPRO and T.NUMB tables. All other variable length entries must be completed in a single buffer, and if enough room is not available, an end mark is written and a new buffer is allocated.

A NAME Table pointer (NAMEP) reference in a table entry points to the flag word of the appropriate NAME entry. Other entries that are pointers will point to the first word of a table entry.

BU Table Formats. The format of the BU Tables generated in Phase One is as follows:

1. T.ARGS Table: An entry is made for each FUNCTION, SUBROUTINE or ENTRY statement that has an argument list. The number of words in each entry is 1+(n/2) where n is the number of dummy arguments. The first word of an entry is:

0 5	6 17	18 35
N	Arg. Count	NAMEP$_1$

where N = 0, if the argument list is for a SUBROUTINE or FUNCTION statement.
= 1,2,...n, if the argument list is for an ENTRY statement.

The sequential numbering for N matches the successive occurrence of ENTRY statements.

Successive entries appear as follows:

0 17	18 35
NAMEP$_2$	NAMEP$_3$
NAMEP$_4$	NAMEP$_5$
NAMEP$_{n-1}$	NAMEP$_n$

2. T.COGO Table: An entry is made for each assigned and computed GO TO. The number of words in each entry is 1 + (n/2), where n is the number of statement numbers (EFN's) in the branch list.

0 2	3 17	18 20	21 35
0	Branch Count	0	P(T.JUMP)$_1$
	⋮		⋮
0	P(T.JUMP)$_{n-1}$	0	P(T.JUMP)$_n$

3. T.COMO Table: A one-word entry is made for each specification of blank or labeled COMMON. The BLOCK NAME for a BLANK COMMON appears in the T.COMO Table as //bbbb. A one-word entry is made for each literal appearance of variables in a COMMON statement.

BLOCK NAME (BCD)		
0	NAMEP$_1$	0
0	NAMEP$_2$	0
	⋮	⋮
4	NAMEP$_n$	0

4. T.DATA Table: An entry is made for each DATA statement. The entry size is n+v+2d where n is the number of lists in the DATA statement, v is the number of variables, and d is the number of implied DO's.

	7	00000	0	00000
NON-SS VAR.	0	NAMEP	0	00000
SHORT LIST VAR.	3	NAMEP	0	(dimension prod.) -1
LEFT PAREN.	5	00000	0	P(T.IMPO)
SS VARIABLE	1	NAMEP	0	P(T.USUB)
RIGHT PAREN.	2	00000	0	P(T.DATA)*

*This entry points to the corresponding left parenthesis entry.

5. T.DIME Table: An entry is made for each assignment of a dimension to an array. The number of words in each entry is 1+(n/2), where n is the number of dimensions.

0 2	3 17	18 20	21 35
n	NAMEP (array name)	f	(Dimension)1
	⋮		⋮
f	(Dimension)n-1	f	(Dimension)n

where n = Dimensionality
f = 4 if the dimension is a constant - (binary).
f = 0 if the dimension is a variable - NAMEP.

6. T.DODO Table: A one-word entry is made for each DO or implied DO.

0 2	3 17	18 20	21 35
0	P(T.IODO)	0	EFN (destination)

7. T.EIFN Table: A one-word entry is made for each executable statement that has a statement number (EFN).

0 17	18 35
EFN	IFN

8. T.EQIV Table: An entry is made for each literal appearance of variables in an EQUIVALENCE statement. The number of words in each entry is 1+(n/2), where n is the number of dimensions appended to a variable.

0 2	3 17	18 20	21 35
N	NAMEP	0	(Subscript)1
	⋮		
0	(Subscript)n-1	0	(Subscript)n

N = 2 for every variable in a group except last.
N = 6 for last variable of each group.

9. T.FEFN Table: A one-word entry is made for each FORMAT that is encountered.

0 2	3 17	18 20	21 35
0	0	0	EFN(Format)

10. T.IMPO Table: A two-word entry is made for each implied DO in a DATA statement.

0 2	3 17	18 20	21 35
0	NAMEP (index)	0	Parameter 1
0	Parameter 2	0	Parameter 3

11. T.INTS Table: A one-word entry is made for each literal appearance of a nonsubscripted integer variable on the left side of an ARITHMETIC statement, in an I/O input list (including NAMELIST), and in the argument list of a CALL statement. A special T.INTS entry is made for each CALL statement to indicate that nonsubscripted integer variables in COMMON have been implicitly redefined.

0 2	3 17	18 20	21 35
0	NAMEP	0	IFN

When a special T.INTS entry is made for each CALL statement encountered, the NAMEP is set equal to zero.

12. T.IODO Table: A three-word entry is made for each DO or implied DO.

	0 2	3 17	18 20	21 35
	0	IFN (origin)	0	EFN (destination)
or	4	IFN (origin)	0	IFN (destination)*
	0	NAMEP (index)	f	Parameter 1
	f	Parameter 2	f	Parameter 3

where f = 0 if Parameter is a variable
f = 4 if Parameter is a constant

*The IFN (destination) is placed in the T.IODO Table when the DO terminus statement is encountered.

13. T.IOLT Table: A one-word entry is made for each left parenthesis that is not used to contain subscripts in a DATA statement. This table is used to check the balance of parentheses in a DATA statement, and it lasts for one statement only.

0 2	3 17	18 20	21 35
0	P(T.DATA) left paren. entry	0	00000

14. T.JUMP Table: An entry is made for each statement that may result in a transfer of control (IF's, GO TO's and nonstandard returns from CALL statements). The number of words entered in the table for each statement depends on the number of branches specified.

	0 2	3 17	18 20	21 35
	0	IFN (origin)	0	EFN (destination)
or	4	IFN (origin)	0	IFN (destination)

The latter entry is generated by the Logical IF.

15. T.LITR Table: A one-word entry is made for each literal string in a DATA statement. An entry of 2 + $\left|\frac{n-1}{6}\right|$ words is also made for each literal appearing in the literal string where n is the number of nonblank characters in the literal except in an alphanumeric field where the blanks are retained.

0 2	3 17	18 20	21 35	
P	REPEAT COUNT	T	N	
	N WORDS OF DATA IN BCD ⋮			
0	00000	0	00000	flag indicating end of string

where P = 0 if REAL
= 1 if INTEGER
= 2 if LOGICAL
= 3 if OCTAL
= 4 if COMPLEX
= 7 if DOUBLE PRECISION

and T = 4 for DEC literal
= 2 for OCT literal
= 1 for BCI literal

16. T.RINT Table: A one-word entry is made for each literal appearance of a nonsubscripted integer variable that is in COMMON or EQUIVALENCE and within a DO loop under the following conditions:

1. When they exist on the right side of an equal sign.
2. In an I/O output list (including NAMELIST).
3. As an argument of a CALL or FUNCTION.
4. As a variable unit assignment in an I/O statement.

5. As a computed GO TO parameter.

A special T.RINT entry is made for each CALL or FUNCTION reference to indicate that nonsubscripted integer variables in COMMON have been implicitly redefined.

0 2	3 17	18 20	21 35
0	NAMEP	0	IFN

17. T.SUBS Table: A two-word entry is made for each literal appearance of a subscripted variable in any statement except a DATA statement.

0 2	3 17	18 20	21 35
0	IFN	0	IFN (Supplemental)
0	NAMEP	0	P(T.USUB)

18. T.USUB Table: An entry is made for each appearance of a unique subscript combination. Subscripts that are actually the same as others are considered unique if there is a difference in the size of the dimensions of the variables that are subscripted. The number of words in each entry is 2n where n is the dimensionality.

0 2	3 17	18 20	21 35
n	Checksum	0	C_1
0	NAMEP (V_1)		a_1
f_x	d_1	0	C_2
0	NAMEP (V_2)		a_2
⋮	⋮	⋮	⋮
f	d_{n-1}	0	C_n
0	NAMEP (V_n)		a_n

where n = dimensionality
f = 4 if dimension parameter, d, is a constant
= 0 if d parameter is a NAME reference (adjustable dimension)
C = coefficient in subscript element
a = (addend in subscript element)-1*
x = 1 if variable is double precision or complex.

*If a is negative, the 2's complement will exist in the table (18-35).

19. T.IPRO Table: This table is composed of fourteen words per entry. Each entry describes unique index computations. Before each index is calculated, this table is checked to determine if the computation has already been done.

0 1718 35

Zeros	IXICTR (Counter for I. erasable storage)
(next seven words)	
Zeros	Index constants
(next six words)	
Zeros	Dimensions

Unused cells within an entry are zero.

20. T.RANG Table: This table of one-word entries contains information concerning the level numbers involved in the evaluation of the arguments of FUNCTION and CALL statements.

0 1718 2627 35

Zeros	B	C

B = lowest level number contained in argument
C = highest level number contained in argument.

21. T.BUGS Table: This table contains a variable number of entries. Each DEBUG statement is stored in this table in BCD form.

0 35

DEBUG statement (next n words in BCD)

22. T.NAMS Table: This table is composed of a variable number of entries. Any NAMELIST statement which occurs with a DEBUG statement is stored here.

0 1718 35

Word Count	Zero
STATEMENT (n words as specified in word count)	

0 1718 35

Word Count (next statement, etc)	Zero

23. T.JUNK Table: This variable length table of entries describes jumps according to the IFN of the origin of the jump and the IFN of the destination of the jump. The table is ordered according to destination.

0 1718 35

IFN (origin)	IFN (destination)

24. T.ARIC Table: This table of multiple-word entries contains the number of T.ARIT items in each level and is used by the re-ordering and optimization section of Phase One.

0 1718 35

A	B

where: A = the T.ARIT pointer for the first item of the level.
B = the number of items in the level.

Note: The Nth entry contains information for level N; for those levels with no T.ARIT items, the entry will be zero.

25. T.ASUP Table: This table is a supplement to the T.ARIT Table to be used when the latter overflows. The entries are the same as the T.ARIT Table described on page 93.

26. T.BLOC Table: The table of two-word entries is used to describe the name and size of BLOCK COMMON's. The first word of the entry contains the BCD name of the BLOCK COMMON. The second word contains the size of the BLOCK COMMON in bits 18-35.

0	35
BLOCK COMMON name in BCD	
0 17	18 35
Zero	Size

27. T.EQCO Table: This is a table of two-word entries. Entries are made for variables which are equivalent to another variable which appears in COMMON.

0 17	18 35
NAME pointer	Zero
T.BLOC pointer	Relative location in BLOCK COMMON of this variable

28. T.LIST Table: This table is composed of a variable number of entries. A set of entries in this table describes a group of variables which appear in more than one equivalence group.

0 17	18 35
T.EQIV pointer	Zero
NAME$_1$ pointer	Addend$_1$
⋮	⋮

0 17	18 35
NAME$_{n-1}$ pointer	Addend$_{n-1}$

0 2	3 17	18 35
7	0	0

Note that the prefix of the last word of a set is equal to seven.

29. T.REQU Table: This table is composed of a variable number of entries. A set of entries is made for equivalence groups containing no COMMON variable.

0 17	18 35
NAME$_1$ pointer	Addend$_1$
NAME$_2$ pointer	Addend$_2$
⋮	⋮

0 2	3 17	18 35
7	0	0

Note that the last word of a set has a prefix of seven.

30. T.BASE Table: This table is composed of a variable number of entries. A set of entries is made for any COMMON reference within an equivalence group.

0 17	18 35
T.REQU pointer	Zeros
$NAME_1$ pointer	$Addend_1$

0 2	3 17	18 35
7	0	0

Note that the last word of a set has a prefix of seven.

31. T.LDXR Table: This single entry table is used in conjunction with the indexer routines of Phase Two. The entries indicate the spill index register assigned to subscripted variables and when they must be loaded. Two types of entries may appear; one for the DO Indexer and the other for the Basic Block Indexer.

(1) DO Indexer Type Entry

0 2	3 17	18 35
+	IFN	Supplementary IFN

(2) Basic Block Type Entry

0 2	3 17	18 35
-	Zero	IXBGCT

32. T.BGIN Table: This single-entry table indicates where instructions stored in the T.COLT Table are to be found and where these instructions are to be merged into the G* file.

0 17	18 35
IFN	IXBGCT-1

33. T.OUTS Table: This single-entry table describes where index register reloading must occur because of jumps out of DO loops.

0 17	18 35
P(T.JUMP)	IXBGCT-1

34. T.PROL Table: This table is composed of a variable number of two-word entries. The table is used by the S.PROL routine for the generation of prologue instructions in Phase Two. The first word takes the form:

0 1	2 17	18 35
P	NAME pointer	Addend

where: P, bit 0 = 0, the IFN is to be compiled in the variable field.
P, bit 0 = 1, the IFN+1 is to be compiled in the variable field.
P, bit 1 = 0, word 2 of the entry contains the IFN and the supplementary IFN.
P, bit 1 = 1, word 2 of the entry contains the IFN and supplement in BCD form.

The second word may be one of the three forms shown below:

0 2	3 17	18 35
0	IFN	Supplementary IFN

0	Zero	IXBGCT

IFN and supplement (BCD form)

35. T.ERAS Table: This table consists of two-word entries for Arithmetic Statement Function Definition counts.

0 1718 35

A. count	AA. count
P. count	PP. count

where: A. count = argument storage count, single precision.
AA. count = argument storage count, double precision.

Note: One-half of this word will be used for each entry.

PP. and P. count = Temporary storage count

36. T.AFDU Table: This table is composed of one-word entries. An entry is made for each Arithmetic Statement Function Definition.

0 1718 35

NAME pointer	Argument Count

37. T.NUMB Table: This table is composed of multiple-word entries. Each entry represents a constant in literal form.

0 1718 35

<table>
<tr><td>TYPE CODE</td><td>Word Count (n)</td></tr>
<tr><td colspan="2">Literal constant in BCD form (next n words)</td></tr>
</table>

where the Type Codes are defined as follows:

TYPE CODE	MEANING
1	Integer
2	Real
3	Double Precision
4	Complex
5	Logical
6	Hollerith

38. T.SISU Table: This variable length entry table handles similar subscripts for subscripting involving the Basic Block or DO Indexers. The format of entries for the DO Indexer are shown below:

0 2	3 5	6 17	18 35
-	XR_A	C_f	G
+	XR_A	0	T.USUB $pointer_1$
+	XR_A	0	T.USUB $pointer_2$
			⋮
+	XR_A	0	T.USUB $pointer_n$

where: XR_A = Assigned index register
C_f = Frequency count
G = Calculated coefficient for address addend

Note: The assigned index register remains the same throughout a set of entries.

The format for the Basic Block Indexer entries follows:

0 2	3 5	6 17	18 35
-	XR_A	C_f *	T.USUB $pointer_1$
+	XR_A	0*	T.USUB $pointer_2$
+	XR_A	0*	T.USUB $pointer_3$
			⋮
+	XR_A	0*	T.USUB $pointer_n$

where: XR_A = Assigned index register
C_f = Frequency count
* = An IFN may be placed in this position after the initial entry has been made.

39. T.OTIN Table: The table is composed of a variable number of two-word entries. Each entry describes jumps in and out of DO loops when saving and restoring of registers becomes necessary. No entry in this table indicates that no saving or restoring is required at the jump location.

0 1718 35

IFN (Jump origin)	IFN (Jump destination)
Zero	IXBGCT

40. T.DBUG Table: This table is composed of a variable number of entries. Each two-word entry points to a DEBUG statement in the T.BUGS table.

0 1718 35

Word Count (of DEBUG Statement)	EFN (binary)
T.BUGS pointer	Zero

41. T.COLT Table: This table is composed of entries consisting of three or more words. Coding generated by GG is stored here for later merging.

?

0 1718 35

IFN	Zeros	
0	18 2930	35
Zeros	Word Count(n)	Zero
BCD Coding (next n words)		

42. T.LINE Table: This table is composed of single entries, one for each line described by the Format Generator.

0 35

T.FORT Tally Word

43. T.FORT Table: This table of variable-length entries contains the BCI text produced by the Format Generator.

0 ... 35
BCI Text (next n words)

44. T.USSR Table: This table of two-word entries describes the appearances of subscripted variables within a defined region (Basic Block). The table is used by the indexer routines of Phase Two. The format of the entries are described below:

0 2	3 5	6 17	18 35
U	XR_A	C_f	T.USUB pointer
(IFN)			T.SUBS IFN

where: U = + or -, the Usage Flag
XR_A = Assigned Index Register
C_f = Frequency count
(IFN) = Location at which the index register value must be recalculated.

45. T.NLIN Table: This table is a compilation of pointers of integer variables in NAMELIST.

0 ... 17	18 ... 35
NAMELIST name in BCD	
NAME pointer_1	Zero
NAME pointer_2	Zero
⋮	
Zero	
Next set of entries	

46. T.ENTY Table: This table consists of one-word entries made for each ENTRY statement.

0 5	6 17	18 35
n	Zero	IFN

where: n = The ENTRY statement number. This number increased by one for each ENTRY statement occurring in the subprogram.

IFN = The IFN of the next executable statement.

47. T.DORT Table: This table is composed of multiple-word entries for nests of implied DO statements.

0 2	3 17	18 35
0	DTALEV	P(T.NAME)
0	INC (1st Subscript)	X
0	INC (nth Subscript)	X_n
All bits present (end of set)		

where: DTALEV = DO level of this variable name.
INC = Increment used to calculate the next addend pertaining to this subscript.
X = Addend of a previous ORG, updated on each pass through the table.

Note: The first name pointer points to the name of the outermost DO. The nth name pointer points to the name of the innermost DO.

BU TABLE ROUTINES. The following routines are used to process entries in the BU Tables.

1. S.TI00--Table Initialization Routine

Purpose. This routine is called at the beginning of Phase One and initializes the pointer cells L.BUFF and H.NAME which are the pointers to the BU Tables and NAME Tables respectively. The routine also sets the table indexes to zero.

Usage. The calling sequence is:

```
TSX1    S.TI00     initialize tables

(Normal Return)
```

Error Returns. There are no error returns.

2. S.TA00--Table Allocator Routine

Purpose. This routine is called to allocate buffers which make up the BU Tables. A call to this routine is made each time a new buffer is needed.

Usage. The calling sequence is:

```
TSX1    S.TA00     allocate a buffer

ARG     n          Table #

(Normal Return)  Tally word is in the A-register and T.TABL+n
```

Error Returns. If an illegal table number is specified in the calling sequence or there is an overlap between BU Tables and NAME Tables, a diagnostic message will be written and return made directly to the executive controller.

3. S.TL00--Table Locator Routine

Purpose. This routine is used to locate the beginning of a BU Table.

Usage. The calling sequence is:

```
TSX1    S.TL00        locate a table
ARG     n             Table #
(Error Return)        No table exists, C(A) 0-35 are zero.
(Normal Return)       C(A) 0-17 contain location of label of
                      first buffer of table, C(A) 18-35 are zero..
```

Error Returns. If an illegal table number is specified a diagnostic message is written and return made directly to the executive controller. The no-table-error return does not cause a diagnostic and can be used when testing for the existence of a particular table.

4. S.TK00--Table Kill Routine

Purpose. When buffers on an entire table are no longer needed, this routine will release specified buffers for reuse by other tables.

Usage. The calling sequence is:

```
      TSX1    S.TK00        release a buffer
      ARG     n             Table #
    Z Zero    **,**         label pointer
      (Normal Return)
```

If C(Z) = 0, the entire table will be deleted.

If C(Z) ≠ 0, C(Z)0-17 points to a label in the BU Table label chain.
C(Z)18-35 = 0, preceding buffers will be deleted.
C(Z)18-35 ≠ 0, succeeding buffers will be deleted.
The buffer pointed to will remain intact.

If table specified has not been allocated, a normal return will be made.

Error Returns. If an illegal table number is specified, a diagnostic message is written and return made direct to the executive controller.

5. EN.TR--Buffer Enter Routine

Purpose. This routine is used to make entries in the BU Tables.

Method. A pointer will be kept in the tally word index (T.TABL+n) which will indicate the next available buffer word to be used for storage and the number of words remaining in the buffer.

Usage. The calling sequence is:

TSX1	EN.TR	make an entry into a table
ARG	n	Table #
TALLY	LOC,n	Location of information, number of words
(Normal Return)		Buffer location of first word of entry in A(0-17) except when an indefinite length entry overlaps buffers.

Restrictions. Upon entering this routine a check is made of the table tally word to determine if there is a partially filled buffer available. If not, a new buffer is allocated and the entry made. The last word of an entry is followed by an end mark (all bits) unless the last word of the entry is placed in the last word of the buffer. Some entries are of indefinite length and make up continuous tables (T.COMO, T.ARGS, T.DATA, T.LITR, T.IPRO, T.NUMB). These entries overlap the end of a buffer and continue in the next buffer. Other entries are of definite length and an entry must be completed in a single buffer. A check is made for an entry of the latter type and if enough room is not available, an end mark is written and a new buffer is allocated. Therefore, all buffers that are partially filled will have an end mark flag.

Error Returns. If an illegal table number is specified a diagnostic message is written and return made to the executive controller.

6. PU.LL--Buffer Pull Routine

Purpose. This routine is called when entries are to be pulled from the BU Tables.

Method. As in routine EN.TR, a pointer will be kept in the tally word index indicating the next word to be pulled and the number of words remaining in the buffer.

Usage. The calling sequence is:

```
TSX1   PU.LL       get next entry from table

ARG    n           Table #

TALLY  LOC,n       Location at which to store entry, number
                   of words.

(End of Table Return)  End mark in location (1) and A-
                       register if table is unassigned.

                       End mark in location(n+1) if there
                       are not enough words in table.

(Normal Return)    Last word pulled in A.
```

Restrictions. When this routine is entered, a check is made of the table tally word (T.TABL+n). If a tally exists, the pull is started at this point in the table. If the tally word has been initialized to zero, the pull is started at the beginning of the specified table. When a table has not been assigned or there are not enough words in the table to satisfy the PU.LL entry, an end-of-table return is made.

Error Returns. If an illegal table number is specified, a diagnostic message is written and return made to the executive controller.

The NAME Table

Introduction. The NAME Table is a dictionary of all variables, FUNCTION names and SUBROUTINE names in the source program. Storage of items in the NAME Table is upward in memory. Available storage for the NAME Table is allocated as described on page 21.

NAME Table Formats. All NAME Table entries are made during Phase One. The NAME Table Routine (page 44) is used to make cumulative entries as each new name is encountered. All references in the BU Tables and in the POOL Table to variable names are made using a relative pointer, R.NAME (page 45). This pointer gives the address of word 2, the flag-word, of the two-word entry in the NAME Table.

In the description below, the bit position, the Phase One mnemonic for that flag, and the meaning when ON, are given.

Bit Position	Phase 1 Mnemonic	Meaning of Flag when ON
0 - 17		Address of buffer entry in T.DIME Table, if applicable.
11	I.LIB.	Library function
12	I.INH.	Inhibit register linkage
13	I.BLT.	Built in function--variable length argument
14	I.NLS.	NAMELIST Table name
15	I.FNM.	Function name in the FUNCTION statement
16	I.CAL.	Subroutine reached by a CALL
17	I.XTN.	External subprogram name
18	I.IMP.	Implicit variable definition
19	I.FCN.	Function name used by this program
20	I.RVR.	True variable (as opposed to function name)
21	I.ASF.	Arithmetic Statement Function Name
22	I.ARA.	Argument in A.S.F. (dummy variable)
23	I.ARG.	Argument to this program
24	I.EXP.	Explicitly typed, that is, none of the bits 25-29 may be reset
25	I.LOG.	Logical type
26	I.CPX.	Complex type
27	I.DBL.	Double Precision type
28	I.REL.	Real type
29	I.ITG.	Integer type
30	I.DIM.	Array name--see bits 0-17
31	I.ADM.	Name of an adjustable dimension size
32	I.EQV.	Equivalenced
33	I.EQR.	Equivalenced more than once
34	I.COM.	COMMON variable
35	I.BCM.	BLANK COMMON variable

S.NAME--The NAME Table Routine

Purpose. This routine is entered when a variable name is to be searched for or added to the NAME Table. If the name is not found in the NAME Table, it is added to the table.

Method. This routine is entered with the variable name in BCI, left adjusted and filled in with blanks, in the A-register.

The NAME Table is built as a forward-stored array. The location of the first word of the table (lowest storage address) is initialized from cell BOT. (bottom of available storage). This is done upon the first entry to the NAME routine.

Two cells are used for each entry made to the NAME Table. The NAME Table contains all variable names of the source program and all the information that can be deduced about them in the form of flag bits.

The NAME Table stored forward in memory has the following structure:

```
BCI Name (1)
Name1 flags
BCI Name (2)
Name2 flags
     .
     .
     .
     .
BCI Name (n)
Namen flags
```

(See page 44 for a description of the flag bits.)

Usage. The calling sequence is:

```
TSX1   S.NAME   (name in the A-register)
RETURN - Name added to table. \
                               > Flags in A-register
RETURN - Name found in table. /
```

On normal return:

L.NAME contains the location of the flag word.
C.NAME contains contents of flag word (zero if name-added return is made).
T.NAME contains the IA of the NAME Table.
R.NAME = L.NAME - T.NAME (relative pointer).

Pointers to the NAME Table are always relative because of a size restriction. In most cases only 15 bits can be spared for the pointer; therefore, the NAME Table cannot safely be greater than 2^{15} words long (16384 variable names).

Error Codes. The last location of the NAME Table (highest in memory) must not overlap the buffers reserved for the BU Tables (page 21). Accordingly, the last entry address is kept in cell H.NAME. It must not be equal to or higher than the address of L.BUFF which contains the lowest location of the BU Table buffers.

If this overlap condition is detected, the following diagnostic message is given and return is made directly to the Executive Routine:

```
TABLE SPACE EXHAUSTED.  SHORTEN PROGRAM.
```

2. THE EXECUTIVE PHASE

The Executive Phase of the FORTRAN IV Compiler is composed of six major segments. Each of the segments is described in detail below.

THE PROCESSOR EXECUTIVE CONTROLLER

This segment of the Executive Phase performs the housekeeping and control functions for the FORTRAN IV Compiler. It communicates with GEFRC for input and output functions and provides the necessary calls to GECOS for the loading of Phase One and Phase Two. The Processor Executive Controller also initializes tables, routines and constants, as well as checking error indications. The flow diagram in Figure 6 details the functions of this segment.

Option Check

The system options specified on the $ FORTRAN card are checked using the Switch Word as defined below:

Switch Word Bit Position	Option	
	Bit ON	Bit OFF
6	COMDK	NCOMDK
7	DECK	NDECK
8	LSTOU	NLSTOU
9	$UPDATE	(NOUPDATE)
10	LSTIN	NLSTIN
11	STAB	NSTAB

(The above options are described in the Comprehensive Operating Supervisor Manual, CPB-1002A).

Boundary Check

The Base Address Register (BAR) is checked to determine the upper bound of allocated memory so that table space may be determined.

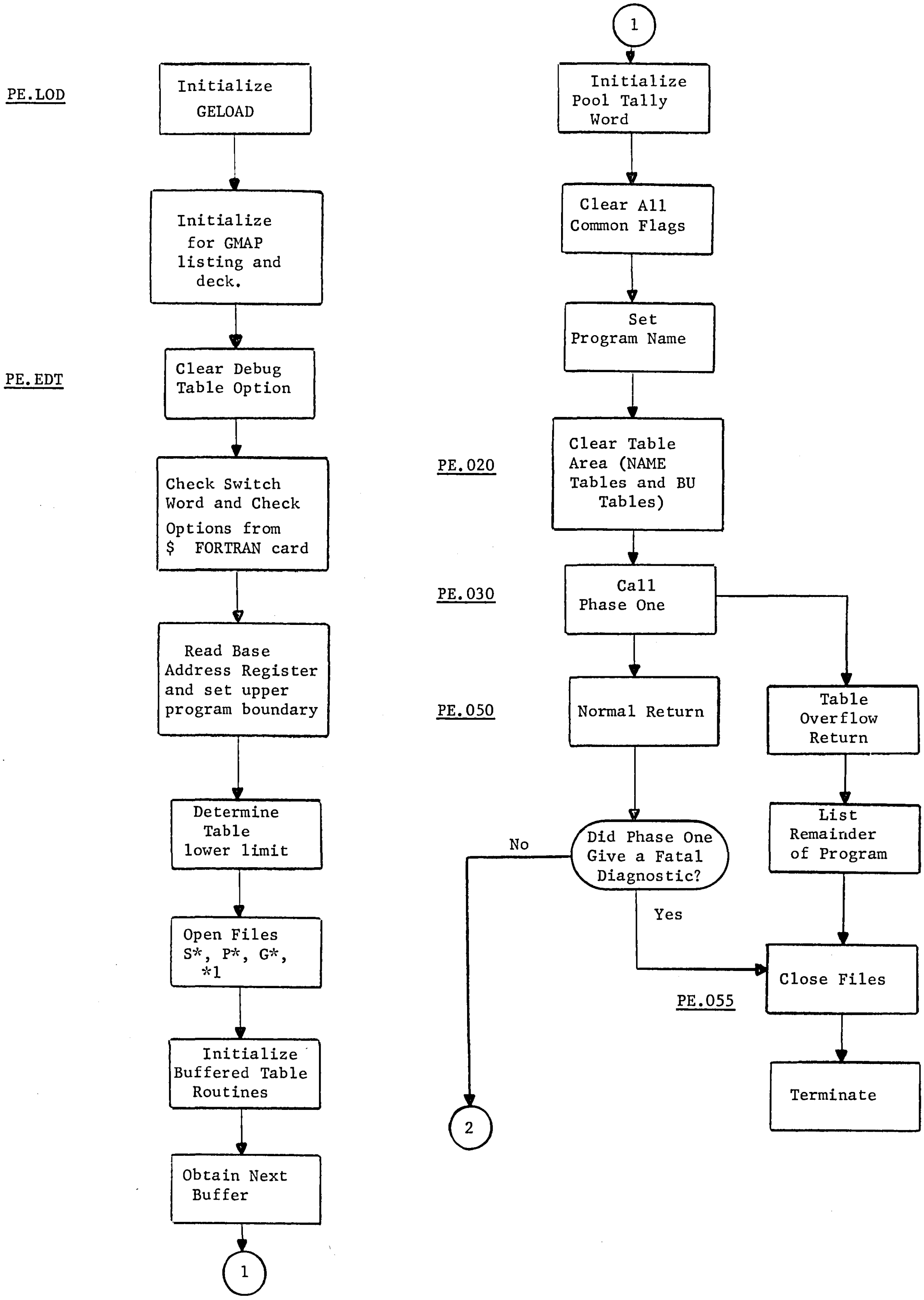


Figure 6. Executive Controller Processor Flow Diagram

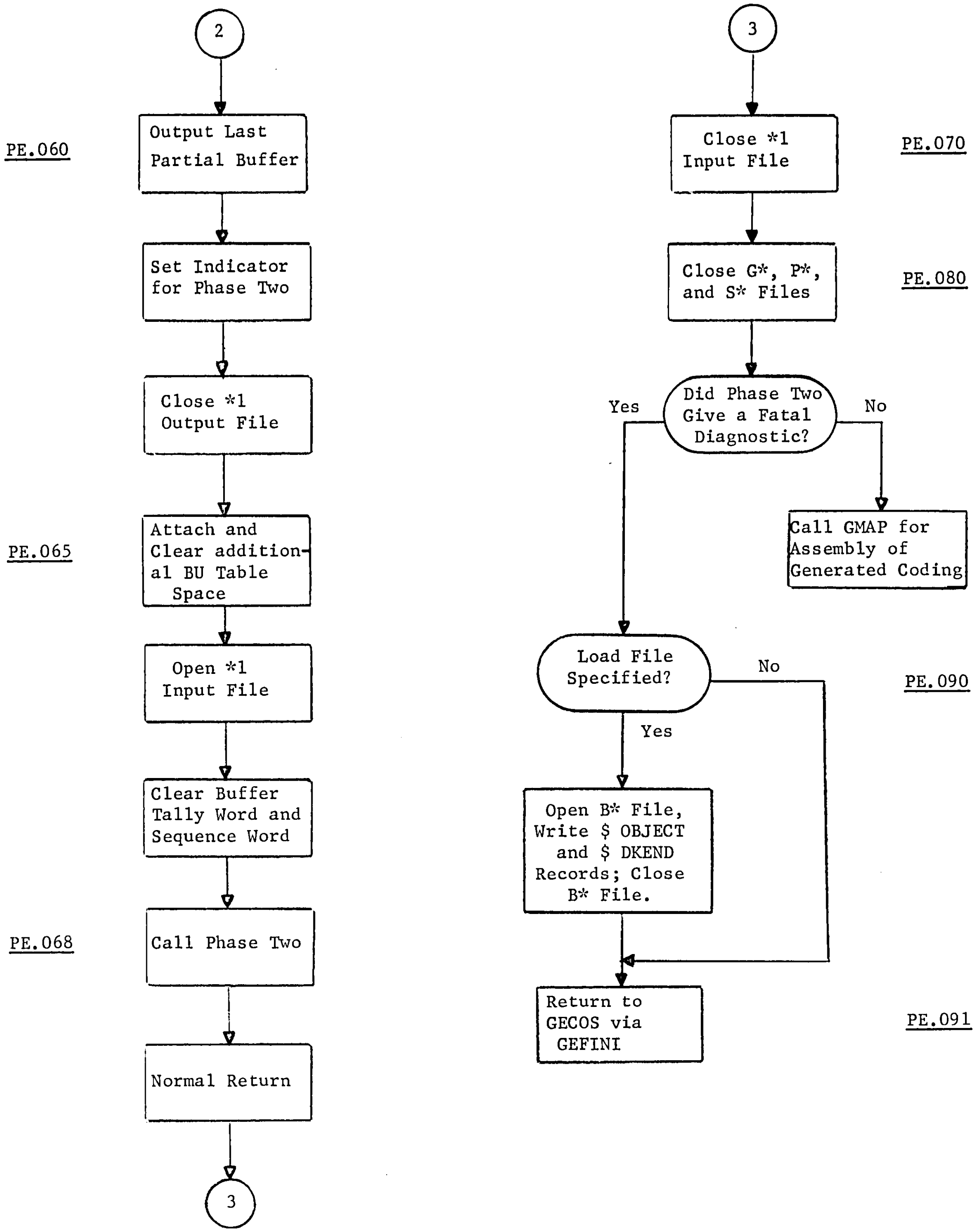


Figure 6. Continued

File Initialization

The S* file is opened as an input file. The P*, G*, and *1 files are opened as outputs.

Initializations

The Buffered Table routines (see "Table Descriptions," page 7) are initialized; the buffer pool tally word is set by storing the POOL Table buffer size in the tally position of S.TP98. All common flags are cleared and the program name is set to all periods (......). The area set aside for the Buffered and NAME Tables is cleared.

Upon completion of its initialization functions, the Processor Executive Controller calls Phase One of the compiler and transfers control. When Phase One is completed, control is returned to the Processor Executive Controller and the location F.DIAG is examined to determine the success of the compilation at this point. If a fatal diagnostic has occurred, the files *1, S*, P* and G* are closed, $ OBJECT and $ DKEND records are written on file B* and control is returned to GECOS through an MME to the location GEFINI. If the location F.DIAG indicates a successful compilation at this point, the phase flag location PHASE. is set to a nonzero value and the scratch file *1 (the POOL Table) is closed as an output file and re-opened with a rewind as an input file. Additional memory is allocated for the Buffered Tables, the available space being computed as the difference between the length of Phase One and the length of Phase Two (Phase Two is shorter than Phase One). Phase Two is then loaded and control transferred to it.

At the completion of Phase Two functions, control is returned to the Processor Executive Controller. The *1, G*, P* and S* files are closed and the location F.DIAG is checked to determine if a fatal diagnostic has occurred. If there has been a source program error which prevented a correct compilation, $ OBJECT and $ DKEND records are written on the B* file and a return to GECOS via GEFINI is executed. If the check of the location F.DIAG indicates no errors, the Processor Executive calls GMAP for the assembly of the generated code.

DIAG, WIAG--Diagnostic Output Routine

Purpose. This routine outputs a diagnostic message in a standard format.

Method. The routine is entered at one of two entry points dependent upon the severity of the diagnostic. An entry to DIAG sets the cell F.DIAG to a nonzero value and provides a fatal diagnostic message describing the error preventing compilation of the source program. The entry point WIAG is used to output a warning diagnostic message. The cell F.DIAG is not referenced and source program compilation may be permitted. Errors found during Phase One processing produce diagnostic messages on the output listing immediately following the improper source statement.

Messages required by checking performed during Phase Two of the compiler will be written on the output listing following the last statement in the user's program.

Usage. The calling sequences are:

```
TSX1   DIAG       For fatal diagnostics
ZERO   LOC,n

 or

TSX1   WIAG       For nonfatal diagnostics
ZERO   LOC,n
```

where

LOC = the address of the message to be printed, and
n = the number of words in the message.

The format of the diagnostic message is:

```
<> <> <> <> <> <> <> <> <> <> <> <> <>

The diagnostic message is written here

<> <> <> <> <> <> <> <> <> <> <> <> <>
```

Restrictions. Diagnostic messages exceeding nineteen words are continued on a second line.

Error Returns. None

S.BB00--Binary Integer to BCD Integer Conversion Routine

Purpose. This routine converts an 18-bit binary integer to a 6-digit BCD integer. Numbers greater than 18 bits are converted modulo 2^{18}.

Method. The binary integer is converted using the special BCD instruction.

Usage. Two calling sequences are used with this routine depending on whether the binary integer to be converted is in the A-register or the Q-register. They are:

```
TSX1    S.BB00     n is in bits 18-35 of the Q-register
(Normal Return)

 or

TSX1    S.BB00+1   n is in bits 18-35 of the A-register.
(Normal Return)
```

The BCD integer is returned in the Q-register left justified and filled out with blanks. If n=0, the BCD integer returned will be one zero digit followed by five blanks.

Error Returns. None

Buffer Table Routines

Six routines associated with the Buffered Tables (see page 21) are included as a part of the Executive Phase of the FORTRAN IV Compiler. These routines (EN.TR, PU.LL, S.TI00, S.TL00, S.TA00 and S.TK00) are described in detail in "BU Table Routines," page 40.

GG--GMAP Code Generator

Purpose. The GMAP Code Generator generates appropriate GMAP code to be written on the G* file for compilation by the GMAP assembly program.

Method. The GMAP Code Generator is a closed subroutine to be used with a specified calling sequence. The argument furnished by the call specifies the location of a list of operations to be performed by the generator.

Usage. The calling sequence is:

```
TSX1   GG
ARG    XLOC
```

where

XLOC = the location of the operation list which is of the general form:

```
TALLYD   PRT1,FLGS,TYPE  )
TALLYD   PRT2,FLGS,TYPE  |
   .           .         |
   .           .          >   Operation
   .           .         |      List
   .           .         |
TALLYD   PRTn,FLGS,TYPE  )
```

The three subfields of the TALLYD operation are examined starting with the third subfield. The TYPE (third) subfield is associated with a numeric code and is interpreted as shown in the table on the following page.

TYPE CODE	TYPE NAME	MEANING
0	.LCF.	Information is to be placed in the Location field.
1	.OCF.	Information is to be placed in the Operation field.
2	.VAF.	Information is to be placed in the Variable field.
3	.CON.	Information is to be concatenated to the most recently specified Location, Operation or Variable field. (For example, the Variable field of a BCI instruction might be generated through the use of several CONCATENATION-type operations.)
4	.HLD.	The information contained in the current line of GMAP coding is to be held in readiness (not written on the G* file) until another TALLYD operation calls for it to be output. (For example, in the generation of a series of arguments for the calling sequence to a subroutine, a HOLD-type operation might be used for each argument until the last argument has been generated.)
5	.CLF.	This Type Name allows the GMAP coding generator to switch output alternately from tables to the G* file. When this Type Code is specified, the first subfield of the TALLYD operation is examined as follows: First subfield = 0 Output to G* file First subfield ≠ 0 Output to Tables (For example, output to tables is required for coding generated by the DO Indexer to be merged into the mainstream coding later.)
6	.TRP.	The operations to be performed will be found in another operation list. The first subfield specifies the location of the remote list. Movement through several levels of operation lists is permitted. Return linkage must pass back through each operation list used. A return to the previous operation list is accomplished with a TALLYD operation having null first and second subfields and a .TRP. as the third subfield.
7	.EPL.	This Type Name specifies the end of an operation list. The generated code is written as a card image.

Only the Type Name. EPL. demands output of the generated card image. The output control device for all other Type Names is based on a comparison of the Type Code. The current Type Code is compared to the previous Type Code. If the current Type Code is less than the previous Type Code, the previous card image is output. The current operation is then examined and construction of the current coding line begins.

The second subfield of the TALLYD operation can be used to specify punctuation characters to prefix or suffix the GMAP coding field. The options listed below are used.

FLGS	MEANING
S.PLS.	Suffix a "PLUS" symbol
P.PLS.	Prefix a "PLUS" symbol
S.MNS.	Suffix a "MINUS" symbol
P.MNS.	Prefix a "MINUS" symbol
S.AST.	Suffix an "ASTERISK" symbol
P.AST.	Prefix an "ASTERISK" symbol
S.SLH.	Suffix a "SLASH" symbol
P.SLH.	Prefix a "SLASH" symbol
S.CMA.	Suffix a "COMMA" symbol
P.CMA.	Prefix a "COMMA" symbol
S.LPN.	Suffix a "LEFT PARENTHESIS" symbol
P.LPN.	Prefix a "LEFT PARENTHESIS" symbol
S.RPN.	Suffix a "RIGHT PARENTHESIS" symbol
P.RPN.	Prefix a "RIGHT PARENTHESIS" symbol
S.PER.	Suffix a "PERIOD" symbol
P.PER.	Prefix a "PERIOD" symbol
S.APS.	Suffix an "APOSTROPHE" symbol
P.APS.	Prefix an "APOSTROPHE" symbol
S.EQS.	Suffix an "EQUALS" symbol
P.EQS.	Prefix an "EQUALS" symbol
S.DOL.	Suffix a "DOLLAR SIGN" symbol
P.DOL.	Prefix a "DOLLAR SIGN" symbol

Also, four additional special codes can be used in the second subfield.

FLAG NAME	MEANING
C.IFN.	The information is to be converted by the IFN technique. This conversion generates an IFN followed by a letter and possibly followed by a supplementary IFN. This converted information occupies either the Location field or the Variable field of the card image.
C.INT.	The information is a binary number to be converted to a BCD integer and placed in the generated card image.
C.BCD.	The information to be placed in the card image is a BCD character string. The first subfield points to a word specifying the location of the string and the number of words the string occupies.
C.EVN.	This Flag Name indicates that an E character must be placed in column 7 of the generated card image for the EVEN location function of the GMAP assembler.

Finally, the first subfield of a TALLYD entry is a pointer to the information to be placed in the card image. The information specified by the pointer may be one of the following.

1. Internal Formula Number encoded with a letter designation, and a possible supplementary IFN
2. An integer
3. A BCD character string pointer
4. A BCD word of six characters

Based on the contents of the third subfield special values may be present in the first subfield as shown below:

FIRST SUBFIELD SPECIAL VALUE	THIRD SUBFIELD TYPE NAME
Always Zero	.HLD.
Zero = Output to G* file Nonzero = Output to tables	.CLF.
Zero = return to previous parameter list Nonzero = address of remote parameter list	.TRP.
Always Zero	.EPL.

The following table describes the possible configurations of TALLYD operation subfields used with the GMAP Code Generator.

<table>
<tr><th>THIRD SUBFIELD</th><th colspan="5">FIRST SUBFIELD</th></tr>
<tr><th>TYPE-NAME</th><th>C.IFN.</th><th>C.INT.</th><th>BCD Pointer</th><th>BCD Word</th><th>Special</th></tr>
<tr><td>.LCF.</td><td>Yes</td><td>Yes</td><td>Yes</td><td>Yes</td><td>No</td></tr>
<tr><td>.OCF.</td><td>No</td><td>No</td><td>Yes</td><td>Yes</td><td>No</td></tr>
<tr><td>.VAF.</td><td>Yes</td><td>Yes</td><td>Yes</td><td>Yes</td><td>No</td></tr>
<tr><td>.CON.</td><td>No</td><td>Yes</td><td>Yes</td><td>Yes</td><td>No</td></tr>
<tr><td>.HLD. *</td><td colspan="4">◄— Always Zero —►</td><td>Yes</td></tr>
<tr><td>.CLF. *</td><td colspan="4">◄— Zero=output to G* —►
Nonzero=output to tables</td><td>Yes</td></tr>
<tr><td>.TRP. *</td><td colspan="4">Zero=Return link
◄— Nonzero=Address of —►
remote operation list</td><td>Yes</td></tr>
<tr><td>.EPL.</td><td colspan="4">◄— Always Zero —►</td><td>Yes</td></tr>
</table>

*Flag codes for TALLYD operations having these Type Names are not checked. For all other Type Names, the Flag will be checked.

Restrictions. None

Error Returns. None

MACERR--Machine Error Dump Routine

Purpose. The Machine Error Dump routine dumps memory if a machine error occurs during the compilation of a FORTRAN IV source program.

Method. This routine is entered using a TSX1 instruction to provide a traceback to the location where the error occurred. The octal value of the error location is stored in a message with the present phase number (1 or 2). The message is written using the diagnostic message routine which is entered at DIAG, the fatal diagnostic entry point. Upon return, the abort code is loaded into the Q-register and a Master Mode Entry is executed to GEBORT.

Usage. The calling sequence is:

```
TSX1   MACERR
(No Return)
```

Error Returns. None

3. PHASE ONE--FORTRAN IV COMPILER

INTRODUCTION

Phase One of the FORTRAN IV Compiler translates the statements of a source program into a series of correlated table entries. This phase is composed of two GMAP assemblies. The first assembly contains Common Routines and the Arithmetic Routines. The second assembly contains Statement Processors and Storage Allocators. The individual routines of Phase One are described in detail below.

EXECUTIVE ROUTINES

The following routines perform executive type functions during the first phase of the FORTRAN IV Compiler.

1. E.1000--Phase One Executive Routine

Purpose. This routine performs the executive control functions for Phase One of the FORTRAN IV Compiler. It handles initializations, processing and correlation of statements for this level and does the final clean-up operations required before Phase Two is called.

Method. The Phase One Executive performs its function through direct coding and the calling of subroutines. Figure 7 illustrates the flow through the executive.

Usage. The Phase One Executive routine is a control program and does not have a calling sequence.

