IBM

PROGRAMMING SYSTEMS ANALYSIS GUIDE

709/7090 32K FORTRAN

Preliminary Copy

©1961 by International Business Machines Corporation
Printed in U. S. A. Form R23-9673

TABLE OF CONTENTS

1.00.00	FORTRAN MONITOR	1
1.01.00	Introduction	2
1.02.00	Start Card and 1-CS	7
1.02.01	Listing of Fortran Start Card	8
1.02.02	Listing of 1-CS	9
1.03.00	Card to Tape Simulator	10
1.04.00	Dump	13
1.05.00	Sign On	14
1.06.00	Fap	16
1.07.00	Scan	17
1.08.00	BSS Control	21
1.09.00	Machine Error	23
1.10.00	Source Error	24
1.11.00	Tape Mover	25
2.00.00	FORTRAN COMPILER	29
2.01.00	Introduction	30
2.01.01	Section I	31
2.01.02	Section I'	36
2.01.03	Section I"	42
2.01.04	Section II	47
2.01.05	Section III	53
2.01.06	Introduction Sect. IV and V	59
2.01.07	Section IV	61
2.01.08	Section V	67
2.01.09	Section V'	74
2.01.10	Section VI	76
2.02.00	Fortran Library	97
2.02.01	Input/Output Library	100
2.03.00	General Diagnostics	106
2.04.00	Tables Generated By Fortran	111
2.04.01	ASSIGNED CONSTANT	111
2.04.02	CALLFN (CALLNM)	111
2.04.03	CLOSUB	112
2.04.04	C.I.T (Compiled Instruction Tables)	112
2.04.05	COMMON	113
2.04.06	DIM	114
2.04.07	DOFILE (C)	115
2.04.08	DOTAG (B)	115
2.04.09	EIFNO	115
2.04.10	END	115
2.04.11	EQUIT	116
2.04.12	FIXCON	117

2.04.13	FLOCON	117
2.04.14	FMTEFN	117
2.04.15	FORMAT	117
2.04.16	FORSUB	118
2.04.17	FORTAG	118
2.04.18	FORVAL and FORVAR	119
2.04.19	FRET	119
2.04.20	HOLARG	121
2.04.21	NONEXC	121
2.04.22	PREDESSOR	122
2.04.23	SIZ	122
2.04.24	SUBDEF	123
2.04.25	SUCCESSOR	123
2.04.26	TAU	123
2.04.27	TDO	125
2.04.28	TIEFNO	125
2.04.29	TIFGO	126
2.04.30	TIFGO FILE	128
2.04.31	TRAD	128
2.04.32	TSTOPS	129

APPENDICES

A Fortran Tape Status By Section	130
B Edit Record Chart	133

FORTRAN MONITOR 1.00.00

INTRODUCTION 1.01.00

The FORTRAN System tape is written as four files. (See Figure 1.) The first file constitutes the major portion of the monitor. Contained as individual records in this first file in order as they appear on tape are 1-CS, Card to Tape Simulator, Dump, Sign On, FAP I and II, Scan, BSS Control, Machine Error record, and the Source Error record. (See Figure 2.). The second file is the FORTRAN Compiler plusthe Tape Mover record and an additional BSS Control. This second BSS Control is used to save time if execution is desired, otherwise the System tape would have to be backspaced to the first file to read BSS Control. All the library subroutines that FORTRAN and FAP require are contained in the third file. The FORTRAN diagnostic routines and error messages are contained in the fourth and last file on the System tape.

The FORTRAN Monitor System may be used in both the monitor mode and single compile mode. In the single compile mode, only FORTRAN compilation can be done. It also might be pointed out that in the single compile mode the only record used in the monitor file will be the Card to Tape Simulator, from this control passes directly to FORTRAN in the second file.

Since the standard method of operation is in the monitor mode, the description of the system will be from this standpoint, and only under special conditions will the differences be pointed out.

Operating in the monitor mode, a large number of jobs may be stacked on the input tape. The limiting factor to the number of jobs that may be stacked is the capacity of the tape reel.

A job may be defined as a basic unit that will be processed by the monitor at any one time. It will consist of at least one, but can contain many programs. The job can be in one of two states, either Execute or Non-Execute. As an Execute job, all programs in the job must be related to one another, these will be executed immediately after any assembly or compilation that is required. As a Non-Execute job, the programs need not be related, since only assemble or compilation can be done. For a more comprehensive explanation of job processing, reference should be made to the 709/7090 FORTRAN Monitor Reference Manual, Form C28-6065.

The FORTRAN Start Card is used to initialize the System. This will rewind the System tape, load the first record, which is 1-CS. 1-CS is a general purpose tape loading routine which remains in lower core storage at all times during an assembly or compilation. It is used to load all monitor and FORTRAN records, and after loading will transfer control to their respective entry points.

Once 1-CS is loaded, it will in turn load the Card to Tape Simulator. The Card to Tape Simulator test the card reader for presence of cards. If the

hopper is empty, the input is assumed to be from tape. This tape will be A2 if in the monitor mode, if not in monitor mode, i. e., single compile, the input tape is B2. If cards are found in the reader, a card to tape simulation will follow until all cards have been read and the End of File is met.

At this point the next record (DUMP) is skipped, control is again given to 1-CS to read in the Sign On record. The Sign On record will read and print the first record onthe input tape, this should be the I. D. card. It is in the Sign On record that the customer may insert his own coding to process accounting information that might be contained in the I. D. card. If an accounting clock is available in the machine, this also could be read at this time.

At the completion of Sign On processing, the next two records (FAP) are skipped, control again passes to 1-CS to read the Monitor Scan record. Monitor Scan will read all remaining control cards, and set up the proper indicators for processing by the appropriate routine. Control now passes via 1-CS to FAP if a FAP control card has been encountered, or to BSS Control if the remaining cards on the input tape are binary and the job is to be executed, if neither of these conditions exist the monitor assumes a FORTRAN compilation, therefore, control passes to FORTRAN in the second file.

After the completion of a FAP, FORTRAN or BSS relocation, control passes back to the Monitor Scan record. This will continue until all pro-programs in the job have been processed. At this time if execution is desired, control once again passes to BSS control to load the relocated program and begin its processing. If this job were not to be executed, control will pass back to the Sign On record, instead of BSS Control, to begin the next job.

This whole process continues until no more jobs are left to process, at this point the card reader will be selected and the program hangs up. The operator now has the option of removing the output tape, loading a new input tape, and re-initializing the system with the Start Card, or depressing the card reader start key for a final stop.

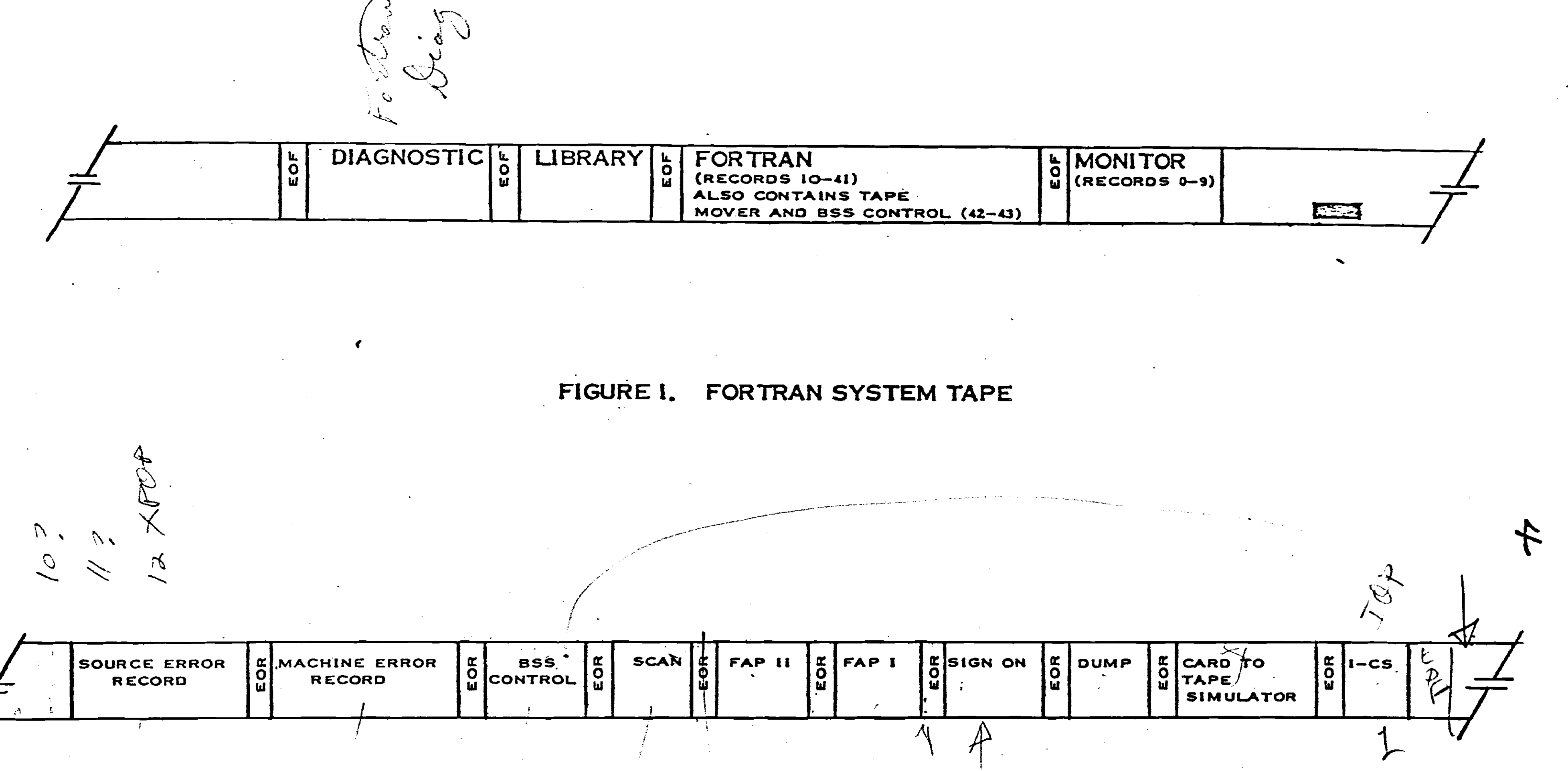


FIGURE 1. FORTRAN SYSTEM TAPE

FIGURE 2. FORTRAN MONITOR

LOGIC OF THE FORTRAN SYSTEM

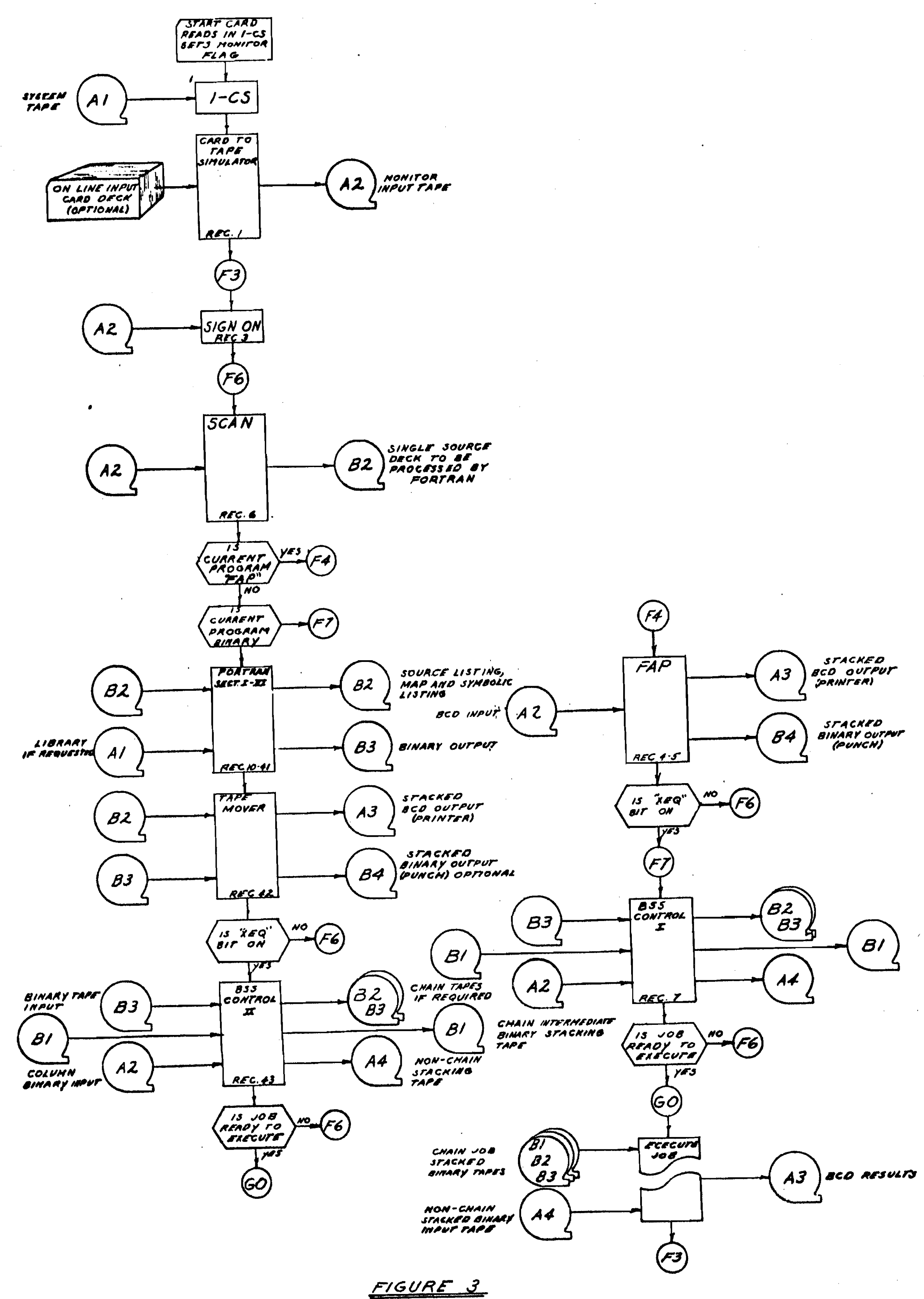


FIGURE 3

The Start Card is a one card self loading binary card. It is used to initialize the FORTRAN monitor system, for batch compiling and/or execution.