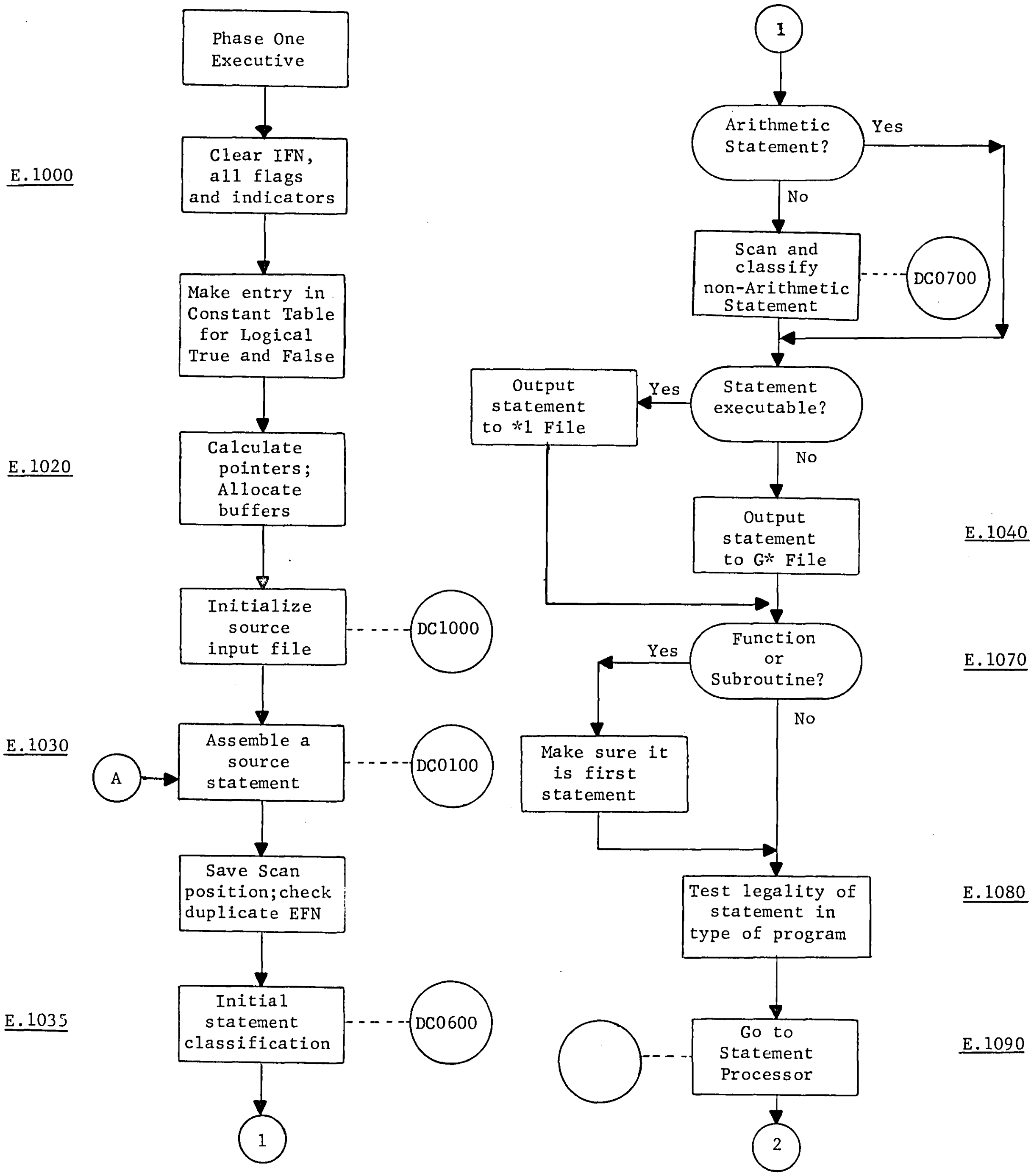


Figure 7. Executive Flow Diagram

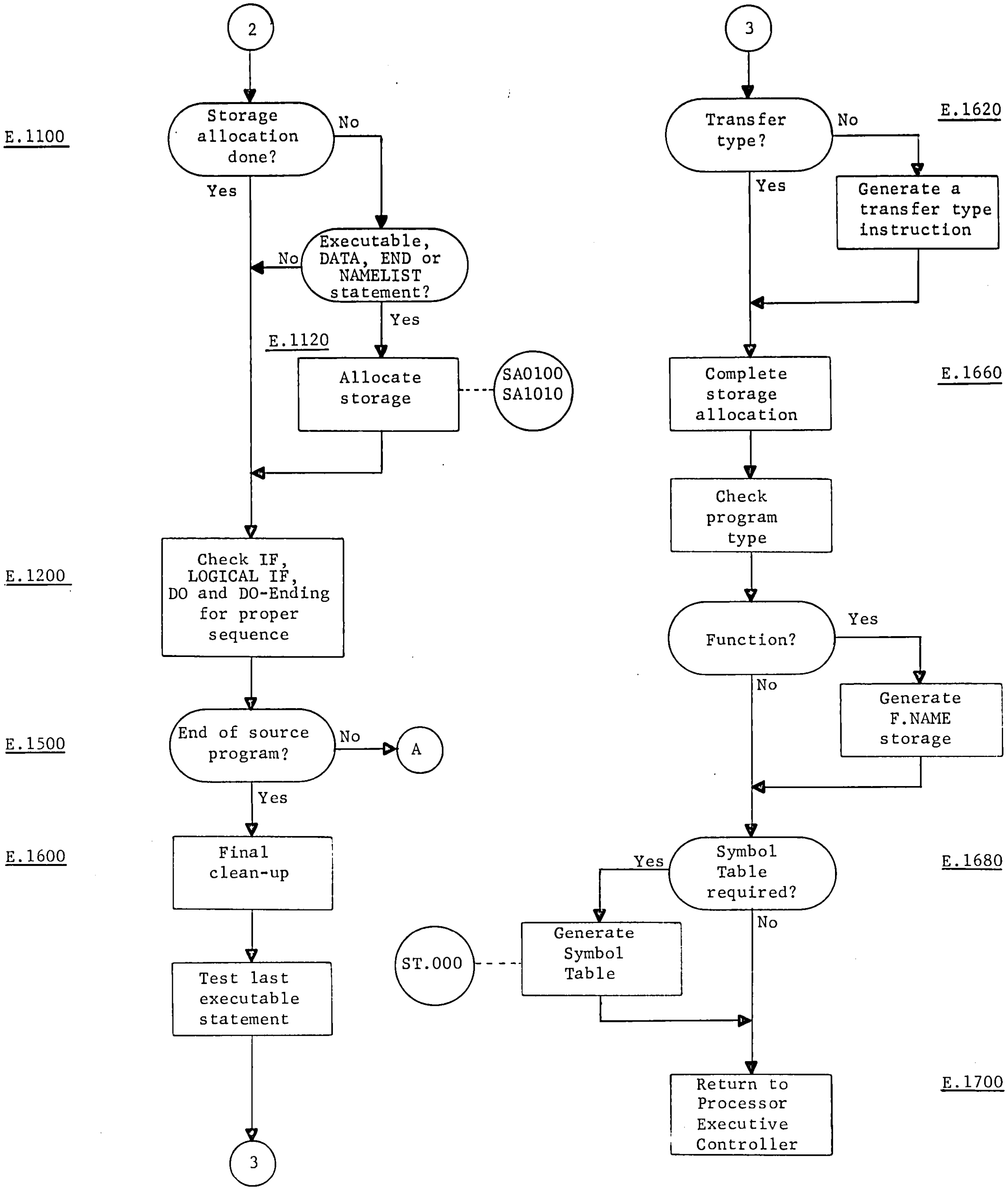


Figure 7 (continued)

Error Returns. There are seven different error messages under the control of the Phase One Executive. In the following cases, an error message is printed, the source input is read and listed until the end is encountered, at which time control is transferred to the "final clean-up" portion of the Phase One Executive.

E.D001	STATEMENT IS EITHER NOT PERMITTED OR MISPLACED IN THE PROGRAM.
E.D002	TRUE CONDITION STATEMENT MUST BE EXECUTABLE BUT NOT -DO- OR -LOGICAL IF-.
E.D003	STATEMENT ILLEGAL TO END RANGE OF -DO-.
E.D005	FUNCTION NAME DOES NOT APPEAR LEFT OF EQUALS ON INPUT LIST.

The above error messages are considered fatal diagnostics. Three error messages can occur which are warnings only. They are:

E.D004	NO END CARD. END CARD SIMULATED.
E.D006	PROGRAM MUST END WITH STOP, RETURN OR TRANSFER. RETURN STATEMENT SIMULATED.
E.D007	END STATEMENT BUT NOT END OF INPUT FILE, WILL LIST AND BYPASS TO EOF.

In the case of the above diagnostic messages, processing will be continued.

2. DC1000--Phase One Initializer

Purpose. This routine performs initialization functions at the beginning of Phase One.

Method. The routine reads one or two cards from the source input and establishes the title and subtitle for the output listing. Label information is accumulated for card output. Alter numbers are assigned to the title and comment card images for reference in the GMAP listing. The initial USE pseudo-operations for .PR000, .MAIN., .TRSH., etc., are output to establish their order of assignment. Upon completion of this routine, the coding may be overlayed by entries for the D.LIT Table which is used in the storage of literals.

Usage. The calling sequence is:

```
TSX1     DC1000
(Normal Return)
```

Error Return. If an unexpected End of File is encountered while reading the source file, the following diagnostic is printed and control is given to the Processor Executive Controller in the Executive Phase:

```
DC1310   UNEXPECTED EOF - MUST TERMINATE
```

STATEMENT ASSEMBLY ROUTINES

The two statement assembly routines locate the next input card image and assemble the entire source statement (including continuation cards) for processing.

1. DC0100--Statement Assembly Routine

Purpose. This routine is called by the Phase One Executive Routine to assemble a complete source statement (1 to 20 cards) in the SS-Region. It also sets an end mark (a word of all bits) in the SS Region following the last nonblank word and initializes the cell SSWW with the number of characters in the statement.

Method. Input to this routine consists of a twelve-word BCI character string received from the input file via the DC0300 routine. Successive card images are obtained and appended to the preceding card image in the SS-Region until one is encountered that does not contain a continuation mark. The continuation mark is a nonzero punch in column six of the card, which indicates that the card is logically connected to the preceding card. An end-of-input condition also terminates the assembly of the current statement.

A backward scan of the SS-Region is performed in order to locate the last nonblank word of the statement. A word of all bits, called the statement end mark, is inserted into the SS-Region following the last nonblank word. Having located the end mark, the Tally Word SSWW is initialized (SS, TALLY) with the number of characters in the statement including the end-mark word for use by the scan routines.

If the statement has an EFN attached to it, it is converted to binary and placed in cell F.EFN. The cell will contain a zero if there is no EFN.

Usage. The calling sequence is:

```
TSX1     DC0100
(Normal Return)
```

Error Returns. The following errors cause diagnostic messages to be given. The scan of the statement will continue.

DC0171	EFN=0 IS ILLEGAL. WARNING UNLESS USED.
DC0175	TOO MANY CONTINUATION CARDS.
DC0179	UNEXPECTED END OF INPUT.

2. DC0300--Locate Input Card Routine

Purpose. This routine is called whenever a next record is required from the input file. The alter count will be incremented by one each time a card is read.

Method. The DC0300 routine uses the .GRDRC entry of the I/O Editor to get a line of input. The line of input received represents a 14-word card image. When an end-of-file condition is encountered, the flag cell F.EOF. is set to a nonzero value. The routine checks the card image for comments cards (a C or * in column 1) and for blank cards. If either type is found, the card image is output to the System Output file, the P* file and the G* file. When a FORMAT GENERATOR statement is recognized, cell DC0199 is set to indicate no searching for continuation cards.

Whenever the DC0300 routine returns to the caller, the next noncomment, nonblank card image to be processed is in the DC0380 buffer (14 words). At this point, the card following (in the input buffer) or an end of file will have been read.

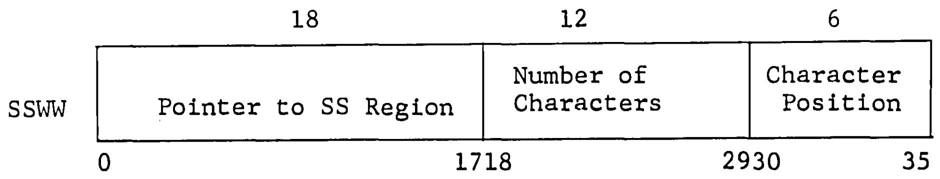
Usage. The calling sequence is:

```
TSX1     DC0300
(Normal Return)
```

Error Returns. None.

SCANNING ROUTINES

Once a source statement has been placed in the region called SS, a scanning process examines the content of the statement. A tally word (SSWW) is used as the scan control word.



The scan control word SSWW is used with SC-type tally modification and is a pointer to the next character to be pulled from the SS-Region.

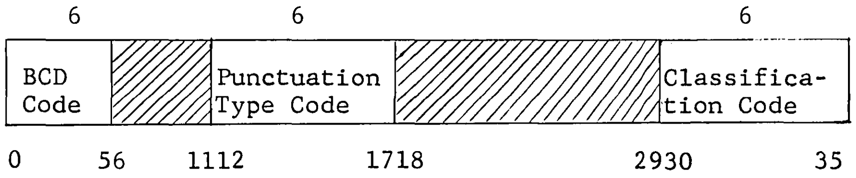
At the beginning of a statement scan, the address part (0-17) of cell SSWW will be set to the value of location SS. The number of characters will be set to the total number of characters in the source statement including blanks that may be needed to complete the last word of the statement. Also included is the end-mark word of all bits. The character position is set to zero. There are other cells used for bookkeeping during the scan. They are:

.FLD.--The output field word where characters are accumulated as they are pulled from the SS-Region.

.TCH.--The termination character where the punctuation type code is stored in bits 12-17.

S.C198--The blank count which indicates the next character position available in .FLD. and the number of blanks attached to the left-justified word in the .FLD. cell.

A table of codes and types is stored at location CH.SET and contains all characters of the GE standard character set.



Classification Code

0	Number
1	Alphabetic
2	Punctuation
3	Illegal
4	Blank

Punctuation Type Codes

SYMBOL	ASSIGNED CODE	TYPE CODE	MEANING
C.PER	1	1	Period
C.COM	2	2	Comma
C.PLS	3	3	Plus
C.MIN	4	4	Minus
C.LPR	5	5	Left Parenthesis
C.RPR	6	6	Right Parenthesis
C.AST	7	7	Asterisk
C.SLS	8	10	Slash
C.EQU	9	11	Equals
C.END	10	12	End-Mark Character delimiting end of statement

The individual scanning routines are presented in detail below.

1. DC0600--Initial Classification Routine

Purpose. This routine is called by the Phase One Executive routine whenever a source statement has been collected in the SS-Region. The function of this routine is to distinguish arithmetic statements from nonarithmetic statements. The general form of an arithmetic statement is a = b. Some statements which contain an arithmetic expression such as IF(b) are classified as nonarithmetic. When the routine has collected sufficient information to classify the statement, it returns control to the Executive routine. The Current Statement cell F.CUST will contain the address of the arithmetic statement processor, S.ARIT, if the statement is classified arithmetic or will be set to zero if the classification has been nonarithmetic.

Method. The statement is examined, one character at a time from left to right. Significant characters used in classifying the statement are: left parenthesis, right parenthesis, comma and equals. Character sequences may also provide the desired classification such as:

, $n\theta$ or /$n\theta$ or ($n\theta$

where θ = the character H or X

These specify Hollerith fields or blank fields and may only appear in nonarithmetic statements. Another sequence is of the form

A(.....)β

If β is not an equal sign, the statement is nonarithmetic. If β is an equal sign, the statement may be arithmetic and the scan continues.

A counter is associated with the occurrence of right and left parentheses. The counter is incremented for each left parenthesis, and decremented for each right parenthesis. In this way, the scan may determine when a comma or equal is enclosed by a pair of parentheses. A comma outside of parentheses, or an equal inside of parentheses is characteristic of a nonarithmetic statement. Conversely, an equal outside of parentheses and all commas inside parentheses is characteristic of an arithmetic statement.

As soon as the scan is able to classify a statement as nonarithmetic, it returns control to the Executive routine with the cell F.CUST set equal to zero. The scan must continue to the end of the statement before the statement can be classified as arithmetic. When the statement is classified arithmetic, the cell F.CUST is set with the address of the arithmetic statement processor.

Usage. The calling sequence is:

```
TSX1     DC0600
(Error Return)
(Normal Return)
```

Error Codes. Illegal characters and blanks are ignored and no diagnostic message is given.

Since parentheses are significant in classifying the statement, the error return will be taken when the parentheses do not balance. The following diagnostic will be given:

```
DC0639   PARENTHESES DO NOT BALANCE.
```

2. DC0700--Dictionary Scan Routine

Purpose. This routine is called by the Executive routine when the statement being processed has been classified as nonarithmetic. The routine will scan a dictionary of statement names comparing them with the source statement. If a successful comparison is made, the Current Statement cell, F.CUST, will be set with the address of the Statement Processor to be used in processing the statement (bits 0-17), and the Statement Control Flags in bits 18-35.

Method. The Dictionary Scan routine calls the S.NC00 routine to obtain N characters ($1 \le N \le 6$) from the SS-Region. The characters are right adjusted in the A-register with leading zeros where necessary.

Two comparison methods are used, a Direct Scan Comparison and a Continued Comparison.

- The Direct Scan Comparison is made starting with the number of characters, N, from the SS-Region equal to 2. The Dictionary of statement names is scanned for a comparison by incrementing N by 1 for each subsequent section of the Dictionary. Comparisons are made against all statement names of two to six characters. If a comparison is not made, the statement is not a legitimate FORTRAN statement.

- After a Direct Scan Comparison has been successful, it is determined if it is necessary to look at more characters. If so, a Continued Comparison is made. The Continued Comparison is made against a referenced word of BCI characters. A successful comparison may call for a further Continued Comparison.

There are three distinct items used in the scan:

- BCI-words. Where there are fewer than six characters, they are right adjusted with leading zeros.

- Key words. There is one Dictionary Control word for each BCI word.

- PL-words. The Processor Location word is unique for each statement processor in the Compiler. It contains the address of the processor, as well as certain control information about the statement to be processed.

There are four distinct tables used in the scan.

- All statement names of six characters or less are kept in a sequential table (General Dictionary) beginning with the two character (minimum statement) statements. For statements of more than six characters, the first six characters are contained in this table. These BCI words are subject to the Direct Scan Comparison.

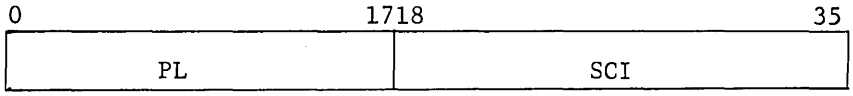
- The keywords corresponding to each BCI word in the previous table are kept in the same sequential order.

- Statement names of more than six characters will have the first six characters in the General Dictionary. The BCI words for the subsequent parts of a statement are placed in the Continued Dictionary with their corresponding keywords immediately adjacent to and following them. No other sequencing requirements are imposed because these are subject to Continued Comparison.

- The Processor Location words are kept in a table. There are no sequencing requirements imposed on this table.

The format of the Keyword and PL-word is as follows:

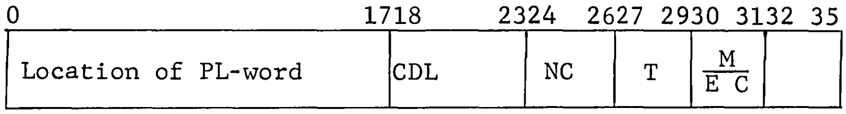
Processor Location Word



PL - Address of Statement Processor
SCI - Statement Control Information

The Statement Control Information is composed of flags that are used by the Phase One Executive routine.

Keyword



Location of PL-word
CDL - Continued Dictionary Location Flag
NC - Number of Characters in Continued Dictionary
T - Type Flag
E - End of Statement Flag
C - Continue Statement Flag
M - Multiple Statement Beginning Flag

The Keyword, one associated with every BCI-word that is compared, contains information about the BCI word.

- End of Statement Flag: This indicates that the characters in the BCI word are the ending characters of a legitimate FORTRAN statement.

- Continue Statement Flag: This occurs when a comparison has been found but more characters have to be scanned for the statement name to be recognized.

- Multiple Statement Beginning Flag: The associated BCI word just scanned and compared is a legitimate FORTRAN statement but it may be the beginning of another statement (for example, END and ENDFILE). This is a signal to continue the search by doing a Continued Comparison. The technique used is to save the PL Flag in case the subsequent Continued Comparison fails. Under these conditions, a Continued Comparison failure does not necessarily indicate an illegitimate source FORTRAN statement.

- Type Flag: Records if statement has type associated with it and, if so, what type it is.
- Continued Dictionary Location Flag: Indicates the relative location of the BCI word in the Continued Dictionary to be used for the Continued Comparison.
- Number of Characters in Continued Dictionary Flag: When a Continued Comparison is called for, this flag tells the number of characters to obtain from the source statement for comparison.
- PL Flag: When a statement is identified, the flag gives the address of the word which contains the PL (address of corresponding processor) and SCI (Statement Control Information).

Usage. The calling sequence is:

```
TSX1     DC0700
(Error Return)
(Normal Return)
```

The Processor Location word is returned in the A-register and also cell F.CUST.

Error Codes. In the event that the source statement does not match any of the entries in the Statement Dictionary, the following diagnostic will be given and the error return taken.

```
DC0781   ILLEGAL FORTRAN STATEMENT.
```

3. NXCHAR--Next Character Macro

Purpose. This macro will generate instructions to fetch the next character from the source statement (SS-Region).

Method. The following instructions will be generated

```
LDQ    SSWW,SC     get next character from SS-Region
LDA    CH.SET,QL   corresponding code from table
EAX0   0,AL        put classification code in XR0
```

C(A)0-5 = BCD code of character
C(A)12-17 = Punctuation type code
C(A)30-35 = Classification code
C(Q)30-35 = BCD code of character
C(XR0) = Classification code

4. S.C000--Scan Routine

Purpose. This routine will scan the source statement for a field and return 1-6 characters in .FLD. and the A-register with the termination code in .TCH. (12-17).

Method. Input to this routine consists of the scan control word SSWW. The first nonblank legal character pulled from the SS-Region determines the type of return that will be made (NUMERIC, ALPHA, PUNCTUATION). Blank characters are ignored and illegal characters cause an error comment to be made. Alpha fields are left adjusted and filled with blanks. If an alpha field is greater than six characters, it is truncated to six characters, a diagnostic message is written and the scan continues until a delimiter is encountered. Numeric fields are left adjusted and filled with blanks. If a numeric field is greater than six characters, the scan routine will return with the six character field in .FLD. and the termination cell .TCH. will be set to zero. An alpha field can only be terminated by a punctuation delimiter but a numeric field can be terminated by having more than six characters (.TCH. = 0), by a punctuation delimiter, or by an alphabetic delimiter, in which case the BCD code is returned as the termination code in .TCH. (12-17).

Usage. The calling sequence is:

```
TSX1     S.C000
(Numeric Return)
(Alpha Return)
(Punctuation Return)
```

Error Codes. There are no error returns but a diagnostic message is given when an illegal character is encountered or an alpha field is greater than six characters.

S.C021 ILLEGAL FORTRAN CHARACTER.

S.C132 THE SYMBOL *______* IS THE RESULT OF TRUNCATION.

5. S.SA00--Start Alpha Scan Routine

6. S.SN00--Start Numeric Scan Routine

Purpose. These routines will scan the statement text and take the appropriate predetermined return.

Method. The output field word .FLD. is set to zero. If the A-register contains a character C(AR) 0-5, it will be inserted into the leftmost character position of .FLD. before the scan is started. If the A-register is zero, scanning will proceed in the normal manner.

Usage. The calling sequence is:

```
          TSX1     S.SA00

           or

          TSX1     S.SN00
```

The returns from either call are:

```
          (Numeric Return)
          (Alpha Return)
          (Punctuation Return)
```

Restrictions. These routines use entry points in the S.C000 routine.

Error Returns. There are no error returns from this routine, but the diagnostic messages S.C021 and S.C132 are written from the routine S.C000.

7. S.CA00--Continue Alpha Scan Routine

8. S.CN00--Continue Numeric Scan Routine

Purpose. These routines will continue the scan in the alpha or numeric mode.

Method. The blank count word, S.C198, determines whether or not there is room for additional characters in .FLD., the output field word. If the A-register contains a character C(AR) 0-5, it will be inserted into the output field word .FLD. before the scan is continued. If room is available, the character will be inserted in the leftmost blank position, otherwise, it will be inserted in the leftmost character position (0-5) of the .FLD. cell. If the A-register is zero, scanning will proceed in the normal manner.

Usage. The calling sequence is:

```
TSX1     S.CA00

 or

TSX1     S.CN00
```

The returns from either call are:

```
(Numeric Return)
(Alpha Return)
(Punctuation Return)
```

Restrictions. These routines use entry points in the S.C000 routine.

Error Returns. There are no error returns in this routine, but the diagnostic messages S.C021 and S.C132 may be written from the S.C000 routine.

9. S.NX00--Next Good Character Scan Routine

Purpose. This routine will pick up the next legal nonblank character in the source statement.

Method. The source statement is scanned until a legal nonblank character is found. Blank characters are bypassed and illegal characters are bypassed after a diagnostic message is written. The character, if it is alpha or numeric, is returned in C(AR) 0-5; if it is a punctuation character, the termination code is returned in C(AR) 12-17.

Usage. The calling sequence is:

```
TSX1     S.NX00
(Numeric Return)
(Alpha Return)
(Punctuation Return)
```

Error Returns. There are no error returns for this routine; however, a diagnostic message is written when an illegal character is found.

S.C021   ILLEGAL FORTRAN CHARACTER.

10. S.NC00--N Character Scan Routine

Purpose. This routine will collect six or less characters from the source statement.

Method. The source statement is scanned until N legal nonblank characters are found. The N characters are returned in the .FLD. cell and the A-register, right adjusted with leading zeros.

Usage. The calling sequence is:

```
TSX1      S.N000
ZERO      N           where N = the number of characters
(Normal Return)
```

Error Returns. There are no error returns but a diagnostic message is given when an illegal character is found.

S.C021   ILLEGAL FORTRAN CHARACTER.

11. S.EMK0--Test for Endmark (Scan Statement for Delimiter)

12. S.EMK1--Test for Endmark in Cell .TCH.

Purpose. These routines are called when the end of statement is expected. Its function is to test a punctuation for the endmark code.

Method. Input to routine S.EMK1 is cell .TCH. which contains the punctuation to be tested. Input to S.EMK0 is the statement in the SS-Region which will be scanned in order to get the punctuation to be tested. This latter punctuation will be returned in cell .TCH. The delimiter code in cell .TCH. is tested against the standard endmark code. A diagnostic message is given if the codes do not match.

Usage. The calling sequences are:

```
TSX1      S.EMK0  (Delimiter must be found first by scanning
                   statement)
(Normal Return)

TSX1      S.EMK1  (Delimiter code to be checked is in cell
                   .TCH.)
(Normal Return)
```

Error Returns. There are no error returns for this routine, but a warning diagnostic is given if the delimiter code is not equal to an endmark.

S.EMK9   EXTRANEOUS CHARACTERS IGNORED. EXPECTED END OF STATEMENT.

CONVERSION ROUTINES

Three number conversion subroutines occupy this section of Phase One of the FORTRAN IV Compiler. Each is described below.

1. S.BI00--BCD Integer to Binary Integer (With Scan)

Purpose. This routine converts a BCD integer to its binary equivalent. Numbers greater than $2^{17}-1$ are set to zero.

Method. There are three entries to this routine depending on whether the number to be converted is in the A-register, the Q-register, or the .FLD. cell. The S.BI00 routine acts as a supervisor by first calling the S.BD00 routine to process the original entry and then alternating between a scan and S.BD00 until a nonzero delimiter is encountered (end of numeric string).

Usage. The calling sequence is:

```
TSX1     S.BI00     N is in the .FLD. cell

 or

TSX1     S.BI00+2   N is in the Q-register

 or

TSX1     S.BI00+3   N is in the A-register
```

All of the calling sequences return to the next line. Upon return, the binary integer is returned in the A-register (18-35). The A-register (0-17) is zero.

Error Returns. There are no error returns but the diagnostic messages S.BI82 and S.BI86 are written from the routine S.BD00.

2. S.BD00--BCD Integer to Binary Integer

Purpose. This routine will convert a six-digit BCD integer to its binary equivalent. Blanks are ignored and will return a zero if all six characters are blank. Numbers greater than $2^{17}-1$ are set to zero.

Method. There are two entries to this routine. Entry is made depending on whether the number to be converted is in the A-register or the Q-register. The conversion is done by successive multiplications and additions. A check is made as each character is collected for illegitimate digits. This routine will convert a BCD number containing any number of digits. To accomplish this, the flag cell S.BD99 must be set nonzero. This flag is used as an indication that this call is a continuation of the preceding call. This flag is always reset to zero before returning to caller.

Usage. The calling sequence is:

```
TSX1     S.BD00    N is in the A-register (0-35)

 or

TSX1     S.BD00+1  N is in the Q-register (0-35)
```

Both calls return to the next line. The binary integer is returned in the A-register.

Error Returns. There are no error returns but the following diagnostic message is written.

S.BD82   ILLEGAL CHARACTER IN INTEGER FIELD.

S.BD86   MAGNITUDE OF DECIMAL INTEGER EXCEEDS 2 TO THE 17TH.

3. S.OB00--Octal to Binary Conversion Routine

Purpose. This routine converts a six-digit BCD octal number to its binary integer equivalent. The conversion is stopped by either a numeric delimiter or a blank.

Method. Input to this routine is the contents of the .FLD. cell. The input word is processed one character at a time. If the character is nonoctal, it is set to zero; a diagnostic is written and the conversion continues. Those characters in excess of six will be truncated; if the cell .TCH. does not contain a termination character, S.CN00 will be called to search for one.

Usage. The calling sequence is:

```
TSX1     S.OB00    N is in the cell .FLD.
(Normal Return)
```

The binary integer is returned in the A-register (18-35).

Error Codes. There are no error returns but the following diagnostic messages exist.

S.D060 OCTAL FIELD TOO LONG.

S.D067 ILLEGAL CHARACTER IN OCTAL FIELD.

PROCESSORS

Most of the statement processors are contained in the second part of Phase One, however, three important ones make their appearance here. Two are concerned with debugging and the third with the processing of subscripts.

1. S.SS00--Subscript Processor Routine

Purpose. This routine scans subscript combinations, segmenting the subscript into its components and makes entries into the T.USUB Table. The general form of subscript combination permitted is:

$$(C_1 * V_1 \pm A_1, \ldots\ldots, C_n * V_n \pm A_n)$$

where C is a constant coefficient
V is a variable element
A is a constant addend
1 to n is the number of elements.

n must agree with the dimensionality specified for the variable to which the subscript is attached, and must also be within the limits of the maximum number of subscripts permitted. The current maximum number of subscripts permitted is seven.

It is necessary to permute the subscripts attached to double-precision and complex variables. The permutation is required to compensate for the storage allocation scheme employed in handling these particular types of arrays. The permutation is performed on the first element of the subscript and may be expressed by the formula:

$$2(C_1 * V_1 \pm A_1) - 1 \text{ and also } 2 * d_1$$

where d_1 is the leading dimension of the array.

The dimension information for each variable is contained in the T.DIME Table in the form:

0 2	3 17	18 20	21 35
N	NAME POINTER	f	d_1
f	d_{n-1}	f	d_n

f = 4 if parameter is a constant
  = 0 if parameter is a NAME reference
n = dimensionality

The subscript elements and their associated dimension sizes are entered into the T.USUB Table in the form:

0 2	3 17	18 20	21 35	
n	Checksum	0	C_1	*
0	NAME POINTER(V1)		A_1	*
fx	d_1	0	C_2	
0	NAME POINTER(V2)		A_2	
	- - -			
f	d_{n-1}	0	C_n	
0	NAME POINTER(V_n)		A_n	

* C and A values will be twice as large for double precision or complex.

n = dimensionality
f = 4, if d parameter is a constant
  = 0, if d parameter is a NAME reference
x = 1, if double precision or complex (bit position 2)

Method. A subscript element may consist of the subelements C, V, and A. Permitted combinations of these subelements are:

C
V
C * V
V ± A
C * V ± A

Subscript combinations other than these cause a diagnostic message to be written. The variable subelement V must not be an array name and must be an integer name. When processing a subscript attached to a double-precision or complex variable, the subelements are permuted as they are collected. An implied coefficient of one is supplied for those subscripts containing a variable subelement but no explicit coefficient. That is, the element V is permuted to 1*V (or 2*V for double precision or complex). The dimension size d is also permuted for double-precision and complex arrays.

The complete element and its associated dimension size are stored, as they are being processed, in a temporary area (S.SS99). A check is made for a one-to-one correspondence between the number of dimensions specified for the array and the number of subscript elements collected as the S.SS99 entry is built. Disagreement of the dimensionality is noted by a diagnostic message.

The routine continues to collect subscript elements until either a right parenthesis is encountered or the number of elements exceeds the maximum permitted by the Compiler. If an error is encountered in processing, the routine always skips to the next right parenthesis, thus allowing the calling statement processor to continue its processing.

When the entire subscript combination has been collected, it is compared to all entries in the T.USUB Table. If a duplicate of this entry exists in the T.USUB Table, then the previously existing entry is used and a new T.USUB entry is not made. However, a new T.USUB entry will be made if this entry is unique. In either case, the routine returns the location of the T.USUB entry to the caller.

Error Codes. The routine scans to a right parenthesis or the end of statement if a right parenthesis cannot be found.

S.SS71	ILLEGAL PUNCTUATION IN SUBSCRIPT OF ' '.
S.SS74	INCORRECT SUBSCRIPT FOR ' '.
S.SS77	ILLEGAL VARIABLE SUBSCRIPT ' '.
S.SS81	INCORRECT ADDEND IN SUBSCRIPT OF ' '.
S.SS84	NO. OF SUBSCRIPTS VS DIMENSIONS FOR ' ' DO NOT AGREE.

Usage. The calling sequence is:

```
TSX1      S.SS00
(Normal Return)
```

The A-register is returned as follows:

A(0) = 0 if variable subscript
A(0) = 1 if constant subscript
A(3-17) = NAME pointer
A(18-35) = T.USUB pointer

If an error is encountered, a zero is returned in the A-register.

2. DEBUG--Debugging Statement Tabling Routine

Purpose. This routine saves DEBUG and NAMELIST statements and makes corresponding entries in the T.NAMS, T.DBUG and T.BUGS tables.

Method. Upon recognition of a DEBUG statement, control is transferred to this routine. Any associated NAMELIST statements are stored in the T.NAMS table. The word count, binary EFN and T.BUGS pointer are stored in the T.DBUG table. The text of the DEBUG statement is stored in the T.BUGS table.

THE [illegible] has to be [illegible]

Usage. The calling sequence is:

```
TSX1      DEBUG
(Normal Return)
```

Error Returns. Incorrect punctuation within the DEBUG statement produces the following fatal diagnostic:

```
DEB201   INCORRECT PUNCTUATION IN DEBUG
```

3. DCBUG--Debug Statement Processor

calls
DCBUG
IFP100
RD0000 (or TPPR00
or FM0000
RT0000)

Purpose. This routine is called to process DEBUG statements.

Method. When it is determined that the EFN of a DEBUG statement matches the EFN of a source program statement, this routine is called. The first time this routine is called, the debug NAMELIST statements (which were saved in the T.NAMS Table by the DEBUG routine) will be processed. The DEBUG statement is retrieved from the T.BUGS Table and processed. For the IF clause, the main IF statement processor is called; either logical or arithmetic as required. Both the IF clauses and the FOR clause produce entries in the POOL Table. Finally, the LIST clause is processed by the READ/WRITE statement processor.

Usage. The calling sequence is:

```
TSX1      DCBUG
(Normal Return)
```

The list clause has the format (fc, namelist) which looks like either the READ (fc, namelist) or WRITE (fc, namelist) pt.
If fc=05, The READ statement processor
[illegible] " WRITE " "

Error Returns. There are three fatal diagnostics produced by this routine:

DCB201 ERROR IN A DEBUG NAMELIST STATEMENT.

DCB211 ERROR IN THE DEBUG STATEMENT.

DCB581 ERROR IN THE -IF- FIELD OF A DEBUG STATEMENT.

STORAGE ALLOCATORS

The two routines concerned with NAME tabling and EFN/IFN replacement are described below.

1. SA7000--Storage Allocation Name Table Scan Routine

Purpose. This routine will scan the NAME Table and assign storage to all variables except those which have been previously assigned or which should not be assigned.

Method. The routine is called at the end of Phase One. Each NAME Table entry flag word is examined to determine if storage should be allocated.

Usage. The calling sequence is:

```
TSX1     SA7000
(Normal Return)
```

Error Returns. There are no error returns but a diagnostic message is written when a variable is implicitly defined; that is, appears only on the right side of =.

SA6013 ________ DOES NOT APPEAR IN READ, DATA, COMMON OR LEFT OF EQUALS (=).

2. SA9000--Storage Allocator/Formula Number Processor Routine

Purpose. This routine will eliminate all destination EFN entries from the T.JUMP Table and replace them with their corresponding destination IFN. The T.JUNK Table, which is a reordered T.JUMP Table, is created.

Method. This routine will process the T.JUMP Table, if one exists, in order to eliminate all destination EFN entries and change them to their corresponding destination IFN. The T.JUNK Table is generated from the modified T.JUMP Table entries. The T.JUNK Table is ordered by destination IFN. If the user specifies the STAB (symbol table for load-time debugging) option on his $ FORTRAN card, the compiler generates the LTAB macro which places the EFN/IFN equivalences in the DEBUG Dictionary.

An error comment is given when a destination EFN in the T.JUMP Table does not have a corresponding EFN entry in the T.EIFN Table.

Usage. The calling sequence is:

```
TSX1      SA9000
(Normal Return)
```

Error Returns. A diagnostic message is written when a destination EFN in the T.JUMP Table does not have a corresponding EFN entry in the T.EIFN Table.

```
SA9090    BRANCH TO NON-EXISTENT EFN __________.
```

UTILITY ROUTINES

These routines which perform a utility-type function and which are called from several different points in the program are presented here.

1. S.TYPO--Implicit Typing Routine (Integer & Real)

Purpose. This routine will implicitly type a variable, either Integer or Real, depending on whether the first character is I,J,K,L,M, or N (Integer type). Any other character types the variable as Real. This typing will only be done when the variable has not appeared in a TYPE statement.

Method. The variable to be typed must exist in the .FLD. cell as input to this routine. The leftmost character is tested for falling in the range I-N. If it does, the variable is typed as Integer, otherwise it is typed as Real. The type is ORed into the flag word (L.NAME*) of the NAME Table and ORed into the cell C.NAME.

Usage. The calling sequence is:

```
TSX1      S.TYPO    (Variable to be typed is in the .FLD.
                     cell)
(Normal Return)
```

The type bit is returned in the A-register and is stored in the NAME Table and cell C.NAME.

Error Codes. None.

2. S.INFO--Increment IFN Counter Routine

Purpose. When called, this routine will generate a new IFN by incrementing the previous IFN by one and make entries into the T.EIFN Table as appropriate. IFNs are assigned only to executable statements. A statement will have only one EFN but may have multiple IFN's.

Method. The cell F.IFN is incremented by one. A comparison of cells F.EFN and S.IFN8 is made and if they differ a new EFN exists. Therefore, an entry will be made into the T.EIFN Table. Only one T.EIFN entry is made for each EFN, this being made when the first of its IFN's is assigned.

Usage. The calling sequence is:

```
TSX1      S.INFO
(Normal Return)
```

The new IFN is returned in bits 0-35 of the A-register.

Error Returns. None.

3. S.AD00--Add a Character Routine

Purpose. This routine will add a character from the A-register (0-5) to the output field word .FLD..

Method. The character in the A-register (0-5) is added to the output field word .FLD. in the leftmost blank position. If .FLD. is full (no blanks), the character is added into the leftmost character position of .FLD. . Using this routine, a zero character can be inserted in .FLD. . The blank count word S.C198 is decremented after an addition is made.

Usage. The calling sequence is:

```
TSX1      S.AD00
(Normal Return)
```

Error Codes. None.

4. S.SB00--Make T.SUBS Table Entry Routine (Subscripted Variables)

Purpose. This routine makes a two-word entry in the T.SUBS Table for subscripted variables appearing in executable statements.

Method. The IFN in cell F.IFN is required as input to this routine. Also required as input is the Name pointer and the T.USUB pointer. The latter two are supplied in the calling sequence as parameters. If the parameter in the calling sequence is zero, return will be made without making a T.USUB Table entry.

Usage. The calling sequence is:

```
TSX1      S.SB00
ZERO      Name Pointer, T.USUB Pointer
(Normal Return)
```

The T.SUBS pointer is returned in the A-register (0-17).

Error Return. None.

5. S.IN00--Make T.INTS Table Entry Routine (Integer Variable)

Purpose. This routine makes entries into the T.INTS Table.

Method. Input to this routine consists of the cell F.IFN and the Name pointer in the A-register (0-17). The Name pointer and the IFN are entered in the T.INTS Table. A one-word entry is made for each literal appearance of a nonsubscripted integer variable on the left side of an arithmetic statement, in an I/O input list (including NAMELIST), and in the argument list of SUBROUTINE subprogram.

A special T.INTS entry is made for each CALL statement encountered in the source program to indicate that integer variables in COMMON may have been redefined.

Usage. The calling sequence is:

```
TSX1      S.IN00
(Normal Return)
```

The T.INTS pointer is returned in the A-register (0-17).

Error Returns. None.

6. S.RIN0--T.RINT Table Entry Routine

Purpose. This routine makes entries into the T.RINT Table.

Method. Control is transferred to this subroutine to make entries into the T.RINT Table described on page 27. The Name pointer is contained in bits 0-17 of the A-register upon entry. This routine places an IFN in bits 18-35 of the A-register to complete the entry for the T.RINT Table. A one-word entry is made for each appearance of a nonsubscripted integer variable on the right side of an arithmetic statement or a call argument list.

Usage. The calling sequence is:

```
TSX1      S.RINO
(Normal Return)
```

Upon entering the routine, the Name pointer will be in bits 0-17 of the A-register. Upon exit from the routine, the T.RINT pointer will be in the A-register.

Error Returns. None.

7. SA0300--Variable Size Computation Routine

Purpose. This routine computes the size required for variables using the Dimension Table.

Method. This size is computed by performing a cumulative multiplication of the dimensions. If the variable is double precision, the size is doubled.

Usage. Upon entry to the routine, the Input-Name Flag is contained in the A-register; the variable name in the Q-register. The calling sequence is:

```
TSX1      SA0300
(Normal Return)
```

Upon leaving the routine, the computed size is located in bits 18-35 of the A-register.

Restrictions. A computed size greater than 2^{18} -1 constitutes an error and a fatal diagnostic is given.

SA0379   DIMENSION -   -GREATER THAN 2 TO 18TH -1.

Variable dimensions constitute an error and a fatal diagnostic is given.

SA0371   ADJUSTABLE DIMENSION ILLEGAL FOR NONARGUMENT -   -.

Error Returns. There are two error returns in this routine as described in "Restrictions" above.

LITERAL COLLECTORS

The routines which process alphameric, complex, decimal and octal literals are described here.

1. S.AH00--Alphameric Collector

Purpose. This subroutine will scan the source statement for an alphameric field defined by wH where w specifies the number of characters in the field.

Method. The characters are collected using the NXCHAR macro and are stored, six to a word, in locations S.LIT.+1 through S.LIT.+n. The number of characters collected will be returned in cell S.LIT. (0-17) and the number of words collected will be returned in the A-register (0-17). The last word collected is left justified and filled out with blanks.

Usage. The calling sequence is:

```
TSX1      S.AH00
(End of Statement Return)
(Normal Return)
```

Error Returns. If an endmark is found before w characters have been collected, a diagnostic message will be given and the end-of-statement return will be made.

S.AH28 UNEXPECTED END OF STATEMENT.

If an illegal character is encountered, a diagnostic message will be given, the character will be set to zero and the scan continued.

S.C021 ILLEGAL FORTRAN CHARACTER.

An alphanumeric field cannot be greater than 1309 characters (DATA statement). If more than the maximum number of characters is requested, a diagnostic message is given and the end-of-statement return is made.

S.AH55 HOLLERITH LITERAL TOO LONG.

2. S.DX00--Complex Literal Collector

Purpose. This routine is used to scan that portion of the source statement which is defined as a complex constant (a pair of decimal constants separated by a comma and enclosed within parentheses).

Method. The scan pointer is pointing at the character immediately following the left parenthesis enclosing the complex pair. The scan will end with the first punctuation character following the right parenthesis enclosing the complex pair. The NXCHAR macro is used to scan the source statement and the delimiting punctuation is returned in .TCH..

Processing consists of determining the proper entry point to the decimal literal collector for each constant in the pair. The literal is collected in a block of storage S.LIT.+1 to S.LIT.+n as one continuous string. The separating comma is retained making the entire string suitable for the variable field of a DEC pseudo-operation. The number of words collected is returned in S.LIT. (0-17) and the Type Code in S.LIT. (18-35). The last word collected is left adjusted and filled out with blanks.

Usage. The calling sequence is:

```
TSX1      S.DX00
(Error Return)
(Normal Return)
```

Error Codes. The following errors stop further scanning of the literal and result in a diagnostic message.

If either constant begins with a punctuation other than (+, -, .), the comma separating the complex pair is missing, the) following the second constant is missing, or the next nonblank character following the) enclosing the complex pair is a numeric or alphabetic character, the following message is given.

S.DX76 ILLEGAL PUNCTUATION IN OR NEAR COMPLEX LITERAL.

Before returning from this routine, a check is made of the type which has been returned from the decimal literal collector. If this type indicates that either of the complex pair was not a real-type variable, the following message is given.

S.DX79 COMPLEX LITERAL PART IS WRONG TYPE.

Note that other messages can be given if an error is encountered in the decimal literal collector as the constant is being processed.

3. S.Dn00--Decimal Literal Collector (S.DI00, S.DN00, S.DP00, S.DS00, S.DNC0)

Purpose. This routine will scan the source statement and collect in BCI, integers, single- and double-precision literals and store them in the area S.LIT.+1 to S.LIT.+n. Blanks are ignored and the last word collected (S.LIT.+n) is left adjusted and filled out with blanks. On return from this routine, S.LIT. will contain the number of words collected (0-17) and the type code (18-35).