To set the FORTRAN monitor in motion, the Start Card is placed in the card reader, the load cards button is depressed, a bootstrap loader will then read in the remainder of the START CARD. Control is passed to the location 144_8, the first instruction beyond the bootstrap. The System tape is first rewound and then the first record on the tape is read in. The first record is the FORTRAN loader called 1-CS, the tape read operation is checked for a redundancy error, if one occurred the tape is backspaced and another attempt is made to read 1-CS. If three unsuccessful attempts are made the program will HALT.

If the Start button is depressed three more attempts will be made and so on. When the read operation is successful a flag bit is set in the sign position of location 42_8 to indicate that the jobs that follow will be processed in the monitor mode. Control now passes to location 50_8 in 1-CS.

1-CS will be used to load all monitor records and FORTRAN Executive records. If redundancy checks occur while reading any record the program will HALT, only one attempt will be made per redundancy check.

The first two words of any System record contain the information necessary to indicate where loading begins, how many words are to be read and where to transfer control when loading is complete. Also contained in word two is the FORTRAN or monitor record number times ten that is being loaded, this will be used if a call is made to the Diagnostic record to ascertain the type of diagnostic message that should print.

The make up of the first two words of any FORTRAN or Monitor record are:

WORD 1	011	WORD COUNT	//////	LOAD ADDRESS
WORD 2	001	RECORD NUMBER X10	//////	TRANSFER ADDRESS

Also contained with 1-CS is the diagnostic record caller. This routine will be used by FORTRAN Section I' through Section VI to call the Diagnostic file (file 4) for any error, source or machine. Prior to reading in the Main diagnostic record the area that it will occupy is saved as a single record on tape A3.

LISTING OF FORTRAN START CARD

The following is a listing of the Start Card and shows the instruction sequence in the card and also the location in storage:

SEQUENCE ON CARD	STORAGE LOCATIONS	OPERATION	ADDRESS, TAG, DECREMENT
0	0	IORT	144, 0, 77777
1	1	TCOA	1
2	2	TTR	144
3	144	LTM	
4	145	AXT	3, 1
5	146	REWA	1
6	147	RTBA	1
7	150	RCHA	160
10	151	TCOA	151
11	152	TRCA	156
12	153	CAL	153
13	154	STP	42 (Monitor Flag)
14	155	TRA	1
15	156	TIX	146, 1, 1
16	157	HTR	145
17	160	IOCP	0,,3
20	161	TCH	0

The following is a listing of 1-CS and shows the instruction sequence on tape and also the locations in storage:

SEQUENCE ON TAPE	STORAGE LOCATIONS	OPERATION	ADDRESS, TAG, DECREMENT
0	0	IORT	23,,77777
1	1	TCOA	1
2	2	TRA	50
3	23	RTBA	1
4	24	RCHA	37
5	25	RTBA	1
6	26	RCHA	37
7	27	WTBA	3
10	30	RCHA	41
11	31	RTBA	1
12	32	RCHA	41
13	33	TCOA	33
14	34	TXI	157
15	35	BSRA	3
16	36	HTR	36
17	37	IORPN	0, 2, 77777
20	40	TCH	37
21	41	IORT	156, 0 4704
22	42	PZE (Monitor Flag Cell)	
23	43	PZE (Chain Flag Cell)	
24	44	PZE (Chain Flag Cell)	
25	45	PZE (Job Lines Output Counter)	
26	46	BSRA	1
27	47	HPR	77777,7
30	50	RTBA	1
31	51	RCHA	54
32	52	TCOA	52
33	53	TRCA	46
34	54	IOSP	55, 0, 2
35	55	IORT	**,,** } [1]
36	56	TXI	**,,** } [1]

[1] First two words of any monitor or FORTRAN record.

CARD TO TAPE SIMULATOR 1.03.00

After the card to tape simulator has been read into storage, by 1-CS, control is passed to it. This will be the only time that this program is used. The monitor flag in 1-CS is tested to determine, if this is a monitor or single compile operation. The card reader is then selected, if an End-of-File is sensed on the first read cycle the program assumes the input is from tape. If the input is from tape, control passes to record 3, Sign On, to begin processing the first job. If we are not in the Monitor mode, card to tape simulator will pass control directly to FORTRAN record 10. If the card reader End-of-File was not sensed, a simulated card to tape operation will follow. Cards with a 7-9 punch in column 1, indicate that they are column binary and therefore must be converted to Row binary format before they may be written on tape. Cards with an 7-8 punch in column 1 will not be written on tape but cause an EOF to be generated on tape. All other cards are considered to be Hollerith and are checked for illegal punching prior to being transcribed onto tape. Illegal Hollerith punching will cause the machine to stop with a HPR 7777, in the storage register. The rules for correcting and reloading the card reader are analogous to a READ CHECK stop on the off line card reader.

The cards are read in double buffered, to allow the card reader to operate at full speed.

When the final EOF is sensed on the card reader, an EOF will be written on tape. The input tape is then rewound, record 2 is skipped and control is passed to, Sign On, record 3.

If the monitor flag is off, at the termination of the card to tape simulation, the remainder of File one will be skipped and control passed directly to, FORTRAN (record 10).