Method. There are five entry points to this routine, use of which is determined by the context of the literal or by the initial character of the constant. Input to this routine is supplied by the output field word .FLD. If further scanning is needed, it is done using the NXCHAR macro.

Leading zeros are retained in the literal string but are ignored in the significant-digit count. A significant-digit count of ten or greater forces double-precision floating point unless the exponent character, E, is found in source statement.

Usage. If the literal is an integer and a decimal point, if found, is not part of the literal, then the calling sequence is:

```
TSX1      S.DI00
(Error Return)
(Normal Return)
```

The .TCH. cell contains the code for the following punctuation.

If the first character of the literal is numeric and a decimal point, if found, is included as part of the literal, then the calling sequence is:

```
TSX1      S.DN00
(Error Return)
(Normal Return)
```

The .TCH. cell contains the following punctuation code or alphabetic character.

If the first character of the literal is a decimal point, then the calling sequence is:

```
TSX1      S.DP00
(Error Return)
(Normal Return)
```

The .TCH. cell contains the following punctuation code.

If the plus or minus sign is the first character of the literal, then the calling sequence is:

```
LDA       WORD       where WORD = BCI 1,±
TSX1      S.DS00
(Error Return)
(Normal Return)
```

The .TCH. cell contains the following punctuation code.

If the latter half of a complex literal is to be processed with the first character being numeric, a special entry exists with the following calling sequence:

```
TSX1      S.DNC0
(Error Return)
(Normal Return)
```

The .TCH. cell contains the following punctuation code.

Error Codes. The following errors stop further scanning of the literal and result in a diagnostic message.

S.DN77	MAGNITUDE OF INTEGER EXCEEDS 2 TO 35.
S.DN79	ILLEGAL CHARACTER IN DECIMAL LITERAL.
S.DN81	NO NUMERIC CHARACTERS IN EXPONENT FIELD (D OR E).
S.DN85	MAGNITUDE OF EXPONENT FIELD EXCEEDS 38.
S.DN87	ONE PART OF COMPLEX LITERAL IS DOUBLE PRECISION TYPE.
S.DN89	NO DIGIT IN DECIMAL LITERAL--IMPOSSIBLE.

Control returns to the caller of the literal routine through the error return.

4. S.DO00--Octal Literal Collector

Purpose. This routine is used to scan that portion of a source statement which is an octal field in a DATA statement.

Method. The scan pointer is pointing at the character immediately following the field definition character, O. The scan will be continued until a punctuation character is found following the octal field. If more than 12 octal digits exist, the literal will be truncated to 12. As each character is collected, it is checked to be a legal octal digit. The literal is stored in BCI in a block of storage S.LIT.+1 to S.LIT.+3 depending on the length of the character string. The number of words collected is returned in S.LIT. (0-17). The following punctuation code is returned in .TCH..

Usage. The calling sequence is:

```
TSX1      S.D000
(Error Return)
(Normal Return)
```

Error Codes. If the octal field is greater than 12 octal digits, the literal is truncated to 12 characters, the scan continues until punctuation is encountered and the following diagnostic message is written.

S.D060 OCTAL FIELD TOO LONG.

The following errors stop further scanning and a diagnostic message is given. Return will be to the error return.

S.D042 EMPTY OCTAL FIELD.

S.D067 ILLEGAL CHARACTER IN OCTAL FIELD.

MISCELLANEOUS ROUTINES

Those routines of part 1 of Phase One which do not fit into a well-defined category or those which have been described in another section of this document are presented here.

1. S.NAME--The NAME Table Routine

This routine is described on page 44 of this publication.

2. S.DB00--DO Beta Assignment Routine

Purpose. This routine is called once for each statement in the source program. IFN entries will be made in the appropriate T.IODO table to replace the EFN.

Method. The routine will test the current EFN against the EFN from the T.DODO Table. When a match is found, it indicates that this statement is the ending for a DO statement. If it is, this routine assigns an IFN to this statement and using the T.IODO pointer from the T.DODO Table replaces the EFN with an IFN. The DO Beta entry is then removed from the T.DODO Table. The search is continued over the entire T.DODO Table because the statement being processed may be the ending for a DO nest.

Usage. The calling sequence is:

```
TSX1      S.DB00
(Normal Return)
```

There are no error returns but a flag, F.DOEN, is set for the Phase One control program if the statement is a DO Ending.

Error Returns. The T.DODO Table is searched from the end backwards and if the entries are out of sequence, a probable error in DO nest exists. The message is written as follows:

S.DBOC ERROR IN DO NESTING FOR DO "EFN" ON INDEX_____.

3. S.CB00--Clear DO Beta Routine

Purpose. This routine is called at the end of Phase One by the Control routine to check if any DO Beta's (T.DODO Table entries) exist that were not referenced by a DO statement.

Method. Routine S.DB00 removes entries from the T.DODO Table when a statement that is a DO Ending is being processed. At the end of Phase One, all entries should have been processed and removed from the T.DODO Table. If any entries remain, the statement designated by the DO statement does not exist and an error message is given.

Usage. The calling sequence is:

```
TSX1      S.CB00
(Normal Return)
```

Error Codes. There are no error returns but a diagnostic message is given when an entry is found in the T.DODO Table.

S.CBOB THERE IS NO STATEMENT NUMBER '______' WHICH WAS INDICATED
AS A DO ENDING.

4. S.DES0--Duplicate EFN Search Routine

Purpose. This routine is called by the Control routine once for each source statement to check for duplicate EFN's.

Method. Using the current statement's EFN, a search is made of the T.EIFN Table. If a match is found, a diagnostic message is written.

Usage. The calling sequence is:

```
TSX1      S.DESO
(Normal Return)
```

Error Codes. There are no error returns but a diagnostic message is written when a duplicate EFN is found in the T.EIFN Table.

```
S.DESC  FORMULA NUMBER IS DUPLICATED IN PROGRAM.
```

5. ST.000--Symbol Table Generator

Purpose. This routine generates a symbol table of variable names. (Note: This routine is used only if the source program has a $ FORTRAN control card with the STAB (symbol table) option for load-time debugging.)

Method. This routine scans the NAME Table searching for variable names. Only true variable names are acceptable; SUBROUTINE names, FUNCTION names, DIMENSION variable names, ARGUMENT or Dummy ARGUMENT names are not used. For each true variable name found, a VTAB macro call with appropriate arguments is generated and GG is called to write it on the G* file. The VTAB macro skeleton is:

```
VTAB    MACRO
        IRPT     ARG2
        BCI      1,ARG2
        VFD      18/ARG2, 12/,06/ARG1
        IRPT     ARG2
        ENDM     VTAB
```

where

ARG1 = octal code for variable type.
(20 is octal)
(21 is integer)
(22 is real)
(23 is double precision)
(24 is complex)
(25 is logical)

ARG2 = variable name in BCD form.

Usage. The calling sequence is:

```
TSX1      ST.000
(Normal Return)
```

Error Returns. None

6. S.TP00--POOL Table Entering Routine

This routine is described on page 20 of this publication.

ARITHMETIC TABLES AND ROUTINES

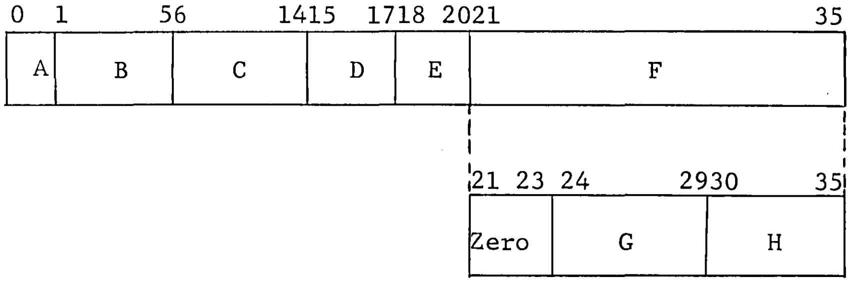
Arithmetic Tables

There are a large number of tables associated with the Arithmetic Scan, Level Analysis and Optimization section of Phase One of the FORTRAN IV Compiler. A description of these tables is presented here to familiarize the reader with the terminology used in the description of the arithmetic routines which follow.

1. T.ARIT Table

Purpose. This table contains all of the information needed by Phase Two of the FORTRAN IV Compiler to generate the proper code for the arithmetic portion of a statement.

Format. The first word of the table is zero. All other words, except the last word, have the following form:



where: A = Level-Save Indicator
= 1, results of level must be saved.
= 0, results of level need not be saved.

B = Operator Code
= 1, operator is .OR.
= 2, operator is .OR..NOT.
= 3, operator is .AND.
= 4, operator is .AND..NOT.
= 5, operator is .GT.
= 6, operator is .GE.
= 7, operator is .EQ.

B = 10, operator is .NE.
= 11, operator is .LT.
= 12, operator is .LE.
= 13, operator is ADD
= 14, operator is SUBTRACT
= 15, operator is MULTIPLY
= 16, operator is DIVIDE
= 17, operator is EXPONENTIATION
= 20, operator is FUNCTION
= 26, operator is DIVIDE INVERTED

C = Level Number Associated with this Operation

D = Operand Mode Indicator
= 0, Logical
= 1, Complex
= 2, Double Precision
= 3, Real
= 4, Integer
= 5, Real, Expand to Double Precision
= 6, Logical and Last of Level

E = Operand Indicator for F, Last 15 Bits of Word
= 0, T.NAME Table Pointer
= 1, T.NUMB (Constant) Table Pointer
= 2, T.SUBS (Subscripts) Table Pointer
= 3, F is a Level Number
= 4, Denotes an ASF Dummy Argument and F is:
Bits 21-23: Zero
Bits 24-29: G, where
G=01, Argument is floating point single-precision real.
=02, Argument is an integer.
=04, Argument is double precision.
=10, Argument is complex.
=20, Argument is logical.
Bits 30-35: H= the argument number (1,2,3, etc.)

F = An Operand, Either a Level Number or Pointer as described above.

The last word in the table contains information about the variable on the left of equals and fields D, E and F will be filled as required. If there is no left of equals, the last word is zero.

2. T.ARIA Table

This table is set up in the Level Analysis portion of the Arithmetic section. It is used in determining the level numbers and ordering of the T.ARIT Table. The table format is:

```
VFD       1/A, 17/B, 3/C, 15/D
```

largest # has highest priority

where:

B = An operator level number assigned as follows:
  = 1, Operators .OR., .OR..NOT.
  = 2, Operators .AND., .AND..NOT.
  = 3, Operators .GT., .GE., .EQ., .NE., .LT., .LE.
  = 4, Operators add (+) and subtract (-).
  = 5, Operators multiply (*) and divide (/).
  = 6, Operator exponentiation (**).
? → = 7, Operator function   function evaluation

what about .NOT. only?

D = A number for the level of the current T.ARIT item.

A = An indicator as follows:
  = 1, The T.ARIA item (Operator level and level) have been used in the T.ARIT table.
  = 0, Otherwise.

C = An indicator as follows:
  = 1, A left parenthesis has been encountered.
  = 0, Otherwise.

If C = 1, the table entry will have the following format:

```
VFD    18/E, 3/C, 15/F
```

where:

E = A pointer to the T.ARIT item preceding the left parenthesis.

F = Is the Operator level of the operator for the T.ARIT item.

Unless this operator is .OR..NOT. or .AND..NOT., then F = 11 or 12 respectively.

3. T.ARIC Table

This table contains the number of T.ARIT items in each level and is used by the Reordering and Optimization routine (ARREOP). This table is a Buffered Table and its complete description appears on page 30 of this document.

4. T.RANG Table

This table contains information concerning the range of levels involved in the evaluation of the arguments of functions and CALL statements. This Buffered Table is described in detail on page 29 of this document.

0 17	18	27 35
← ZERO →	LOWEST LEVEL # IN ARG LIST	HIGHEST LEVEL # IN ARG LIST

5. T.ARLF Table

This table is used in the construction of the T.LTAG Table described below. An entry is made each time a function is encountered in the level analysis of a statement. The table format is:

```
VFD   18/A, 18/B
```

where: B = Zero (always)

A = A pointer to the T.ARIT Table if the operator preceding the function is .OR..NOT.; if not, this portion will be zero.

6. T.LTAG Table

This table is similar to the T.RANG Table except that entries are made only for arguments which are logical. The table format is:

```
VFD  18/A, 18/B
```

where: A = The lowest level in the argument

B = The highest level in the argument

7. AROTST Table

This table is used in the reordering section of the Reordering and Optimization routine (ARREOP). Its entries of a T.ARIT item whose level is the lowest of all levels within the arithmetic string being reordered. The table format is the same as that for entries in the T.ARIT Table described earlier.

8. AROSUP Table

This table is used by the reordering section of the Reordering and Optimization routine (ARREOP). The table is designed for reordering of logical levels. A T.ARIT item whose operator is logical, and whose operand is a level number yet to be reordered, is placed in the AROSUP Table. The format of entries is the same as that for T.ARIT Table entries described earlier.

9. AROTSU Table

This table is used by the reordering section of the Reordering and Optimization routine (ARREOF) to construct the AROSUP Table described earlier. The AROTSU Table contains all of those T.ARIT items of a given level whose operator is logical and whose operand is a level number. When the end of a level is reached in the T.ARIT Table, the items of the AROTSU Table are relocated to the AROSUP Table in reverse order. The format of the AROTSU entries is the same as T.ARIT Table entries.

10. ARO.TC Table

This table is one of three tables used in the optimization of logical strings in the Reordering and Optimization routine (ARREOP). The other two tables are called ARO.TB and ARO.TD and are described below. The entries of the ARO.TC Table consist of a series of pointers. A pointer points to the T.ARIT Table item which is the first item of a logical string. When it has been determined that this string is the argument of a function or a call, a "store bit" is placed in the first item. The table format is:

```
VFD   18/A, 18/B
```

where: A = The T.ARIT Table pointer

B = Zero (always)

11. ARO.TB Table

This table is used in the construction of the ARO.TC Table described earlier. When a T.ARIT item of a new logical level is encountered, the first item of this level is placed in the ARO.TB Table. When the last item of the level is encountered, the entry for the level is removed from the ARO.TB Table. The format of this table is the same as the T.ARIT Table described earlier.

12. ARO.TD Table

This table is used to determine the legality of combined arithmetic/logical expressions. During the processing of a logical level, a T.ARIT item with a level number for an operand causes the next item to be examined. If the level of the next item is not the same as the level/operand, then the level/operand is entered into the ARO.TD Table. During the processing of arithmetic strings, the levels encountered are compared with those in ARO.TD. If a match is found, the operators for that level must be relational or a function with a logical result. The format of the table is:

```
VFD  6/A, 9/B, 21/C
```

where:

A = Zero

B = The level number

C = Zero

13. AROMLV Table

This table is used by the optimization section of the Reordering and Optimization routine (ARREOP) to determine the modes of level number operands and the over-all statement. If the result of a level is arithmetic, then the mode and the level will be entered into this table. The format of this table is:

```
VFD  6/0, 9/C, 3/D, 18/0
```

where:

C = The level number

D = The mode.

14. ARO.TA Table

This table is constructed from the T.RANG Table (described earlier) and is used in the processing of arguments. The format of the table is:

```
VFD 1/D, 17/E, 9/B, 9/C
```

where:

E = A pointer to the first item of an all real level that occurs in an argument. E will be zero if there is none.

D = 1, The argument is double precision
  = 0, Otherwise

B = Lowest level number contained in the argument.

C = Highest level number contained in the argument.

15. T.ASUP Table

This table is used to hold the reordered T.ARIT items if the T.ARIT Table is over one-half full. This Buffered Table is described in detail on page 31.

Arithmetic Routines

1. ARCNTL--Arithmetic Control Routine

Purpose. This routine controls the processing of arithmetic strings.

Method. Entry to this routine is achieved in two ways:

- When an arithmetic expression within another statement must be processed, the relevant statement processor transfers control to this routine.

- All other arithmetic expressions are handled by this routine when control is transferred here from the Statement Classification routine, DC0600, through the entry routine, S.ARIT.

If the arithmetic expression is contained within another statement, no call is made to the subroutine, ARLFEQ, for "left of equals" processing. All other arithmetic expressions call the following subroutines:

(1) ARLFEQ--Process "left of equals"
(2) ARRFEQ--Process "right of equals"
(3) ARREOP--Reorder and optimize

When optimization of the T.ARIT Table has been completed, it is written in the POOL Table on the *1 file, except in the case of arithmetic expressions contained in other types of statements. In the latter, the T.ARIT information will be used by the statement processor originally in control. Before return from this routine, the T.ASUP, T.ARIC and the T.RANG Tables are cleared. If an Arithmetic Statement Function was being processed, the T.AFDU Table is also cleared. Prior to exit, the fatal diagnostic flag is checked to determine if a serious error was found by one of the subroutines called. If yes, an error return is taken; if not, the normal return is taken.

Usage. The calling sequence is:

```
TSX1     ARCNTL
(Error Return)
(Normal Return)
```

Error Returns. There are no error messages directly associated with this routine; only those produced by subroutines called by this program.

2. S.ARIT--Process Arithmetic Statement Entry Routine

Purpose. This routine serves as an entry routine to the subroutine ARCNTL.

Method. Upon entry to this routine, the "left of equals" indicator is turned on and control is transferred to the subroutine ARCNTL. Upon return from this subroutine, the "left of equals" indicator is turned off and control returned to the calling program.

Usage. The calling sequence is:

```
TSX1     S.ARIT
(Normal Return)
```

Error Returns. None.

3. ARLFEQ--Left of Equals Processor

Purpose. This subroutine processes information to the left of the equals character in an arithmetic statement or an Arithmetic Statement Function.

Method. Information to the left of the "equals" character may be one of the following:

- A nonsubscripted name (a simple variable).
- A Subscripted dimensioned variable.
- Name and arguments of an Arithmetic Statement Function.

For the first two types of information an IFN is assigned. If the nonsubscripted name is an integer, an entry is made in the T.INTS Table through the subroutine S.IN00. If not an integer, no action is taken at this time. ~~Dimensioned~~ Subscripted variables require a call to the subroutine S.SS00 for subscript processing and a call to the subroutine S.SB00 to make appropriate entries in the T.SUBS Table. Arithmetic Statement Functions are handled by subroutine calls to S.NAME (and S.TYP0, if required) to ensure that all names are entered in the NAME Table. Entries are made for the arguments in the Buffered Table T.AFDU by the subroutine EN.TR.

Usage. The calling sequence is:

```
TSX1     ARLFEQ
(Normal Return)
```

Error Returns. The following error messages are written as fatal diagnostics in this subroutine.

ARLECA	INVALID CHARACTERS LEFT OF EQUALS.
ARLECB	VARIABLE DIMENSION SYMBOL ________ IS USED IN WRONG CONTEXT.
ARLECC	SUBPROGRAM NAME ________ IS USED IN WRONG CONTEXT.
ARLECD	FORTRAN EXPECTS LEFT PARENTHESES OR EQUALS.
ARLECE	AN ARITHMETIC STATEMENT FUNCTION APPEARS AFTER AN EXECUTABLE STATEMENT.
ARLECF	THE ARITHMETIC STATEMENT FUNCTION ________ HAS THE SAME NAME AS A SUBPROGRAM.
ARLECG	INVALID CHARACTERS IN ARITHMETIC STATEMENT FUNCTION ARGUMENT LIST.

4. ARRFEQ--Right of Equals Processor

Purpose. This subroutine processes information to the right of the equals character in an arithmetic expression.

Method. This routine initializes appropriate constants and sets indicators, then calls the subroutine ARNEXT to obtain ARNWRD (the next operator/operand pair). A call to ARLEAN performs the level analysis. This process is repeated for the entire arithmetic expression. If a function is present in the expression it is checked for a valid name. Except for functions in the FORTRAN Library, external functions, functions in DO loops or having a DO index not in COMMON or EQUIVALENCE, an entry will be made in the T.RINT Table. The parentheses of the expression are checked.

Usage. The calling sequence is:

```
TSX1    ARRFEQ
(Normal Return)
```

Error Returns. Only one error message is directly produced by this subroutine. However, the subroutines called by this program may print error messages. If the latter occurs, an error return to this routine is performed and an error count accumulated for the expression. When the error count reaches 3, processing terminates. (Deeper analysis of errors is not feasible because of the difficulty in determining the proper restart position when an error has occurred.)

This error message for this routine is:

ARRFCA   STATEMENT CONTAINS TOO MANY NESTED FUNCTIONS.

5. ARNEXT--The Next Word Routine

Purpose. This subroutine is used to fill in the next word (ARNWRD) from a source statement.

Method. The next word (ARNWRD) is composed of an operator/operand pair. This routine obtains the next field in an arithmetic expression in preparation of the next word. Not only is it necessary to obtain the next field, but it must be examined to determine its type--alphabetic, numeric or a punctuation character. Each type of field is further tested to ensure that it is valid, properly used and that it fits within the context of its surroundings.

In addition, this routine also examines the termination character; that is, the character following the next field. Frequently, the type of termination character determines the extent to which the next field must be examined. In general, the field beginning with the termination character will become the operator of the operator/operand pair when the routine is next entered. However, this is not always the case and is mentioned here only for clarity in a simple example.

Depending on the type of the next field and its additional characteristics, entries will be made to specific tables. For example, if the next field is a nonsubscripted integer variable, entries are made in the T.INTS table. Though extensive and complex in its examination, the sole purpose of the next word (ARNWRD) is to subsequently provide information from which the T.ARIT Table is constructed.

Usage. The calling sequence is:

```
TSX1     ARNEXT
(Error Return)
(Normal Return)
```

Error Returns. Due to the extensive analysis performed by this routine, there are many fatal diagnostic messages which may occur. They are:

ARNCAA	MISSING OPERATOR BEFORE THE SYMBOL ________.
ARNCAB	SUBPROGRAM NAME __________ INCORRECTLY USED.
ARNCAC	SUBSCRIPTS ARE NOT PERMITTED IN AN ARITHMETIC STATEMENT FUNCTION DEFINITION.
ARNCAD	THE SUBROUTINE NAME __________ IS USED AS A FUNCTION.

ARNCAE	THE SYMBOL ________ IS USED AS A FUNCTION.
ARNCAF	FUNCTION ________ CALLS ITSELF.
ARNCAI	ILLEGAL PUNCTUATION FOLLOWING THE SYMBOL ________.
ARNCAJ	TOO MANY EQUALS.
ARNCAK	STATEMENT ENDS WITH AN OPERATOR.
ARNCAL	PARENTHESES DO NOT BALANCE.
ARNCRL	TOO MANY RIGHT PARENTHESES.
ARNCLL	TOO MANY LEFT PARENTHESES.
ARNCAM	MISSING OPERATOR BEFORE OR AFTER SYMBOL ________.
ARNCAN	ILLEGAL USE OF A COMMA.
ARNCAO	DOUBLE OPERATOR BEFORE OR AFTER THE SYMBOL ________.
ARNCAS	NO PERIOD AFTER THE REL/LOGICAL OPERATOR ________.
ARNCAU	ILLEGAL REL/LOGICAL OPERATOR.
ARNCAV	HOLLERITH ILLEGAL EXCEPT AS ARGUMENTS.
ARNCAX	EFN IS ZERO IN A NONSTANDARD RETURN.
ARNCAY	IMPROPER SYMBOL.

There is one warning diagnostic written by this routine.

ARNCAH	SUBPROGRAM ________ HAS SAME NAME AS BUILT IN FUNCTION-WARNING ONLY.

6. ARLEAN--Level Analysis Routine

Purpose. This routine performs the level analysis for that portion of an arithmetic or logical expression to the right of the equals character.

Method. This routine utilizes the operator/operand pairs in the current word (ARCWRD) and in the next word (ARNWRD) to perform the level analysis and build the T.ARIT Table entries. After the reordering process (described later in this section as the ARREOP routine) is performed, the information of the T.ARIT Table is used to generate the necessary coding for arithmetic and logical expressions.

To translate an arithmetic or logical expression into information ready for reordering, the structure of the expression must be analyzed and associated with its corresponding levels. The relationship of these levels is hierarchical; the higher levels must be processed first and the lower levels last. The chart below shows the relative order of levels for the FORTRAN IV Compiler operations:

RELATIVE LEVEL		OPERATOR
(Lowest)	1	.OR.
	2	.AND.
	3	Relational Operators (.GT., .GE., .LT., .LE., .EQ., and .NE.)
	4	+ and - (Add and subtract)
	5	* and / (Multiply and divide)
	6	** (Exponentiation)
(Highest)	7	FUNCTION

The concept of levels is best illustrated by an example:

Given the expression: A-C+B*E/F*D

In the above expression there are two levels; the first is for the + and - operators and the second is for the * and / operators. If we arrange the expression in tabular form, we have:

Operator	Operand
+	A
-	C
+	Z

where Z is a level within this expression. (Note that it is the highest level in this expression also.) The level Z can be broken down and illustrated in tabular form as shown below:

Operator	Operand
*	B
*	E
/	F
*	D

To attach some meaningful cross-reference to levels within an expression, let us assign a number, N, and substitute it for the level intended. If N is used for the first level encountered (not necessarily the highest or lowest), then the other levels within the expression may be expressed as N+i and N-i as necessary. The combined tables would appear as follows:

Level	Operator	Operand
4 N	+	A
4 N	-	C
4 N	+	N+1 5
5 N+1	*	B
5 N+1	*	E
5 N+1	/	F
5 N+1	*	D.

LEVEL	OP	OPERAND
7	+	CMPLX

By assigning levels to arithmetic or logical expressions, the T.ARIT Table entries may later be reordered to ensure that the processing will be done in the proper sequence for the generation of coding to evaluate the expression.

Usage. The calling sequence is:

```
TSX1     ARLEAN
(Error Return)
(Normal Return)
```

ARLEAN calls on:
EN.ART puts entries in T.ARIT/T.ARIC table
EN.RNS " " " T.RANS/T.LTAG "
ARL.SF " " T.ARLF table

Error Returns. There are four fatal diagnostics written by this routine as follows:

ARL820	ARITHMETIC STATEMENT IS TOO LONG.
ARL821	TOO MANY SUBEXPRESSIONS IN AN ARITHMETIC STATEMENT.
ARL822	TOO MANY ARGUMENTS OR SUBEXPRESSIONS IN AN ARITHMETIC STATEMENT.
ARL823	TOO MANY FUNCTIONS IN AN ARITHMETIC STATEMENT.

7. EN.ART--T.ARIT Table Entering Routine

Purpose. This subroutine is called by the Level Analysis routine (ARLEAN) to place entries in the T.ARIT and T.ARIC Tables.

Method. After the level analysis of the operator/operand pairs of current word and next word have been completed, this subroutine is called to make appropriate entries in the T.ARIT Table. Entries are also made by this subroutine in the T.ARIC Table. The T.ARIC Table contains a count of the number of T.ARIT items at each level.

Usage. The calling sequence is:

```
TSX1     EN.ART
(Normal Return)
```

Error Returns. Two error conditions are detected in this routine; however, control is returned to the ARLEAN routine for the actual writing of error messages. The messages written are described as ARL820 and ARL822 in the ARLEAN routine.

8. EN.RNG--T.RANG Table Entering Routine

Purpose. This subroutine makes entries into the T.RANG and T.LTAG Tables. (The T.LTAG Table is the same as the T.RANG Table except entries are made only for logical arguments.)

Method. During the level analysis routine (ARLEAN), the occurrence of a FUNCTION or CALL with arguments will require entries to be made to the T.RANG/T.LTAG Tables concerning levels. The T.RANG Table contains the range of levels for the arguments of FUNCTIONS and SUBROUTINES. If the FUNCTION is logical, the range of levels for the arguments are placed in the T.LTAG Table. This subroutine has two entry points:

1st Entry Point:	EN.RN1	Entries will be made to both tables if necessary.
2nd Entry Point:	EN.RN2	Entries will be made in the T.RANG Table only.

Usage. The calling sequence is:

```
TSX1     EN.RN1  — calls no ARL.SF.

 or

TSX1     EN.RN2
```

Return is to the next line.

Error Returns. None.

9. ARL.SF--T.ARLF Table Entering Routine

Purpose. This routine is used to make entries into the T.ARLF Table during the level analysis.

Method. The table entries this routine makes are used in the construction of the T.LTAG Table. An entry is made to the T.ARLF Table each time a FUNCTION is encountered during the level analysis of a statement.

Usage. The calling sequence is:

```
TSX1     ARL.SF
(Normal Return)
```

Error Returns. There is no error return for this routine, however, if the T.ARLF Table becomes too large, control will be returned to the Level Analysis routine (ARLEAN) where the error message ARL823 is written.

10. ARREOP--Reordering and Optimization Routine

Purpose. This subroutine reorders and optimizes the information contained in the T.ARIT Table.

Method. This subroutine is called from the Arithmetic Control routine (ARCNTL) after the level analysis of an arithmetic or logical expression has been completed. Three phases comprise this subroutine.

The first phase reorders the T.ARIT Table and eliminates redundant entries. Arithmetic strings are reordered by level in descending sequence; logical strings are reordered by level in ascending sequence.

The reordered T.ARIT Table information may be stored in one of two places. If less than one-half of the T.ARIT Table has been used, then the next available location is defined as the beginning of the T.ARIN Table and the reordered information is stored there. If more than one-half of the T.ARIT Table has been used, then a Buffered Table, T.ASUP, is used for the storage of the reordered information. Upon the completion of the reordering process, the reordered information is placed back into the T.ARIT Table.

In the second phase, common arithmetic subexpressions are eliminated. Common subexpressions are defined as having the same number of elements with the same operator/operand combination in the same order; such as, X=(A*B/C) - (B*A/C) - (A*B/C). The first and last subexpressions (A*B/C) are common. This process requires the examination of the levels contained in the T.ARIT Table information against a given level; that is, find two levels that have the same number of entries. Those levels found equal to the given level are further compared item-for-item. For those levels which satisfy this matching test, the expression will be deleted, but the level will be saved so that other operands which reference this deleted level may be changed to reference the given level.

The third phase of the reordering subroutine assigns a mode to each item. The codes are represented by coded numbers as follows:

Mode	Code Number
Logical	0
Complex	1
Double Precision	2
Real (Floating)	3
Integer	4
Real/Double (Single precision must be expanded to double.)	5
Last logical	6

It should be noted that if the operand is primitive, the mode can be determined from the Name Table (variables) or the Number Table (constants). The subroutine ARO.SD described later in this section is used for the mode determinations. Nonprimitives (levels) are determined by searching the Mode Level Table. If the level number at hand is the same as the level of an item in the Mode Level Table, then the table level is used for mode determination. If the level number at hand is not in the Mode Level Table, it is assumed to be logical. Upon determination of modes, the legality of combined modes is checked. The codes described earlier were chosen such that the logical and combination of codes for illegal mode combinations will be zero.

Also in the third phase, a determination is made as to which levels must be saved in erasable storage. For those levels that must be saved, a "store bit" (item A as described in the T.ARIT Table diagram) is inserted into the first T.ARIT item of the level.

For example:

A*B - C*D

The level A*B must be saved in erasable storage while the expression C*D is evaluated. In addition, single-precision operations are reordered within a level so that they will be processed first.

The final portion of the third phase searches the T.ARIT Table for items having store bits. A count is incremented for each reference to the level by an operand. If this count is 1 at the completion of the search, then the store bit can be eliminated since the result will be used immediately and need not be stored (linkage by registers).

Usage. The calling sequence is:

Calls ; Phase I – ARO.[illegible]
ARO.[illegible]
PHASE II – ARO.SE
PHASE III – ARO.SD
ARO.SET

```
TSX1     ARREOP
(Error Return)
(Normal Return)
```

Error Returns. The following diagnostics are written by this routine:

RO.701	ARITHMETIC EXPRESSION - LOGICAL OPERATOR.
RO.702	ARITHMETIC EXPRESSION SHOULD HAVE A RELATIONAL OPERATOR.
RO.704	ARITHMETIC OPERATOR - LOGICAL OPERAND.
RO.705	TYPES ILLEGALLY COMBINED.
RO.706	ILLEGAL USE BY EXPONENTIATION OR RELATIONAL OPERATOR.
RO.707	TYPE ILLEGALLY COMBINED BY EXPONENTIATION.
RO.708	TYPES ILLEGALLY COMBINED BY A RELATIONAL OPERATOR.
→ RO.711	LEFT OF EQUALS AND RIGHT OF EQUALS ARE INCOMPATIBLE. { Complex V = non-complex expression
RO.713	COMPLEX TYPE ILLEGAL.

11. ARO.SE--T.RANG Table Search Routine

Purpose. This subroutine searches the T.RANG Table to determine if a given T.ARIT item level falls within the range of levels of any item in the T.RANG Table.

Method. This subroutine is called by the Reordering and Optimization routine (ARREOP) during the third phase of its operation. Sequential entries in the T.RANG Table are tested against the item level in the T.ARIT Table. Separate returns are made if there is or is not an item in the T.RANG Table with the proper range of levels.

Usage. The calling sequence is:

```
TSX1     ARO.SE
(First Return, level is in T.RANG Table)
(Second Return, level is not in T.RANG Table)
```

The subroutine is entered with the T.ARIT pointer in index register 5. Upon return, if the level is in the T.RANG Table, the T.RANG Table pointer will be in index register 0.

Error Returns. None.

12. ARO.SA--T.ARIT Level Search Routine

Purpose. This subroutine searches the T.ARIT Table for the next entry with the same level as a given level.

Method. This subroutine is called by the Reordering and Optimization routine (ARREOP) during Phase One of its operation. The corresponding level search is part of the reordering process.

Usage. The calling sequence is:

```
TSX1     ARO.SA
(Normal Return)
```

Upon entry to the subroutine, the given level is specified as a count in index register 4.

Error Returns. None.

13. ARO.SB--T.ARIN/T.ASUP Table Entering Routine

Purpose. This subroutine makes entries into the T.ARIN or T.ASUP Tables.

Method. This routine is called by the Reordering and Optimization routine (ARREOP) during the first phase of its operation. As described in the ARREOP routine, T.ARIT items are reordered by level and placed in the T.ARIN or T.ASUP Table. If the T.ARIN Table is used, entries are made directly. If the T.ASUP Table is used, the subroutine EN.TR is called to make the entry to the Buffered Table T.ASUP.

Usage. The calling sequence is:

```
TSX1     ARO.SB
(Normal Return)
```

Upon entering and leaving this subroutine, the entry to be placed in the T.ARIN/T.ASUP Table is contained in the A-register.

Error Returns. None.

14. ARO.SC--T.ARIC Table Search Routine

Purpose. This subroutine searches the Buffered Table T.ARIC for items corresponding to a given level.

second phase.

Method. This subroutine is called by the Reordering and Optimization routine (ARREOP). A given level is specified in the upper Q-register. The T.ARIC Table is searched for an entry having a corresponding level. When found, the T.ARIC Table entry specifies the number of T.ARIT items for that level.

Usage. The calling sequence is:

```
TSX1     ARO.SC
(Normal Return)
```

Upon entering this subroutine, the given level is specified in the upper Q-register. Upon leaving the subroutine, the T.ARIC item found is contained in the A-register.

Error Return. None.

15. ARO.SD--T.ARIT Item Mode Determination Routine

Purpose. This subroutine determines the mode (Logical, Complex, Double Precision, Real/Floating, Integer, Real/Double or Last Logical) of a T.ARIT Table item.

Method. This subroutine is called by the Reordering and Optimization routine (ARREOP) during the third phase of its operation. If the operand of the T.ARIT Table entry is primitive, that is, a variable or constant, the mode can be determined from the Name Table or the Number Table. However, if the operand is a level, then the Mode Level Table (AROMLV) must be searched. If the operand level is the same as the level of an item in the AROMLV Table, then the operand level is assumed to be the same. If no comparison is found, the mode of the operand level is assumed to be logical.

Usage. The calling sequence is:

```
TSX1     ARO.SD
(Normal Return)
```

Upon entry the item to be typed is contained in the A-register. Upon return, the mode is placed in the item in the A-register. The mode is also in the upper Q-register.

Error Returns. None.

16. ARO.SH--Operator/Level Processor Routine

Purpose. To process a level containing the operators + (plus) and - (minus) or * (multiply) and / (divide).

Method. This subroutine is called to process levels which are not all real (single precision) and contain no logical operations. The items of a level are examined so that proper modes can be established and items reordered as required. Reordering is performed within a level containing real and double-precision operations so that the real (single-precision) operations are compiled first.

In addition, divide operations are replaced by divide-inverted where appropriate.

Usage. The calling sequence is:

```
TSX1    ARO.SH

 or

TSX1    RO.SHA

 or

TSX1    RO.SHB
```

The first entry is used when the level is not all real. The second entry is used when the level is real/double. The third entry is used when a level is all of one type. Return for all entries is always to the next line.

Error Returns. None.

17. ARO.SF--All Real String Routine

Purpose. This subroutine is used to process an all real string.

Method. The T.ARIT information is examined to determine the presence of an all real level that is not in the T.RANG Table. When a level is found, the subroutine ARO.SI (described on page 113) is called to process it.

Usage. The calling sequence is:

```
TSX1     ARO.SF
(Normal Return)
```

Error Returns. None.

18. ARO.SI--All Real Level Routine

Purpose. This subroutine is called to process all real levels.

Method. The modes for this level are set. If the first operator is functional, exponential or relational, the mode is set to either real or real/double. (Double-precision requires the use of the real/double mode.) If the first operator is arithmetic, then the mode of the last item in the previous level is found to be used later with the ARO.SH routine.

Usage. The calling sequence is:

```
TSX1     ARO.SI
(Normal Return)
```

Error Returns. None.

19. ARO.SG--AROMLV Table Entering Routine

Purpose. This subroutine makes entries into the Mode Level Table (AROMLV).

Method. The routine is entered with the AROMLV entry in the A-register. The entry count is increased and the entry stored in the table.

Usage. The calling sequence is:

```
TSX1     ARO.SG
(Normal Return)
```

Error Returns. None.

20. AR.PNT--Two's Complement Computation Routine

Purpose. This subroutine is used to compute the two's complement of the Buffered Table pointer in an item stored in the T.ARIT Table.

Method. The pointer is placed in the upper A-register, the NEG instruction is executed and return is made to the calling routine.

Usage. The calling sequence is:

```
TSX1     AR.PNT
(Normal Return)
```

Upon entry, the T.ARIT item is in the A-register. Upon return the two's complement of the pointer is in the upper A-register.

Error Returns. None.

STATEMENT PROCESSORS

1. IFP100--IF Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as an IF statement. Both Arithmetic and Logical IF statements are processed by this routine. The actual processing of the parenthesized expression is done by the Arithmetic Processor which returns a flag, F.LOGL, indicating whether the expression is Arithmetic or Logical and a parameter describing the length of the T.ARIT Table. After control is returned, the routine continues to process the statement according to the information returned.

Method. The general form of the statement is:

```
IF(A)B
```

The IF processor does not process the expression within parentheses, but calls the Arithmetic processor to treat the expression A. If A is Arithmetic, then B must be three branch EFN's. If A is a logical, the B must be a statement other than another Logical IF or a DO. The BCD text is made available via the scanning routines. For the Arithmetic IF, the three branches are collected, entered into the T.JUMP Table, and the appropriate POOL Table entry is made. The T.JUMP Table entries for the destination are EFNs.

For the Logical IF, a partial T.JUMP Table entry is made for the false condition. A pointer to this T.JUMP Table entry is saved in cell LGIF.P (bits 0-17) so that the Executive Routine can complete the T.JUMP Table entry with a destination IFN after the True statement has been processed. The appropriate POOL Table entry is made.

Table Entries

T.JUMP Table

Arithmetic IF

0 2	3 17	18 20	21 35
0	IFN Origin	0	Destination EFN_1
0	IFN Origin	0	Destination EFN_2
0	IFN Origin	0	Destination EFN_3

Logical IF

0 2	3 17	18 20	21 35
0	IFN Origin	0	Destination IFN

POOL Table

Arithmetic IF

0 2	3 17	18 20	21 35
7	5_8	0	IFN
	T.ARIT String . . .		
0	3(count)	0	$P(T.JUMP)_1$
0	$P(T.JUMP)_2$	0	$P(T.JUMP)_3$

Logical IF

0 2	3 17	18 20	21 35
7	6_8	0	IFN
	T.ARIT String . . .		
0	1(count)	0	P(T.JUMP) False

Usage. The calling sequence is:

```
TSX1      IFP000
(Normal Return)
```

Error Returns. There are no error returns but the following diagnostic messages are written:

IFIER1	AN IF STATEMENT WITH NO LEFT PARENTHESIS WAS ENCOUNTERED.
IFIER3	A STATEMENT NUMBER IN N, N, N IS NOT NUMERIC.
IFIER4	NO COMMA EXISTS BETWEEN STATEMENT NUMBERS.
IFIER5	CHARACTER FOLLOWING LOGICAL IF IS NOT VALID.

Additional error messages may be written by the Arithmetic Processor.

2. GTO100--GOTO Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a GO TO statement. The routine will process unconditional GO TO and the conditional computed and assigned GO TO statements. The three statement forms are:

GO TO A_n	Unconditional
GO TO (A_n.......),B	Computed
GO TO Z(A_n......)	Assigned

Method. The BCD text is made available via the scanning routines. The routine determines the type of GO TO by analysis of the character following the words GO TO. The typing is done according to the following logic. If the character is numeric, then the type is "unconditional." If it is alphabetic, then the type is "assigned." If it is punctuation, then the type is "computed."

For unconditional GO TO statements, the routine collects the destination EFN and makes an entry in the T.JUMP Table and a two-word entry is placed in the POOL Table.

For assigned GO TO statements, the routine collects Z and places it in the NAME Table (if it is not already there) and places the Assigned GO TO control word in the POOL Table prototype. The routine then calls the S.GTOL routine to process the branch list. It completes processing by entering the NAMEP for this branch symbol, Z, in the POOL Table prototype and making the POOL Table entry.

For Computed GO TO statements, the routine places the Computed GO TO control word in the POOL Table prototype and then calls the S.GTOL routine to process the branch list. Upon successful return from S.GTOL, it collects the branch symbol, B, and enters it into the NAME Table if it has not already been entered. The NAMEP is placed in the POOL Table prototype. Processing is completed by making the POOL Table entry.

The POOL Table entry for Assigned and Computed GO TO statements is variable in length (3 + n/2) where n is the number of statement numbers in the branch list.

Table Entries

T.JUMP Table

0 2	3 17	18 20	21 35
0	IFN (origin)	0	EFN (destination)

POOL Table

Unconditional GO TO

0 2	3 17	18 20	21 35
7	7_8	0	IFN
0	P(T.JUMP)	0	0

Conditional GO TO

0 2	3 17	18 20	21 35
7	T	0	IFN
0	BRANCH COUNT	0	$P(T.JUMP)_1$
	.		.
	.		.
	.		.
0	$P(T.JUMP)_{n-1}$	0	$P(T.JUMP)_n$
0	(NAMEP) ϕ	0	0

where T = 10_8 for assigned GO TO
T = 11_8 for computed GO TO
ϕ = branch symbol

Usage. The calling sequence is:

```
TSX1     GT0100
(Normal Return)
```

Error Returns. The following errors stop further scanning of the statement and result in fatal diagnostic messages:

GTOER1	Illegal punctuation.*
GTOER3	COMPILER EXPECTS A LEFT PARENTHESIS TO OPEN BRANCH LIST.
GTOER4	EXTRANEOUS INFORMATION FOLLOWING END OF STATEMENT HAS BEEN FOUND.
GTOER5	_______ FOLLOWING RIGHT PARENTHESIS IS ILLEGAL. COMPILER EXPECTS A BRANCH NAME.
GTOER7	THE BRANCH NAME _______ MUST BE AN INTEGER VARIABLE.
GTOER8	THE BRANCH NAME HAS APPEARED AS A DIMENSIONED VARIABLE OR FUNCTION/SUBROUTINE NAME.
GTOER9	THE EFN IS ZERO.

The following errors cause only a warning message and the scan is continued:

GTOER2	A COMMA IS MISSING IN FRONT OF LIST OF STATEMENT NUMBERS. WARNING ONLY.
GTOER6	COMMA IS MISSING FOLLOWING THE RIGHT PARENTHESIS.

*One of the following fatal diagnostics will be written when illegal punctuation is encountered:

S.PC30	ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL _______.
S.PC35	THE PUNCTUATION MARK _______ WAS USED INCORRECTLY.
S.PC40	MISSING PUNCTUATION AT THE END OF STATEMENT.

3. S.GTOL--Branch Collector for GO TO Statements Routine

Purpose. This routine is called by the GT0100 routine to process the branch list when the source statement is determined to be a conditional GO TO.

Method. The BCD text is made available via the scanning routines. The scan begins with the first character following the left parenthesis that encloses the branch list and terminates when a right parenthesis is encountered.

For each EFN encountered, an entry is made in the T.JUMP Table and also the T.COGO Table. The T.COGO Table is used as erasable storage to collect the entries that will be placed in the POOL Table string.

When returning to the caller, the A-register will contain a word describing the POOL routine parameter to be used for making the POOL Table entry.

Bits	0-17	Location of first word in T.COGO
Bit	18	1 indicating intermediate parameter to S.TP00
Bits	19-35	Number of words in T.COGO

Each word except the first in the T.COGO Table will contain pointers to the T.JUMP Table for two EFN's in the branch list.

Table Entries

T.JUMP Table

0 2	3 1718 20	21 35
0	IFN (origin)	EFN (definition)

T.COGO Table

0 2	3 1718 20	21 35
0	BRANCH COUNT 0	$P(T.JUMP)_1$
	.	.
	.	.
	.	.
0	$P(T.JUMP)_{n-1}$	$P(T.JUMP)_n$

Usage. The calling sequence is:

```
TSX1     S.GTOL
(Error Return)
(Normal Return)
```

Error Returns. The following errors stop further scanning of the statement and result in a diagnostic message being given.

S.GTER	ALPHABETIC CHARACTERS OR ILLEGAL PUNCTUATION FOUND IN BRANCH LIST.
S.GTR2	AN EXTERNAL FORMULA NUMBER IN THE BRANCH LIST IS ZERO.
S.PCER	Illegal punctuation results in one of the following diagnostics:
S.PC30	ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL ________.
S.PC35	THE PUNCTUATION MARK ________ WAS USED INCORRECTLY.
S.PC40	MISSING PUNCTUATION AT THE END OF STATEMENT.