CARD TO TAPE SIMULATOR

MONITOR RECORD #1

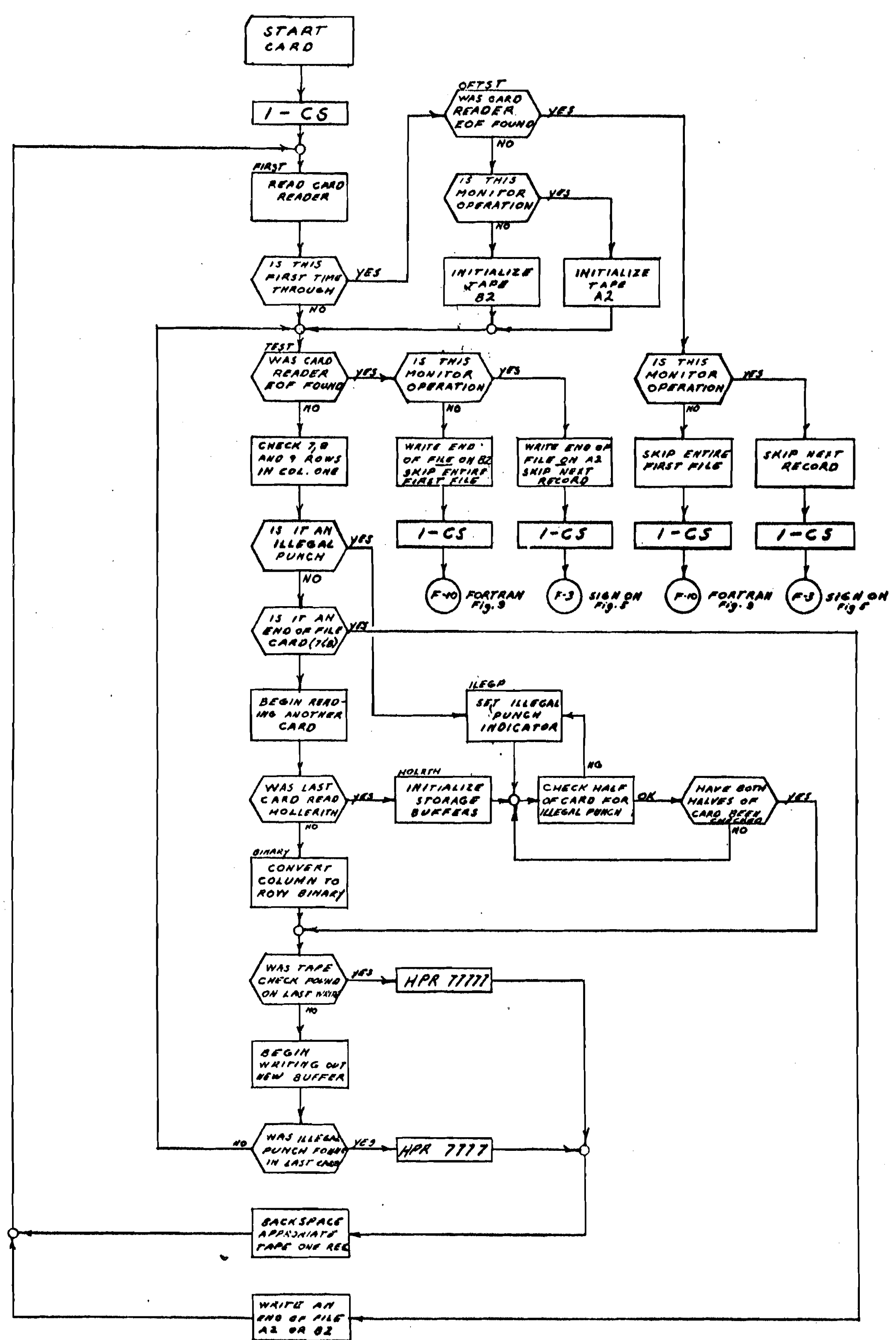


FIGURE 4

DUMP RECORD WITH DUMP CARD

The Dump is used when trouble is experienced during compilation or execution of a job. This will give the Programmer or Customer Engineer a printed record of the contents of core storage and the OP panel indicators, which can be used to ascertain the possible cause of trouble.

Actually the Dump record can be called by three different methods. Two of these are through the use of FORTRAN statements, these are CALL DUMP (?) and CALL PDUMP (?). For the description and use of these see the 709/7090 FORTRAN MONITOR, Reference Manual (Form Number C28-6065). The third method of calling the Dump record is by using the Dump card, which should be available at all installations using the FORTRAN System.

The Dump Card will ohly destroy the first three locations in storage. This is accomplished by initiating a Write tape to dump the first 3500_8 locations of storage, then delaying the information on the card from coming in until the locations necessary for storing this information have been dumped on to tape. This delay is accomplished by reading the first fifteen words of the card into locations 0, 1 and 2 and causing the program to transfer to itself at location 1. After this delay the System tape is rewound, the first and second records skipped and then the Dump record is read in, at which time control is passed to it.

The Dump program will cause the entire contents of memory to be written onto tape A3, following any FORTRAN output from this job or from previous jobs. The storage entry switches are interrogated to determine whether or not mnemonics are to be included and to see what course of action to take after completing the dump. These options are fully explained in the 709/7090 FORTRAN MONITOR, Reference Manual (Form Number C28-6065).

SIGN ON (RECORD 3) 1.05.00

The Sign On record is called only at the beginning or end of job. The number of lines of output from the last job is picked up from the line count storage cell in 1-CS. This number if greater than zero, is converted to decimal and written on and off line. If Sense Switch 6 is up, an End of File will be written on tape B4, the stacked column binary punch tape. At this time a test is made to determine if the input tape is positioned to read a new job (at the beginning of file). If not, the tape is spaced forward until an End of File is encountered. The first record of the file is then read and it is then determined if the first character is an asterisk. If no asterisk is found, a comment is printed on line indicating this, then the System tape (A1) is spaced to the Machine Error record and control passes to 1-CS. If the asterisk is found the record is further scanned to determine if it is an End Tape card. If it is, the End Tape card is written on and off line, and a load card button sequence is simulated to end monitor operation. If the first card is not an End Tape card, it is assumed to be a true I. D. card. At this point, space is provided for the individual installation to insert coding for accounting purposes; therefore, at this point, differences may exist from one installation to another. As the standard program exists, the I. D. card will merely be written on and off line. After treating the I. D. record, the System tape is spaced to the Scan record (record 6), control is then passed to 1-CS.