4. DOP100--DO Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a DO statement. The statement will be tested for errors and if none are found, the statement will be broken down into various table entries.

Method. The routine will scan the source statement commencing with the first character following the letter O in the word DO and terminating at the end mark. The BCD text is made available through the scanning routines. The DO statement is of the form:

$$\text{DO} \quad n \quad i \;=\; m_1,\ m_2,\ m_3$$

where:

n = the EFN of the statement that is the terminus of the DO loop.

i = a nonsubscripted integer variable index.

m = are three parameters which may be integer constants or nonsubscripted integer variables describing the control of indexing within the DO loop.

The first two parameters, m_1 and m_2, must be present and separated by a comma. If the third parameter, m_3, is not given, an implied m_3 with a value of 1 is supplied.

This routine will process the statement by collecting the terminating EFN, the index i, and the parameters, m. An entry will be made in the T.IODO Table. A POOL Table entry will be constructed using the T.IODO Table pointer.

An entry consisting of the terminating EFN and the pointer to the T.IODO Table will be made in the T.DODO Table. This latter entry will be processed by the DO Beta Assignment routine, S.DBOO, to determine when a statement is the end of a DO loop. This provides a means of completing the T.IODO entry by replacing the terminating EFN with a terminating IFN.

Table Entries

T.DODO Table   1 word entry

0 2	3 17	18 20	21 35
0	P(T.IODO)	0	EFN (DO termination)

T.IODO Table   3 word entry

	0 2	3 17	18 20	21 35
	0	IFN (origin)	0	EFN (destination)
or	0	IFN (origin)	0	IFN (destination)*
	0	NAMEP (index)	N	m_1
	N	m_2	N	m_3

where:

N = 0, if m is the NAME Table pointer

N = 4, if m is a binary constant

*The IFN (destination) is placed in the T.IODO Table by the S.DBOO routine.

POOL Table

0 2	3 17	18 20	21 35
7	21_8	0	IFN
0	P(T.IODO)	0	0

Usage. The calling sequence is:

```
TSX1      DOP100
(Normal Return)
```

Error Returns. The following errors stop further scanning of the statement and result in a diagnostic message being given.

DOPER1	THE EFN, ________ IS ILLEGAL.
DOPER2	ILLEGAL PUNCTUATION EXISTS FOLLOWING THE EFN.
DOPER4	THE INDEX NAME OR PARAMETER IS NOT AN INTEGER VARIABLE.
DOPER5	AN INDEX NAME OR PARAMETER MAY NOT BE DIMENSIONED.
DOPER6	THE INDEX NAME OR PARAMETER HAS PREVIOUSLY APPEARED AS A FUNCTION/SUBROUTINE NAME.
DOPER7	COMPILER EXPECTS AN EQUAL SIGN FOLLOWING INDEX NAME.
DOPER8	ONE OF THE PARAMETERS N1, N2, N3 IS ZERO.
DOPER9	Punctuation error causes one of the following diagnostics:
S.PC30	ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL ________.
S.PC35	THE PUNCTUATION MARK ________ WAS USED INCORRECTLY.
S.PC40	MISSING PUNCTUATION AT THE END OF STATEMENT.

5. DOPCK1--Numeric Checker Routine

Purpose. This routine checks the numeric values assigned to a DO index.

Method. The numeric values assigned as a DO (or implied DO) index are checked for the following:

- Numerics greater than 6 characters.
- Numerics greater than 2^{17}.
- Numerics equal to zero.

If one of the above conditions is found, a fatal diagnostic is written and an appropriate exit is taken from the routine.

Usage. The calling sequence is:

```
TSX1     DOPCK1
(Error Return 1)
(Error Return 2)
(Normal Return)
```

where:

Error Return 1 = Numeric greater than 6 characters.
Error Return 2 = Numeric equal to 2^{17} or 0.

Error Returns. The following fatal diagnostics are written:

DOPCK7	NUMERIC VALUE ASSIGNED TO DO INDEX OR IMPLIED DO HAS MORE THAN 6 CHARACTERS.
DOPCK8	A NUMBER IN K = K1, K2, K3 IS ZERO.
DOPCK9	A NUMBER IN K = K1, K2, K3 IS BIGGER THAN 2 TO 17TH.

6. ASN100--ASSIGN Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as an ASSIGN statement. The statement will be tested for errors and, if none are found, the statement will be broken down into various table entries. The form of the ASSIGN statement is:

ASSIGN i TO n

where:

i is an EFN
n is a nonsubscripted integer variable

Method. The BCD text is made available via the scanning routines. The scan will begin with the first character following the N in the word ASSIGN and terminate with the end mark.

The routine collects the EFN, i, and converts it to binary for entry in the POOL Table prototype. It then checks for the two characters TO following i. The assign parameter, n, is then collected and entered into the NAME Table if it is not already there. The NAMEP is entered in the POOL Table prototype and then a two-word POOL Table entry is made.

Table Entries

POOL Table

0 2	3 17	18 20	21 35
7	20_8	0	IFN (Binary)
0	(NAMEP)n	0	EFN (Binary)

Usage. The calling sequence is:

```
TSX1     ASN100
(Normal Return)
```

<u>Error Returns.</u> The following errors stop further scanning of the statement and result in fatal diagnostic messages being written:

ASNER1	THE FORMULA NUMBER FOLLOWING THE WORD, ASSIGN, IS MISSING.
ASNER2	COMPILER EXPECTS THE WORD -TO- FOLLOWING THE STATEMENT NUMBER.
ASNER3	THE STATEMENT MUST HAVE AN INTEGER NAME FOLLOWING THE WORD TO.
ASNER4	THE VARIABLE NAME FOLLOWING THE WORD -TO- HAS BEEN USED AS A SUBPROGRAM NAME.
ASNER5	THE VARIABLE NAME FOLLOWING THE WORD -TO- MAY NOT BE A SUBSCRIPTED VARIABLE.

7. CAL100--CALL Statement Processor

<u>Purpose.</u> This routine is called after the Dictionary Scan has classified the source statement as a CALL statement. If an argument list exists, it is processed by the Arithmetic Processor. Otherwise, processing of the statement is confined to this routine.

<u>Method.</u> The BCD text in the SSWW region is made available via the scanning routines. The routine will save the Scan position in case it is necessary to rescan the first part of the statement as is the case when an argument list is present. Since a subroutine may redefine integer variables in COMMON, special T.INTS and T.RINT Table entries are made to indicate that this statement is a possible redefinition of COMMON nonsubscripted integer variables. The subroutine name flags are checked for proper usage. If not already set, the subroutine-reached-by-CALL flag, I.CAL, and the external-subprogram-name flag, I.XTN, are set ON.

At this point, determination is made whether or not the statement contains an argument list. If it does not have an argument list, the CAL100 routine sets up the appropriate T.ARIT string and entry is made into the POOL Table. An entry is also made in the T.RINT Table if the following conditions are met:

- The statement being processed exists within a DO loop.
- The current DO index name is in COMMON or EQUIVALENCE.

The cell, F.DONM, contains a pointer to the current index name and is used in making the two previous tests. The T.RINT entry is made to indicate that this statement is a possible redefinition of the nonsubscripted integer variable that exist in COMMON.

If an argument list exists, the Scan position is restored to the beginning of the subprogram name and the Arithmetic Processor is called. Flag words are passed to the Arithmetic Processor to indicate that it should treat the statement as a function with an argument list. The Arithmetic Processor stops processing when it encounters the closing right parenthesis of the argument list and returns a parameter word, ARITCT (bits 0-17), describing the length of the T.ARIT string. The POOL Table parameter list is set up, a POOL Table entry is made, and control is returned to the caller.

Processing the argument list may give rise to NAME Table entries and additional T.INTS and T.RINT entries. If the subprogram NAME has not previously appeared in the source program, it will also be entered into the NAME Table. If the statement has an EFN, it will be entered into the T.EIFN Table. An argument which is a nonstandard return is treated as a special constant with the statement number of the return entered in the T.NUMB Table. An entry is also made in the T.JUMP Table.

Table Entries

T.RINT Table

0 2	3 17	18 20	21 35
0	NAMEP (index)	0	IFN

T.NUMB Table

0	5 17	18 35
	1	1
13	0	EFN

EFN = Statement number of a nonstandard return.

T.JUMP Table

0 2	3 17	18 35
1	IFN	EFN

EFN = Statement number of a nonstandard return.
1 = A bit in position 2 indicates that a transfer to the next IFN is not necessary.

T.INTS Table

0 2	3 17	18 20	21 35
0	00000	0	IFN

Where zero NAMEP indicates CALL statement with no argument list.

POOL Table

CALL (with arguments)

0 2	3 17	18 20	21 35
7	26_8	0	IFN
T.ARIT information			

CALL (without arguments)

0 2	3 17	18 20	21 35
7	26_8	0	IFN
0	00000	0	00000
0	CODE	0	NAMEP (subroutine)
0	00000	0	00000

Where CODE is the Function Operator code needed by the Arithmetic Processor in Phase Two.

<u>Usage.</u> The calling sequence is:

```
TSX1     CAL100
(Normal Return)
```

<u>Error Returns.</u> The following error will interrupt the scan to give a diagnostic message. The scan then continues in the normal manner:

CALER3	THE SUBROUTINE NAME WAS PREVIOUSLY DEFINED AS A FUNCTION SUBPROGRAM.

The following errors stop further scanning of the statement and result in a diagnostic message being given:

CALER1	THE SUBROUTINE NAME SHOULD FOLLOW THE WORD CALL. PUNCTUATION OR NUMERICS FOUND INSTEAD.

CALER2	THE SUBROUTINE NAME WAS USED AS A VARIABLE PREVIOUSLY.
CALER4	THE SUBROUTINE NAME WAS PREVIOUSLY DEFINED AS AN ARITHMETIC STATEMENT FUNCTION OR A BUILT-IN FUNCTION.
CALER6	SUBROUTINE CALLS ________ ITSELF.
CALER5	Punctuation errors cause one of the following diagnostics and the scan to be discontinued.
S.PC30	ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL ________.
S.PC35	THE PUNCTUATION MARK ________ WAS USED INCORRECTLY.
S.PC40	MISSING PUNCTUATION AT THE END OF STATEMENT.

8. PAS100--PAUSE Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a PAUSE statement. An entry is made in the POOL Table.

Method. The routine makes a two word entry into the POOL Table, then checks for an end mark.

Table Entry

POOL Table

0 2	3 17	18 35
7	15_8	IFN
0	0	N

where: N = An octal integer constant, 1-5 digits. (If N is nonexistent, a zero is entered for N.)

Usage. The calling sequence is:

```
TSX1     PAS100
(Normal Return)
```

Error Returns. There are no error returns, but the following diagnostic is written:

PASBD1	ILLEGAL CHARACTER(S) FOLLOWING PAUSE.

9. RET100--RETURN Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a RETURN statement. The return number will be processed and a POOL Table entry made.

Method. The BCD text is made available via the Scanning routines. The routine collects the return number, if any, and if it is numeric, then it will be converted to binary form. If the return number is a variable, it is entered in the NAME Table if it has not been previously entered. The NAMEP or the return number is entered in the POOL Table prototype. A two-word POOL Table entry is made. Return to the calling program is made upon completing the POOL Table entry.

Table Entries

POOL Table

0 2	3 17	18 35
7	27_8	IFN
X	Y	0

where: X = 0, the return is an integer variable, and Y = NAMEP.

X = 4, the return is a constant, and Y = the return number.

Usage. The calling sequence is:

```
TSX1     RET100
(Normal Return)
```

Error Returns. There is no error return for this processor, but the following diagnostic messages are written:

RETBD1	A MAIN PROGRAM SHOULD NOT CONTAIN A RETURN STATEMENT.
RETBD2	RETURN IS NOT AN INTEGER VARIABLE.
RETBD4	NONSTANDARD RETURNS NOT ALLOWED.

10. END100--END Statement Processor

Purpose. This routine will set the flag, F.END., on.

Method. The F.END. flag is set to indicate that an END card has been encountered. A check is made for an end mark and control returned to the caller. The Executive Routine makes a one-word POOL Table entry for the END statement.

Table Entries

POOL Table

0 2	3 17	18 20	21 35
7	25_8	0	IFN

Usage. The calling sequence is:

```
TSX1     END100
(Normal Return)
```

Error Returns. None.

11. STP100--STOP Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a STOP statement. The sole purpose of this routine is to make a POOL Table entry.

Method. The routine makes a one word POOL Table entry.

Table Entries

POOL Table

0 2	3 17	18 20	21 35
7	23_8	0	IFN

Usage. The calling sequence is:

```
TSX1     STP100
(Normal Return)
```

Error Returns. None.

12. CNT100--CONTINUE Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a CONTINUE statement. The sole purpose of this routine is to make a one-word POOL Table entry.

Method. The routine makes a one-word POOL Table entry and then checks for an end mark.

Table Entries

POOL Table

0 2	3 17	18 20	21 35
7	22_8	0	IFN

Usage. The calling sequence is:

```
TSX1     CNT100
(Normal Return)
```

Error Returns. None.

13. DTA100--DATA Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a DATA statement. The statement will be tested for errors and, if none are found, the statement will be broken down into various table entries.

Method. This routine will scan the source statement commencing with the first character following the final letter A in the word DATA and terminating at the end mark. The BCD text is made available via the Scanning routines. The DATA statement is processed in sections:

- The List Processor which scans the list portions of the statement is logically divided into two sections. One section processes a list within implied DO parentheses, making entries in the T.IOLT, T.IMPO and T.DATA Tables. The other section of the List Processor will handle lists containing only nonsubscripted variables, variables with constant subscripts and array variables, using the short list notation (an array name with no subscripts). This latter section will make entries into the T.DATA Table.

- The Literal String Processor scans the literal strings and makes entries into the T.LITR Table.

The List Processor recognizes and ignores redundant parentheses. Whenever a right parenthesis is encountered an entry is made in the temporary T.IOLT Table and a partial entry is made in the T.DATA Table. When the matching right parenthesis is encountered, the incomplete T.DATA entry can be completed and the last entry in the T.IOLT Table is deleted. The T.IOLT Table entry is essentially a parentheses count that will reduce to zero if no parenthetical errors have occurred in processing a list.

Table Entry

T.IOLT Table

0 2	3 17	18 20	21 35
0	P(T.DATA) Left parenthesis entry	0	00000

The List Processor uses the S.SS00 routine to scan the subscripts of each dimensioned variable in the list.

The variable name is added to the NAME Table (if not already there) with the Type flag (I.REL or I.ITG) set ON implicitly. For each variable in the statement, proper usage is checked and true variable usage flag, I.RVR, and the explicit type flag, I.EXP, are set ON in the NAME Table flag word. Thus, usage of a variable name in a DATA statement has the same binding strength as usage in an arithmetic statement.

In processing dimensioned variables, the subscript processor, S.SS00, ensures that there is a unique T.USUB Table entry for each appearance of a subscripted variable.

In addition to the NAME Table and T.USUB Table entries, the DATA statement information is broken down into three other buffer tables for use at the end of Phase One to allocate storage.

Table Entry

T.DATA Table. The list part of the DATA statement will cause a T.DATA Table entry string to be generated. One entry is made to identify the beginning of the string. An entry is made for each variable name with a parameter in bits 0-2 defining the appearance of subscripted, nonsubscripted and short list notation variables. An entry is made for each left parenthesis encountered and each nonredundant right parenthesis.

The left parenthesis entries are incomplete at the time they are entered in T.DATA. They are completed with a pointer to the T.IMPO Table when a right parenthesis is encountered following indexing information.

T.DATA Table

	0 2	3 17	18 20	21 35
	7	00000	0	00000
NON-SS VAR.	0	NAMEP	0	00000
SH. LIST VAR.	3	NAMEP	0	(dimension product) - 1
LEFT PAREN.	5	00000	0	P(T.IMPO)
SS VARIABLE	1	NAMEP	0	P(T.USUB)
RIGHT PAREN.	2	00000	0	P(T.DATA) Corresponding left parenthesis entry

Table Entry

T.IMPO Table. This table is generated when implied DO indexing is encountered. A two-word entry describing the index range information is generated.

T.IMPO Table

	0 2	3 17	18 20	21 35
N = N1,N2,N3	0	(NAMEP) index	0	N1
	0	N2	0	N3

Table Entry

T.LITR Table. The literal string part of the DATA statement will cause a T.LITR Table entry string to be generated. An entry of

$$2 + \frac{n - 1}{6}$$

words is made for each literal in the string, where n is the number of nonblank characters in the literal except in an alphanumeric field where the blanks are retained. An entry of all zeros is made as a flag that the literal string entry is complete. There is a one-to-one correspondence between T.DATA and T.LITR Table entry strings. The literal collector routines are used in processing the literal string.

T.LITR Table

0 2	3 17	18 20	21 35
P	REPEAT COUNT	T	N = WORD COUNT
	N WORDS OF DATA IN BCD		
0	00000	0	00000

Where P = 0 if REAL
P = 1 if INTEGER
P = 2 if LOGICAL
and T = 4 for DEC literal
T = 2 for OCT literal
T = 1 for BCI literal

P = 3 if OCTAL
P = 4 if COMPLEX
P = 7 if DOUBLE PRECISION

Usage. The calling sequence is:

```
TSX1     DTA100
(Normal Return)
```

Error Returns. The following errors stop further scanning of the statement and result in a diagnostic message:

DTAER1	THE VARIABLE NAME __________ BEGINS WITH A NUMERIC CHARACTER.
DTAER3	NO LITERAL LIST WAS FOUND IN THE DATA STATEMENT.
DTAER4	Illegal Punctuation*
DTAER5	TOO MANY RIGHT PARENTHESES IN DATA STATEMENT.
DTAER6	TOO MANY LEFT PARENTHESES IN DATA STATEMENT.
DTAER8	AN ARRAY NAME WITH VARIABLE DIMENSIONS IS OUTSIDE THE RANGE OF AN IMPLIED DO.
DTAER9	PUNCTUATION FOLLOWING A RIGHT PARENTHESIS IS MISSING.
DTAE13	THE SYMBOL __________ IS USED INCORRECTLY FOLLOWING AN = SIGN.
DTAE14	A LITERAL BEGINS WITH AN ILLEGAL CHARACTER.
DTAE15	MISSING PUNCTUATION FOLLOWING A LITERAL.
DTAE16	THE ALPHANUMERIC FIELD COUNT IS TOO BIG.

The following errors interrupt the scan, output a diagnostic message and then continue the scan of the statement:

DTAER7	A VARIABLE,________ CONTAINS ADJUSTABLE DIMENSION(S).
DTAE10	A VARIABLE,________ WITH SHORT LIST NOTATION IS WITHIN AN IMPLIED DO.
DTAE11	THE NONDIMENSIONED VARIABLE,__________ IS WITHIN AN IMPLIED DO.
DTAE12	THE ARRAY,__________ WITHIN AN IMPLIED DO SEQUENCE HAS CONSTANT DIMENSIONS.
DTAE18	A LITERAL IS TOO LONG.
DTACE6	A VARIABLE,__________ IS A SUBPROGRAM NAME OR ARGUMENT.
DTACE7	A VARIABLE, __________ IN THE DATA STATEMENT IS IN BLANK COMMON.
DTACE8	A VARIABLE, __________ NOT IN COMMON IS IN A BLOCK-DATA SUBPROGRAM.
DTACE9	A VARIABLE,__________ IN COMMON IS IN A NONBLOCK DATA SUBPROGRAM.
DOPCK7	NUMERIC VALUE ASSIGNED TO DO INDEX OR IMPLIED DO HAS MORE THAN 6 CHARACTERS.
DOPCK8	A NUMBER IN K = K1, K2, K3 IS ZERO.
DOPCK9	A NUMBER IN K = K1, K2, K3 IS BIGGER THAN 2 TO 17TH.
	*One of the following diagnostics will be given when illegal punctuation is encountered:
S.PC30	ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL __________.
S.PC35	THE PUNCTUATION MARK __________ WAS USED INCORRECTLY.
S.PC40	MISSING PUNCTUATION AT THE END OF STATEMENT.

14. DTACK1--Check Variable Names Routine

Purpose. This routine is called by the DATA Statement Processor whenever the name of a variable is encountered.

Method. This routine calls the NAME Table Routine, S.NAME, to enter the variable name in the NAME Table if it has not yet been entered. If a new entry is made, the Variable Typing Routine, S.TYPO, is called to type the variable. Upon return from the S.NAME Routine the variable type is checked to be sure it is consistent with its usage in the DATA statement. Three returns are made from this routine; one for nonsubscripted variables, one for subscripted variables and one for short list notation.

Usage. The calling sequence is:

```
TSX1     DTACK1
(Return 1)
(Return 2)
(Return 3)
```

Where:

Return 1 = Nonsubscripted variable
Return 2 = Subscripted variable
Return 3 = Short list notation

Error Returns. There are no error returns but the following fatal diagnostics are written:

DTACE6	A VARIABLE,________, IS A SUBPROGRAM NAME OR ARGUMENT.
DTACE7	A VARIABLE,________, IN THE DATA STATEMENT IS IN BLANK COMMON.
DTACE8	A VARIABLE,________, NOT IN COMMON IS IN A BLOCK-DATA SUBPROGRAM.
DTACE9	A VARIABLE,________, IN BLOCK COMMON IS IN A NONBLOCK DATA SUBPROGRAM.

15. S.VAER--Illegal Variable Name Error Routine

Purpose. This routine prints a diagnostic message to identify illegal variable names.

Method. Upon entering this routine, the illegal variable name is contained in the A-register. The name is stored in a message line and the routine DIAG is called to write the message. Return is then given to the calling program.

Usage. The calling sequence is:

```
TSX1     S.VAER
(Normal Return)
```

Error Returns. There are no error returns, but a fatal diagnostic message is written:

S.VAE5   THE NUMERAL ________ PRECEDES A VARIABLE NAME.

16. S.PCER--Illegal Punctuation Error Routine

Purpose. This routine writes a fatal diagnostic message concerning illegal or omitted punctuation.

Method. This routine attempts to determine the symbol nearest the the punctuation error and place it in a diagnostic message line. If the cell .FLD. does not contain a symbol, another message line is selected containing the termination character except when it is an end mark. An end of statement error message is written if the termination character is an end mark.

Usage. The calling sequence is:

```
TSX1     S.PCER
(Normal Return)
```

Error Returns. There are no error returns but three fatal diagnostic messages are written:

S.PC30	ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL __________.
S.PC35	THE PUNCTUATION MARK __________ WAS USED INCORRECTLY.
S.PC40	MISSING PUNCTUATION AT END OF STATEMENT.

17. RD0000--READ Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a READ statement. The form of the statement is:

READ n, list	Read on-line (standard file assignment)
READ (u,n) list	Read BCD file
READ (u) list	Read Binary file

where u is an unsigned integer constant or an integer variable referencing a file, and n is a FORMAT statement reference.

A classification will be made as to whether the statement is READ on-line or READ file. The routine will call either the On-Line Statement Processor OL0000, or the File Processor, TPPR00.

Pg 139

Method. The cell IINOUT is reset to zero to indicate the mode of I/O is input.

The BCD text in the SSWW region is made available via the Scanning routines. The routine scans the source statement following the word READ in order to determine the type of the statement. If punctuation is found, the statement is assumed to be READ file. If a numeric or alphameric FORMAT reference is found, the statement is assumed to be READ on-line. Once the classification has been made, the appropriate type flag is set in the POOL Table prototype. Control is then transferred to the On-line Processor if the statement has been classified as such. Otherwise, control is transferred to the File Processor. The routine called will complete the processing of the statement making the appropriate POOL Table entries. After the statement is processed, control is returned through the READ routine to the Executive routine.

Usage. The calling sequence is:

```
TSX1     RD0000
(Normal Return)
```

Error Returns. None. Errors are noted by the routines which scan the statement.

18. PR0000--PRINT Statement Processor
PN0000--PUNCH Statement Processor
RT0000--WRITE Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a PRINT, PUNCH or WRITE statement. The form of the statement is:

PRINT	n, list	Print on-line
PUNCH	n, list	Punch cards on-line
WRITE	(u,n) list	Write BCD file
WRITE	(u) list	Write Binary file

where u is an unsigned integer constant or an integer variable referencing an output file, and n is a FORMAT statement reference.

The routine will classify the statement and call the appropriate routines to do the processing.

Method. The cell IINOUT will be set nonzero to indicate that the I/O mode is output.

There are three entry points to this routine which will set the appropriate type flag in the first word of the POOL Table prototype. If the entry to this routine was PRINT or PUNCH, control will be transferred to (pg 139) the On-line Processor, OL0000. The WRITE entry will transfer control to the File Processor, TPPR00. The routine called will process the statement making the appropriate POOL Table entries. After the statement is processed, control is returned through the PRINT, PUNCH or WRITE routine to the Executive routine.

Usage. The calling sequences are:

```
TSX1     PR0000

 or

TSX1     PN0000

 or

TSX1     RT0000
```

(Control is returned to the next line.)

Error Returns. None. Errors are noted by the routines which scan the statement.

19. BK1000--BACKSPACE Statement Processor
RW1000--REWIND Statement Processor
EN1000--END FILE Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a BACKSPACE, REWIND or END FILE statement. This routine will make a two-word entry in the POOL Table. The form of the statement is:

```
BACKSPACE   u
REWIND      u
END FILE    u
```

where u is an unsigned integer constant or an integer variable.

Method. There are three entry points to this routine which will set the proper type flag in the first word of the POOL Table prototype. The routine then calls the UNT000 routine to process the file reference, u, and complete the POOL Table prototype. The two-word POOL Table entry is made and control returned to the Executive routine.

Table Entries

POOL Table

0 2	3 17	18 20	21 35
7	Type	0	IFN
f	u	0	0

where Type = 34_8 for BACKSPACE
= 33_8 for REWIND
= 32_8 for END FILE
and f = 4 for a constant file reference
= 0 for a symbolic file reference

Usage. The calling sequences are:

```
TSX1     BK1000

 or

TSX1     RW1000

 or

TSX1     EN1000
```

(Return is to the next line of coding.)

Error Returns. None. For an illegal file reference, an error message is written by the UNT000 routine.

20. OL0000--On-Line Processor

Purpose. This routine is called by the READ, PRINT, and PUNCH Statement Processors. The routine will call the appropriate routines to process the FORMAT reference, the list, if one exists, and the routine to end I/O, S.ENDI. The On-line Processor makes a two-word POOL Table entry (Begin I/O) which are the first two words of the I/O POOL Table entry string.

The general form of the statement processed by this routine is:

S n t

where: S is either READ, PRINT or PUNCH
n is the FORMAT reference
t is either a comma or an end mark

If t is a comma, then the expected form of the statement is:

S n, L

where: L is a list of elements to be transmitted.

Method. This routine is entered with the POOL Table type flag in the A-register and the scan pointer positioned at the FORMAT reference n. An IFN is generated and the first word of the POOL Table prototype is set up. It then calls the FM0000 routine to process the format reference n. A two-word POOL Table entry is made at the beginning of an I/O string. If t is a comma, the LIST00 routine is called to process the list. Whether or not the statement contained a list, the S.ENDI routine is called to end the POOL Table string for the I/O statement. Control is returned to the caller.

Table Entries

POOL Table

0 2	3 17	18 20	21 35	
7	Type	0	IFN	Begin I/O
0	0	f	n	

Type = 14_8 for READ
= 13_8 for PRINT
= 12_8 for PUNCH
f = 4 if n is a constant FORMAT reference
= 0 if n is a variable FORMAT reference

Usage. The calling sequence is:

```
TSX1     OL0000
(Normal return)
```

See on line routine
p 81 86

Error Returns.

OLDIAG COMMA OR END EXPECTED AFTER FORMAT REFERENCE.

Other error messages may be given by the routines which scan and process the statement.

21. TPPR00--File Processor

Purpose. This routine is called by the READ Processor when the statement is classified as READ File, and by the WRITE Processor. The File Processor will call the appropriate routines to process the file reference, to process the FORMAT reference and list if they exist. A call is also made to the S.ENDI routine to end the I/O POOL Table string. The routine makes a two-word POOL Table entry (BEGIN I/O) which is the first two words of the I/O POOL Table entry string. The forms of the statements which refer to file input or output are:

```
READ or WRITE  (u,n)  t
READ or WRITE  (u)  L
```

where: u is the file reference
n is the format reference

If t is not an end mark, then the expected form is:

```
READ or WRITE  (u,n)  L
```

where: L is a list of elements to be transmitted.

Method. This routine is entered with the POOL Table type flag in the A-register and the scan pointer positioned at the left parenthesis preceding the file reference. An IFN is generated and the first word of the POOL Table prototype is set up. It then calls the UNT000 routine to process the file reference u. If the file reference is followed by a comma, it calls the FM0000 routine to process the FORMAT reference or NAMELIST name, n. A two-word POOL Table entry is then made as the beginning of an I/O string.

If the t following the right parenthesis of the u,n specification is not an end mark, the list processor LIST00 is called. Whether or not the statement contained a list, the S.ENDI routine is called to end the POOL Table string for the I/O statement. Control is then returned to the caller.

Table Entries

POOL Table

0 2	3 17	18 20	21 35
7	Type	0	IFN
f_1	u	f_2	n

Type = 31_8 for READ
= 24_8 for WRITE
f_1 = 4 if u is a constant
= 0 if u is a variable
f_2 = 4 if n is a constant
= 0 if n is a variable or zero (binary read/write)

Usage. The calling sequence is:

```
TSX1     TPPR00
(Normal Return)
```

Error Returns.

NLPDIG	LEFT PARENTHESIS IS EXPECTED AFTER THE WORD READ OR WRITE.
NRPDIG	THERE SHOULD BE A COMMA OR RIGHT PARENTHESIS AFTER THE FILE, OR A RIGHT PARENTHESIS AFTER THE FORMAT.

The following diagnostic gives a warning message:

EXCOMA	UNNECESSARY COMMA. CORRECT FORM IS (FILE, FORMAT) LIST.

Other error messages may be given by the routines which scan and process the statements.

22. S.ENDI--END I/O Routine

Purpose. This routine is called by the On-line Processor and the File Processor routines to make a POOL Table entry which terminates the I/O POOL Table string.

Method. The condition of cell IINOUT is checked to determine whether the I/O statement being processed is input or output (zero if input, nonzero if output). The first word of the POOL Table prototype is set up for either END READ or END WRITE depending upon the condition of the cell IINOUT. The second word of the POOL Table prototype is identical to the second word of the BEGIN I/O POOL entry. The routine then makes a two-word END I/O POOL Table entry terminating the I/O string for the source statement being processed.

Table Entries

POOL Table

0 2	3 17	18 20	21 35	
7	Type	0	IFN	END I/O
f_1	u	f_2	n	

where Type = 37_8 for END READ
= 40_8 for END WRITE
f_1 = 4 if u is a constant file reference
= 0 if u is a variable file reference
f_2 = 4 if n is a constant FORMAT reference
= 0 if n is a variable FORMAT reference

Usage. The calling sequence is:

```
TSX1     S.ENDI
(Normal Return)
```

Error Return. None.

23. FM0000--FORMAT Reference Collector Routine

Purpose. This routine is called by the I/O Statement Processors when a FORMAT reference is to be processed. The FORMAT reference information will be placed in the second word of the BEGIN I/O POOL Table prototype.

Method. The scan of the statement will begin with the first character of the FORMAT reference. The BCD text of the source statement is made available via the Scanning routines. A FORMAT reference that is a constant (EFN) will be converted to binary and set in the POOL Table prototype. If the FORMAT reference is a variable name, the name will be entered in the NAME Table (if it is not already there) and the Implicit flag, I.IMP, will be set ON in the flag word. A check is made as to whether or not the variable is a NAMELIST name. A check is then made for legal usage of the FORMAT reference name, and if legal, it is placed in the second word of the BEGIN I/O POOL Table prototype. Return is then made to the caller. The second word of the POOL Table prototype is as follows upon return:

	0 2	3 17	18 20	21 35
		file reference	f_2	n
or	0	0	f_2	n

depending on whether or not there is a file reference in the source statement.

f_2 = 4 if the FORMAT reference n is a constant
= 0 if the FORMAT reference n is a variable or a NAMELIST name

Usage. The calling sequence is:

```
TSX1     FM0000
(Error Return)
(Normal Return)
```

Error Returns.

EM0060 A NAMELIST NAME HAS BEEN DEFINED ELSEWHERE.

DIMDIG	FORMAT VARIABLE IS NOT DIMENSIONED.
FMTDIG	PUNCTUATION OR ILLEGAL FORMAT REFERENCE HAS BEEN FOUND INSTEAD OF FORMAT REFERENCE.

24. UNT000--File Reference Collector Routine

Purpose. This routine is called by the I/O Statement Processors when a file reference is to be processed. The file reference information will be placed in the second word of the BEGIN I/O POOL Table prototype.

Method. The BCD text of the source statement is made available via the Scanning routines. The scan will begin with the first character of the file reference.

A constant file reference will be converted to binary and placed in the second word of the BEGIN I/O POOL Table prototype. If the file reference is a variable, the name will be entered in the NAME Table (if not already there) and the implicit flag, I.IMP, will be set ON in the flag word. A check is made of cell F.DONM which will indicate whether or not a DO is active at the present time. If a DO is active, the I/O statement being processed is contained within the DO, and a check has to be made as to whether a T.RINT Table entry must be made. A T.RINT Table entry will be made if either of the following conditions exist.

- The DO index and the file reference are equal.
- The DO index and the file reference are both equivalenced variables (equivalence flag, I.EQV, is set in flag word of both variables).

The variable name is then checked for proper usage and placed in the second word of the BEGIN I/O POOL Table prototype.

The second word of the POOL Table prototype is as follows upon return to the calling routine:

0 2	3 17	18 20	21 35
f	u	0	0

where: f = 4 if u is a constant
= 0 if u is a variable

Usage. The calling sequence is:

```
TSX1     UNT100
(Error Return)
(Normal Return)
```

Error Returns.

UNDIAG	PUNCTUATION OR ILLEGAL FILE REFERENCE HAS BEEN FOUND INSTEAD OF FILE REFERENCE.

25. LISTOO--I/O List Processor

Purpose. This routine is called by the On-line Processor and the File Processor when it is determined that a I/O list exists. The list will be scanned and checked for errors, and if none exist, the list will be broken down into various table entries.

Method. The I/O List Processor will scan from left to right until a left parenthesis is found which is not a subscript parenthesis, building as it proceeds various tables: POOL, T.IODO, T.USUB, T.SUBS, T.INTS, T.RINT. Upon finding such a left parenthesis (which may be a DO-implying parenthesis), a right to left scan will be performed, beginning with the end mark, and continuing until the initiating left parenthesis is again found. The Backward Scan establishes which left parentheses are DO-implying, and which are redundant and are to be disregarded. The Forward Scan then resumes.

Each item in the I/O list is associated with an IFN, which is either the same or higher by one than that of the preceding item. Each time the IFN is incremented, an entry is made in the I/O POOL Table string. The entry will consist of a list of one word entries preceded by a one-word label. The label word will contain the IFN and a flag (SETIN or SETOUT) depending on whether an input statement (READ) or an output statement (WRITE, PUNCH) is being processed. The list following the label will contain the NAME Table pointers, a flag to indicate whether the variable is subscripted or not (if subscripted, a pointer to the T.SUBS Table will be included), and an indication of whether or not the short-list notation was used.

The following conditions result in the IFN being incremented:

1. The first item in the list increases the IFN, where item is either a variable name or a DO-implying left parenthesis.
2. A DO-implying parenthesis produces a higher IFN, a T.IODO POOL entry, and an incomplete entry (with the IFN (origin) in it) in the T.IODO Table.
3. The variable following the DO-implying left parenthesis produces a higher IFN and thus a new SETIN/SETOUT POOL entry list.
4. When the indexing parameters of the implied DO are encountered, the IFN is incremented, and the indexing parameters and the newly created IFN are used to complete the T.IODO Table entry that was generated in 2 above. Since the IFN has been increased, an (END DO) POOL Table entry is also made.

5. The item following the right parenthesis of the implied DO produces a higher IFN. Thus if the next item is a variable, a new SETIN/SETOUT POOL list is begun; if it is a DO-implying left parenthesis, procedure 2 is followed; if the next item is the index of an implied DO (making it a nested DO), procedure 4 is followed.

6. A nonsubscripted integer variable which is not dimensioned in an input list produces a T.INTS Table entry. At this point the POOL Table entry string (including the entry for the variable that caused the T.INTS entry) is entered in the POOL Table. This in effect causes the item following to receive a higher IFN.

A buffer area, beginning with PTTEMP, is used to accumulate the SETIN and SETOUT lists before they are transferred to the POOL Table.

At the conclusion of the Backward Scan, LPRCNT (left parenthesis count) has the number of the last left parenthesis encountered in the scan; namely,the left parenthesis that initiated the Backward Scan. As the Forward Scan continues, the LPRCNT is decremented by 1 each time a left parenthesis is encountered (except for initiating left parenthesis). For all nonsubscript left parentheses, it compares its LPRCNT number against the last item in the CN.IDT Table (implied DO Table). If the numbers match, a DO-implying left parenthesis has been found. The cell TBLCNT, which is a count of the number of entries in the CN.IDT Table, is reduced by 1.

The Backward Scan is entered when a left parenthesis which is not a subscripting parenthesis is encountered. The Scan then begins at the end mark, looking only for left and right parentheses, and equal signs. The statement is scanned backwards to and including the parenthesis that initiated the Scan.

The counter, PARBAL, is increased by 1 for each right parenthesis encountered and decreased by 1 for each left parenthesis. At the conclusion of the Scan, PARBAL should be zero, and if not, an error message will be given indicating the left and right parenthesis do not balance.

The left parentheses, as they are encountered are numbered from 1 (LPRCNT). When an equal sign is encountered, DOCNTR (DO counter) is increased by 1. This incremented value is the number of remaining DO-implying left parenthesis that must be found.

If the routine, as explained below, decides that a given left parenthesis is a DO-implying parenthesis, it stores the number of that parenthesis (LPRCNT) in CN.IDT (implied DO Table). A count of the number of entries in CN.IDT is kept in TBLCNT (table counter). DOCNTR (DO counter) is also reduced by 1, thus, DOCNTR indicates the balance between equal signs and DO-implying left parenthesis.

An equal sign is taken as signifying an implied DO. However, if PARBAL has a value of zero, the equal sign was not preceded (in the Backward Scan) by a right parenthesis and an error message is given. What is wanted when the equal sign is found is to find the left parenthesis which is the mate of the right parenthesis which immediately preceded the equal sign. To accomplish this, PARBAL is given a value of 1; when this counter falls to zero, we have found its mate. However, when that left mate is found we wish to restore PARBAL and to continue the scan. The restored value will be 1 less than it had been when the equal sign was found. This will set PARBAL to the parenthesis twice removed from the equal sign. This can be done because scanning from the right parenthesis preceding the equal sign up to and including the DO-implying left parenthesis cancel each other out, and so add nothing to PARBAL.

When a left parenthesis is found, PARBAL is decreased by 1. If the result is not zero, and if DOCNTR (DO counter) is not zero, indicating that an implied DO has not been accounted for, a DO-implying left parenthesis has been found. This causes an entry (LPRCNT) to be made in the CN.IDT Table (implied DO) and the CN.IDT Table counter, TBLCNT, to be increased by 1. PARBAL is then restored to the value it had just prior to the implied DO, and the Scan continues.

Up to 8 nested DO's are handled by this routine. More than 8 nested DO's yield an error message.

Example (Backward Scan)

All numbers represent condition of cell after character has been scanned. PARBAL that are crossed out represent setting PARBAL to 1 when equal sign is encountered.

0 0001 2	~~2~~	1 2	~~2~~	1 2	1 ~~2~~1	PARBAL
(A,((((B),	I=1,3),	(C),	J=1,3),	(D),	K=1,3))	SCAN
7 6543		2		1		LPRCNT
012	3		2		1	DOCNTR
321						TBLCNT
	1		1		1	CNSAVE
654						CN.IDT

At 1st equal sign, PARBAL is set to 1, and 2-1 = 1 goes into CNSAVE
At 2nd equal sign, PARBAL is set to 1, and 2-1 = 1 goes into CNSAVE+1
At 3rd equal sign, PARBAL is set to 1, and 2-1 = 1 goes into CNSAVE+2

At LPRCNT4, PARBAL is 0 and DOCNTR is 3, therefore 4(LPRCNT) goes into CN.IDT and PARBAL is restored to CNSAVE+2, or 1.

At LPRCNT5, PARBAL is 0 and DOCNTR is 2, therefore, 5(LPRCNT) goes into CN.IDT+1 and PARBAL is restored to CNSAVE+1, or 1.

At LPRCNT6, PARBAL is 0 and DOCNTR is 1, therefore, 6(LPRCNT) goes into CN.IDT+2 and PARBAL is restored to CNSAVE, or 1.

At LPRCNT1, PARBAL is 0 and DOCNTR is 0 (indicating implied DO's are satisfied), therefore, LPRCNT is merely a redundant parenthesis.

Usage. The calling sequence is:

```
TSX1     LIST00
(Error Return)
(Normal Return)
```

Error Returns. The following errors cause the Forward Scan to be discontinued, a diagnostic message given and return made to the calling program:

E.ERR1	________ SHOULD BE AN ALPHABETIC VARIABLE.
E.ER10	TOO MANY NESTED IMPLIED DOS.
E.ER11	PUNCTUATION USED IMPROPERLY AFTER =.
E.ER12	AT LEAST 2 PARAMETERS MUST FOLLOW = IN AN IMPLIED DO.
E.ER13	IMPROPER PUNCTUATION AFTER ________.

The following errors will cause a diagnostic message to be given and the forward scan will continue:

E.ERR2	________ SHOULD BE PRECEDED BY A COMMA.
E.ERR3	ILLEGAL PUNCTUATION AFTER ________.
E.ERR4	LEFT PAREN AFTER ________ SHOULD BE PRECEDED BY A COMMA.
E.ERR5	EXTRA COMMA AFTER ________.
E.ERR6	________ IS A SUBSCRIPTED VARIABLE AND NOT DIMENSIONED.
E.ERR7	AN EQUAL SIGN IS NOT ENCLOSED BY PARENS.
E.ERR8	IMPROPER USE OF RIGHT PAREN AFTER ________.
E.ERR9	AN EQUAL SIGN IS NOT PRECEDED BY AN INDEX.
E.ER17	________ IS AN ADJUSTABLE DIMENSION AND SHOULD NOT BE REDEFINED.
E.ER18	________ IS A FUNCTION OR SUBROUTINE NAME.
E.RR19	________ IS AN INDEX AND SHOULD BE AN INTEGER.
E.RR22	LEFT AND RIGHT PARENS DO NOT BALANCE.

E.RR23 ________MUST BE AN INTEGER IN AN IMPLIED DO.

E.RR24 A PARAMETER OF ZERO HAS BEEN USED IN AN IMPLIED DO.

The following errors are possible when processing the DO indexing parameters. The first two will allow the Scan to continue; the third one will cause a return to the calling routine.

E.ER14 A RIGHT PAREN MUST FOLLOW________.

E.ER15 ANOTHER PARAMETER IS NEEDED AFTER ________.

E.ER16 COMMA OR RIGHT PAREN MUST FOLLOW.

The following errors are possible when the statement is being scanned backwards. The error message is given and the Backward Scan will continue:

RTPDIG A RIGHT PAREN MUST FOLLOW AN EQUAL SIGN.

LFPDIG A LEFT PAREN HAS NO MATE.

26. S.CHEK--Type Variable Routine

Purpose. This routine determines the type of variable which appears in the list of an INPUT/OUTPUT statement.

Method. This subroutine is called by the list of I/O Statements Processor, LIST00, to determine the type of variable. The actual typing is done by the S.TYP0 routine. Upon return from S.TYP0 the type is checked for input or output. If an input type, control is immediately returned to the calling routine, LIST00. If an output type, the implicit flag is turned ON in the flag word and control returned to LIST00.

Usage. The calling sequence is:

```
TSX1     S.CHEK
(Normal Return)
```

Error Returns. None.

27. S.IMFG--Check Implicit Flag Routine

Purpose. This routine types a variable and adds the implicit flag.

Method. This routine is called by the list of I/O Statements Processor, LIST00, immediately following a call to the NAME Table routine, S.NAME. The variable is typed by the routine, S.TYP0, the implicit flag is added to the flag word and control returned to the calling program.

Usage. The calling sequence is:

```
TSX1     S.IMFG
(Normal Return)
```

Error Returns. None.

28. S.AJSO--Variable Check Routine for T.INTS and T.RINT

Purpose. This routine examines the variables in the list of an I/O statement and makes appropriate table entries.

Method. The variable is examined and if it is an output integer, an entry is made in the T.RINT Table. If the variable is a FUNCTION name, the cell F.FUNM is set ON. All integer variables will cause an entry to be made to the T.INTS Table. If the short-list notation is used, a POOL Table flag is set. Control is returned to the calling program, LIST00.

Usage. The calling sequence is:

```
TSX1     S.AJSO
(Normal Return)
```

Error Returns. None.

29. CN0000--Backward Scan Routine

Purpose. This routine is called to perform a Backward (right-to-left) Scan of the list of an I/O statement.

Method. This routine scans the list of an I/O statement backwards and checks for implied DO's and a balance of parentheses.

Usage. The calling sequence is:

```
TSX1     CN0000
(Normal Return)
```

Error Returns. There are no error returns for this routine, but the following fatal diagnostics are written:

RTPDIG	A RIGHT PAREN MUST FOLLOW AN EQUAL SIGN.
LFPDIG	A LEFT PAREN HAS NO MATE.

30. S.NTRO--POOL Table Entry Routine

Purpose. This subroutine makes entries into the POOL Table as required during the processing of a list of an I/O statement.

Method. Variable length I/O string entries are made in the POOL Table using S.TP00, the POOL Table routine. (Refer to Figure 5, page 19). Return is made to the calling program.

Usage. The calling sequence is:

```
TSX1     S.NTRO
(Normal Return)
```

The call LISTNO contains the number of entries.

Error Returns. None.

31. S.SUB1--Parameter Processor (I/O List and Implied DO's)

Purpose. This subroutine checks the parameters of an implied DO in an I/O list.

Method. The parameters of an implied DO in the I/O list are checked. If there is a third parameter, the T.IODO Table entry is completed. If no error is encountered in examination of the parameters, the IFN and the end number type are placed in word 1 of the POOL Table entry. Control is returned to the calling program.

Usage. The calling sequence is:

```
TSX1     S.SUB1
(Normal Return)
```

Error Return. There is no error return in the calling sequence, but if an error is encountered, one of the following fatal diagnostic messages is written and control is transferred to location LIST04 in the calling program LIST00.

E.ER14	A RIGHT PAREN MUST FOLLOW ________.
E.ER15	ANOTHER PARAMETER IS NEEDED AFTER ________.
E.ER16	COMMA OR RIGHT PAREN MUST FOLLOW ________.

32. NMLS00--NAMELIST Processor

Purpose. This routine processes and compiles NAMELIST statements.

Method. The NAMELIST statement is examined and the required code generated as shown below:

```
                     Generated Code

          ( NAME  BCI     1,NAME       }      1st Entry
          |       BCI     1,VNAME      }
          |       TALLYD  VNAME,D,FT   }
          |       ZERO    SIZE, D1     }      2nd Entry
          |       ZERO    0,D1*D2      }
    Set   {       BCI     1,VNAME2     }
          |       TALLYD  VNAME2,D,FT  }
          |       ZERO    SIZE, D1     }      Additional Entry
          |       ZERO    0,D1*D2      }
          |       --------------------
          (       ZERO    0            }      After Last Entry
```

where:

D = 7777, for single cell variables
= 1, for 1- or 2-dimension variables
= 2, for 3-dimension variables
SIZE = the product of all of the dimensions
FT = 0, not applicable
= 1, integer type
= 2, real type
= 2, double-precision type
= 4, complex type
= 5, logical type
NAME = the name of the NAMELIST
VNAME = the name of the variable

A set of generated coding is produced for each NAMELIST name appearing in a NAMELIST statement. The last word in a set is zero. The first word in a set specifies the NAMELIST name as the location symbol and as the variable field of a BCI instruction. Within a set, there may be one or more entries. Each entry consists of two, three or four words. Only the first two words are used if the variable generating the entry is a single-cell variable. Line 4 is omitted if 1- or 2-dimension variables generate the entry. There will be one entry for each variable in the list.

Usage. The calling sequence is:

```
TSX1     NMLS00
(Normal Return)
```

Error Returns. There are no error returns in the calling sequence but the following fatal diagnostics are written:

E.NM01	ILLEGAL PUNCTUATION.
E.NM02	________ IS AN ILLEGAL SYMBOL.
E.NM03	SLASHES MUST ENCLOSE NAMELIST NAME.
E.NM04	A NAMELIST NAME, ________, HAS BEEN PREVIOUSLY DEFINED.
E.NM05	COMPILER EXPECTS LIST BEFORE END OF STATEMENT.
E.NM06	THE NAMELIST VARIABLE, ________, HAS ADJUSTABLE DIMENSIONS.

33. FRMT00--FORMAT Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a FORMAT statement. The statement will be tested for errors and if none are found, a call of the GG routine will be made to output a card image to the G* file.

Method. The scan of the statement will begin with the first character following the T in FORMAT and terminate at an end mark. The FORMAT EFN is placed in the T.FEFN Table; this table will be utilized when processing the FORMAT reference in an I/O statement to determine whether or not the FORMAT exists.

The scan of the statement essentially consists of transferring the format in BCI into a buffer region of six words, which when full, is placed on the G* file by the GG routine. The location field of the first card image contains the FORMAT EFN which is suffixed with the period character. Blanks, not in alphanumeric fields, are deleted from the text.

Example:

```
28    FORMAT (12Abb6)  FORTRAN source card

28.   BCI 1,(12A6)     GMAP card image
```

Usage. The calling sequence is:

```
TSX1     FRMT00
(Normal Return)
```

Error Returns. There are no error returns but the following error conditions stop further scanning of the statement and result in a diagnostic message being given:

E.FM02	NO OPENING LEFT PAREN IN FORMAT.
E.FM03	ILLEGAL FORTRAN CHARACTER.

E.FM04	ILLEGAL FIELD CONVERSION CHARACTER.
E.FM05	THE CHARACTER X IN OR PRECEDING ________ IS WRONG.
E.FM06	AN H FIELD IS TOO WIDE.
E.FM07	END OF STATEMENT FOUND BEFORE H FIELD EXHAUSTED.
E.FM10	ERROR IN A FW.D TYPE FIELD.
E.FM11	END OF FORMAT DOES NOT FOLLOW PAREN BALANCE.
E.FM12	RIGHT AND LEFT PARENS DO NOT BALANCE.

The following errors will interrupt the scan to give a diagnostic message. The scan then continues in the normal manner:

E.FM01	NO EXTERNAL FORMULA NUMBER ON FORMAT.
E.FM08	ILLEGAL CHARACTER IN H FIELD. excluded are ! and
E.FM09	MORE THAN 3 NESTED LEFT PARENS.

34. FG0000--FORMAT Generator Processor

Purpose. This routine processes FORMAT Generator statements, produces the necessary FORMAT statements and compiles the FORMAT statements into GMAP coding.

Method. Statements are moved to the SS-Region for scanning. (Since cards of this type represent a special case, the collecting of cards normally done by the routine DC0100 is done directly by this subroutine.) FORMAT statements are generated from parameter cards and control-type cards (SPACE, REPEAT, RESTORE and END OF FORMAT) are processed.

Usage. The calling sequence is:

```
TSX1     FG0000
(Normal Return)
```

Error Return. There are no error returns in the calling sequence but the following fatal diagnostics are written:

FG0301	ILLEGAL CONVERSION CHARACTER IN FORMAT.
FG0311	ERROR IN LOCATING TABLE AT LOC ________.
FG0316	NO EXTERNAL FORMULA NUMBER ON FORMAT.
FG0319	IN REPEAT N, THE N IS MISSING.
FG0321	IN REPEAT N,A THE A CONTAINS AN ILLEGAL CHARACTER.

35. SEQ000--Sequence Error Routine

Purpose. This subroutine writes a fatal diagnostic error message when statements of the source program are out of sequence.

Method. If a TYPE, COMMON, DIMENSION, EQUIVALENCE, EXTERNAL, FUNCTION, SUBROUTINE or BLOCK DATA statement occurs after the first EXECUTABLE (or DATA and/or NAMELIST) statement of the FORTRAN source program, control is transferred to this subroutine. An error message is written and control is returned to the calling routine.

Usage. The calling sequence is:

```
TSX1     SEQ000
(Normal Return)
```

Error Returns. There are no error returns, but the following fatal diagnostic message is written:

SEQ090	STATEMENT IS OUT OF SEQUENCE, PLACE BEFORE EXECUTABLE, DATA AND NAMELIST STATEMENTS.

36. CMN100--COMMON Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a COMMON statement. The statement will be tested for errors and, if none are found, the statement will be broken down into various table entries.

Method. This routine will scan the source statement commencing with the first character following the N in COMMON and terminating at an end mark. The BCD text is made available via the Scanning routines. If any variables are dimensioned, the COMMON Processor will call the Dimension Subscript Processor routine to process the actual array dimensions. As the scan progresses, a check for errors is made and appropriate table entries are made.

Variable names are added to the NAME Table (if not already there) with the Type flag (I.REL or I.ITG) set ON implicitly. For each variable in the statement, a common flag, I.COM, is set ON. If the variable is in BLANK COMMON, the I.BCM flag is set ON. If the variable is dimensioned, the I.DIM flag is set ON and a pointer to the T.DIME Table entry is entered in the NAME Table flag word (0-17).

Entries are made in the T.DIME Table for each variable that is dimensioned. The subscripts defining the dimension of an array are restricted to unsigned integer constants.

The COMMON block name is isolated or if no block name exists, COMMON is established to be BLANK COMMON. The block name in BCI (//bbbb for BLANK COMMON) is inserted in the T.COMO Table. For each COMMON variable, the NAME Table pointer is entered in a separate, successive word of the table.

Table Entries

T.COMO Table

0 2	3 17	18 20	21 35
	BLOCK NAME (BCI)		
0	NAMEP (var 1)	0	00000
0	NAMEP (var 2)	0	00000
	.		.
	.		.
	.		.
4	NAMEP (var n)	0	00000

<u>Usage.</u> The calling sequence is:

```
TSX1     CMN100
(Error Return)
(Normal Return)
```

<u>Error Returns.</u> The following error conditions stop further scanning of the statement and result in a diagnostic message being given:

CMNER1	THE VARIABLE NAME ________ BEGINS WITH A NUMERIC CHARACTER.
CMNER2	Illegal punctuation.*

The following errors will interrupt the scan to give a diagnostic message. The scan then continues in the normal manner:

CMNER5	THE VARIABLE NAME ________ HAS PREVIOUSLY APPEARED AS A SUBPROGRAM NAME.
CMNER6	THE VARIABLE NAME ________ HAS PREVIOUSLY APPEARED IN COMMON.
CMNER8	THE VARIABLE NAME ________ HAS PREVIOUSLY BEEN DIMENSIONED.
SEQ090	STATEMENT IS OUT OF SEQUENCE. PLACE BEFORE EXECUTABLE, DATA AND NAMELIST STATEMENTS.

The following errors cause a warning message to be given:

CMNER3	THE COMMON BLOCK NAME/________/HAS NO VARIABLE NAMES. THE BLOCK NAME WAS IGNORED. WARNING ONLY.

CMNER7 ONE OR MORE REDUNDANT COMMAS EXIST. WARNING ONLY.

*One of the following diagnostics will be given when illegal punctuation is encountered.

S.PC30 ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL _________.

S.PC35 THE PUNCTUATION MARK _________ WAS USED INCORRECTLY.

S.PC40 MISSING PUNCTUATION AT THE END OF STATEMENT.

37. EQV100--EQUIVALENCE Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as an EQUIVALENCE statement. The statement will be tested for errors and, if none are found, the statement will be broken down into various table entries.

Method. This routine will scan the source statement commencing with first character following the final E in EQUIVALENCE and terminating at an end mark. The BCD text is made available via the Scanning routines. As the scan progresses, a check for errors is made and appropriate table entries are made.

The variable name being equivalenced is added to the NAME Table (if not already there) with the Type flag (I.REL or I.ITG) set ON implicitly. If the Equivalence flag, I.EQV, is OFF, it is set ON. If the Equivalence flag is ON, the Repeatedly Equivalenced flag, I.EQR, is set ON.

Entries are made in the T.EQIV Table for each variable and its associated subscripts. The T.EQIV prototype is:

0 2	3 17	18 20	21 35
N	NAMEP	0	Subscript-1_1
.	.	.	.
.	.	.	.
.	.	.	.
0	Subscript-1_{n-1}		Subscript-1_n

Where: N = 2 for each variable except last
N = 6 for last variable of each group

Usage. The calling sequence is:

```
TSX1     EQV100
(Error Return)
(Normal Return)
```

Error Returns. The following error conditions stop further scanning of the statement and result in a diagnostic message being given.

EQVER1	AN EQUIVALENCE GROUP WITH NO LEFT PARENTHESIS WAS ENCOUNTERED.
EQVER2	THE VARIABLE NAME ________ BEGINS WITH A NUMERIC CHARACTER.
EQVER7	Illegal punctuation.*

The following errors will interrupt the scan to give a diagnostic message; the scan then continues in the normal manner:

EQVER3	A VARIABLE, ________ HAS APPEARED AS A SUBPROGRAM NAME OR ARGUMENT.
EQVER4	SUBSCRIPT IN EQUIVALENCE BEGINS WITH ALPHABETIC CHARACTER, ________ .
EQVER5	SUBSCRIPT IS LARGER THAN 17 BITS.
EQVER6	TOO MANY SUBSCRIPTS ON ONE OR MORE VARIABLES.
EQVER8	ONLY 1 VARIABLE EXISTS IN EQUIVALENCE GROUP.
SEQ090	STATEMENT IS OUT OF SEQUENCE. PLACE BEFORE EXECUTABLE, DATA AND NAMELIST STATEMENTS.

The following error causes only a warning message:

EQVER9	COMMA MISSING BETWEEN EQUIVALENCE GROUPS. WARNING ONLY.

*One of the following diagnostics will be given when illegal punctuation is encountered:

S.PC30	ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL ________.
S.PC35	THE PUNCTUATION MARK ________ WAS USED INCORRECTLY.
S.PC40	MISSING PUNCTUATION AT THE END OF STATEMENT.

38. DIM100--DIMENSION Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a DIMENSION statement. The statement will be tested for errors and, if none are found, the statement will be broken down into various table entries.

Method. This routine will scan the source statement commencing with the first character following the final N in DIMENSION and terminating at an end mark. The BCD text is made available via the Scanning routines. The DIMENSION Processor uses the DIMENSION Subscript Processor routine to process the actual array dimensions. As the scan progresses, a check for errors is made and appropriate table entries are made.

The variable being dimensioned is added to the NAME Table (if not already there) with the Type flag (I.REL or I.ITG) set ON implicitly. A pointer to the T.DIME Table entry is set in the flag word of the NAME Table entry (0-17).

Entries are made in the T.DIME Table for each variable and its associated subscripts. The T.DIME prototype is:

0 2	3 17	18 20	21 35
N	NAMEP	f	DIM_1
.	.	.	.
.	.	.	.
.	.	.	.
f	DIM_{n-1}	f	DIM_n

Where:

N = Dimensionality
f = 0 if dimension is a variable name pointer
f = 4 if dimension is a constant

Usage. The calling sequence is:

```
TSX1     DIM100
(Error Return)
(Normal Return)
```

Error Returns. The following error conditions stop further scanning of the statement and result in a diagnostic message being given:

DIMER1	THE VARIABLE NAME ________ BEGINS WITH A NUMERIC CHARACTER.
DIMER2	Illegal punctuation.*

The following errors will interrupt the scan to give a diagnostic message. The scan then continues in the normal manner:

DIMER3	THE VARIABLE NAME ________ HAS ALREADY APPEARED AS AN ARRAY NAME.
DIMER4	THE VARIABLE NAME ________ HAS PREVIOUSLY APPEARED AS A FUNCTION OR SUBROUTINE NAME.
SEQ090	STATEMENT IS OUT OF SEQUENCE. PLACE BEFORE EXECUTABLE, DATA AND NAMELIST STATEMENTS.

*One of the following diagnostics will be given when illegal punctuation is encountered.

S.PC30	ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL ________.

S.PC35	THE PUNCTUATION MARK________WAS USED INCORRECTLY.
S.PC40	MISSING PUNCTUATION AT THE END OF STATEMENT.

39. TP1000--TYPE Statement Processor

Purpose. This subroutine processes and compiles the TYPE statement.

Method. The next BCI group is obtained via a call to the S.C000 routine. Upon return, numerics and punctuation are treated as errors. Alphabetics are checked to be sure the name is entered in the NAME Table. The type code (REAL, INTEGER, DOUBLE PRECISION, COMPLEX or LOGICAL) is obtained from the cell .TYPE. (lower 3 bits) and the type in the NAME Table is set accordingly. The remainder of the statement is scanned until the termination character is recognized.

Usage. The calling sequence is:

```
TSX1     TP1000
(Normal Return)
```

Error Returns. There are no error returns. The following fatal diagnostics are written:

TP1046	THE SYMBOL________APPEARS IN SEVERAL TYPE STATEMENTS.
TP1050	COMMA OR END EXPECTED INSTEAD OF__________.
TP1054	__________HAS BEEN PREVIOUSLY DIMENSIONED.
TP1060	__________IS A SUBROUTINE NAME.
TP1066	ILLEGAL USE OF PUNCTUATION OR NUMERIC CHARACTER.

40. S.SCRP--Dimension Subscript Processor

Purpose. This routine is called by the Type, COMMON and DIMENSION Statement Processors to scan the subscript combinations that exist within these statements. The subscript will be tested for errors and, if none are found, a T.DIME Table entry will be made.

Method. This routine will scan the BCD text beginning with the first character following a left parenthesis and ending in a right parenthesis. The BCD text is made available via the Scanning routines. As the scan progresses, a check for errors is made and appropriate table entries are made.

The NAME Table pointer for the array name is entered in the T.DIME Table. If the dimension size is a constant, it is converted to a binary integer and entered, with an appropriate flag, in the T.DIME Table. If the dimension size is specified by a variable, the following takes place:

1. The adjustable dimension flag, I.ADM, is set ON in the NAME Table flag word.
2. The NAME Table pointer of the variable dimension is entered in the T.DIME Table.

Variable dimensions are not allowed in a COMMON statement. The flag cell F.COMO is set when a COMMON statement is processed and, if on when the Subscript Processor encounters an adjustable dimension, a diagnostic message will be given.

Usage. The calling sequence is:

```
TSX1     S.SCRP
(Error Return)
(Normal Return)
```

Error Returns. The following error conditions stop further scanning of the statement and result in a diagnostic message being given:

S.PC30	ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL ________.
S.PC35	THE PUNCTUATION MARK ________ WAS USED INCORRECTLY.
S.PC40	MISSING PUNCTUATION AT THE END OF A STATEMENT.

The following errors will interrupt the scan to give a diagnostic message; the scan then continues in the normal manner:

SCRER1	THE COMMON ARRAY ________ HAS VARIABLE DIMENSIONS.
SCRER2	THE ARRAY NAME ________ MUST BE AN ARGUMENT IN THE FUNCTION/ SUBROUTINE STATEMENT.
SCRER3	THE ADJUSTABLE DIMENSION ________ MUST BE AN ARGUMENT IN THE FUNCTION/SUBROUTINE STATEMENT.
SCRER4	THE ADJUSTABLE DIMENSION ________ IS NOT AN INTEGER TYPE.
SCRER5	THE ARRAY ________ HAS TOO MANY DIMENSIONS.
SCRER6	THE ARRAY ________ HAS A DIMENSION OF ZERO.
SCRER7	SUBSCRIPT IS LARGER THAN 17 BITS.

41. XTN100--EXTERNAL Statement Processor

Purpose. This subroutine processes the EXTERNAL statement.

Method. The statement is scanned, symbols are checked for inconsistent usage and if none, the external flag is set. A call is made to the GG routine to generate the appropriate code.

Usage. The calling sequence is:

```
TSX1    XTN100
(Normal Return)
```

Error Returns. There are no error returns. The following fatal diagnostic is written:

XTN138 __________ WAS PREVIOUSLY USED IN A CONFLICTING WAY.

42. FNC100--FUNCTION Statement Processor
SBR100--SUBROUTINE Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as a FUNCTION or SUBROUTINE statement. This routine also processes FUNCTION statements which are Typed (REAL FUNCTION, INTEGER FUNCTION, LOGICAL FUNCTION, COMPLEX FUNCTION and DOUBLE PRECISION FUNCTION). Testing for syntax errors in the source statement will be done and, if none are found, the statement will be broken down into various table entries. If a Type FUNCTION is being processed, the cell, .TYPE., will contain the appropriate type flag.

Method. This routine will scan the source statement commencing with the first character following the final N in FUNCTION or the final E in SUBROUTINE, and terminating at an end mark. The BCD text is made available via the Scanning routines. As the scan progresses, a check for errors is made and appropriate table entries are made.

The Subprogram name is entered in the NAME Table with the Function Name flag, I.FNM, set ON. If it is a Type FUNCTION statement, the Explicit flag, I.EXP, and the appropriate Type flag are set ON; otherwise the Type flag, I.REL or I.ITG, is set ON by the implicit rule. Any arguments found are entered in the NAME Table with a Type flag set by the implicit rule. The Program Argument flag, I.ARG is also set ON.

When there are arguments to the subprogram, an entry is made in the T.ARGS Table. In that these arguments are dummy arguments, the T.ARGS Table will be used to generate the program prologue. The number of asterisks (*) that are used to indicate nonstandard returns is counted for reference later when compiling RETURN statements.

Table Entries

0 17	18 35
Arg. Count	NAMEP_1
NAMEP_2	NAMEP_3
.	.
.	.
.	.
NAMEP_{n-1}	NAMEP_n

The BCD name of the FUNCTION or SUBROUTINE is placed in the cell PR.NAM as the source program name. The cell PR.TYP which was initialized to zero is set with the NAMEP (Name Table pointer) of the subprogram name in bits 18-35. If the subprogram is a FUNCTION subprogram, a 1 is placed in bit position 0.

Usage. The calling sequences are:

```
        TSX1    FNC100   process a FUNCTION statement
   or   TSX1    SBR100   process a SUBROUTINE statement
        (Error Return)
        (Normal Return)
```

Error Returns. When an error is encountered, scanning of the statement is terminated and the error return is taken. The following diagnostic messages exist.

FNCER1	MORE THAN ONE FUNCTION/SUBROUTINE STATEMENT APPEARS IN PROGRAM.
FNCER2	THE FUNCTION/SUBROUTINE NAME BEGINS WITH A NUMERIC CHARACTER.
FNCER3	Illegal punctuation.*
FNCER4	THE FUNCTION/SUBROUTINE STATEMENT IS NOT FIRST IN THE PROGRAM.
FNCER5	A FUNCTION STATEMENT MUST HAVE AN ARGUMENT LIST.
FNCER6 S.VAER	THE VARIABLE NAME ________ BEGINS WITH A NUMERIC CHARACTER.
FNCER7 S.PCER	Illegal punctuation.*
FNCER8	AN ARGUMENT ________ APPEARS PREVIOUSLY IN THE PROGRAM OR IN AN ARGUMENT LIST.

*One of the following diagnostics will be given when illegal punctuation is encountered.

S.PC30	ILLEGAL OR MISSING PUNCTUATION AFTER OR NEAR THE SYMBOL ________.
S.PC35	THE PUNCTUATION MARK ________ WAS USED INCORRECTLY.
S.PC40	MISSING PUNCTUATION AT END OF STATEMENT.

43. BLD100--BLOCK DATA Statement Processor

Purpose. This routine is called after the Dictionary Scan has determined that this BLOCK DATA statement is the first statement of the program.

Method. Processing consists of turning on a switch, F.BLOC, to indicate that the program is a BLOCK DATA subprogram. The cell PR.TYP (bit 2) is set to 1. Control is then returned to the calling program.

The status of the F.BLOC switch is recognized by the Dictionary Scan in Phase One as allowing only the following statements in a BLOCK DATA subprogram: COMMON, EQUIVALENCE, DIMENSION, DATA, END and TYPE. The appearance of any others will cause a diagnostic message. The F.BLOC switch is initialized to off by the Phase One Initialization section. It is located in the Phase One COMMON block.

In the Storage Allocator section of Phase One, the status of the PR.TYP cell is tested as follows:

1. PR.TYP cell (0-2) is zero, DATA variables may not be in any COMMON.
2. PR.TYP cell (bit 2) is a one, DATA variables _must_ be in labeled COMMON.

Phase Two is called to complete the compilation of DATA.

Usage. The calling sequence is:

```
TSX1     BLD100
(Normal Return)
```

Error Returns. None.

44. SA0100--Storage Allocator for Blank and Block Common

Purpose. This subroutine and the subroutine SA0200 process the COMMON Tables and compile the required code.

Method. The COMMON Table (T.COMO) contains the information to be processed. An entry in this table consists of block names in BCI form and pointers to the variable list. The pointer is contained in bits 3-17; the last pointer in a variable list is prefixed by a bit in position 0. The block name and size is entered in the T.BLOC Table. Equivalenced variable names with the T.BLOC pointer and the relative T.BLOC pointer are entered in the T.EQCO Table. If the fatal diagnostic flag is OFF, the appropriate instructions are generated.

The list of variables in the COMMON statement are processed by the routine SA0200 which is a separate portion of this routine. Each variable name is checked for legality. An entry is made in the T.EQCO Table if the variable is equivalenced. The size of the variable is computed and the appropriate instructions are compiled.

Sample COMMON Statements

```
7
 COMMON  SHRT1, SHRT2, SHRT3, VFORM, DIAG
```

The above statement specifies that the list of variables shall reside in BLANK COMMON. The variables are either nondimensioned or dimensioned as follows:

```
7
 DIMENSION  SHRT3 (2,2,2), VFORM (20)
 DOUBLE PRECISION  SHRT1 (2,2)
 COMPLEX  SHRT2 (3)
 INTEGER  DIAG
```

The generated code for the list of variables in the BLANK COMMON statement above would be:

```
1       8       16
        BLOCK
 SHRT1  BSS     8
 SHRT2  BSS     6
 SHRT3  BSS     8
 VFORM  BSS     20
 DIAG   BSS     1
```

A BLOCK COMMON statement might appear as follows:

```
7
 COMMON  /BLDAT/  CPVAR, XA,  SSVAR(4)
```

The name of this BLOCK COMMON area is BLDAT. The variables in the list are dimensioned as follows:

CPVAR(3,2) - Complex
XA(3) - Double precision
SSVAR(4) - Real

The generated code would appear as follows:

1	8	16
	BLOCK	BLDAT
CPVAR	BSS	12
XA	BSS	6
SSVAR	BSS	4

Usage. The calling sequence is:

```
TSX1     SA0100
(Normal Return)
```

Error Returns. There are no error returns in the calling sequence. The following fatal diagnostic messages are written:

SA0183 // COMMON ASSIGNMENT ILLEGAL IN BLOCK DATA SUBPROGRAM.

The following messages are written by the subroutine SA0200:

SA0276 ILLEGAL VARIABLE TYPE IN COMMON.

SA0279 ODD LOCATION FOR DP OR COMPX IN COMMON.

45. SA1010--Storage Allocator for Equivalenced Variables

Purpose. This subroutine processes EQUIVALENCE Tables and generates the required coding. The subroutines SA2010, SA3010 and SA4010 are component parts of this subroutine.

Method. The EQUIVALENCE statement has the general form:

7
EQUIVALENCE (a,b,c), (d,e,f),......

The list of variables within a set of parentheses, for example (a,b,c) is called a group. The EQUIVALENCE Table, T.EQIV is examined by groups. During this scan, another table, T.LIST, is constructed. The T.LIST Table contains entries for those variables which appear in more than one equivalence group. An entry is composed of a T.EQIV Table pointer, the NAME Table pointers and an addend locating the variables within their arrays.

EQUIVALENCE (A,B,C), (A,X,Y), (B(2),Q,W)

A = B(1) = C = X = Y
B(2) = Q = W

T.LIST Table

Entry 1	T.EQIV Table Pointer to first group
	T.NAME Table Pointer to variable A
	T.NAME Table Pointer to variable B
	Addend for A = 0
	Addend for B = 0
Entry 2	T.EQIV Table Pointer to second group
	T.NAME Table Pointer to variable A
	Addend for A = 0
Entry 3	T.EQIV Table Pointer to third group
	T.NAME Table Pointer to variable B
	Addend for B = 1

The T.LIST Table is used to construct another table, T.REQU, which is a reordered or reorganized T.EQIV Table. All groups which can be linked by a variable appearing in both groups are consolidated into a single group. During this process, redundant and inconsistent equivalences are detected. The redundancies produce warning diagnostic messages; the inconsistencies produce fatal diagnostic messages. While the T.REQU Table is being constructed, the T.BASE Table is also being formed. For each group, a base variable and addend are stored. The base variable may be:

- A COMMON variable which appears in the group
- The last variable of the group

A check is made for redundancies and/or inconsistencies during creation of the T.BASE Table. Finally, using the T.BASE and T.REQU Table information, the necessary code is generated for storage assignment.

Sample EQUIVALENCE Statements

```
7
COMMON CPVAR
COMPLEX CPVAR(3,2)
EQUIVALENCE (EDATA,CPVAR)
```

Generated code: (for EQUIVALENCE only)

```
1      8     16
EDATA  EQU   CPVAR
```

The entire code for the above 4 statements would be:

```
1      8      16
       BLOCK
CPVAR  BSS    12
EDATA  EQU    CPVAR
```

Given this source card sequence,

```
      7
      DIMENSION  EREAD(48)
      EQUIVALENCE  (EREAD, CPVRX), (EREAD(13),XAX)
```

this code will result:

```
1      8      16
EREAD  BSS    48
CPVRX  EQU    EREAD
XAX    EQU    EREAD+12
```

Usage. The calling sequence is:

```
TSX1     SA1010
(Normal Return)
```

Error Returns. There are no error returns in the calling sequence. The following diagnostics are written:

SA1082 ________ INCONSISTENCY IN EQUIVALENCE.

*SA1084 ________ REDUNDANCY IN EQUIVALENCE. WARNING ONLY.

*A warning message. Compilation continues.

The following messages originate in routine SA3010:

SA3062 VARIABLE ______ OF ADJUSTABLE DIMENSIONS MAY NOT BE EQUIVALENCED.

SA3065 A SINGLE VARIABLE ________ MAY NOT BE REFERENCED AS AN ARRAY IN EQUIVALENCE.

SA3068 EQUIVALENCE AND DIMENSION STATEMENTS DO NOT AGREE FOR ________.

The following messages originate in routine SA4010:

SA4521 ________ AND ________ IN DIFFERENT COMMON BLOCKS ILLEGALLY EQUIVALENCED.

SA4531 __________ AND __________ IN SAME COMMON BLOCK INCONSISTENTLY EQUIVALENCED.

*SA4531 __________ AND __________ IN SAME COMMON BLOCK REDUNDANTLY EQUIVALENCED. WARNING ONLY.

SA4551 NONCOMMON __________ ILLEGAL FOR EQUIVALENCE IN BLOCK DATA SUBPROGRAM.

SA4561 __________ EXTENDS COMMON BLOCK BEYOND ORIGIN.

SA4571 __________ (DP OR CX) REQUIRES EVEN ADDEND WITH BASE.

46. ENTY00--ENTRY Statement Processor

Purpose. This routine is called after the Dictionary Scan has classified the source statement as an ENTRY statement.

Method. If there is an argument list with the ENTRY statement, an entry is made in the T.ARGS Table.

OHOH! It looks like the entry names are put into the T.NAME table BUT is the I.ARG flag (bit 23) set ON? and is a check for I.EQV (bit 32) set ON made?

Table Entries

T.ARGS Table

0 5	6 17	18 35
X	No. of Arguments	P(T.NAME)Arg 1
P(T.NAME)Arg 2		P(T.NAME)Arg 3
etc.		etc.

where X = the number of this argument list. This number is increased by 1 for each ENTRY statement.

Entries are also made in two other tables.

Table Entries

T.ENTY Table

0 5	6 17	18 35
X	0	IFN

where X = the number of this entry.
IFN = the IFN of the next statement.

T.JUMP Table

0 5	6 17	18 35
5	IFN	IFN

Usage. The calling sequence is:

```
TSX1     ENTY00
(Normal Return)
```

Error Returns. There are no error returns in the calling sequence but the following fatal diagnostic messages are written:

ENTYM1	ENTRY IS MISPLACED.
ENTYM2	ENTRY IS WITHIN A DO.
ENTYM3	ENTRY NAME BEGINS WITH A NUMERIC.
ENTYM4	ENTRY NAME IS MISSING.
ENTYM5	ENTRY NAME USED PREVIOUSLY.
ENTYM7	FUNCTION ENTRY MUST HAVE AN ARGUMENT LIST.

Does not catch an argument ~~that~~ appears in an EQUIVALENCE ~~statement~~ (illegal (see Rule #5 for Arguments pg 44 of CPB1006,

4. PHASE TWO

Phase Two of the FORTRAN IV Compiler uses the information contained in the Buffered, POOL and NAME Tables and generates the required GMAP code. The table information is the actual source program statements in a form which can be processed by the compiler.

After Phase One has been completed, control is returned to the Executive Phase for loading of Phase Two. When the two assemblies which comprise Phase Two are loaded, control is transferred to the Phase Two Executive Routine, PH2000. This Executive Routine controls the processing of the table information through calls to various subroutines.

Initially, the Phase Two Executive Routine checks the source program flow to ensure that every statement can be reached at execution time. This verification is performed by comparing the IFN+1 of each transfer-type statement in the program with a table of destination IFNs for each transfer-type statement in the program. If there is no match, an inaccessible statement is indicated and the compiler writes a diagnostic message indicating the nearest EFN.

A second function is performed as GMAP coding is generated for DATA statements contained in the source program. The subroutine CK.DOS is called to check the legality of transfers into the range of DO statements. An error sensed during this check causes a diagnostic message and control is returned to the Executive Phase.

The required initializations are performed and the main compiler begins processing each statement as it is retrieved from the POOL Table. If a statement is a comment, it is simply written on the G* file and the next statement is obtained. Noncomment statements are tested for the following:

1. Beginning of a DO loop?
2. An END statement?
3. A DO-ending?
4. Beginning of a Basic Block?

The DO Indexer routine analyzes and generates all coding for DO statements. As each DO statement is encountered, whether it occurs singly or in a nest, the DO indexer generates the required coding for the beginning and end of each DO statement. The instructions generated by the beginning of the outermost DO statement are written directly on the G* file. The coding for the end of a DO statement is written into the T.COLT Table for use later. In the case of nested DO statements, the coding for both the beginning and the end of a DO statement is also written into the T.COLT Table. All subscripted variables in the DO range are noted and coding is generated for the initialization and incrementation of the indexes. It should be noted that the DO Indexer does not return control to the main compiler until the entire range of a DO statement has been processed. If intermediate statements occur within the range of a DO, they are not processed until all of the DO statement is finished. When the DO statement, or nest of DO statements, is completed, the main compiler will process the intermediate statements and merge the required DO statement instructions from the T.COLT Table into the program.

The END statement causes control to pass to the final section of the main compiler.

The Basic Block Indexer is called to examine a Basic Block and to generate the necessary indexing instructions. A Basic Block is defined as a linear stretch of code having only one entry and one exit and not within the range of a DO loop. If the indexer finds subscripts in the Basic Block during its examination, the generation of indexing instructions is required; the absence of subscripts reduces the function of the indexer to a simple process.

All statements other than DO, DO-ending or END are processed by the main compiler. In most cases individual subroutines will be called as required to process the table information which constitutes the source statement. The GMAP Code Generator Routine (GG) is frequently called to build and output lines of GMAP coding.

Finally the prologue logic of necessary save and initialization instructions is compiled including the erasable storage to be used at execution time. The linkages will be restored and control is returned to location PE.070 in the Executive Phase. The following figure shows the flow of Phase Two of the FORTRAN IV Compiler.

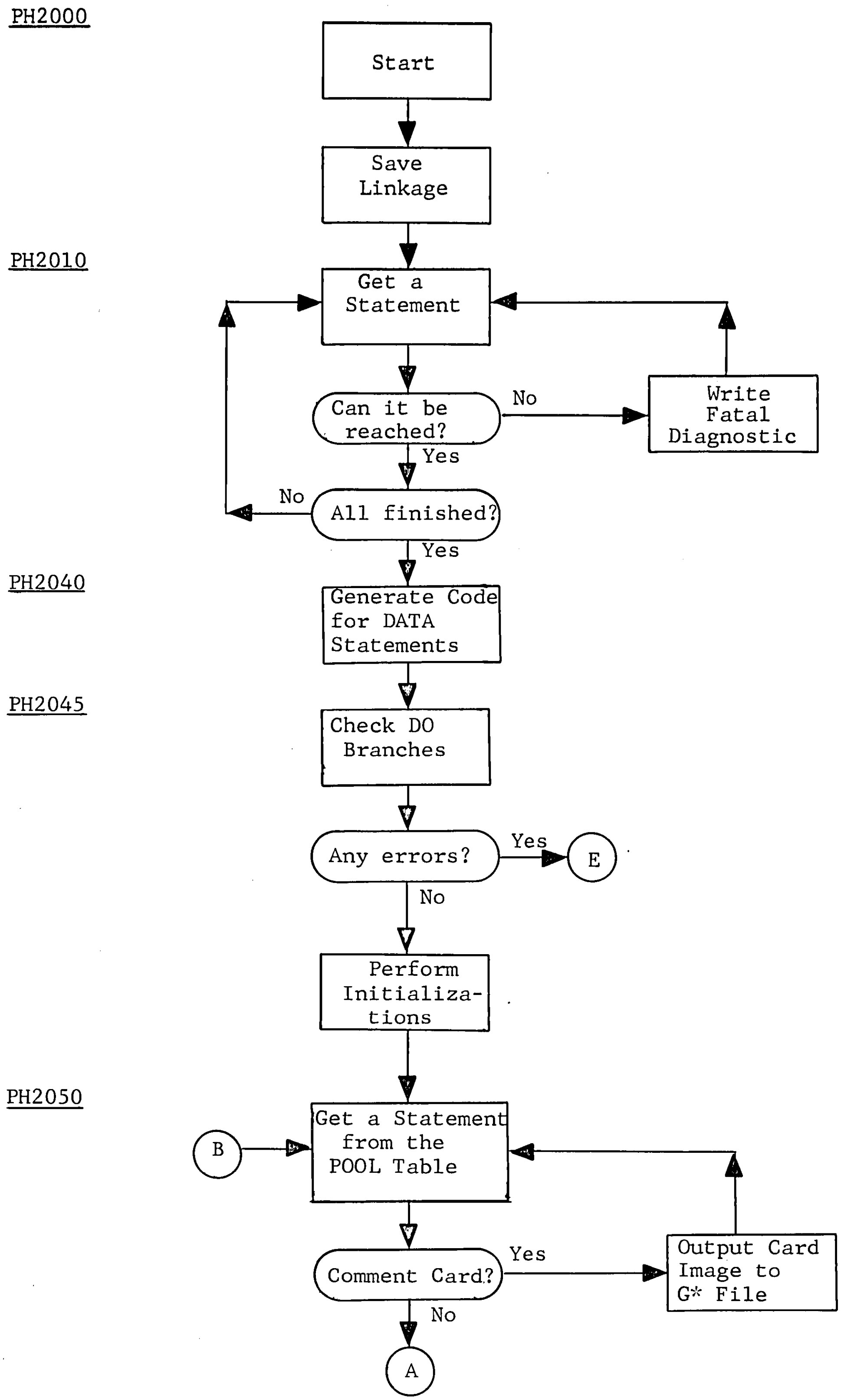


Figure 8. Phase Two Flow Diagram

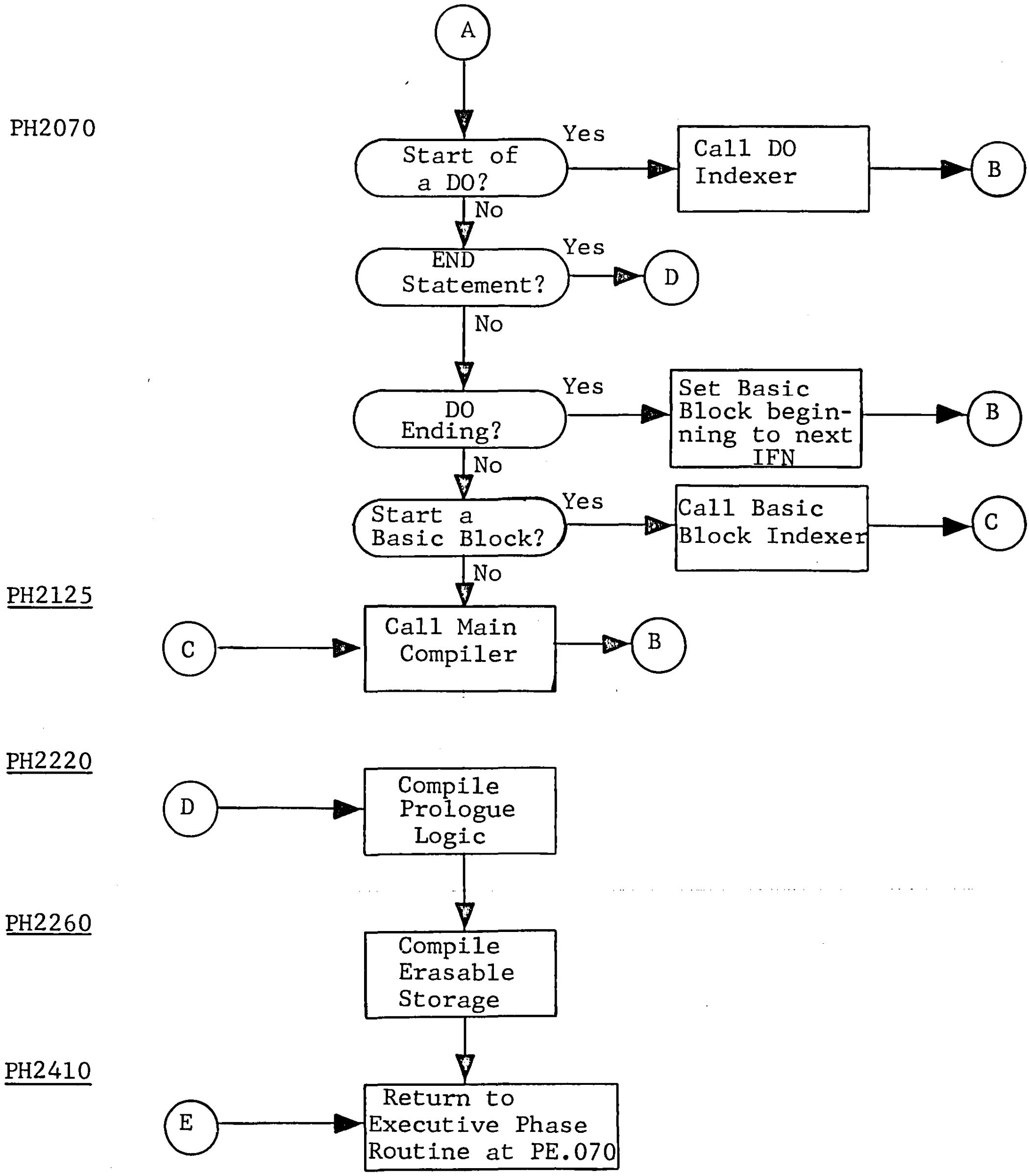


Figure 8. (continued)

1. S.TPOO--POOL Table Routine

Purpose. This subroutine retrieves entries from the POOL Table.

Method. This routine is described in Chapter 1 of this manual.

Usage. The calling sequence is:

```
TSX1     S.TPOO
(Normal Return)
```

Upon return the tally word (TA,CT) is in the A-register.

Error Returns. If an error is encountered in the POOL Table, return is made directly to location PE.070 in the Executive Phase. The following fatal diagnostics are written:

S.TP61 UNEXPECTED EOF READING T.POOL TABLE.

S.TP63 T.POOL TABLE SEQUENCE ERROR.

2. CK.EFN--EFN Check Routine

Purpose. This routine checks the EFN on a FORMAT statement to see if the same EFN appears on an executable statement.

Method. Entries from the T.FEFN Table are compared with entries from the T.EIFN Table. If the same EFN appears on an executable statement and a FORMAT statement, a warning diagnostic message is written.

Usage. The calling sequence is:

```
TSX1     CK.EFN
(Normal Return)
```

Error Return. A warning diagnostic message is written.

CK.F40 THE EFN__________ APPEARS ON A FORMAT STATEMENT AND AN EXECUTABLE STATEMENT.

3. S.PROL--Prologue Compile Routine

Purpose. This subroutine compiles the prologue instructions for the source program.

Method. The subroutine is entered once for each FUNCTION, SUBROUTINE and ENTRY statement with arguments in the T.ARGS Table. The Argument Name Pointer is in index register 4. The subroutine matches the AN pointer with T.NAME Table pointers retrieved from the T.PROL Table. Instructions are compiled as indicated by the T.PROL entry.

Usage. The calling sequence is:

```
TSX1     S.PROL
(Normal Return)
```

Error Returns. None.

4. PH2BGN--Search, Match, Find and Merge Routine

Purpose. This subroutine is used to determine if coding is to be merged into the G* file; the coding to be merged is also located by this routine in the T.COLT Table.

Method. The subroutine searches the T.BGIN Table for an IFN equal to the current IFN. If a match is found, the T.COLT Table is searched to find a block of coding to be merged into the GMAP coding being generated.

Usage. The calling sequence is:

```
TSX1     PH2BGN
(Normal Return)
```

Error Returns. None.

5. PH2KIL--Table Kill Routine

Purpose. This subroutine releases Buffered Tables.

Method. This subroutine calls the Buffered Table Kill Routine, S.TK00, to kill the T.BGIN, T.OUTS, T.LDXR and T.COLT Buffered Tables.

Usage. The calling sequence is:

```
TSX1     PH2KIL
(Normal Return)
```

Error Returns. None.

6. MC2000--Phase Two Main Compile Routine

Purpose. This subroutine examines the type number of a POOL Table entry and transfers control to the appropriate routine which will compile the executable statement.

Method. The POOL Table entry is obtained; all information is masked out except the type number which is placed in the upper Q-register. Control is then given to a table of transfer instructions; one for each type of statement. The appropriate transfer instruction in the table is selected by the type number in QU modifying the basic address of the transfer table. For executable statements, control will be given to the appropriate statement processor. For nonexecutable statements, control returns normally to the calling program.

Usage. The calling sequence is:

```
TSX1     MC2000
TALLY    L(T.POOL) ENTRY, WORD COUNT
ZERO     L(IFN)
(Normal Return)
```

Error Returns. None.

7. IFBN00--Operation Compile Routine

Purpose. This subroutine compiles a transfer-type instruction with an address of an IFN or a B.N where N is an integer.

Method. This subroutine is called when compiling instructions for an Arithmetic or Logical IF, a Computed GO TO or for an Unconditional GO TO statement. The T.OUTS Table is examined. If there is an entry in the T.OUTS Table corresponding to the transfer-type instruction to be compiled, the B.N symbol will be used for the address. If there is not an entry in the T.OUTS Table, the IFN will be used. Entries in the T.OUTS Table are dependent upon entries in and out of DO loops.

Usage. Upon entry to this routine the T.JUMP pointer is contained in the A-register. The calling sequence is:

```
TSX1     IFBN00
ZERO     L(Operation in BCD)
(Normal Return)
```

Error Returns. None.

8. BN.000--T.JUMP Pointer Check Routine

Purpose. This subroutine checks the T.OUTS Table to see if the T.JUMP pointer is there.

Method. Upon entering this routine the T.JUMP pointer is contained in the A-register. Consecutive entries are obtained from the T.OUTS Table and compared. If the T.JUMP pointer is found, return is made with a bit in position zero of the Q-register, N of B.N is in QL and the IFN is in QU. If the T.JUMP pointer is not found, return is made with only the IFN in the upper half of the Q-register.

Usage. The calling sequence is:

```
TSX1     BN.000
(Normal Return)
```

Error Return. None.

9. BCD200--BCD Comment Routine

Purpose. This subroutine is called to process a BCD comment from the POOL Table.

Method. The GG routine is called to generate the coding to assemble the BCD comment as a GMAP card image. Comment cards of less than 72 characters are appended with blanks. All cards are appended with blanks in columns 72-83 and an asterisk is placed in column 84.