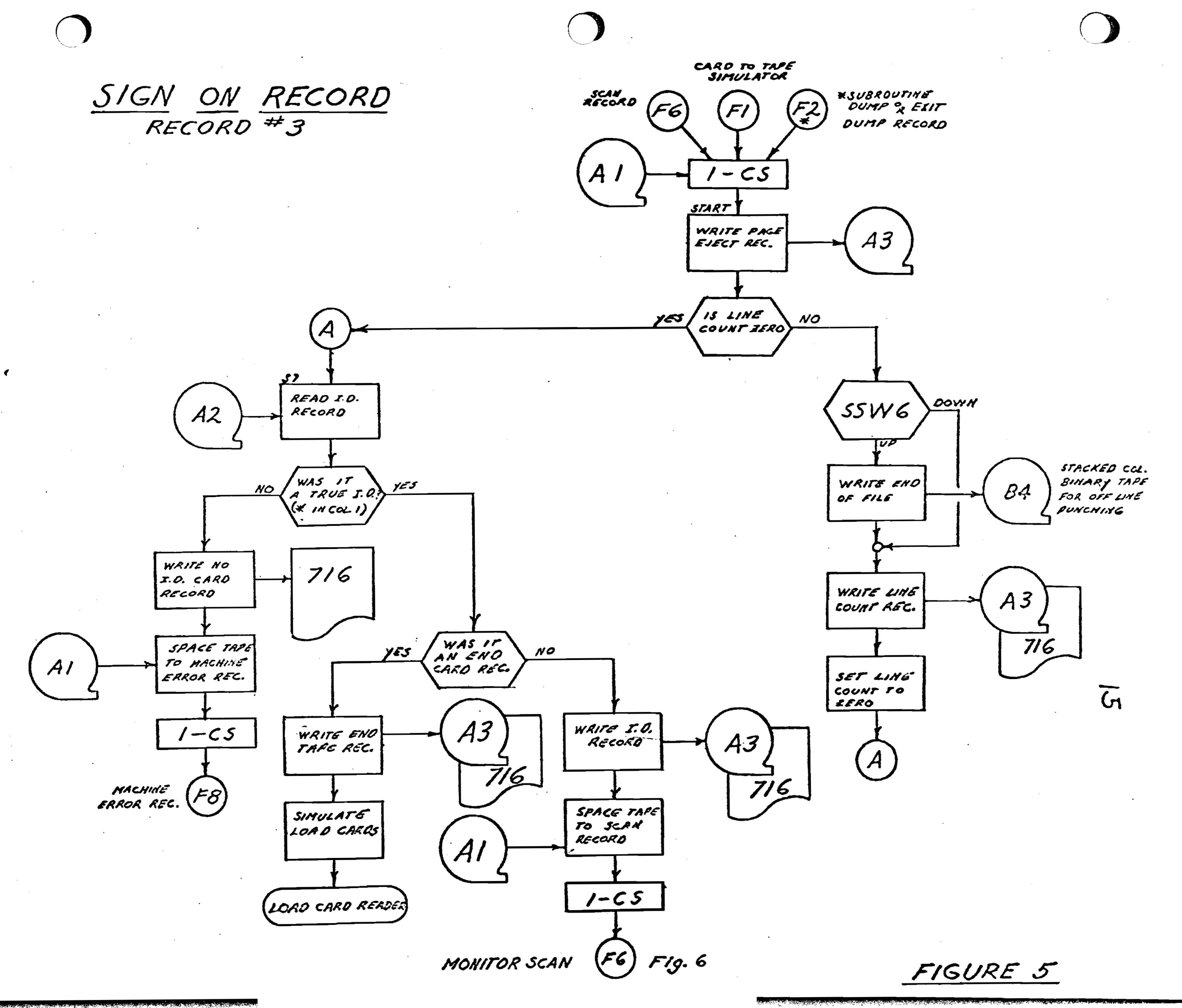


FIGURE 5

FAP (FORTRAN ASSEMBLY PROGRAM) was written, by the Western Data Processing Center at Los Angeles, to satisfy the need to produce machine language sub-programs for use with FORTRAN. FAP is also a fast, versatile general purpose assembler for non-FORTRAN main programs which has the additional advantage of operating within a monitor system. Operating in the monitor mode, it is possible to assemble and run a FAP assembled main program with the same load.

When assembling FORTRAN sub-programs, FAP provides all necessary information for direct communication with FORTRAN programs, including the program card and appropriate transfer vector. Also, FAP output occupies the binary card format required by the BSS loader.

The output may also be punched in the standard absolute binary format, to be run independently of any operating system.

Operating under FORTRAN Monitor control it is possible to input the symbolic deck on-line or off-line, however, the listing will only be written off-line.

FAP main programs may call upon FORTRAN subprograms, FORTRAN library functions or other FAP subprograms. The Monitor system and the BSS loader provide the necessary communication, based upon information given by the programmer in the calling sequences. Because the Monitor will accept programs in either binary or symbolic form, all programs need not be assembled or compiled at the same time.

They may be assembled in stages, thus providing a very useful method of debugging the main program. One section at a time.

Detection of assembly errors does not stop the assembly, but does suppress card punching and execution. Diagnostic information is given in the assembly listing. Control will pass to the Machine Error record or Source Error record, depending upon the error detected.

MONITOR SCAN (RECORD 6) 1.07.00

Record 6 is the primary monitor record in that it interprets the control cards which specify different system programs to be called. It also scans FORTRAN programs and prepares a single-compile input tape for the compiler. Control is passed to Monitor Scan in the following circumstances:

a) From record 3 (Sign On) after processing an I. D. card at the beginning of a job

b) From record 5 (FAP) after completing an assembly not for execution.

c) From record 7 or 43 (BSS) after relocating a series of binary programs when there are more symbolic programs remaining in the job.

d) From record 8 (machine error) or record 9 (source error) or record 2 (dump) when it has been determined that the job should be continued after an error

e) From the restart card "CONTINUE"

Operation is as follows: All input is from A2. Records are read double buffered and scanned first for an asterisk in column 1. If this is found, the mnemonics on the card are scanned and compared with a dictionary of control card mnemonics. If no asterisk is found, the card is assumed to be part of a FORTRAN program and a routine called SP is used. If the card is column binary, and an XEQ control card has been encountered earlier in the job, control is passed to BSS control (record 7). If the XEQ flag is off, column binary cards are ignored. Asterisk cards not in the dictionary are printed on and off line as remarks and then ignored. FORTRAN source program cards are scanned and then transcribed onto tape B2 (FORTRAN input). A FORTRAN source card with a CALL CHAIN (N, Bn) will be changed to CALL CHAIN (N, n). Upon encountering an END card, a fabricated END card is simulated onto tape B2 containing output options as indicated by control cards, previously encountered. Programmer's END card options will be preserved if not in conflict with control cards, which have precedence. Asterisk (control cards) found in the dictionary, are treated as follows:

a) XEQ - A flag in 1-CS is set indicating execution is desired. A word of zeros is written on the beginning of tape B1 to indicate that there is no snapshot (see record 7).

b) CHAIN ( ) - If the execution flag is off, this is treated as a remark card. If on, the parameters are examined and a unique control word is written on B1 (in front of the zero word) and stored in a cell (curchn) in 1-CS. If this is the 1st link, it is stored in a different cell (1st chn). A chain flag is set in 1-CS (FLGBX).

c) FAP - An END card is simulated onto B2 containing control card output options and control is passed to FAP Pass 1 (record 4).

d) DATA - This should be encountered only if there was no execution flag (or if execution has been deleted). Control is passed to Sign On unless the execution flag is on, in which case an error message (incorrect deck set up) is printed and control is passed to the source error record (record 9).

e) CARDS ROW, LIBE, ETC. - A flag is set for the END card routine to set the appropriate END card options

In summary, control is then passed as follows:

Upon Recognizing:	Go To:
a) FORTRAN END card	Record 10 (FORTRAN)
b) Column binary card	Record 7 ([illegible] Control)
c) FAP control card	Record 4 (FAP)
d) Deck error	Record 9 (Source Error)
e) Machine error	Record 8 (Machine Error)

Note: Monitor Scan has its own diagnostic message and prints them on and off line.