Usage. The calling sequence is:

```
TSX1     BCD200
(Normal Return)
```

Error Returns. None.

10. IFA200--Arithmetic IF Routine

Purpose. This subroutine compiles the necessary transfer instructions for an Arithmetic IF statement after the coding for the arithmetic expression has been generated.

Method. The general form of the IF statement is:

```
IF(---) M1, M2, M3
```

The expression within the parentheses is compiled by the Arithmetic Processor, ARCODE. The symbols M1, M2 and M3 represent the branches of the statement. This routine checks to determine if any of the branch EFNs are the same or if any branch is the same as the EFN of the next statement. The appropriate transfer instructions are compiled as needed.

The subroutine IFBN00 is used to compile the code when it is given a destination IFN and the transfer type operation.

Usage. The calling sequence is:

```
TRA      IFA200
(Return to routine PH2000 at 2,1)
```

Upon entering the routine the A-register contains the POOL Table tally word.

Error Returns. None.

11. IFL200--Logical IF Routine

Purpose. The subroutine compiles the necessary code for Logical IF statements.

Method. The logical expression within the parentheses is compiled by the Arithmetic Processor, ARCODE. Then this subroutine compiles one instruction. The location field contains the current IFN with a suffix of "F." for the false condition; the operation field contains "FEQU" and the variable field contains the IFN obtained from the T.JUMP Table, which is the IFN of the statement following the Logical IF. A suffix of "T." is prepared by this routine for use in the location field of the GMAP code for the true side of the Logical IF statement. The sequence of code will be similar to the following:

```
         T--      nF.

nF.      FEQU     IFN

nT.      NULL
          .
          .
          .
```

Usage. The calling sequence is:

```
TRA      IFL200
(Return to PH2000 at 2, 1)
```

Upon entering this routine the A-register contains the POOL Table tally word.

Error Returns. None.

12. GT0200--Unconditional GO TO Routine

Purpose. This subroutine calls the IFBN00 routine to compile a transfer instruction for the GO TO statement.

Method. The POOL Table entry contains the T.JUMP pointer. The subroutine IFBN00 is then called to compile a TRA instruction to the destination using the information from the T.JUMP Table. Either of the two following instructions will be generated:

```
          TRA     IFN
   or
          TRA     B.N
```

Usage. The calling sequence is:

```
TRA      GT0200
(Return to PH2000 at 2,1)
```

When entering this routine, the A-register contains in bits 0-17 the pointer to the first word of the T.POOL Table entry.

Error Returns. None.

13. GTA200--Assigned GO TO Routine

Purpose. This routine sets up coding to cause a transfer to the statement number last assigned to the GO TO variable.

Method. Prior to setting up the transfer coding, this routine checks to determine if any statement number or numbers are in the T.OUTS Table. If so, coding must be set up to transfer to the appropriate B.N IFN instead of the IFN of the statement number assigned.

The generated code for a simple case will be:

```
TRA      v,I
```

For the case where the transfer IFNs appear in the T.OUTS Table (loading and restoring of index registers will be required) the generated code will appear as follows:

```
LDA      v
LDX1     0,DU
RPT      m,2,TZE
CMPA     *+3,1
TTF      -1,1*
TRA      v,I
ZERO     IFN
ZERO     B.n1
ZERO     IFN
ZERO     B.n2
  .        .
  .        .
  .        .
ZERO     IFN
ZERO     B.nn
```

Usage. The calling sequence is:

```
TRA      GTA200
(Return to PH2000 at 2,1)
```

Upon entry to this routine the A-register contains the POOL Table pointer.

Error Returns. None.

14. GTC200--Computed GO TO Routine

Purpose. This routine compiles the necessary code for a Computed GO TO statement.

Method. The following code is generated by this routine:

```
LDQ      (VAR)            (Name of variable)
LDA      (IFN),DL         (IFN of Current Statement)
EAX1     0,QL
TZE      .CGTE.           Call Error Trace
CMPX1    (BRC+1),DU       (Branch Count)+1
TRC      .CGTE.           Call Error Trace
XEC      *,1
TRA      (IFN)            (One TRA for each branch)
TRA      (IFN)
 .
 .
 .
```

If the variable name is an argument, an entry is made in the T.PROL Table and the variable field is compiled as **. The routine IFBN00 is called to compile the TRA instructions at the end of the list.

A flag, location .CGTE., is set to tell the controlling routine that the sequence of instructions for the error trace call must be generated at the end of the compilation.

Usage. The calling sequence is:

```
TRA      GTC200
(Return to PH2000 at 2,1)
```

Upon entry the A-register (0-17) contains the T.POOL Table pointer.

Error Returns. None.

15. ASN200--ASSIGN Routine

Purpose. This routine compiles the required code for the ASSIGN statement from its POOL Table entry.

Method. The routine compiles either

```
         LDA      (IFN),DU
         STA      (VAR. NAME)
```

or

```
   (SYM) LDA      (IFN),DU
         STA      **
```

The latter form is used when the variable name is an argument.

The statement number (EFN) assigned to the variable is in the POOL Table. This EFN is matched against the EFNs in the T.EIFN Table to obtain the corresponding IFN to be compiled. If the EFN is not in the T.EIFN Table a fatal diagnostic message is written.

Usage. The calling sequence is:

```
TRA      ASN200
(Return to PH2000 at 2,1)
```

Upon entry the A-register (0-17) contains the T.POOL Table pointer.

Error Returns. There are no error returns, but the following fatal diagnostic message may be written.

ASN298 THE ASSIGN STATEMENT AT IFN________ REFERS TO A NON-EXISTENT EFN.

16. PAS200--PAUSE Routine

Purpose. This subroutine generates code to handle the PAUSE statement.

Method. This subroutine generates code to perform two functions.

1. Type the message:

   -PAUSE N HIT EOM TO CONTINUE-

   where N = the integer number on the PAUSE source statement.

2. Delay (with no message) until the operator hits the EOM key.

The generated code is as follows:

```
(IFN1)  MME     GEINOS
        OCT     130000000002
        ZERO    (IFN7),(IFN2)
        RTYP
        ZERO    0,(IFN4)
        ZERO    (IFN6)
        MME     GEROAD
        LDA     =031000,DU
        CNAA    (IFN6)
        TZE     (IFN1)
        TRA     (IFN7)+1
(IFN2)  IOTD    (IFN3),8
(IFN3)  OCT     770017171717
        BCI     6,-PAUSE  HIT EOM TO CONTINUE-
        OCT     770017171717
(IFN4)  IOTD    (IFN5),1
(IFN5)  BCI     1,
(IFN6)  ZERO
        ZERO
(IFN7)  BCI     1,0000T*
```

?

repeats until EØM is read

(5-spaces/N ⓝ

Usage. The calling sequence is:

```
TRA     PAS200
(Return to PH2000 at 2,1)
```

Upon entry to this subroutine the A-register contains the POOL Table tally word.

Error Returns. None.

17. RET200--RETURN Routine

Purpose. This routine compiles the required code for a RETURN statement.

Method. Two sets of code can be compiled by this routine. If the return is from a subroutine program:

```
RETURN   (PR.NAM)            Standard return

  or

RETURN   (PR.NAM),K          Nonstandard return

  or

LDQ      (VAR)               Variable nonstandard return
EAX1     0,QL
LDA      (IFN),DL
CMPX1    (MAX. RET. NO.+1),DU
TRC      .CGTE.
STX1     *+2
RETURN   (PR.NAM),0
```

If the return is from a function program:

```
DFLD     F.NAME              Double Precision

 or

LDAQ     F.NAME              Complex

 or

FLD      F.NAME              Real

 or

LDQ      F.NAME              Integer/Logical
```

Each of the above items is followed by:

```
RETURN   (PR.NAM)
```

The cell PR.NAM contains the BCD name of the subprogram. The cell PR.TYP contains the T.NAME pointer in bits 18-35; bit position zero is a 1 for functions; a 0 for subroutines.

Usage. The calling sequence is:

```
TRA      RET200
(Return to PH2000 at 2,1)
```

Upon entry to this routine the A-register contains the POOL Table tally word.

Error Returns. None.

18. STP200--STOP Routine

Purpose. This routine generates the code required for a STOP statement.

Method. The code generated is:

```
CALL .FEXIT
```

Usage. The calling sequence is:

```
TRA      STP200
(Return to PH2000 at 2,1)
```

Error Returns. None.

19. CAL200--CALL Routine

Purpose. This subroutine calls the Arithmetic Processor, ARCODE, to generate the required code for the CALL statement.

Method. Control is transferred immediately to the ARCODE routine.

Usage. The calling sequence is:

```
TRA      CAL200
(Return to PH2000 at 2,1)
```

Upon entry to this routine the A-register contains the POOL Table tally word.

Error Returns. None.

20. RC2000--On-Line Routine
PR2000
PN2000

Purpose. This routine compiles the required coding for READ (cards), PRINT and/or PUNCH statements.

Method. This subroutine checks for a format or namelist reference; if one is found the flag at cell WR2BNF is set for a BCD indication. A CALL to the appropriate library routine with the arguments designating the file code and the format reference is generated. The generated code will appear as follows:

.FRCD

CALL ~~.FCRD.(~~ For READ cards fo=41

or

CALL .FPRN.(For PRINT fo=42

or

CALL .FPUN.(For PUNCH fo=43

and ETC file code, format)'IFN'

There are three entry points to this routine; RC2000 for READ (cards), PR2000 for PRINT and PN2000 for PUNCH.

Usage. The calling sequence is:

```
          ⎧ RC2000
TRA       ⎨ PR2000
          ⎩ PN2000
(Return to PH2000 at 2,1)
```

Upon entry to this routine the A-register contains the POOL Table tally word.

Error Returns. There are no error returns for this routine in the calling sequence, but the following fatal diagnostic message may be written.

PH2021 THE NAMELIST NAME ________ MAY NOT BE USED IN AN ON-LINE INPUT/OUTPUT STATEMENT.

21. RDT200--READ and WRITE Routines
WR2000

Purpose. This subroutine generates the required code for READ and WRITE statements addressing input and output files.

Method. The input/output status is determined. The linkage and T.POOL Table pointer are saved. The file designation is examined to determine if it is a variable or constant. A check is made for FORMAT reference. The mode, BCD or binary, is also analyzed. The appropriate code is generated by calls to the GG routine.

The generated code for the decimal read or write will be:

```
              CALL     .FRDD.(            For READ
         or
              CALL     .FWRD.(            For WRITE

        and   ETC      file code, format)'IFN'
```

The generated code for a binary read or write will be:

```
              CALL     .FRDB.(            For READ
         or
              CALL     .FWRB.(            For WRITE

        and   ETC      file code)'IFN'
```

Usage. The calling sequence is:

```
TRA      {RDT200
         {WR2000
(Return to PH2000 at 2,1)
```

Upon entry to this routine the A-register contains the POOL Table tally word.

Error Returns. None.

22. RW2000--REWIND, BACKSPACE and END FILE Routines
EF2000
BK2000

Purpose. This subroutine compiles the necessary code for REWIND, BACKSPACE and END FILE statements.

Method. There are three entry points to this routine, RW2000 for REWIND statements, EF2000 for END FILE statements and BK2000 for BACKSPACE statements. Upon entry the appropriate I/O subroutine name is retrieved and stored for later use. The linkage is then saved and it is determined if the file reference is a constant or a variable. The routine GG is called to write the compiled coding on the G* file.

The generated coding will be:

```
REWIND

     CALL     .FRWT.(
     ETC      file code)'IFN'

END FILE

     CALL     .FEFT.(
     ETC      file code)'IFN'

BACKSPACE

     CALL     .FBST.(
     ETC      file code)'IFN'
```

Usage. The calling sequence is:

```
              ⎧EF2000
     TRA      ⎨RW2000
              ⎩BK2000
     (Return to PH2000 at 2,1)
```

Upon entry to this routine the A-register contains the POOL Table tally word.

Error Returns. None.

23. WR2CNV--File Routine

Purpose. This subroutine checks the number of the file assigned in an I/O statement.

Method. The file number is checked; if it is greater than 40, a diagnostic is given. A correct number is converted to BCD, prefixed with an "equals" character and returned in the Q-register.

Usage. The calling sequence is:

```
TSX1     WR2CNV
(Error Return)
(Normal Return)
```

Upon entry the file number is contained in the upper Q-register in binary form.

Error Returns. If an error is encountered in the file number (greater than 40), a fatal diagnostic is written and the error exit in the calling sequence is taken.

```
WR2CER    THE FILE NUMBER MUST BE LESS THAN 41.
```

24. SI2000--SETIN and SETOUT Routine

Purpose. This routine removes the input/output list from the POOL Table and generates the required GMAP coding.

Method. This routine is entered at either of its two entry points dependent upon whether an input or an output function is to be performed. The type of variable is determined and special coding for indexing may be required for short list notation. Finally, the coding is generated to call the proper routine for input or output at execution time.

The generated code for the short list notation for decimal input/output is:

```
            CALL     .FSLI.(var,       For input

        or

            CALL     .FSLO.(var,       For output

        and ETC      =n)'IFN'
```

where n is the size of the array.

The generated code for the short list notation for binary input/output is:

```
            CALL     .FBLI.(var,       For input

        or

            CALL     .FBLO.(var,       For output

        and
            ETC      =n)'IFN'
```

where n is the size of the array.

The generated code for other list variables is basically (aside from the indexing instructions) as follows:

```
            LD-      var               For output
            TSX1     .FCNV.
                  or
            TSX1     .FCNV.            For input
            ST-      var
```

Usage. The calling sequence is:

```
        TRA   {SI2000
              {SO2000
        (Return to PH2000 at 2,1)
```

Error Returns. None.

25. ER2000--END READ and END WRITE Routines

Purpose. This subroutine generates the code required for the compilation of input and output.

Method. Upon entry to the appropriate entry point (ER2000 or EW2000) in this routine, index register 0 is set to 0 or 1 to indicate a READ or WRITE condition respectively.

Also, depending on whether binary or BCD READ or WRITE, a CALL to the appropriate library subroutine is generated.

The generated code for binary input/output is:

```
        CALL    .FRLR.          For input
    or
        CALL    .FNLR.          For output
```

The generated code for decimal input/output is:

```
        CALL    .FRTN.          For input
    or
        CALL    .FFIL.          For output
```

Input/output using NAMELIST does not require an END call.

Usage. The calling sequence is:

```
        TRA   {ER2000
              {EW2000
        (Return to PH2000 at 2,1)
```

Error Returns. None.

26. WRCHEK--FORMAT Reference Check Routine

Purpose. This subroutine checks for nonexistent FORMAT statements being referenced.

Method. The T.FEFN Table is examined to determine if the FORMAT statement referenced exists. If not, a fatal diagnostic message is written. At entry, the EFN is in the A-register.

Usage. The calling sequence is:

```
TSX1     WRCHEK
(Error Return)
(Normal Return)
```

Upon leaving this routine on the error return, the A-register contains the EFN.

Error Returns. A fatal diagnostic is written and the error return of the calling sequence is taken when a nonexistent FORMAT statement is referenced.

WRCK10   THE NONEXISTENT FORMAT STATEMENT ________, IS REFERRED TO.

27. WRDIAG--Nearest EFN Diagnostic Routine

Purpose. This routine finds the nearest EFN and writes a diagnostic message.

Method. This routine takes the IFN in cell F.IFN and searches the T.EIFN Table to find the nearest EFN. When found it is converted to BCD and stored in a diagnostic message.

Usage. The calling sequence is:

```
TSX1     WRDIAG
(Normal Return)
```

Error Returns. The following diagnostic comment is written to complete some previous message.

WRD122   AT OR NEAR EXTERNAL FORMULA NUMBER ________.

28. CMPRLG--Prologue Initialization Routine

Purpose. This subroutine generates coding for prologue initialization of variables which will appear in a CALL instruction and are arguments to the subprogram.

Method. Upon entry to this routine the NAME pointer is contained in index register 3. If the variable is found to be an argument to the subprogram, an entry to the prologue table is made and an EQU instruction is generated to establish a location symbol.

Usage. The calling sequence is:

```
TSX1     CMPRLG
ZERO     P
(Normal Return)
```

where P = * + P initialization. Upon return from this routine, the Name or ** (argument) is returned in the A-register.

Error Returns. None.

29. S.PRLO--Determine and Assemble Next IFN Routine

Purpose. This subroutine determines and assembles the next supplementary IFN for this statement.

Method. The subroutines AR.SYM and AR.IFN are used to determine and assemble the next IFN for this statement. The subroutine then enters this symbol and the NAME pointer into the prologue table. The NAME pointer is in bits 0-17 of the A-register upon entry to this routine. At exit, the A-register contains the IFN symbol, left adjusted with trailing blanks. This routine is called by ASN200, GTA200 and CMPRLG.

Usage. The calling sequence is:

```
TSX1     S.PRLO
(Normal Return)
```

Error Returns. None.

30. DTA200--DATA Statements Storage Allocator

Purpose. This routine assigns the values in DATA statements to the proper variable or relative location of an array.

Method. This routine utilizes information stacked in the T.DATA, T.LITR, T.IMPO and T.USUB Tables during Phase One. Another table, T.DORT, is constructed during Phase Two but killed before final return to Phase Two control. The entire storage assignment logic includes the subroutines TDT000, KDT000, MDT000, PDT000 and CDT000 which are described later in the section. There is only one return from DTA200 to the calling program; namely 0,1. If an error occurs such that the source program cannot be compiled, the F.DIAG flag cell is set by the diagnostic routine so that the controlling routine will know when a fatal error has occurred.

Usage. The calling sequence is:

```
TSX1    DTA200
(Normal Return)
```

Error Returns. There is no error return in the calling sequence, but the following fatal diagnostics may be written:

DT2BD1	DATA STATEMENT. LITERAL LIST IS LONGER THAN VARIABLE LIST.
DT2BD2	DATA STATEMENT. NO SUBSCRIPT CORRESPONDING TO THE DO INDEX __________ FOR THE VARIABLE ___________.
DT2BD3	DATA STATEMENT. NO DO INDEX CORRESPONDING TO THE SUBSCRIPT __________ FOR THE VARIABLE ___________.
DT2BD4	DATA STATEMENT. A BCI LITERAL EXCEEDS THE DIMENSION OF A SHORT LIST VARIABLE.
DT2BD6	A VARIABLE, __________, IN A DATA STATEMENT IS IN BLANK COMMON.

31. TDT000--T.USUB Entry Pull Routine

Purpose. This routine extracts information from the T.USUB Table.

Method. The routine pulls from the T.USUB Table all information from a single entry. An entry includes the coefficient, addend, name and dimension of all subscripts of a variable. The routine handles variables with constant subscripts or with variable subscripts. Each subscript of an entry puts a word in each of the following arrays:

TDTDIM,3 - Product of dimensions, so far.

TDTADD,3 - The addend -1

TDTCOF,3 - Coefficient

TDTNAM,3 - Pointer to T.NAME and the Name of Index. (This will be zero for constant subscripts.)

Usage. The calling sequence is:

```
TSX1     TDT000
(Normal Return)
```

The routine is called from the DTA200 routine.

The dimensionality is returned in bits 0-17 of the cell DTADIM upon exit.

Error Returns. None.

32. MDT000--Index Match Routine

Purpose. This routine is called by the DTA200 routine to match index names and calculate addend information.

Method. This routine is used only when implied DOs occur. It matches the index names found in the T.IMPO Table (DTADON,5) with the index names from the T.USUB Table (TDTNAM,3). If there is not a one-to-one correspondence, an error message is printed, but processing is continued to find more possible errors. For each subscript where a match is found the routine calculates the following:

DTALLM,3 - Lower limit for subscript.

DTAULM,3 - Upper limit for subscript.

DTAINC,3 - Increment for successive addends.

This routine returns control to location DTA000 when the information for all subscripts of a single variable name are calculated. The routine is entered once for each variable name within an implied DO.

Usage. The calling sequence is:

```
TSX1     MDT000
(Normal Return)
```

Error Returns. Error messages required by this routine are written from the DTA200 routine.

33. KDT000--Subscript/Dimension Check Routine

Purpose. This subroutine ensures that subscript notation for a variable does not violate the dimensions of the variable.

Method. This subroutine compares all of the subscripts of a variable against the corresponding dimensions from entries in the T.DIME Table. If any value of the subscript (C*I+A) for any I defined by the implied DO is less than zero or greater than the dimension, a diagnostic is printed. Note that the dimension is doubled for complex or double precision variables. The remaining subscripts are checked even though an error has occurred.

Usage. The calling sequence is:

```
TSX1    KDT000
(Normal Return)
```

Error Returns. There is no error return in the calling sequence, but the following diagnostic message may be written:

```
KDTB1A    DATA STATEMENT, SUBSCRIPT _________ OF THE VARIABLE,
          _________, IS OUTSIDE THE RANGE OF THE DIMENSION.
```

34. PDT000--T.DORT Entry Pull Routine

Purpose. This routine compiles the ORG, DEC and similar type operations for all variables and addends within an implied DO statement or a nest of implied DOs.

Method. The CDT000 subroutine (described later in this Chapter) is called for every variable/addend combination to set up the actual GMAP code. The variable name and its addend are determined from information saved in the T.DORT Table. One pass is made through the T.DORT Table for each variable/addend combination to be compiled. Each pass also updates the table for future passes. The logical flow through the T.DORT Table is controlled by the DO-ending information stored in the DTADO(X) buffers.

The T.DORT Table is built up in Phase Two for each nest of implied DO statements. It is killed when the routine PDT000 reaches completion before control is returned to the DTA200 routine.

Usage. The calling sequence is:

```
TSX1    PDT000
(Error Return)
(Normal Return)
```

Error Returns. The error return is in the calling sequence. No error messages are written by this routine.

35. CDT000--GG Code Setup Routine

Purpose. This routine compiles instructions for the DATA statement.

Method. The compiled code appears as follows:

```
ORG     A+N

OPR     LIST
```

where:
A = Input to the routine as DATNAM.
N = Input to the routine as DTAFAD.
OPR = DEC, BCI or OCT.
LIST = A string of BCI literal information pulled from the T.LITR Table.

Before compiling "ORG A+N," the routine checks the previous name and addend to see if another ORG is needed.

If a literal is to be repeated (indicated by the CDTRPT flag on), no entries are pulled from the T.LITR Table since the previous entry is to be used again.

Usage. The calling sequence is:

```
TSX1     CDT000
(Error Return)
(Normal Return)
```

Error Returns. The error return in the calling sequence is taken when the literal list is too long. The following error messages are written.

CDTB32	A NONDIMENSIONED VARIABLE CONTAINS TOO MANY WORDS OF HOLLERITH INFORMATION.
CDTB0A	THE ENTIRE ARRAY OF A SHORT LIST VARIABLE MUST BE FILLED.
CDTBD0	DATA STATEMENT, VARIABLE LIST IS LONGER THAN LITERAL LIST.

36. CDT500--Type Consistency Check Routine

Purpose. This subroutine checks the variable name being compiled against the type of literal assigned to the name.

Method. The variable name being compiled is checked against the type of literal assigned to the name as indicated in the T.LITR Table. The check is performed by comparing the flags I.LOG, I.CPX, I.DBL, I.REL, and I.ITG with the code in the first 3 bits of the T.LITR control word.

The bits are assigned as follows:

```
Prefix = 0, Real
       = 1, Integer
       = 2, Logical
       = 3, Octal *
       = 4, Complex
       = 7, Double Precision
```

*Flags not checked further.

The routine also checks if the variable/literal combination has been checked before. This procedure prevents duplication of error messages if in a DO loop or if a short list and the literal is being repeated.

Usage. The calling sequence is:

```
TSX1        CDT500
(Normal Return)
```

Error Returns. There are no error returns in the calling sequence, but the following messages are written.

CDT585   A VARIABLE,__________ IS INCONSISTENT WITH A LOGICAL LITERAL.

CDT595   A VARIABLE,__________ IS INCONSISTENT WITH THE LITERAL __________.

37. DBUG20--Debug-Time Test Code Generator

Purpose. This routine generates the required coding for the FOR clause of the DEBUG statement.

Method. The general form of the FOR clause is:

FOR m_1, m_2, m_3

where m_2 and m_3 are optional. These parameters are obtained from the POOL Table. The following code is generated if only m_1 is present:

```
(IFN)   LDX0     0,DU
        ADLX0    1,DU
        STX0     (IFN)
        CMPX0    m1,DU
        TNZ      (IFN+3)
```

If m_1, m_2, and m_3 are present, the following code will be generated:

```
(IFN)     LDX0      0,DU
          ADLX0     1,DU
          STX0      (IFN)
(IFN/1)   CMPX0     m1, DU
          TNC       (IFN+3)
          CMPX0     m2+1,DU
          TRC       (IFN+3)
          LDX0      (IFN/1)
          ADLX0     m3,DU
          STX0      (IFN/1)
```

Usage. The calling sequence is:

```
TRA       DBUG20
(Return to PH2000 at 2,1)
```

Upon entry to this routine, the A-register contains the POOL Table tally word.

Error Returns. None.

38. DIFA00--Debug Arithmetic IF Code Generator

Purpose. This routine generates coding for the DEBUG Arithmetic IF clause.

Method. The subroutine ARCODE is called to generate the code for the arithmetic expression. The conditions which may be indicated by YES, NO, EXIT, or DUMP are analyzed and the appropriate transfers or calls are generated.

Usage. The calling sequence is:

```
TRA       DIAF00
(Return to PH2000 at 2,1)
```

Upon entry to this routine, the A-register contains the POOL Table tally word.

Error Returns. None.

39. DIFLOO--Debug Logical IF Code Generator

Purpose. This routine generates the required coding for the DEBUG Logical IF statement.

Method. The subroutine ARCODE is called to generate the code required for the logical expression. An FEQU instruction is generated to define the "False" location symbol.

Usage. The calling sequence is:

```
TRA      DIFLOO
(Return to PH2000 at 2,1)
```

Upon entry to this routine the A-register contains the POOL Table tally word.

Error Returns. None.

40. AR.SYM--Location Symbol Generator

Purpose. This subroutine generates a two character BCD symbol.

Method. The two character BCD symbol is appended to the IFN. The entire symbol then becomes the location field (columns 1-6) of a GMAP operation. A new symbol is generated with each call to this subroutine.

Usage. The calling sequence is:

```
TSX1     AR.SYM
(Normal Return)
```

Error Returns. There are no error returns,but the following diagnostic message is written:

```
AR.S70    FORTRAN STATEMENT IS TOO LONG.
```

41. AR.TRC--Logical Expression Check Routine

Purpose. This subroutine makes entries in the ARLO.T Table (the logical operation table). Upon entry to this routine, a "look-back" over the logical statement up to this point is performed to determine what type of transfer (true or false) should be compiled at this time. A branch symbol is also provided as the address of the chosen type of transfer operation.

Method. This subroutine performs a true/false traceback to the lower level of logical operation for optimization of testing instructions. The Logical Operator Table (ARLO.T) is examined. The format of this table is:

```
VFD  1/X,5/Operator+Y,12/Level,18/Symbol
```

where:

X = 1, if the level is closed out.
= 0, if the level is not closed out.
Y = 4, if the item is the last of the level.
= 0, if the item is not the last of the level.
Symbol = a two character symbol provided for the level.

Another table, the Level and Logical Store Table (ARLS.T) is also used. The format of this table is:

```
VFD  18/Level,18/000000
VFD  18/Symbol F,18/Symbol T
```

where:

Symbol F = A created two character symbol associated with the false point of level.

Symbol T = A created two character symbol associated with the true point of level.

The subroutine is entered with the ARLO.T Table item for the current operator/operand (primitive) in the A-register. Output from the subroutine is contained in the cells AR.TZE, AR.TFS, AR.TSM. If the cell AR.TZE is equal to zero, then the TZE operation should be used after a SZN. Otherwise, the TNZ operation should be used. The cell AR.TFS contains either "T." or "F." or the created symbol.

Usage. The calling sequence is:

```
TSX1      AR.TRC
(Normal Return)
```

Error Returns. None.

42. SS.ETN--T.ARIT Item Fetch Routine

Purpose. This subroutine fetches items from the T.ARIT Table.

Method. After the T.ARIT Table item is retrieved, the set of flags are turned ON as follows:

Flag Name	Definition
AR.NOP	Operator Flag
AR.NOD	Operand Flag
AR.NID	ID Flag
AR.NST	Store Bit Flag
AR.NTP	Type (Mode) Flag
AR.NLV	Level Flag

Usage. The calling sequence is:

```
TSX1      SS.ETN
(Normal Return)
```

Error Returns. None.

43. ARCODA--Arithmetic Statement Entry Routine

Purpose. This subroutine provides an entry to the Arithmetic Statement Processor, ARCODE.

Method. Upon entry to this routine, control is transferred to the routine ARCODE. Control from ARCODE is returned to this routine and then back to routine PH2000 at location 2,1.

Usage. The calling sequence is:

```
TRA       ARCODA
(Return is to PH2000 at 2,1)
```

Error Returns. None.

44. AH.RAS--Erasable Storage Addend Routine

Purpose. This subroutine is used to compute the addend for erasable storage.

Method. This routine is called by the Phase Two Arithmetic Processor, ARCODE. There are two entry points for this routine; one for single precision (AR.RAS) and one for double precision (AR.RAD) erasable storage.

Usage. The calling sequence is:

```
TSX1      AH.RAS
(Normal Return)
```

The subroutine is entered with the level in the A-register. Upon exit the addend is in the Q-register.

Error Returns. None.

45. AR.COM--Operation Compile Routine

Purpose. This subroutine is used to compile a GMAP line of code from an operator/operand combination.

Method. Upon entry to this subroutine the operator is in the A-register. The operand is described by the flags set by the routine SS.ETN described earlier. If index register zero is a zero, the current flags are used. If it is a one, the next flags are used.

Usage. The calling sequence is:

```
TSX1      AR.COM
(Normal Return)
```

Error Returns. None.

46. ARCODE--Arithmetic Expression Coding Generator

Purpose. This subroutine is called to generate the GMAP coding for arithmetic expressions.

Method. The arithmetic expressions of the source program are broken down into a series of table entries during Phase One of the FORTRAN IV Compiler. This routine retrieves that information from the POOL Table and compiles the required coding.

The subroutine SS.ETN is called to retrieve arithmetic expression information from the POOL Table. This information is distributed to the flag words as shown in the description of the routine SM.OVE later in this Chapter.

Initially the routine determines if the expression is logical or arithmetic. If arithmetic, the initial level is taken and processed. For the first item of a level, load-type instructions must be generated according to the mode of the operand. It should be noted that all elements of a given level are completely processed before passing to the next level.

Tests are made for the mode; that is, double precision, real, integer, complex or real double. For each of these the necessary instructions are compiled for addition, subtraction, multiplication or division. If the exponentiation operator is present, the mode of the operand determines whether a call to a subroutine, a series of multiply instructions or a repeat-multiply sequence will be generated. For the function operator, a call is made to the AR.FUN subroutine for processing. At the end of a given level, the store bit is tested to determine whether store-type instructions will be required or whether linkage by registers will be used.

Logical expressions are processed according to the type of operator; logical or relational. For the logical operator, testing instructions are compiled followed by either a transfer-zero or a transfer-nonzero type instruction to the true or false side of the equation.

For relational operators, the two operands require generation of comparison instructions (which are determined by the mode of the operands) followed by test and transfer-type instructions to complete the sequence.

For the left side of the equals in an arithmetic expression, the mode is tested again and the proper store-type instructions will be generated. For the left side of the equals in a logical expression, the generated coding will be terminated with store-type instructions. The instructions provide for storing, either a zero or a one, to indicate either a false or a true condition, respectively.

Usage. The calling sequence is:

```
TSX1      ARCODE
(Normal Return)
```

Error Returns. None.

47. AR.ARG--Argument Compile Routine

Purpose. This subroutine is called to compile the required code for a CALL statement.

Method. There are two entries to this subroutine, AR.ARG and AR.ARH. The former compiles an ETC in the operation field and the argument in the variable field. The AR.ARH entry concatenates the argument onto the variable field. The argument will be prefixed from the location

AR.PFX and suffixed from the location AR.SFX. If index register zero is a zero, then the next flags are used. Otherwise, the current flags are used.

Usage. The calling sequence is:

```
TSX1    ⎧AR.ARG
        ⎩AR.ARH
(Normal Return)
```

Error Returns. None.

48. AR.IFN--IFN Conversion Routine

Purpose. This routine converts the IFN to BCD and adds a suffix of a two character symbol for use as the location symbol in a GMAP coding line.

Method. This routine calls the conversion routine S.BB00 to convert the IFN. Upon entering the routine, the two character symbol is contained in the A-register right adjusted with zeros.

Usage. The calling sequence is:

```
TSX1     AR.IFN
(Normal Return)
```

Upon exit, the entire symbol is in the Q-register, left adjusted with blanks.

Error Returns. None.

49. AR.ALC--Erasable Counts Routine

Purpose. This subroutine computes the erasable counts for the Storage Allocator.

Method. The following erasables:

```
 N.
NN.
 X.
XX.
```

and the Arithmetic Statement Function erasables:

P.N
PP.N
A.N
AA.N

where N is the level number for the ASF, comprise the erasables for which counts are computed.

There are three entry points to this routine as follows:

Entry Point	Definition
AR.ALC	For A.N and AA.N
AR.ALP	For P.N and PP.N or N. and NN.
AR.ALX	For X. and XX.

Usage. The calling sequence is:

$$\text{TSX1}\ \left\{\begin{array}{l}\text{AR.ALC}\\ \text{AR.ALP}\\ \text{AR.ALX}\end{array}\right.$$

(Normal Return)

Error Returns. None.

50. AR.FUN--Function and Arguments Compile Routine

Purpose. This subroutine is called to compile FUNCTION statements and their arguments.

Method. This subroutine compiles the required GMAP code for FUNCTION statements and their arguments including linkages, calls to appropriate routines and returns.

Usage. The calling sequence is:

```
TSX1      AR.FUN
(Normal Return)
```

Error Returns. None.

51. AR.CLS--Logical Levels Close Routine

Purpose. This subroutine is called to close out intermediate logical levels.

<u>Method.</u> This subroutine generates the required coding to set true or false indicators for variables as required by logical operations.

<u>Usage.</u> The calling sequence is:

```
TSX1      AR.CLS
(Normal Return)
```

<u>Error Returns.</u> None.

52. AS.FNC--Arithmetic Statement Function Definition Compile Routine

<u>Purpose.</u> This routine is called to compile the required instructions for an Arithmetic Statement Function Definition.

<u>Method.</u> The information for the ASF definition is taken from the POOL Table and examined by this routine. The GG routine is called to generate the required GMAP coding.

<u>Usage.</u> The calling sequence is:

```
TSX1      AS.FNC
(Normal Return)
```

<u>Error Returns.</u> None.

53. SM.OVE--Flag Shift Routine

<u>Purpose.</u> This subroutine shifts the flags described in the SS.ETN routine description.

<u>Method.</u> The Current Flags are shifted to the Previous Flags. The Next Flags are moved to the Current Flags. The Next Flags are set by the SS.ETN routine. The flag names are:

1. Next Flags

AR.NOP	Operator
AR.NOD	Operand
AR.NID	ID
AR.NST	Store Bit
AR.NTP	Mode
AR.NLV	Level
AR.NLK	Linkage

2. Current Flags

AR.COP	Operator
AR.COD	Operand
AR.CID	ID
AR.CST	Store Bit
AR.CTP	Mode
AR.CLV	Level
AR.CLK	Linkage

3. Previous Flags

AR.POP	Operator
AR.POD	Operand
AR.PID	ID
AR.PST	Store Bit
AR.PTP	Mode
AR.PLV	Level
AR.PLK	Linkage

Usage. The calling sequence is:

```
TSX1      SM.OVE
(Normal Return)
```

Error Returns. None.

54. SUBCOM--Subscripted Operand Compile Routine

Purpose. The subroutine compiles GMAP coding for subscripted operands.

Method. This subroutine is entered with the operation to be compiled in the A-register. The Q-register contains the T.SUBS pointer for the subscripted operand. The subroutine uses the T.SUBS Table and the T.LDXR Table.

Usage. The calling sequence is:

```
TSX1      SUBCOM
(Normal Return)
```

Error Returns. None.

55. CK.DOS--Transfer Check Routine

Purpose. This subroutine checks the legality of transfers into and out of DO loops and creates a table for those jumps which must pass through the save/restore sequences.

Method. This subroutine uses the information contained in the Buffered Table T.IODO, T.JUMP and T.JUNK. The table T.OTIN may be generated by this routine.

Usage. The calling sequence is:

```
TSX1      CK.DOS
(Normal Return)
```

Error Returns. There are no error returns in the calling sequence, but the following fatal diagnostic message may be written:

```
CK.875   ILLEGAL TRANSFER INTO THE RANGE OF A DO.
```

56. AR.XPC--Literal Exponents Check Routine

Purpose. This subroutine is called to check for literals in exponentiation.

Method. If literals are found, ETC operations are compiled for the arguments.

Usage. The calling sequence is:

```
TSX1      AR.XPC
(Return 1)          Literals present
(Return 2)          No literals present
```

Error Returns. None.

57. BX.000--Basic Block Indexer

Purpose. The Basic Block Indexer is called at the beginning of a basic block; that is, a linear stretch of code with only one entry, one exit and not in the range of a DO.

Method. Initially, the basic block is checked for subscripted variables. If none exist; indexing will not be required and the indexer is bypassed.

The Basic Block Indexer uses several tables and they are described in detail below for ready reference. It should be noted that the T.INTS and T.SUBS Tables are created in Phase One and remain throughout the duration of the compile. The T.USSR and T.SISU Tables are created and used only for the duration of a basic block.

1. T.USSR Table--A table of appearances of subscripted variables within a defined region; that is, a basic block.

0	1 5	6 17	18 35
-	XR	F	USUB Pointer
(IFN)			T.SUBS IFN

where: XR = the assigned index register.
F = frequency of appearance.
(IFN) = location at which XR must be recalculated.
\- = usage flag.

2. T.INTS Table--A table of each literal appearance of a nonsubscripted integer variable on the left side of an Arithmetic Statement or in an I/O list.

0 17	18 35
NAME Pointer	IFN

3. T.SISU Table--A table of T.USSR Table entries grouped according to similar USUB entries or calculated coefficients.

0	1 5	6 17	18 35
-	XR	( ) F	USUB Pointer
XR		( )	USUB Pointer

(Additional entries may be included in this group. The entry having a minus sign always marks the beginning of a group.)

where: - = signal for first word of a similar group.
XR = assigned index register.
F = frequency of use for entire group.
( ) = later an IFN placed here.

4. T.SUBS Table--A table of each literal appearance of a subscripted variable.

0 17	18 35
IFN	Supplementary IFN
NAME Pointer	USUB Pointer

After replacements are performed this table appears as shown below:

<table>
<tr><th colspan="2">0 17</th><th>18 35</th></tr>
<tr><td colspan="2">LDXR Pointer</td><td>Supplementary IFN</td></tr>
<tr><td>5
XR</td><td>6
NAME Pointer</td><td>Addend</td></tr>
</table>

The Basic Block Indexer performs its functions in several distinct steps and calls additional subroutines as needed. These subroutines which are mentioned below will be described in detail later in this Chapter. The general flow of the Basic Block Indexer is as follows:

1. The T.SUBS Table is examined for subscript usage within the basic block. Through the use of the subroutine IX1000 the T.USSR Table is constructed. When a subscript integer appears in the T.INTS Table, within the range of the basic block and the T.INTS IFN precedes the T.SUBS IFN, then the IFN+1 for the occurrence is also placed in the T.USSR Table. Duplicate entries in the T.USSR Table are eliminated and accounted for by incrementing a frequency count in the first word of an entry. The frequency count is later used in determining the priority for index register assignment.

2. The T.USSR Table is used to construct the T.SISU Table (similar subscripts). The first word of a T.USSR entry is placed in the T.SISU Table and set minus. The subroutine X19000 is then used to find all T.USSR entries which contain a variable subscript. When found, the USUB pointer of the T.USSR entry is compared to the T.SISU entry. If a match is found, the frequency count in the applicable T.SISU entry is incremented.

   If no match is found, the simple coefficient for the T.USSR entry is computed and compared again with the calculated coefficient of the T.SISU Table entry. If a match is found on this second compare, the frequencies are added and stored back into the first word of the T.SISU entry and the USUB pointer from the T.USSR Table is entered into the sequence. If there is no match on the second compare, a new entry is made in the T.SISU Table and the subroutine returns for the next T.USSR entry.

3. The entries in the T.SISU are examined for frequency counts and index register assignments are made accordingly. The subroutine X07000 is called to perform this function. The T.SISU Table is examined to determine the most frequently used group of similar subscripts. This group will be assigned the next available index register. The process is repeated until all of the available index registers have been used. Remaining groups, if any, are assigned to the spill register and the Overflow-Spill Flag is set. Additional information on the subroutine X07000 appears later in this Chapter.

4. The T.SISU and T.USSR Table entries are compared by USUB pointer and the index register assignments of the T.SISU entries are transferred to the matching T.USSR entries.

5. The T.SUBS Table is examined to find subscripted variables within the basic block. When one is found, the T.USSR Table is searched for a match on the USUB pointer. The T.SUBS IFN is then compared with the T.USSR's IFN (the point to recalculate the index value). If the T.SUBS occurrence is earlier than that indicated by T.USSR, then the next T.USSR Table entry is obtained and tested. If there is a successful match the T.SUBS IFN is compared to the T.SUBS IFN stored in the T.USSR entry.

   If this subscripted variable is not within the range of this unique subscript, the subroutine returns to process the next T.USSR Table entry. If the T.USSR entry has already been done, control passes to step 7 described below. If the T.USSR subscript is a constant, control passes to step 8 described below.

   The assigned index register and the T.SUBS (or Basic Block origin) IFN are saved. The subroutine IX9070 is called to compile the IFN in the location field position. The subroutine IX9050 is called to compute the indexing coefficient. The subroutine X08000 is called to compile the computation instructions for index register loading. These subroutines are described in detail later in this Chapter.

6. The assigned index register is then tested to determine if it is the spill register. If not, the GG routine is called to compile the code for the load instruction (EAXn 0,QL). If the spill register is required, additional instructions such as

```
EAX0        0,QL
STX0        B.BGCT
```

   where

```
B.BGCT   LDX0     **,DU
```

are compiled. Upon return from the GG subroutine, the X11000 subroutine is called to mark the T.USSR and T.SISU Table entries used.

7. The assigned index register number is placed in the T.SUBS Table. If the assigned index register is the spill register, then an entry is made in the T.LDXR Table and the T.LDXR pointer is placed in the T.SUBS Table entry replacing the IFN.

8. The addend is computed through a call to the subroutine X20000 and placed in the T.SUBS Table entry replacing the USUB pointer. Control is then returned to step 5 above for examination of the next T.SUBS Table entry.

9. If the subscript contained adjustable dimensions, the subroutine X01000 is called to compile instructions to build a simple USUB Table of dimensions and constants, to compute coefficients and to compile prologue instructions for the computation of the addend. A loop is set up within this subroutine so that prologue instructions will be compiled for all ENTRY points that have adjustable dimensions as arguments. On return from the X01000 subroutine, completion of this loop is tested before control is returned to step 5 above.

The following diagram outlines the general flow through the Basic Block Indexer.

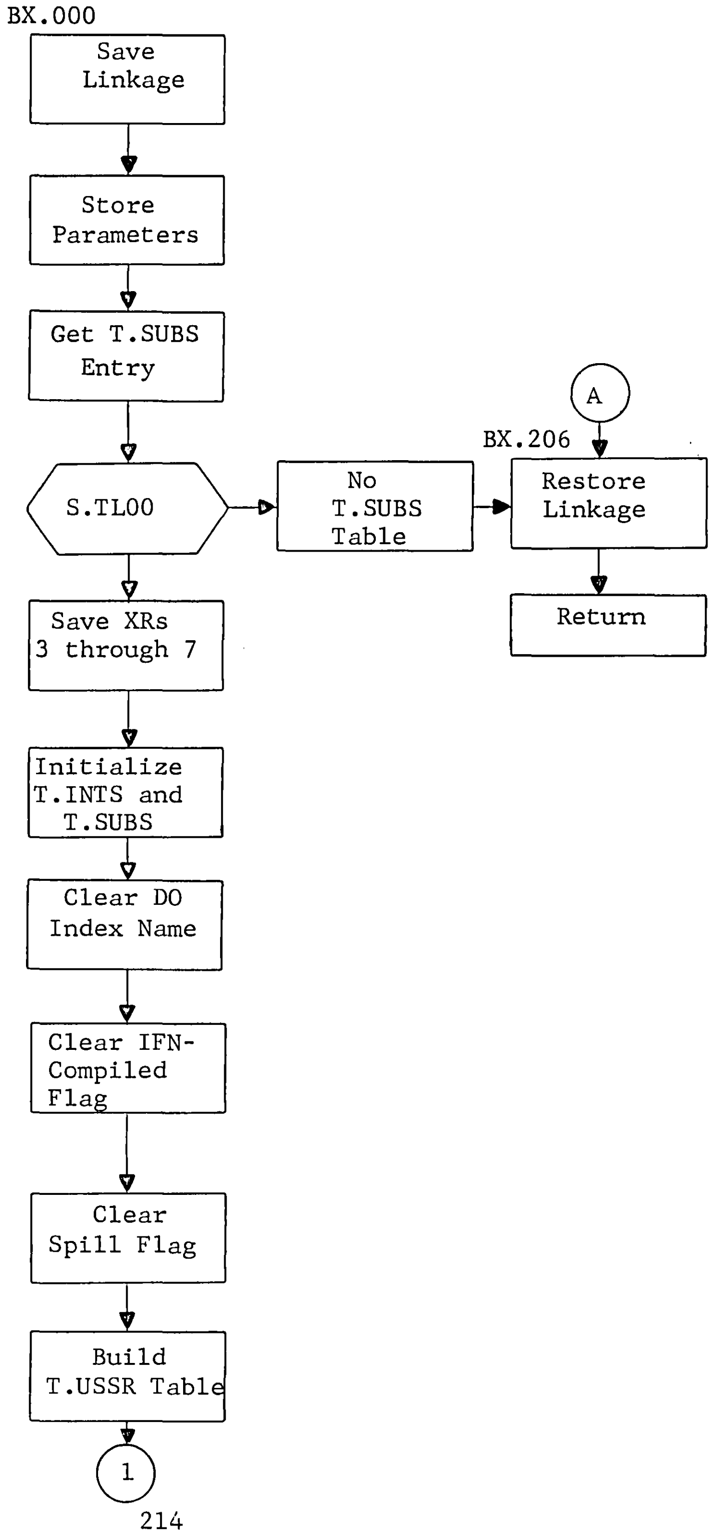
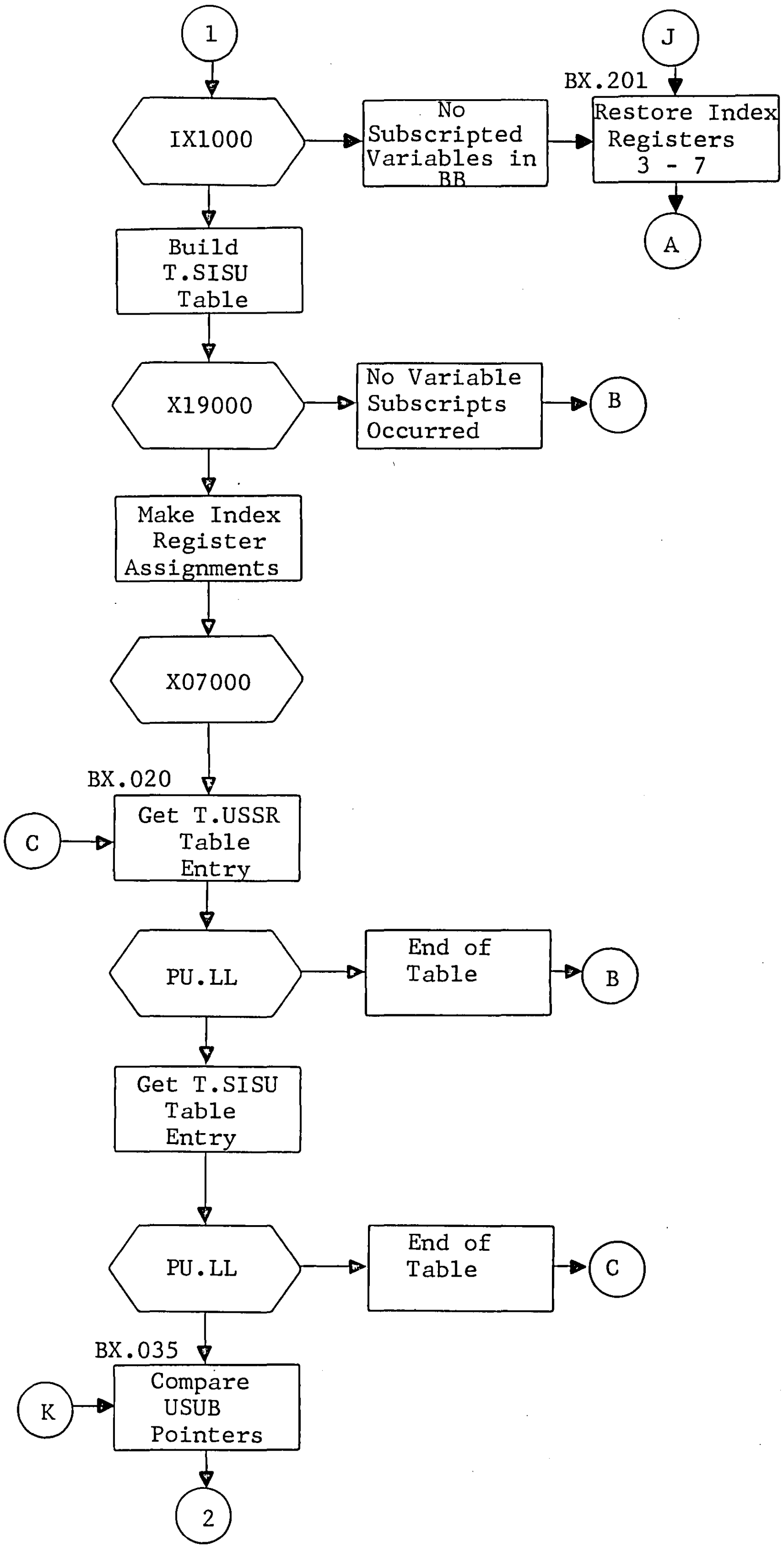
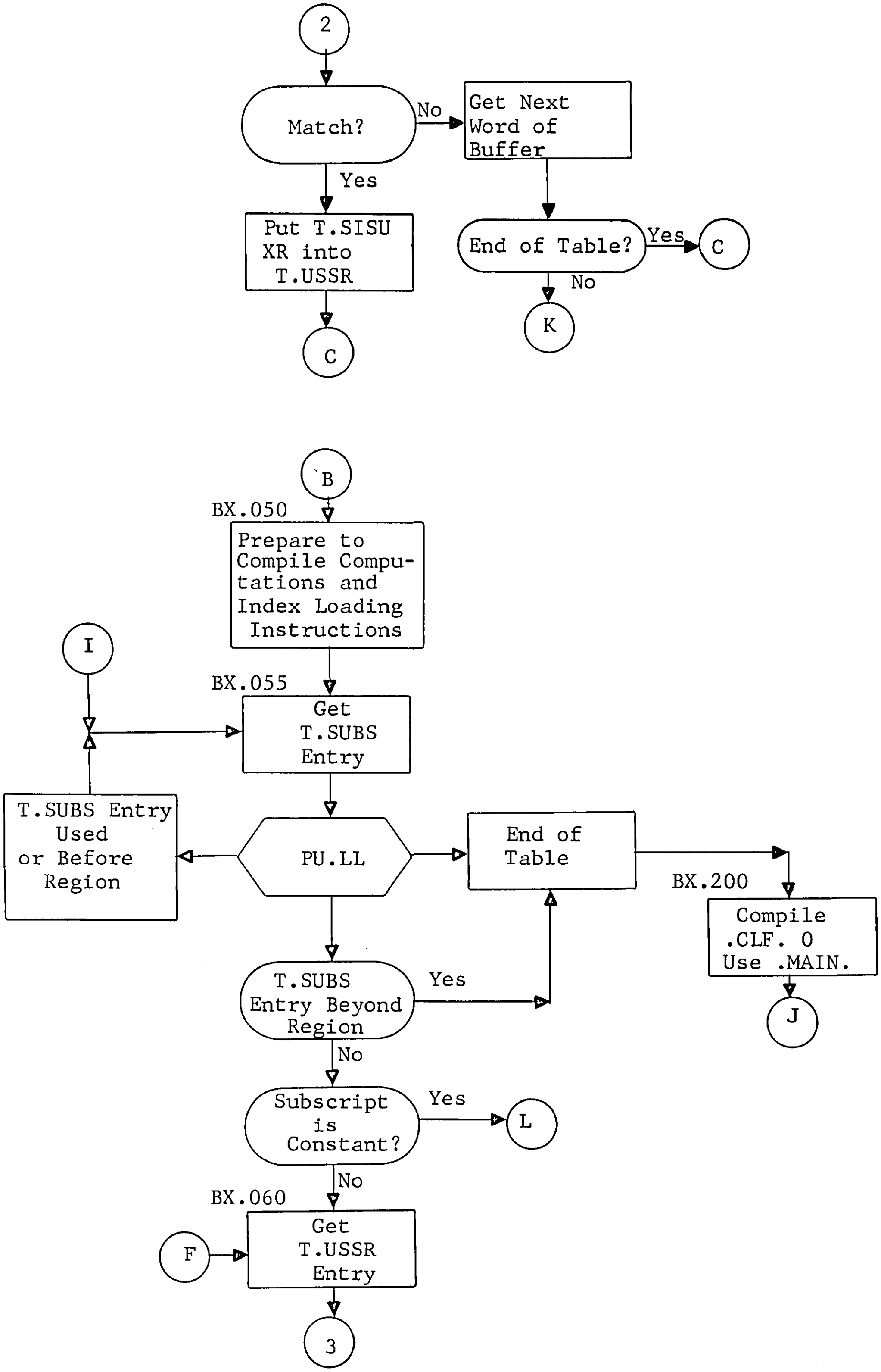


Figure 9. Basic Block Indexer Flow Diagram



215

Figure 9. (continued)



216

Figure 9. (continued)

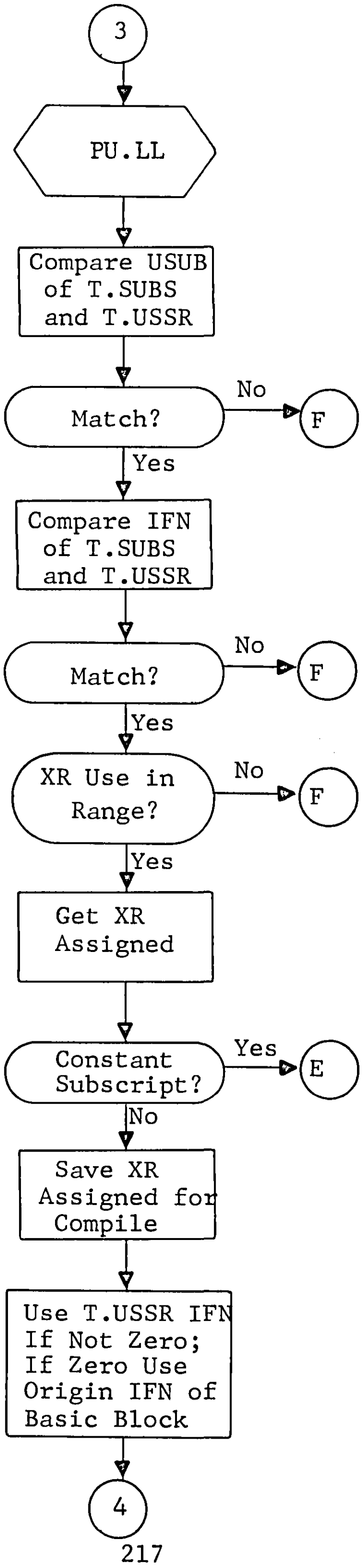
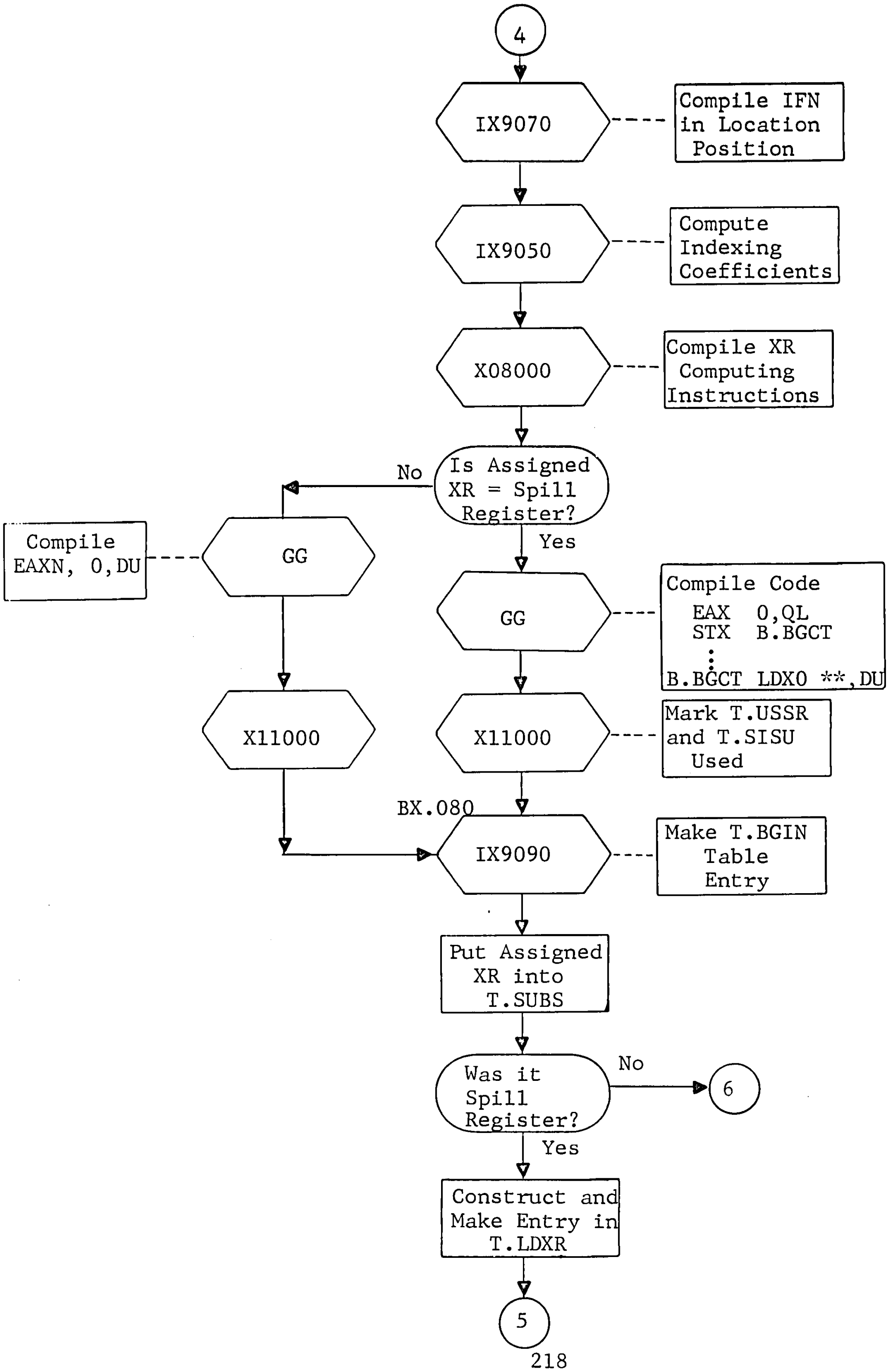
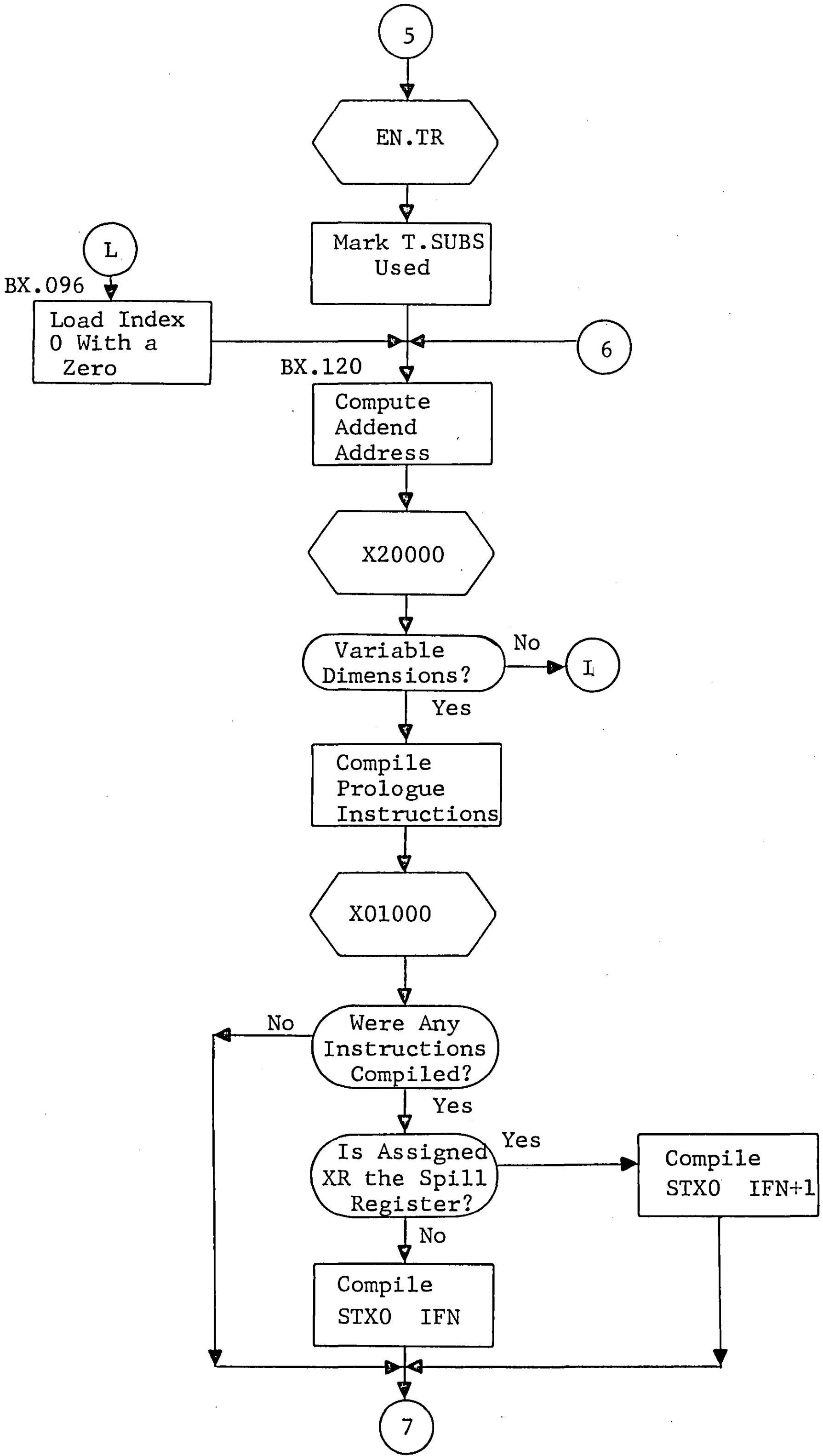


Figure 9. (continued)



218

Figure 9. (continued)



219

Figure 9. (continued)

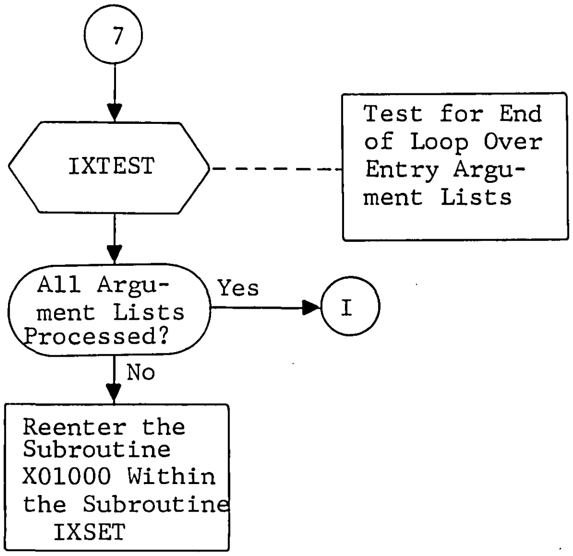


Figure 9. (continued)

Usage. The calling sequence is:

```
TSX1        BX.000
ARG         L (BB Origin)
ARG         L (BB End)
(Normal Return)
```

The following subroutines related to indexing are called by this subroutine:

IX1000
X19000
X07000
IX9070
IX9050
X08000
X11000
IX9090
X20000
X01000

Each of these subroutines is described in detail later in this Chapter.

Error Returns. None. (There is one transfer to the machine error routine, MACERR, at symbolic location BX.060+1.)

58. DX.000--DO Indexer Subroutine

Purpose. This subroutine is called from the Phase Two Executive Routine (PH2000) whenever a new DO nest is encountered. All of the instructions for the computation and initialization of all DO indexing and subscripting of variables within the DO nest are generated by this subroutine.

Method. For the processing of a DO nest there are three DO pointers maintained at all times. They are:

IXDOOP, Basic DO Pointer--Points to the outermost DO.

IXDOCP, Current DO Pointer--Points to the current DO being processed.

IXDOWP, Working DO Pointer--Points within the current DO being processed.

The processing done in this subroutine is performed in a series of steps, some of which are repeated until all of a particular type of information has been completed. In line with this design, the method of the DO indexer subroutine is presented in step form below:

1. Upon entry to this subroutine, the linkage is saved from index register 1. The "IFN Compiled" flag is cleared as well as the "XR Overflow" flag, the "XR Assign" flag and the "DO-XR Overflow" flag. The Basic DO pointer is retrieved from the calling sequence and saved. This pointer is also established as the working DO pointer.

2. The basic DO entry is retrieved from the T.IODO Table using the PU.LL subroutine and the origin and destination of the DO are saved.

3. The tables, T.SUBS, T.INTS, T.JUNK, T.JUMP and T.RINT are set to their respective beginnings. Any previous T.SISU Table is cleared and the first entry is made in the new T.SISU Table for the DO index. The format of the T.SISU Table as used by the DO Indexer is shown below:

T.SISU Table--DO Indexer

0 2	3 17	18 35
-XR	F	G
XR		USUB Pointer$_1$
XR		USUB Pointer$_2$

. . . . etc . . . .

where:
- = the first word of a group of entries.
F = the frequency of appearance; later replaced by an IFN.
G = the calculated coefficient.

The format of the entry made in the T.SISU Table for the DO index is:

0 17	18 35
- 1	1

The subroutine X14000 is then called to construct the T.SISU Table for all of the variables on the DO index only. When completed, the frequency count of the T.SISU entry for the DO index is replaced with the high value of octal sevens so that it will take first priority in the assignment of index registers. Upon completion of the T.SISU Table, initialization is performed for the assignment of index registers. The actual assignment of index registers is performed through a call to the subroutine X07000; values are placed in the T.SISU Table.

4. The subroutine X02000 is called to compile instructions for the initialization, incrementation and testing of the DO index register. This subroutine also compiles the instructions necessary for the initialization and incrementation of the other index registers used inside the DO loop. The next DO pointer is obtained from the T.IODO Table by a call to the PU.LL subroutine. The origin and end of this new DO are saved. A check is performed for duplicate DO names. If found, a warning diagnostic message will be written. It should be noted that within the DO nest, all nested DOs of the same level will have the same DO index assigned to provide the most efficient use of index registers.

5. A call to the subroutine X06000 is made to check for branches into and out of the range of the DO. When present, coding must be generated for the saving and restoring of the indexes. Again within the DO nest, the subroutine IX9120 is called to locate the next DO loop on the same level.

6. The T.SUBS Table is examined in order to find all the variables within the range of the DO which are subscripted on the DO index and the index register assignment will be transferred from the T.SISU Table to the corresponding entry in the T.SUBS Table. The subroutine X15000 is called to perform this function. The next DO is obtained from the information in the T.IODO Table; this becomes the current DO and steps 3 through 6 above are repeated until all DO indexes are compiled and all variables subscripted on the DO index are processed.

7. At this point in the DX.000 subroutine, all of the DO indexing has been completed. All pointers and indicators are reset to the beginning of the basic DO.

8. The T.USSR Table is now constructed through a call to the X13000 subroutine. This table is composed of all of the subscripted variables using the DO index. The T.USSR Table information is then used to construct a new T.SISU Table. The subroutine X19000 is called to perform this function. Using the subroutine PU.LL, entries from the T.USSR Table are processed against the T.SISU Table entries. All similar regions are marked through a call to the subroutine X11000.

9. The subroutine IX9070 is called to compile the IFN in the location field of the GMAP instruction. The subroutine IX9050 is called to compute the coefficient and the subroutine X08000 is called to compile the instructions to compute the constant part of the addresses of the subscripted variables. In the case of a subscripted variable in which there is no DO index involved; then the entire reference address is computed. Additional information on computed values is described in a later paragraph.

10. Entries in the T.SUBS Table are examined to find subscripted variables within the range of this DO. These entries are matched against entries in the T.SISU Table. The names within these entries

are also matched as well as the IFNs. Successful matches on all of these conditions result in a call to the subroutine X20000 for computation of the addend. This addend is then compared to a previously computed one; if there is an unsuccessful compare, control returns to obtain the next T.SUBS entry. If the compare is successful, then the T.SUBS entry will be marked used and the generated code completed with either a "STXO IFN" or a "STXO IFN+1" instruction. Processing is contained through the T.SISU, T.USSR and T.SUBS Tables until all of the subscripted variables within the range of the DO have been completed. The next DO is then obtained and the process repeated.

11. When all of the DOs have been finished, the subroutine X16000 is called to compute the addends for all of the subscripted variables. The T.LDXR Table is examined and the pointers are transferred to the T.SUBS Table. The remote compile is turned off; the coding

```
USE        .MAIN.
```

is compiled and control is returned to the Phase Two Executive Routine (PH2000).

The total address is expressed as:

$$V+((A_1-1)+C_1I_1)+((A_2-1)D_1+C_2I_2D_1)+((A_3-1)D_1D_2+C_3I_3D_1D_2)$$

and so forth for up to seven dimensions where:

V = the location of the variable
A = the subscript addend
C = the subscript constant
D = the dimension
I = the index value

All of the constant parts are computed and placed in the address at compile time or held as a constant, if necessary, to compute the rest. All parts which vary in the program, but not in the loop, are computed and added to the constant part and stored in the address at execution time. All parts which vary in the loop are placed in the index register. For example:

A three dimensioned variable in a nested DO on all three indexes. The variable appears in the innermost DO on the third index.

$XR=C_3D_1D_2I_3$

$V+(A_1-1)+D_1(A_2-1)+D_1D_2(A_3-1)$ is computed in the compiler and used as a constant in address computation (Format V+n).

$C_1I_1+C_2I_2D_1$ is computed immediately prior to the DO, added to the above constant and stored as the address of the variable.

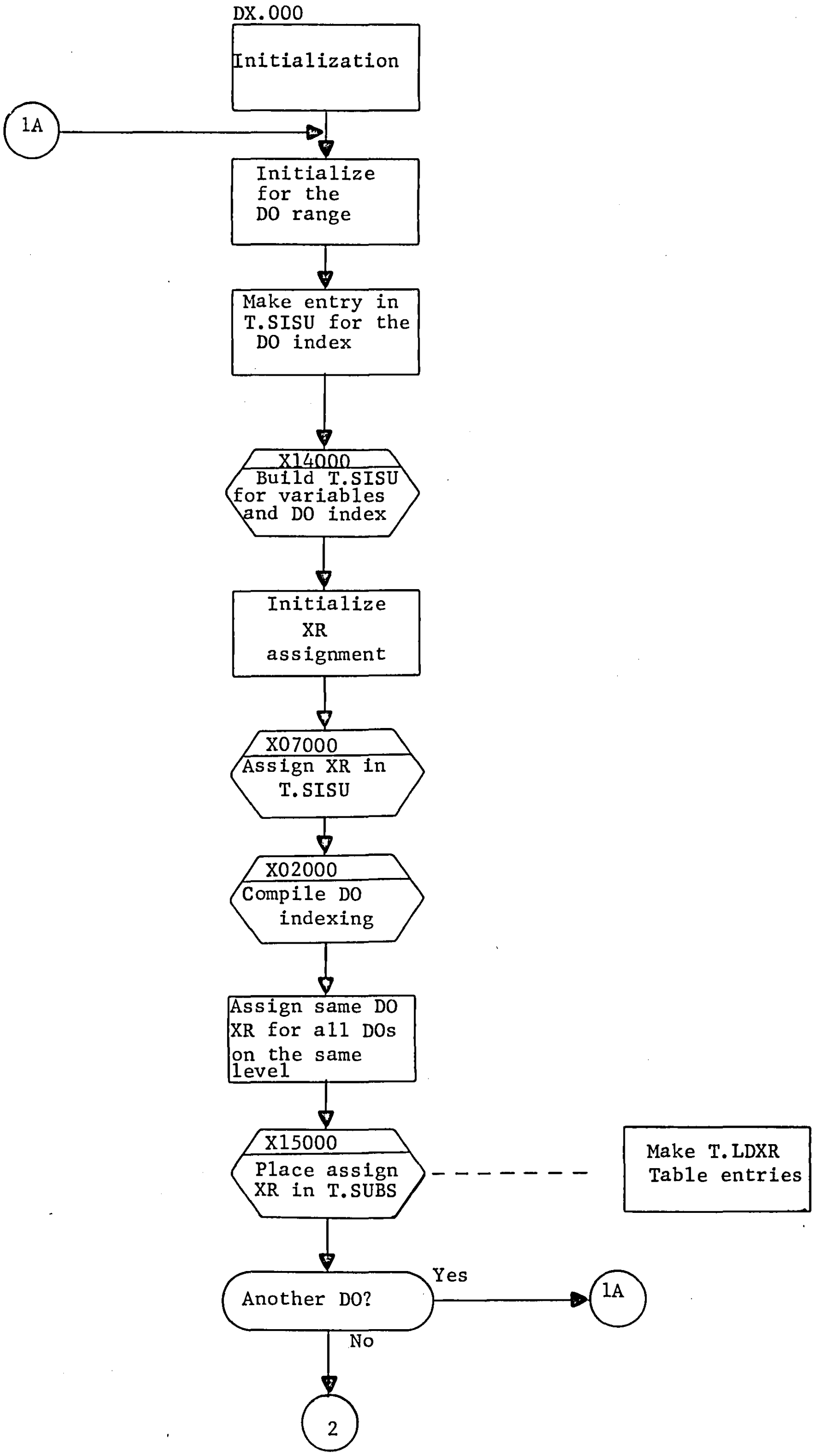


Figure 10. DO Indexer Flow Diagram

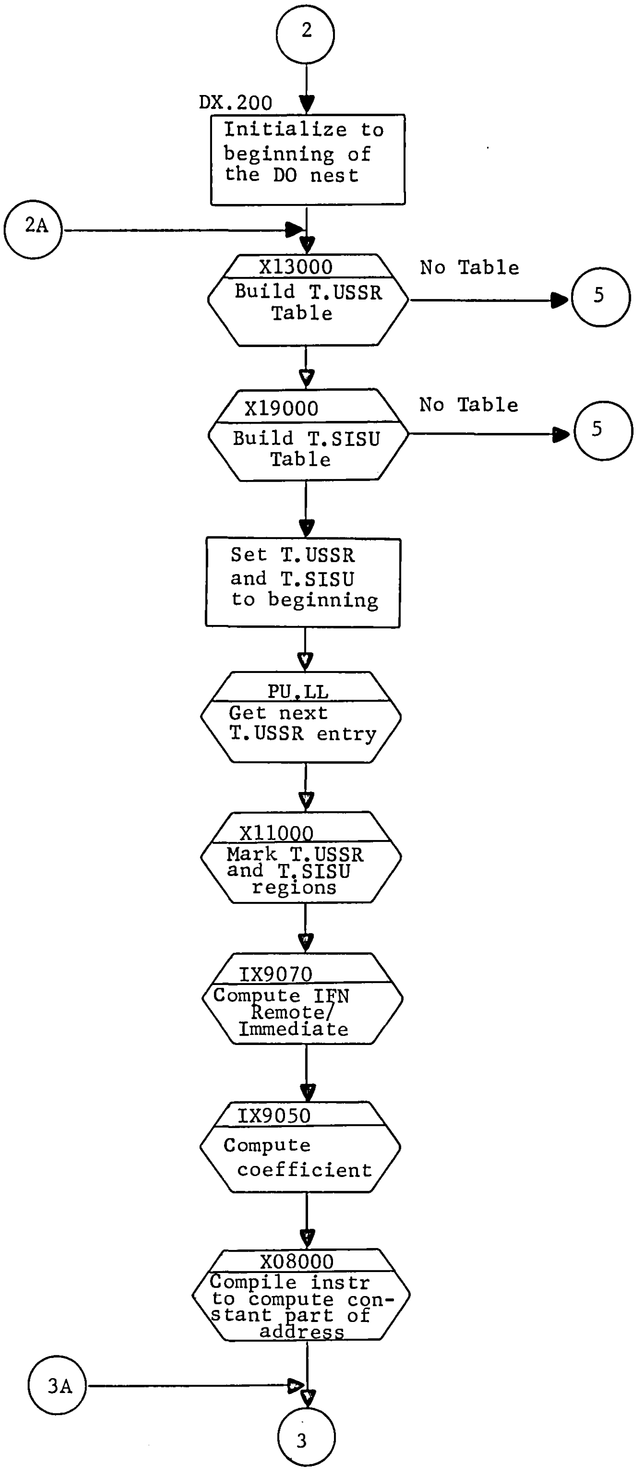


Figure 10. (continued)

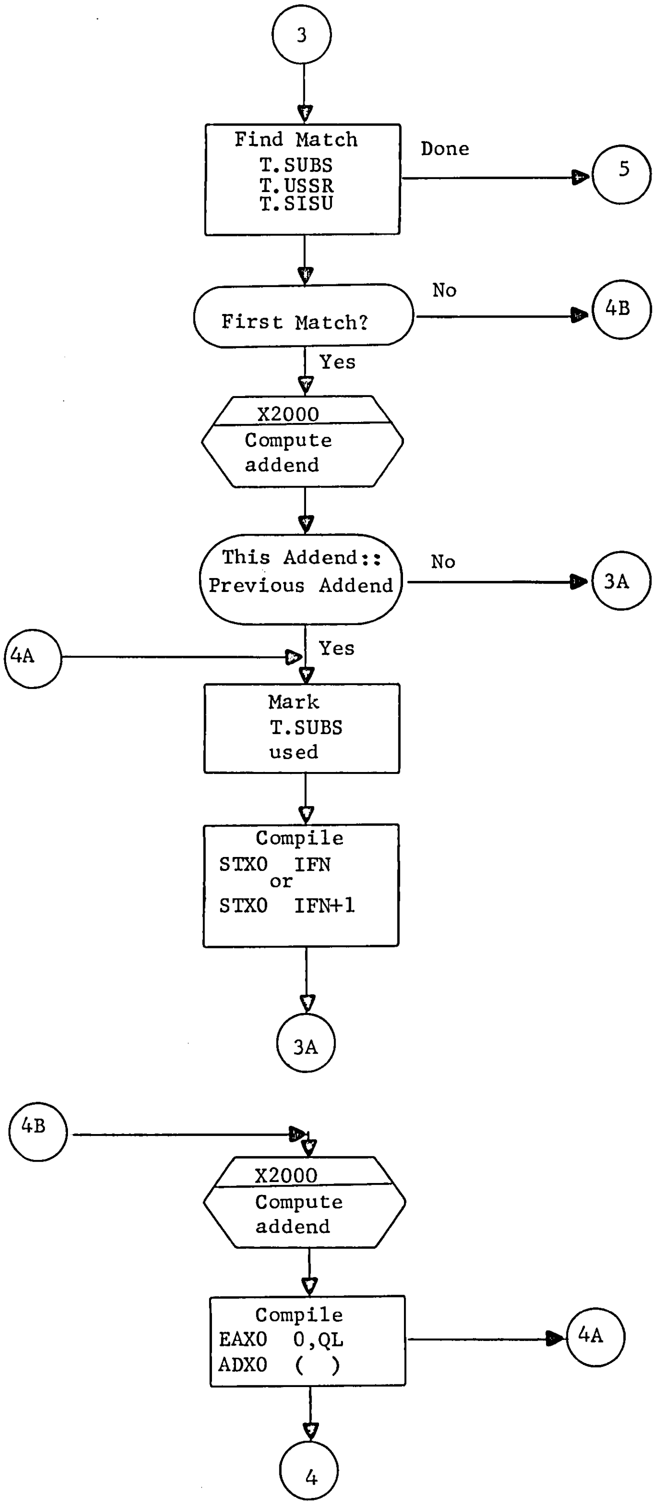


Figure 10. (continued)

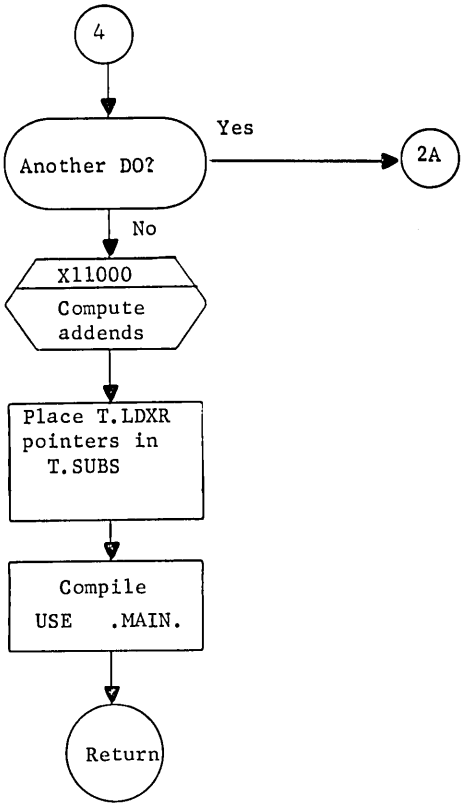


Figure 10. (continued)

Usage. The calling sequence is:

```
TSX1      DX.000
(DO Pointer)
(No Return)
```

Error Returns. There are no error returns in the calling sequence of this subroutine, but the following warning diagnostic message may be written when duplicate DO names are found.

```
DX.850   WARNING.  DO INDEX NAME -_____- APPEARS MORE THAN ONCE IN A
         DO NEST.
```

59. X01000--Addend Compile Subroutine

Purpose. This subroutine is called to compile instructions for the computation of the addend of a data address when arguments for adjustable dimensions are involved.

Method. Input to this subroutine consists of the USUB pointer and the NAME pointer for this subscripted variable.

Initially, this subroutine constructs a simple USUB Table having the dimension and the addend of the index. A loop is initialized in subroutine IXSET so that prologue instructions will be compiled for all ENTRY points that have adjustable dimensions as arguments. The end of the loop is tested at various points outside of this subroutine. The simple USUB Table is then used by a call to the X09000 subroutine which builds a Coefficient or Dimension Table which will be used for the prologue computations. Upon return from X09000, the coefficients are tested for zero or nonzero values.

If the coefficient is not zero, the subroutine X1000 is called to compile the prologue computations. When completed, there are two possible returns from the X10000 subroutine. The first return is taken when the prologue computations were already performed. In this case the following coding will be compiled.

```
USE       .PROXX
LDQ       I.+ICTR
```

where XX is the prologue number assigned to the ENTRY point being processed. Processing then continues at the same point entered when the second return from the subroutine X1000 is taken, which is also the processing path for coefficients which are zero. The instruction:

```
LDX0      ARGPOS,1
```

or, if there were previous instructions compiled to compute the addend, the instructions:

```
EAX0      0,QL
ADLX0     ARGPOS,1
```

are generated.

Usage. The calling sequence is:

```
TSX1      X01000
(Normal Return)
```

Error Returns. None.

60. X02000--DO Index Compile Subroutine

Purpose. This subroutine is called by the DO indexer, DX.000, to compile the instructions required to initialize the start and to test the end of a DO.

Method. Input to this routine consists of the DO origin, destination, name, and the parameters N_1, N_2, and N_3. The assigned index register is examined to compile either:

```
          LDXn      G,DU

              or

B.n       LDXn      **,DU
```

where G = N_1 or a computed value

Store instructions are generated when the assigned index register is the spill register. This process is repeated for each group of entries in the T.SISU Table.

Next, the DO index value is checked to see if it must be stored for future use. If not, this section is bypassed; otherwise an

```
EAQ       0,X     (X=the DO index)
QRL       18
    and
STQ       Name
    or
STQ       **
```

coding sequence is generated.

Instructions are generated to increment all index registers loaded for the DO and for all other indexing required. Prologue instructions may be compiled if arguments exist. Generally the compiled instruction is:

```
                    ADXn      DELTA,DU
                          or
                    ADXn      **,DU
where
                    DELTA = parameter N3 or a computed value.
```

The above instruction is compiled at the end of the DO loop.

The instructions compiled for the DO loop test are:

```
                    CMPXj     N2+1,DU
                          or
                    CMPXj     **,DU
    followed by
                    TNC       IXIFN,1

where               j = the DO index number
                    N2= parameter N2 of the DO statement.
```

Prologue instructions may be required.

Usage. The calling sequence is:

```
          TSX1      X02000
          (Normal Return)
```

Error Returns. None.

61. X03000--DO Parameter N_1 Compile Subroutine

Purpose. This subroutine is called by the subroutine X02000 to compile the DO indexing instructions associated with the parameter N_1.

Method. Upon entry to this subroutine, the parameter N_1 is tested to determine if it is a constant. If it is, then a test is performed to determine if prologue instructions are required. If no prologue is required, then the subroutine IX9180 is called to compile the prologue instructions. Next the subroutine IXTEST is called to determine if all of the required prologue instructions have been compiled for all of the sets of arguments. When compilation for the arguments has been completed, a USE .MAIN. instruction is compiled and the remote compile indicator is set. Return is then given to the calling program.

If the parameter N_1 is a variable, then the prologue instructions are compiled followed by the instruction EAXN 0,QL and return is made to the calling program.

Usage. The calling sequence is:

```
TSX1         X03000
(Return 1)             N1 is a variable
(Return 2)             N1 is a constant
```

When Return 2 is taken the A-register contains zero if a prologue was compiled; otherwise, the A-register will be nonzero.

Error Returns. None.

62. X04000--DO Parameter N_3 Compile Subroutine

Purpose. This subroutine is called by the subroutine X02000 to compile the DO indexing instructions involving parameter N_3.

Method. If the parameter N_3 is a constant, return is immediately made to the calling program. If N_3 is a variable, the prologue instructions will be compiled, followed by:

```
          EAX0     0,QL

      and

          STX0     DIFN+1  (For the spill register)

      or

          STX0     B.n     (Not the spill register)
```

Usage. The calling sequence is:

```
TSX1         X04000
(Return 1)             N3 is constant
(Return 2)             N3 is variable
```

Error Returns. None.

63. X05000--Saves and Restores Compile Subroutine

Purpose. This subroutine is called by the X02000 subroutine to compile save and restore instructions for index registers in nested DO loops when index register assignments overflow.

Method. Upon first call to this subroutine, the flag X05900 is set for the subroutine X02000. Later, this flag will be checked in the subroutine X02000 and a transfer will be made to location X05500 to compile the corresponding restore instructions.

The location XXXR specifies the number of index registers to be saved (restored) and the number of the first index register. These quantities are located in the right and left halves of location XXXR respectively. Save (restore) instructions are generated for all of the required registers starting with the initial register as specified in location XXXR.

Usage. The calling sequence is:

```
TSX1      X05000   Generate save instructions
(Normal Return)
```

or

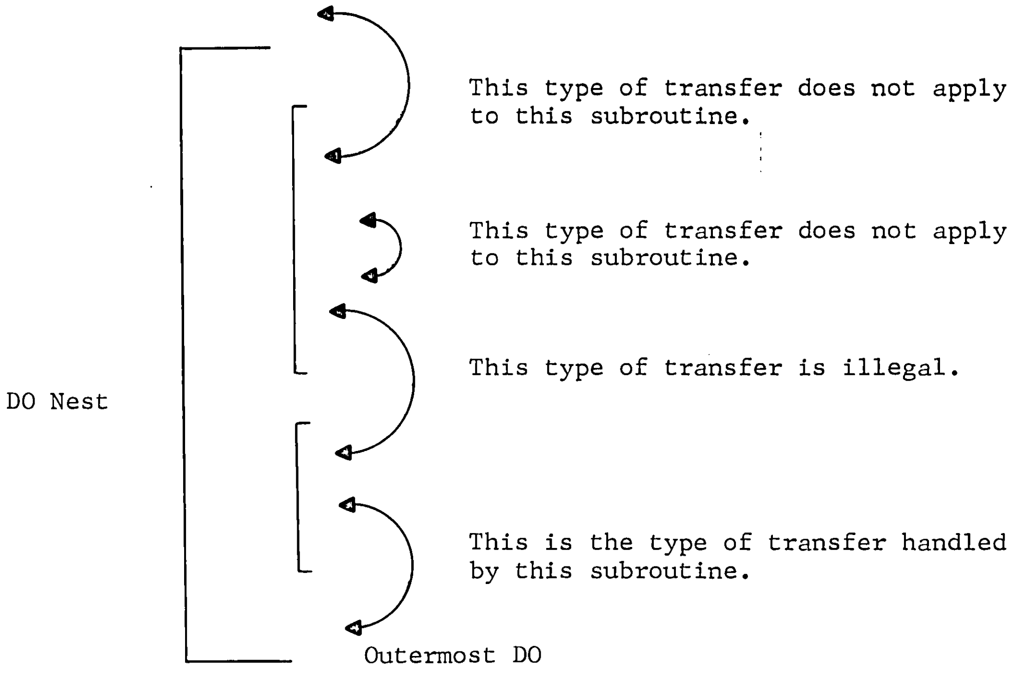
```
TSX1      X05500   Generate restore instructions
(Normal Return)
```

Error Returns. None.

64. X06000--Check Jump Table Subroutine

Purpose. This subroutine is called by the DO indexer, DX.000, to check the T.JUMP Table for transfers into and out of DOs and to compile save and restore instructions where necessary.

Method. Entries are retrieved from the T.JUMP Table using the PU.LL subroutine. If the jump origin and destination are in the range of the basic DO further tests are made. If the jump is from an inner DO to an outer DO of the same nest, then instructions to save the appropriate index registers will be generated. The following diagram illustrates some transfers in a DO nest.



The following instructions are generated:

```
               STXn         B.n+m

          and
               LDQ          (DO Name)
               EAXn         0,QL

          or

           B.n LDQ          **        For arguments
               EAXn         0,QL

followed by
               LDXn         **,DU

          and
               TRA          XXJUMP    The destination
```

It should be noted that instructions to save the DO index are not generated.

Entries are made in the T.OUTS Table so that a transfer statement causes an instruction to be compiled to go to the index register save instructions.

Usage. The calling sequence is:

```
TSX1      X06000
(Normal Return)
```

Error Returns. None.

65. X07000--Index Register Assignment Subroutine

Purpose. This subroutine is used to assign index registers to T.SISU Table information.

Method. The subroutine examines the T.SISU Table information. The most frequently used group of similar subscripts, as determined by the count in the first word of a group of T.SISU Table entries, will be assigned the next available index register. The index register designation is stored into each word (bits 3-5) of the group. This index register assignment process is repeated until each T.SISU Table group has been assigned or until the number of available index registers has been exhausted.

Initially there are six index registers available, numbers 2 through 7 in that order. For a given basic block each register in sequence is used as required. If all six are used and another is needed, then special index register instructions for loading an index register each time its use is called for must be generated by the compiler. This register is termed the Spill Register.

If less than six index registers are used within a basic block, then for a subsequent basic block the next available index register will be the first one used. Additional required index registers will be used until a total of six have been exhausted. For example, if index register 5 was the first to be assigned for a given basic block, then the subsequent assigned index registers would be 6, 7, 2, 3, and 4 in that order.

Usage. The calling sequence is:

```
TSX1      X07000
(Normal Return)
```

Error Returns. None.