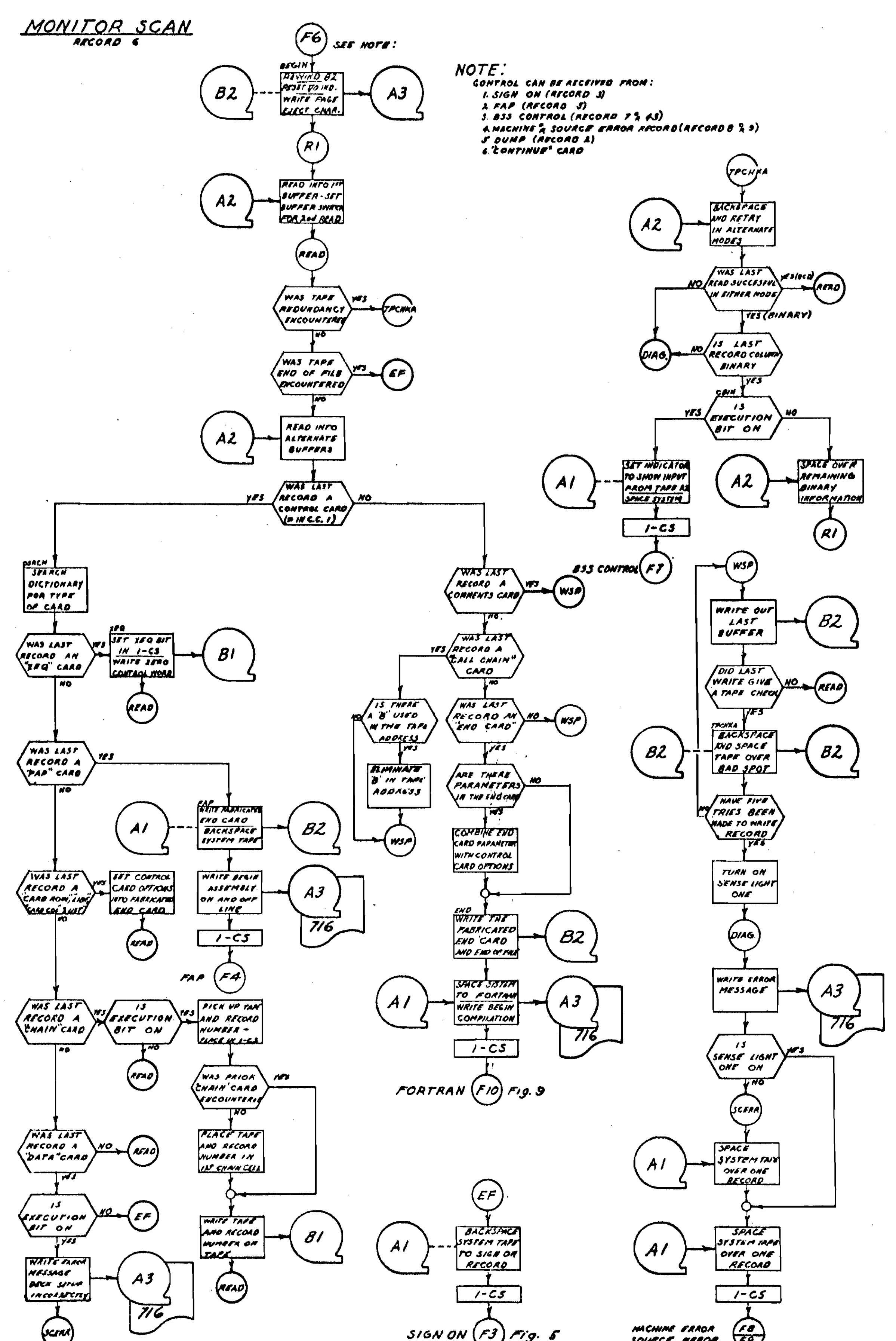


FIGURE 6

BSS CONTROL RECORD (RECORD 7 OR 43) 1.08.00

Records 7 and 43 are identical except for tape positioning, which of course, makes the decrement of the second word on the System tape, the record number, different. This record is duplicated in order to make it quickly accessible either from FORTRAN (second file) or from Monitor Scan when column binary cards are encountered or from a just completed FAP assembly. BSS accepts card image input from A2 (column binary cards), B3 (FAP or FORTRAN binary tape), or A1 (library subroutines) using a generalized double buffered read routine. The BSS program is located in the top of memory, occupying the standard Common region.

BSS will locate binary card images into locations 144_8 to 73000_8. 73000_8 to 74456_8 is used for a table of BCD program names, a missing subroutine table, and a Transfer Vector table. These tables together with several loading counts are referred to as the Snapshot.

Upon entry to BSS the Snapshot, from previous locations in the same job is read from tape B1. If this is the first time BSS has been entered for this job, a zero word will be read indicating that this is the first entry. This zero word was written by Monitor Scan if execution was called for.