66. X08000--Index Loading Instructions Subroutine

Purpose. This subroutine is called to compile computing instructions for index register loading. If arguments are present, a prologue may be required.

Method. Input to this subroutine consists of the table, IXCOEF, constructed by the X01000 subroutine. For a simple case where all of the coefficients are equal to 1, the following coding is generated:

```
          LDQ         Name
     or
          LDQ         **        For arguments.  Prologue required.
     and
          ADLQ        Name
     or
          ADLQ        **        For arguments.
```

This process is repeated until all of the dimensions have been completed.

If the coefficients are not equal to 1, then each coefficient will be checked individually. For each that is greater than 1, this subroutine will generate an Address Macro call through a call to the subroutine X10000. A complete description of the Address Macro is given in the X10000 subroutine writeup. Depending on which dimension is being processed (2nd or larger), the following code will be generated:

```
          ADLQ        E.
     and
          STQ         E.        This instruction is not generated for
                                the last dimension processed.
```

This process is repeated until all of the dimensions have been processed. When complete, the code:

```
          ADLQ        Name
     or
          ADLQ        **        For arguments
```

is generated for as many dimensions as required. Control is then returned to the calling program.

Usage. The calling sequence is:

```
          TSX1        X08000
          (Normal Return)
```

Error Returns. None.

67. X09000--Constant and Dimension Table Generator Subroutine

Purpose. This subroutine is called by the three subroutines X01000, X08000 and IX9180 to build a constant and dimension table in preparation for prologue compilation.

Method. A subscript is composed of three elements as follows:

$$S = C * I \pm A$$

where:

C = the constant
I = the index
A = the addend

The table of constants, IXCON, consists of up to seven entries, C1 through C7. An entry is made if the index name matches the input name; otherwise a zero is entered.

The table of dimensions, IXDIM, consists of up to six entries (no entry is made for the seventh dimension). If the dimension is constant, then D_n is placed in the respective entry. For adjustable dimensions, the argument number is placed in the entry. These entries are marked with a bit in bit position 0.

Usage. The calling sequence is:

```
TSX1      X09000
(Normal Return)
```

Error Returns. None.

68. X10000--Variable Dimension Prologue Compile Subroutine

Purpose. This subroutine is called by the three subroutines X01000, X08000 and IX9180 to compile prologue instructions for the computation of indexes when adjustable dimensions may be involved.

Method. Input to this subroutine consists of the Constant and Dimension Tables, IXCON and IXDIM, which are generated by the subroutine X09000. The indicator for double precision/complex is checked, and if on, C2 through C7 of the Constant Table will be doubled. The T.IPRO Table is checked to see if computation has already been made. If it has, then a return is made to the calling program with the location IXICTR containing the I storage counter for the value. No USE .PROXX instruction is generated.

If the computation has not already been made, then IXCON and IXDIM are placed in the T.IPRO Table with the respective IXICTR. The subroutine then computes the values and constructs the arguments for the ADRES macro (described later). The generated macro will be preceded by a USE .PROXX instruction and followed by a STQ I.ICTR instruction. The second return in the calling sequence is then taken to return to the calling program.

The ADRES macro is described in detail below:

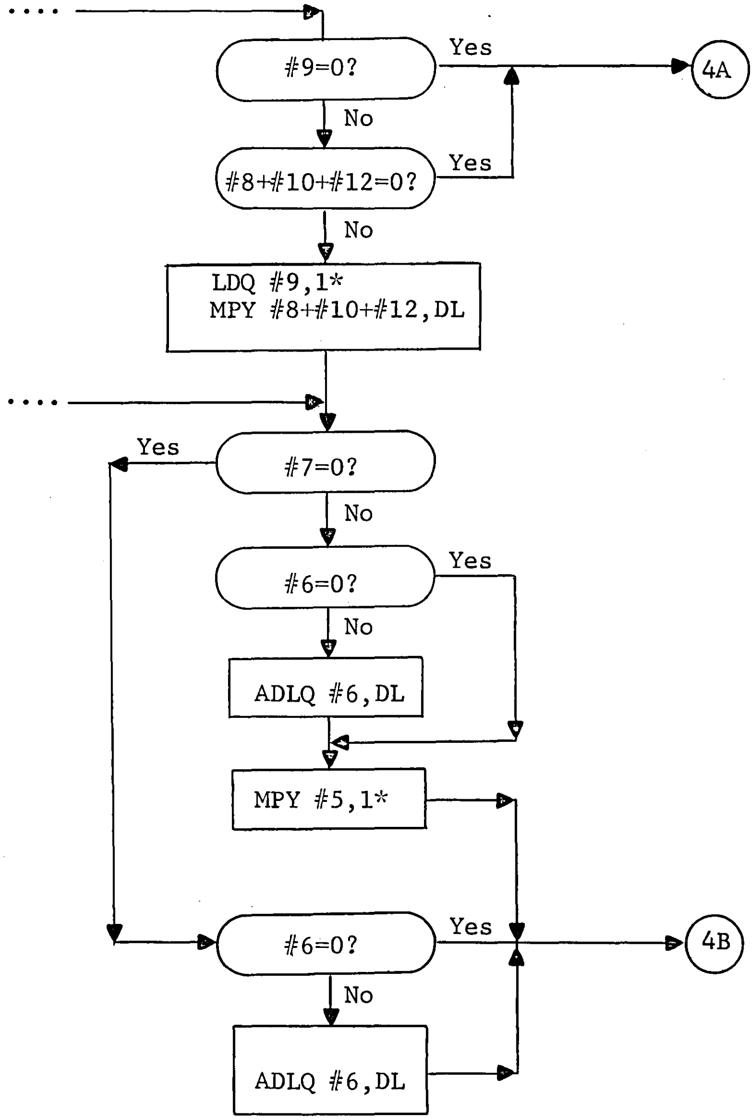
Address Macro
Compute: C1 + (D1 * C2) + (D1 * D2 * C3) . . .

Macro Arguments:

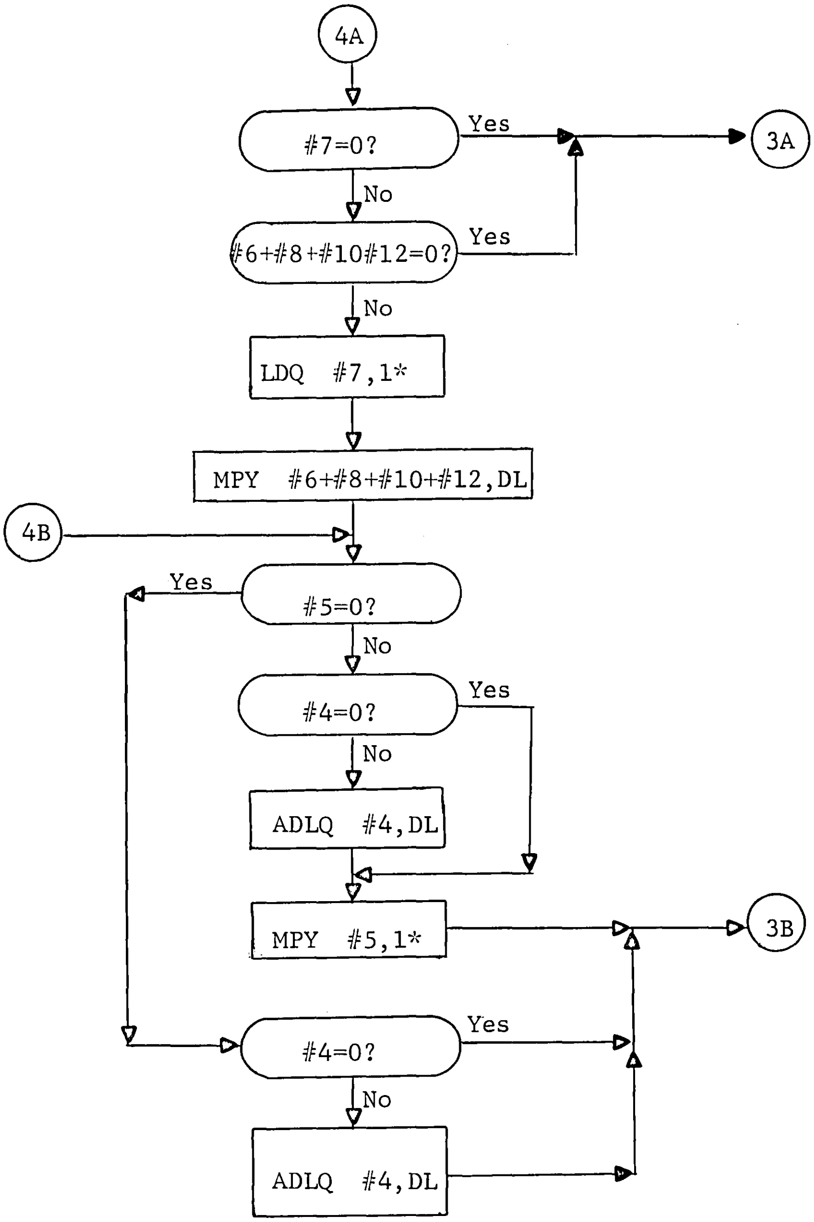
#1 = 0 or C1
#2 = 0, or C2, or D1*C2
#3 = 0, if constant D1; Argument number if variable D1
#4 = 0, C3, D1*C3, D2*C3, or D1*D2*C3
#5 = 0 if constant D2; Argument number if variable D2
#6 = 0, C4, D1*C4, D3*C4, D1*D2*C4, D2*D3*C4, or D_1*D_2*D_3*C_4
#7 = 0, if constant D3; Argument number if variable D3
. . . and so forth for all 13 arguments (seven dimensions)

The following flow diagram illustrates the Address Macro. (There are two additional flow paths for the 6th and 7th dimensions which are not shown, but are indicated by a series of dots .... . These additional paths are symmetrical to the paths for the 4th and 5th dimensions.)

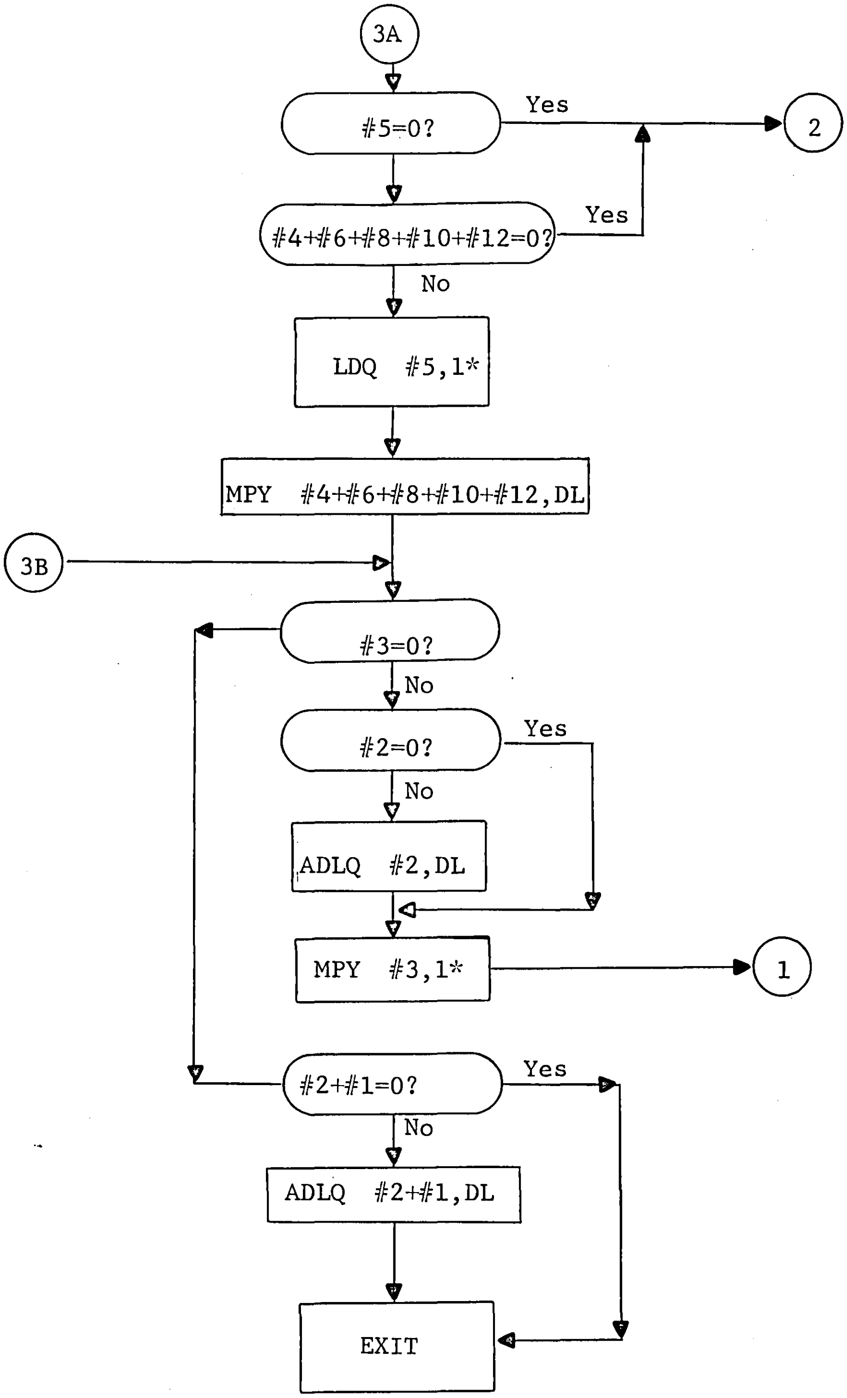
Five Dimensions

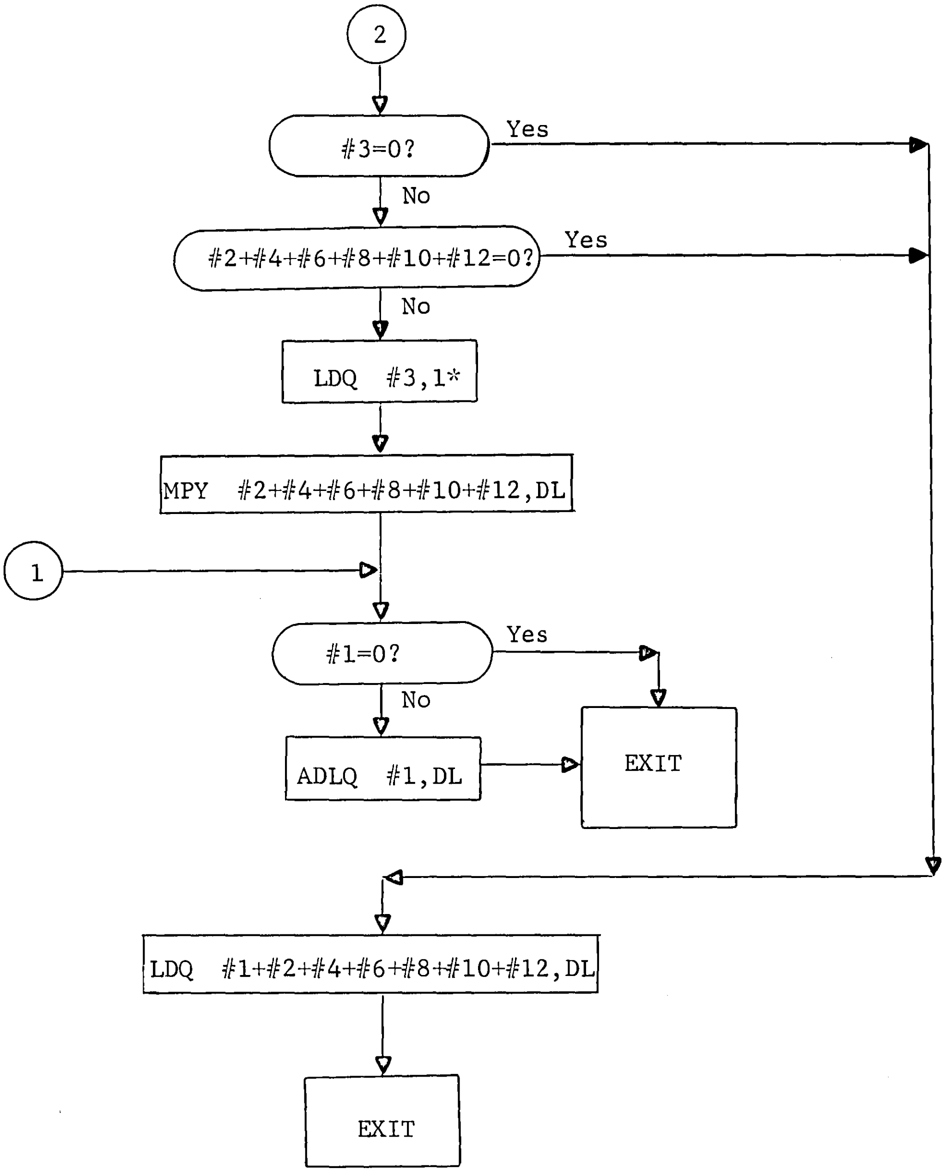


Four Dimensions



Three Dimensions




2
#3=0?
Yes
No
#2+#4+#6+#8+#10+#12=0?
Yes
No
LDQ #3,1*
MPY #2+#4+#6+#8+#10+#12,DL
1
#1=0?
Yes
No
ADLQ #1,DL
EXIT
LDQ #1+#2+#4+#6+#8+#10+#12,DL
EXIT

A listing of the Address Macro is shown below:

```
ADRES  MACRO
       INE      #13,0,11
       INE      #12,0,10
       LDQ      #12,DL
       MPY      #13,1*
       INE      #11,0,4
       INE      #10,0,1
       ADLQ     #10,DL
       MPY      #11,1*
       INE      0,0,7
       INE      #10,0,6
       ADLQ     #10,DL
       INE      0,0,4
       INE      #11,0,11
       INE      #10+#12,0,10
       LDQ      #10+#12,DL
       MPY      #11,1*
       INE      #9,0,4
       INE      #8,0,1
       ADLQ     #8,DL
       MPY      #9,1*
       INE      0,0,7
       INE      #8,0,6
       ADLQ     #8,DL
       INE      0,0,4
       INE      #9,0,11
       INE      #8+#10+#12,0,10
       LDQ      #8+#10+#12,DL
       MPY      #9,1*
       INE      #7,0,4
       INE      #6,0,1
       ADLQ     #6,DL
       MPY      #7,1*
       INE      0,0,7
       INE      #6,0,6
       ADLQ     #6,DL
       INE      0,0,4
       INE      #7,0,11
       INE      #6+#8+#10+#12,0,10
       LDQ      #6+#8+#10+#12,DL
       MPY      #7,1*
       INE      #5,0,4
       INE      #4,0,1
       ADLQ     #4,DL
       MPY      #5,1*
       INE      0,0,7
       INE      #4,0,6
       ADLQ     #4,DL
       INE      0,0,4
       INE      #5,0,11
       INE      #4+#6+#8+#10+#12,0,10
       LDQ      #4+#6+#8+#10+#12,DL
       MPY      #5,1*
```

```
INE      #3,0,4
INE      #2,0,1
ADLQ     #2,DL
MPY      #3,1*
INE      0,0,7
INE      #1+#2,0,10
ADLQ     #1+#2,DL
INE      0,0,8
INE      #3,0,6
INE      #2+#4+#6+#8+#10+#12,0,5
LDQ      #2+#4+#6+#8+#10+#12,DL
MPY      #3,1*
INE      #1,0,3
ADLQ     #1,DL
INE      0,0,1
LDQ      #1+#2+#4+#6+#8+#10+#12,DL
ENDM     ADRES
```

Usage. The calling sequence is:

```
TSX1     X10000
(Return 1)       Prologue compilation previously performed.
(Return 2)       Prologue compilation not previously per-
                 formed.
```

Upon leaving this subroutine at either return, the location IXICTR contains the I storage counter.

Error Returns. None.

69. X11000--T.USSR and T.SISU Tables Match Subroutine

Purpose. This subroutine is called to match the T.USSR and T.SISU Table entries and mark them used. The T.SISU group to which the T.USSR belongs is determined. Each and every T.USSR entry is then compared with the T.SISU group and when a match occurs the T.USSR entry is marked used.

Method. Upon calling this subroutine, a T.USSR entry is furnished as input. A T.SISU entry is obtained and the USUB pointer of the T.SISU entry is compared with the USUB pointer of the T.USSR entry. If a match does not occur when using the first entry of a group within the T.SISU Table, the next entry of the group is examined. When a match occurs, the subroutine will reposition itself to the beginning of the T.SISU Table group. (The beginning of a group in the T.SISU Table is indicated by a word with a 1 in bit position 0.) Then, starting with the first entry in the T.SISU Table group, each T.USSR entry is tested for a match on the USUB pointer. Whenever a match is found, the T.USSR entry will be marked used with a 1 in bit position 0 of the first word. In addition, the frequency flag is removed from the T.USSR entry and replaced with the IXBGCT. Also the IFN is removed from the T.USSR entry and transferred into the T.SISU Table entry, but only if it is greater than the IFN which is already there. When the end of the T.USSR Table is reached, the next T.SISU Table entry in the T.SISU Table group will be used in a similar way. This process continues until every entry in the T.SISU Table group has been compared with the T.USSR entries. Control is then returned to the calling routine.

Usage. The calling sequence is:

```
TSX1     X11000
(Normal Return)
```

Error Returns. None.

70. X13000--T.USSR Table Construct Subroutine

Purpose. This subroutine is called by the DO Indexer, DX.000, to construct the T.USSR Table for all subscripted variables in a given basic DO.

Method. This subroutine computes the points at which initialization must take place for subscripts using the information contained in the T.INTS, T.JUNK, T.JUMP or T.IODO Tables. The initialization point information is stored in the T.USSR Table.

The subroutine finds the last entry in each of the T.SUBS, T.INTS and T.JUNK Tables located just beyond the end of the basic DO. Working backwards, and always in combination based on the IFN, the subroutine finds the subscripts which belong in the T.USSR Table. When the initialization point is found, an entry is made in the T.USSR Table and marked used (-). The entry is made with the basic DO origin and end.

For example:

```
     DO 4, I = 1,12,3

     A(I)=........
     J = ......         (T.INTS)
     B (I,J) = ......
     IF (X) 11,2,11
 2   J = ......         (T.INTS) (T.JUNK)
     B(I,J) .....
11   A(I) ......        (T.JUNK)
 4   B(I,J) .....
```

The T.USSR Table will have one entry for A(I) with a frequency of 2, but there must be three separate entries of B(I,J), each with a different IFN-at-which-to recalculate the index because of the T.INTS Table and T.JUNK Table entries.

Usage. The calling sequence is:

```
TSX1     X13000
(Normal Return)
```

Error Returns. None.

71. X14000--T.SISU Table Construct Subroutine

Purpose. This subroutine is called by the DO Indexer, DX.000, to construct the T.SISU Table (similar subscripts) for all subscripts involving the DO index.

Method. This subroutine locates the subscripts in the range of the DO and examines the DO name. When found, the previous entries into the T.SISU Table are examined for a match on the USUB pointers. If there is a match, the frequency in the first word of the T.SISU Table entry is incremented and the subroutine returns to get the next subscript. If there is no match on the USUB pointers, then the subroutine computes the coefficient G of the subscript and compares it with the entries in the T.SISU Table. A match on the coefficient G causes the subroutine to put the USUB pointer into the appropriate T.SISU Table group and to increment the frequency count. If the coefficient G cannot be matched, then a new group entry is made in the T.SISU Table containing the coefficient G and the USUB pointer.

Usage. The calling sequence is:

```
TSX1     X14000
(Normal Return)
```

Error Returns. None.

72. X15000--Find T.SUBS Subroutine

Purpose. This subroutine is called by the DO Indexer, DX.000, to find all T.SUBS Table entries in a DO region.

Method. For all of the T.SUBS Table entries within the DO region, the index register assignment is taken from the corresponding T.SISU Table entry and placed in the appropriate T.SUBS Table entry. If the assigned index register is the spill register, then an entry is also made in the T.LDXR Table.

This subroutine processes nested DOs. The locations BBORG and BBEND define the range of the basic DO. NDORG and NDEND specify a nested DO. All subscripts must be found which fall between BBORG and NDORG. When that range is covered, the next DO on the same level and within the basic DO is found and the procedure is repeated. In this way all of the subscripts under the direct influence of the basic DO are located.

Usage. The calling sequence is:

```
TSX1     X15000
(Normal Return)
```

Error Returns. None.

73. X16000--T.SUBS Table Entries Addend Computation Subroutine

Purpose. This subroutine is called by the DO Indexer, DX.000, to compute addends for all T.SUBS Table entries in the DO area.

Method. This subroutine calls the subroutine IX9040 to scan the unique subscripts (USUBs). If the subroutine IX9040 returns to the first line of the calling sequence, then the subscript is a constant or the DO index. In this case, the subroutine X20000 is called to compute the addend. There are two possible returns from the X20000 subroutine. The first return results in a prologue compilation (subroutine X01000) if the addend has variable dimensions. However, if a prologue was already compiled, then a

```
STX0     IFN       (Not Spill Register)
```

or

```
STX0     IFN+1     (Spill Register)
```

instruction will be generated.

The second return from the X20000 subroutine results in placement of the addend in the right half of the second T.SUBS Table entry word and the entry is marked used.

A return by the subroutine IX9040 to the second line of the calling sequence indicates a variable subscript and a value of zero is used for the addend and the entry is marked used.

Usage. The calling sequence is:

```
TSX1     X16000
(Normal Return)
```

Error Returns. None.

74. X17000--DO Index Name Usage Subroutine

Purpose. This subroutine is called by the X02000 subroutine to check the DO index name to see if it is used in a calculation.

Method. If the DO index value is required in a calculation or for reinitialization of the index, then it is necessary to store it each time it is changed. The T.RINT and T.JUMP Tables are used to check the usage of the DO index name.

The DO is tested for the following:

1. The DO index is used in a calculation in the range of the DO.
2. DO name is in COMMON.
3. There is a transfer out of the range of the DO.
4. Inner DOs are present.

A return is made to the calling program to indicate whether or not instructions must be generated to save the DO index.

Usage. The calling sequence is:

```
TSX1      X17000
(Return 1)         Save instructions needed.
(Return 2)         Save instructions not needed.
```

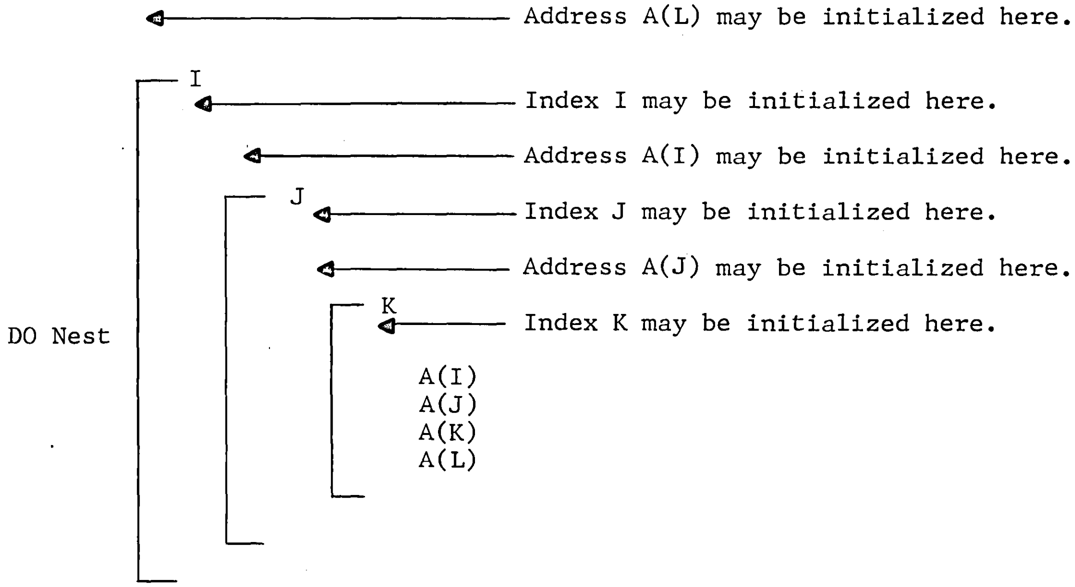
Error Returns. None.

75. X18000--Find Target Subroutine

Purpose. This subroutine is called by the X02000 and IX3000 subroutines to find the point in an area bounding or beyond the DO basic block at which address modification must occur for a subscripted variable. This point is referred to as the target point.

Method. If required, the target is located at the beginning of the lowest level DO bounding or beyond the DO basic block, containing no definition of the index variable between that DO and the DO basic block.

For example:



The initialization of addresses is always performed as far as possible outside of the DO nest in order to reduce the number of times the initialization instructions will be executed.

Usage. The calling sequence is:

```
TSX1     X18000
(Normal Return)
```

Error Returns. None.

76. X19000--Indexer T.SISU Table Build Subroutine

Purpose. This subroutine is called by the Basic Block Indexer and the DO Indexer to build the T.SISU Table from the information contained in the T.USSR Table.

Method. This subroutine groups entries from the T.USSR Table by similar USUB pointers. For T.USSR entries having identical subscripts the frequency count in the corresponding T.SISU entry is incremented. If an identical subscript is not found, the subscript values are evaluated and compared again. If a match occurs, the USUB pointer is inserted into the T.SISU group and the frequency counts are incremented. If a match does not occur, a new entry is made into the T.SISU Table and the subroutine returns for the next T.USSR entry.

Additional information on this routine is included in the descriptions of the Basic Block Indexer and the DO Indexer described in this Chapter.

Usage. The calling sequence is:

```
TSX1     X19000
(Return 1)
(Normal Return)
```

Return 1 is taken when there are no variable subscripts; hence no T.SISU Table.

Error Returns. None.

77. X20000--Addend Computation Subroutine

Purpose. This subroutine is called to compute an addend or a given USUB entry.

Method. The USUB dimension is examined. Depending on its value, 1 to 7, the calculated addend will be:

```
(A1-1)                          Single Dimension
(A1-1)+D1(A2-1)                 Two Dimensions
(A1-1)+D1(A2-1)+D1D2(A3-1)      Three Dimensions
 . . . . . . . . . . . . .
 . . . . . . . . . . . . .
 . . .  etc . . . . . .
 . . . . . . . . . . . . .
(Continued up to a maximum of seven dimensions)
```

Usage. The calling sequence is:

```
TSX1     X20000
(Return 1)
(Return 2)
```

Return 1 is taken for variable dimensions, no addend. Return 2 is taken for constant dimensions with the addend in the A-register and in the location XXADND.

Error Returns. None.

78. IX1000--Indexer T.USSR Table Build Subroutine

Purpose. This subroutine is called by the Basic Block Indexer to build the T.USSR Table.

Method. Given the origin and end of a basic block, this subroutine collects all unique subscript usage for the region from the T.SUBS Table. Each is identified as to its IFN and if and where an integer variable may have been evaluated. This latter information comes from the T.INTS Table. Duplicate entries are eliminated from the T.USSR Table in this subroutine and are indicated by the frequency count in the T.USSR Table entry. This frequency count is also used as a basis for the assignment of index registers.

Usage. The calling sequence is:

```
TSX1     IX1000
(Return 1)
(Normal Return)
```

Return 1 of the calling sequence is taken when the basic block is such that no T.USSR Table is constructed.

Error Returns. None.

79. IX3000--T.USSR Table Mark Subroutine

Purpose. This subroutine is called by the X13000 subroutine to check and mark the T.USSR Table entries.

Method. This subroutine performs four distinct functions:

1. For all integers in the basic DO, the T.USSR Table is checked for the appearance of the integer name in the USUB entry and the T.USSR Table entry is marked used.
2. For all jumps from an inner DO to an outer DO, all T.USSR Table entries are marked used.
3. For all T.USSR Table entries not initialized so far, the point of initialization (target) is found and the T.USSR Table entry is marked used.
4. All tables referenced by this subroutine are properly repositioned before exit occurs. The T.JUNK Table is positioned so that the destination just precedes the origin of the next DO. The T.INTS Table is positioned so that the IFN just precedes the origin of the next DO. The T.SUBS Table is positioned at the next unused subscript just preceding the next DO origin.

Usage. The calling sequence is:

```
TSX1     IX3000
(Normal Return)
```

Error Returns. There are no error returns in the calling sequence, but the following fatal diagnostic may be written.

```
IX3890     ILLEGAL NESTED DO NAME -      -.
```

80. IX9000--Table Backup Subroutine

Purpose. This subroutine is called by the three subroutines X13000, IX1000 and IX9140 to obtain the previous entry from a Buffered Table.

Method. There are two entry points to this subroutine, IX9000 and IX9010. The first entry assumes the table pointer is at the current entry and the user wants the previous entry. This subroutine will move the pointer backwards N words, then call the subroutine PU.LL to obtain the previous entry. If the entry point IX9010 is used, the subroutine assumes the table pointer is located at the next entry and that the user wishes to obtain the entry previous to the current one. In this case the subroutine backs the pointer up by N*2 words and then calls the subroutine PU.LL to obtain the desired entry.

Usage. The calling sequence is:

```
TSX1      IX9000    Pointer at current entry.
```

or

```
TSX1      IX9010    Pointer at next entry.
```

followed by

```
ARG       T.XXXX    Name of table
TALLY     LOC,N     where N = number of words
(Return 1)
(Return 2)
```

Return 1 is taken if there is no table or if the table is empty. Return 2 is the normal return.

Error Returns. None.

81. IX9020--Name Check Subroutine

Purpose. This subroutine is called by the subroutines X13000, X18000, IX3000 and IX1000 to perform a check if a given name is contained in a particular USUB entry.

Method. Upon entry to this routine the USUB pointer is contained in location XXUSUB and the given name is in location SXNAME. If the contents of SXNAME are zero, then a CALL statement is indicated, and an additional check is performed to determine if the USUB contains names in COMMON. If the name is in COMMON, the second return is taken in the calling sequence.

Usage. The calling sequence is:

```
TSX1      IX9020
(Return 1)          Name not found
(Return 2)          Name was found
```

Error Returns. None.

82. IX9040--USUB Entry Check Subroutine

Purpose. This subroutine is called by the X16000 and IX1000 subroutines to examine a USUB entry and determine the type of subscript.

Method. Upon entry to this subroutine, the USUB pointer is contained in location XXUSUB. The USUB entry is examined to determine if the subscript is constant (including the DO index name) or if the subscript is variable (other than the DO index name). Two returns are provided in the calling sequence for either case.

Usage. The calling sequence is:

```
TSX1      IX9040
(Return 1)          Constant subscript (including DO name)
(Return 2)          Variable subscript (other than DO name)
```

Error Returns. None.

83. IX9050--Compute Coefficient Subroutine

Purpose. This subroutine is called to build a table of coefficients.

Method. Input to this subroutine consists of the USUB pointer and the DO name. The Index name is compared to the DO name. It should be noted that the DO name for a Basic Block is equal to zero. If the comparison produces a match, then this dimension will be bypassed. If the comparison does not produce a match (as is always the case in a basic block), then the Index name will be stored and the subroutine IX9060 is called to compute the numeric coefficient. Upon return the coefficient and the Index name are placed in the simple table IXCOEF. This process is repeated for as many dimensions as are present.

Usage. The calling sequence is:

```
TSX1     IX9050
(Normal Return)
```

Error Returns. None.

84. IX9060--Compute Numeric Coefficient Subroutine

Purpose. This subroutine is called by the X14000 and IX9050 subroutines to compute the numeric coefficient for a given USUB and Name.

Method. Upon entry to this subroutine, the USUB pointer is contained in location XXUSUB and the Name is contained in location SXNAME. If the dimension is variable, an immediate return to the calling program is taken.

Usage. The calling sequence is:

```
TSX1     IX9060
(Normal Return)
```

Upon return from the subroutine, the numeric coefficient is contained in the location XXGAMA and in the A-register; if a variable dimension, the A-register is zero.

Error Returns. None.

85. IX9070--IFN Location Field Compile Subroutine

Purpose. This subroutine is called to set the IFN for compilation in the location field of a GMAP symbolic instruction.

Method. The input IFN is located in cell IXIFN. It is compared to the current IFN. If they are equivalent, the remote compile is turned off. A test is then performed to determine if the IFN has already appeared in the location field. If it has, the subroutine returns control to the calling program. If not, the IFN is compiled into the location field of a GMAP instruction and held. The "Compiled IFN Flag" is then set, the remote compile is turned on and control is transferred to the calling program.

Usage. The calling sequence is:

```
TSX1     IX9070
(Normal Return)
```

Error Returns. None.

86. IX9080--B.n Compile and Hold Subroutine

Purpose. This subroutine is called by the subroutines DX.000, X02000, X03000, X04000, X06000 and X08000 to compile and hold the B.n for the location field of a GMAP instruction.

Method. This subroutine calls the GG subroutine to place B.n in the location field for the next instruction being generated.

Usage. The calling sequence is:

```
TSX1     IX9080
(Normal Return)
```

Error Returns. None.

87. IX9090--T.BGIN Table Entry Subroutine

Purpose. This subroutine is called to make an entry in the T.BGIN Table to indicate where to collate instructions from the remote compile table (T.COLT) into the GMAP code being written on the G* file.

Method. Upon entry to this subroutine, the input IFN (in location IXIFN) is compared with entries already in the table. If there is a match, the compile flag is cleared and begin count (BGCT) is moved into the entry in the T.BGIN Table. When there is no match, an entry will be made in the T.BGIN Table. This new entry will consist of the IFN and the BGCT. When the input IFN (IXIFN) is equal to the current IFN, the entry in the T.BGIN Table will be marked with a 1 in bit position 0. The equality of the IFNs indicates the beginning of a basic block.

Usage. The calling sequence is:

```
TSX1     IX9090
(Normal Return)
```

Error Returns. None.

88. IX9100--T.SISU Table Push Down Subroutine

Purpose. The subroutine is called by the X14000 and X19000 subroutines to place an entry in the T.SISU Table.

Method. The T.USSR Table is searched to build the T.SISU Table. The T.SISU Table consists of groups of entries of similar subscripts. The first word of a T.SISU Table group is indicated by a minus sign. As additional words of the T.USSR Table are examined, it may be necessary to add words to a group in the T.SISU Table. This requires that the rest of the entries in the T.SISU Table be pushed down.

Usage. The calling sequence is:

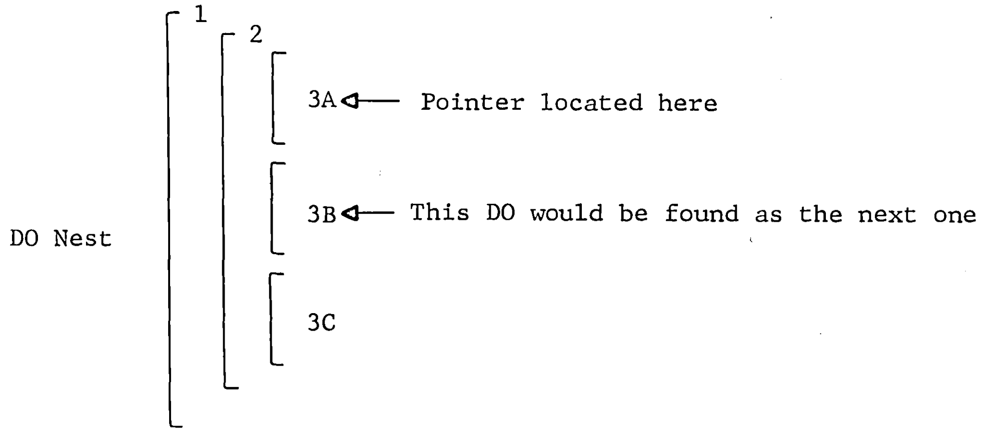
```
TSX1     IX9100
(Normal Return)
```

Error Returns. None.

89. IX9120--Next DO Entry Subroutine

Purpose. This subroutine is called by the DX.000, X13000, X14000, X15000 and X16000 subroutines to find the next DO entry.

Method. This subroutine searches the T.IODO Table for the next DO entry with an origin greater than the current DO end. When the entry is located, the origin and destination are stored in NDORG (next DO origin) and NDEND (next DO end) respectively. If the end of table is encountered during the search before an entry is found, then the origin and destination will be set to all bits (octal 777777777777).



Usage. The calling sequence is:

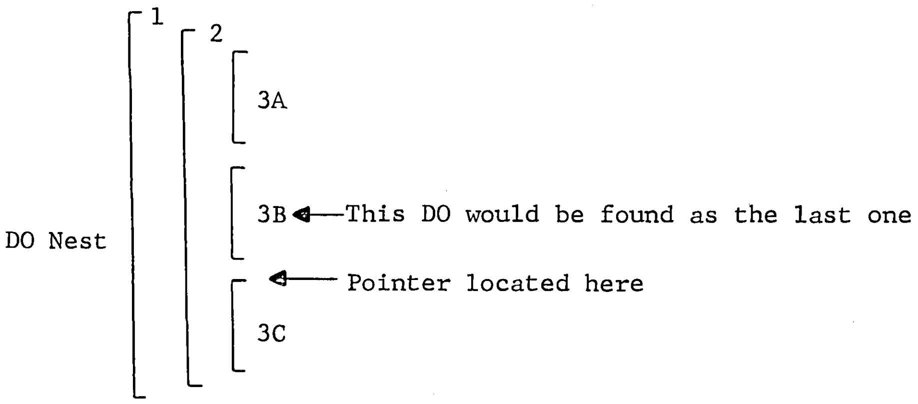
```
TSX1      IX9120
(Normal Return)
```

Error Returns. None.

90. IX9140--Last DO Entry Subroutine

Purpose. This subroutine is called by the X13000 and IX3000 subroutines to find the last DO entry.

Method. This subroutine searches backwards within the DO nest looking for the last DO entry with a destination less than the current DO origin and greater than the previous DO nest. When found, the DO origin and destination will be saved in NDORG (next DO origin) and NDEND (next DO end) respectively. If the table runs out, the beginning is encountered since this is a backwards search, a value of zero will be used for the origin and destination.



Usage. The calling sequence is:

```
TSX1      IX9140
(Normal Return)
```

Error Returns. None.

91. IX9160--Variable Name/Argument Table Subroutine

Purpose. This subroutine is called by the X01000 and X09000 subroutines to find the position of the variable name (location ARGNAM) in the Argument Table (T.ARGS).

Method. Upon entry to this subroutine the T.ARGS Table is searched for the argument list that is indicated in location IXALN. If this list is not found, then the first return is taken. Within the proper list a search is made for the variable name. If the variable name is not found, a return is made to the first line of the calling sequence. If the variable name is found, then the second line of the calling sequence is taken with the relative table position contained in the A-register. The relative table position is actually the relative position plus one to account for the TRA instruction and the error linkage in the calling sequence.

Usage. The calling sequence is:

```
TSX1     IX9060
(Return 1)
(Return 2)
```

where:

Return 1 = variable name not found.
Return 2 = variable name was found.

Error Returns. None.

92. IX9180--N_1/N_3 Constant Subroutine

Purpose. This subroutine is called by the subroutines X03000 and X04000 when parameters N_1 or N_3 of a DO are constant to construct a simple coefficient table and to compile any prologue instructions required by the DO.

Method. The subroutine PU.LL is called to obtain an entry from the T.SISU Table. The USUB pointer and DO name are saved. The subroutine IXSET is called to set up a loop to go over the argument lists. The X09000 subroutine is called to construct the Coefficient and Dimension Table. Upon return the subscript coefficient value is multiplied by N. Finally the subroutine X10000 is called to compile the prologue if required and return is given to the calling program.

Usage. The calling sequence is:

```
TSX1     IX9180
(Normal Return)
```

Error Returns. None.

93. IX9200--DO Variable Parameters Compile Subroutine

Purpose. This subroutine is called by the subroutines X03000 and X04000 to process the parameters N_1 or N_3 of a DO, if they are variable.

Method. This subroutine compiles any prologue instructions which may be necessary because the parameters of the DO are variable.

Usage. The calling sequence is:

```
TSX1     IX9200
(Normal Return)
```

Error Returns. None.

94. IX9220--USUB Entry Replace Subroutine

Purpose. This subroutine is called by the IX9200 subroutine to examine the USUB entry.

Method. This subroutine contains two separate entry points. Each entry point enables the subroutine to perform a different function.

The entry point IX9220 causes the subroutine to examine the USUB entry and replace any given N name (contained in location IXNN) with the current DO name.

The entry point IX9225 causes the subroutine to examine the USUB entry and replace the current DO name with the N name.

Usage. The calling sequence is:

```
          TSX1     IX9220
          (Normal Return)
     or
          TSX1     IX9225
          (Normal Return)
```

Error Returns. None.

95. IXSET--Loop Set Subroutine

Purpose. This subroutine is called by the subroutines X01000, X08000 and IX9180 to initialize a loop to compile prologues for all argument lists.

Method. Upon entering this subroutine the indicator (location IXCNTR) for incrementing the "n" part of B.n is reset. In preparation for an error check the count of noncompiles is set to zero. The count of prologue compiles is set to zero initially. Subsequent entries to this subroutine store new values as the XX portion of a USE .PROXX instruction. Return is given to the calling program.

Usage. The calling sequence is:

```
TSX1     IXSET
(Normal Return)
```

Error Returns. None.

96. IXTEST--Loop Test Subroutine

Purpose. This subroutine is called by the subroutines BX.000, DX.000, X03000, X04000, X08000 and X16000 to test for the end of the loop when compiling prologues for argument lists.

Method. Upon entering this subroutine, a test is made to see if all of the prologues for ENTRY statements have been done for all argument lists by comparing the number of ENTRYs (location NO.ENT) against the count of prologues compiled (location IXALN). If all of them have not been done, a one is added to location IXALN and the subroutine IXSET is reentered at symbolic location IXSET1.

If all of them have been done, then a check on the number of prologues compiled is performed. If none have been compiled, a fatal diagnostic message is written and control returns to the calling program. If one or more prologues have been compiled, the location IXCNTR is checked to see if the "n" part of B.n needs incrementation. If stepping is required, it is done and return is made to the calling program. If no stepping is required, control is immediately given to the calling program.

Usage. The calling sequence is:

```
TSX1     IXTEST
(Normal Return)
```

Error Returns. There are no error returns in the calling sequence, but the following fatal diagnostic may be written.

```
IXTST2    ARGUMENT LISTS ARE IMPROPER.
```

Progress Is Our Most Important Product

GENERAL ELECTRIC

Computer Department • Phoenix, Arizona

LITHO IN U.S.A.