The appropriate input tape is selected by examination of the indicator register which contains a control word left by the calling record. If an assembly (FAP) or compilation (FORTRAN) has just been completed, this will be tape B3, otherwise the input tape will be A2 in the case of column binary cards. The input tape is read in binary, transfer vectors are peeled off and stored at 144_8. When a new set of transfer vectors are met, the relocated block is saved as a single record on tape B1. The first word of this record is a control word specifying the size of the program and whether or not it has a transfer vector. If transfer vectors do exist a second control word is written giving the count. If B3 was the first input tape, when an End of File is met, the input is switched to tape A2. If this input is binary, the process of reading in, saving the transfer vector and relocating the binary deck starts anew. However, if BCD information is met, it is scanned and compared with a dictionary of control words. An XEQ card is ignored, as it is obviously in the wrong place. Any control card other than CHAIN or DATA cause control to be passed back to monitor Scan.

If a DATA or CHAIN card is recognized, the table of Transfer Vectors is searched against the table of BCD names to form a table of missing subroutines (MISUB). The System tape is then spaced to file 3 (library), the library is scanned for the missing subroutines, when found they are read in and relocated in memory. This search continues until the table of missing subroutines is zero or two passes have been made over the library. If subroutines are still missing they are listed on and off line with an appropriate error message, the execution bit is deleted and control passes back to Monitor Scan to finish any compilation that may be left in this job.

Encountering the DATA control card indicates that all programs for this job have been processed, relocated and written on tape B1. Tape B1 is now read, the Transfer Vector table is changed to TTR's with their proper relocated addresses, and written on tape A4 in absolute binary form.

A test is now made to see if this is a CHAIN job. If not, a small execution loader is moved over 1-CS. The word "Execution" is printed, and control passes to the execution loader. The loader reads the absolute program from Tape A4 into memory. The last record on tape A4 is the transfer word to the program.

If it is a CHAIN job, and the DATA card has not been encountered, tape B1 is backspaced to the current CHAIN I. D. word. The current link is then stacked on tape B1. BSS is refreshed, and the process begins again reading tape A2.

When the DATA card is encountered for a CHAIN job, the chain links are edited from tape B1 and are moved to the specified chain link tape. The execution loader is placed over 1-CS and the first link is read in as a single job, except that it is read from B1, B2 or B3 instead of from A4. After initial loading of the first chain link, loading of subsequent links will be done by the CHN subroutine. BSS control carries a large set of diagnostic messages that print on and off line. After a diagnostic message is printed, control will be given to the Machine Error record or Source Error record which ever is appropriate.

MACHINE ERROR RECORD (RECORD 8) 1.09.00

If during processing of monitor or non-monitor jobs a supposed machine error occurs, an attempt will be made to identify the failure. After this failure is identified, an error message will be written on and off line. At this point the System tape (A1) is positioned at the Machine Error record. The routine will first find out in what routine the error was detected, (i.e., FAP, MONITOR SCAN, GENERAL DIAGNOSTIC, BSS Control, or FORTRAN Section I") then print options to continue this job, restore memory or to retry. Since the exit from this routine is dependent upon the routine that called it, a brief description of each entry will follow:

If Monitor Scan called the Machine Error record sense light 3 will be on. In this case the error message "JOB DELETED BECAUSE OF MACHINE ERROR, PUSH START TO BEGIN NEXT JOB" will be printed. The System tape will be backspaced to Sign On then the machine will halt. When the Start switch is depressed, control will transfer to 1-CS to be read in Sign On.

If FAP called the Machine Error record, sense light 1 will be on. The errors that FAP believes are machine error are persistent tape errors, overflow of various tables and table search errors. In some cases table overflow errors may be caused by incorrect deck setup or faulty coding. In any event instructions will be printed on line, indicating in most cases the type of error and the course of action to be taken. Depending on the instructions and the action taken by the operator, the program can be deleted or continued, in either event control will be returned to Monitor Scan.

When sense light 4 is found on, control was received from the General Diagnostic. Since FORTRAN may be run in either the single compile or monitor modes, the options to delete or retry will vary with the mode. In either mode, to retry, control is passed back to FORTRAN record 10. Likewise to restore memory and halt is the same in both modes. In the monitor mode if deletion is called for, the source program is transcribed from B2 to A3 then control passes back to the Monitor Scan record.

When BSS control calls the Machine Error record sense light 2 will be on. These errors are persistent tape checks that BSS control cannot get around. The indication of the error is printed on line, the execution bit is deleted and control is passed to the Monitor Scan record.

The last case will be with all sense lights off, when the diagnostic within Section I of FORTRAN calls the Machine Error record. The printed messages and the options are the same for this as they where when control was received from the Main Diagnostic record.

SOURCE ERROR RECORD 1.10.00

The Source Error record is called when a source error is detected. All routines that can call the Machine Error record (see 1.09.00) can also call this record.

The only processing that is done will be to delete the execution bit in 1-CS if it exists and write off line, the reason for no execution.

In the Monitor mode if any record other than Monitor Scan called this record control is passed to Monitor Scan. If Monitor Scan calls this record control is passed to the Sign On record.

However, in the case of a source error in the single compile mode the card reader is selected and the program will hang up.

TAPE MOVER (RECORD 42) 1.11.00

The Tape Mover record is entered at the termination of FORTRAN. If the system is in monitor mode, the information from the single compile print tape (B2) will be transferred to the stacked print tape (A3). Since the third file (symbolic listing) is optional, FORTRAN in Section VI will turn on Sense Light 2 to indicate to tape mover that the third file is needed. The information on the binary output tape (B3) will be transferred to the stacked binary tape if: (1) Sense Switch 6 is up and, (2) Sense Light 1 is ON (left on if column binary cards were not called for in Section 6). When one or both tapes have been processed the execution bit is checked, if ON control passes to BSS Control (Record 43), if OFF control passes to Monitor Scan (Record 6).

Upon initial entry to Tape Mover if the system were not in monitor mode, the card reader would be selected and the program would hang up.

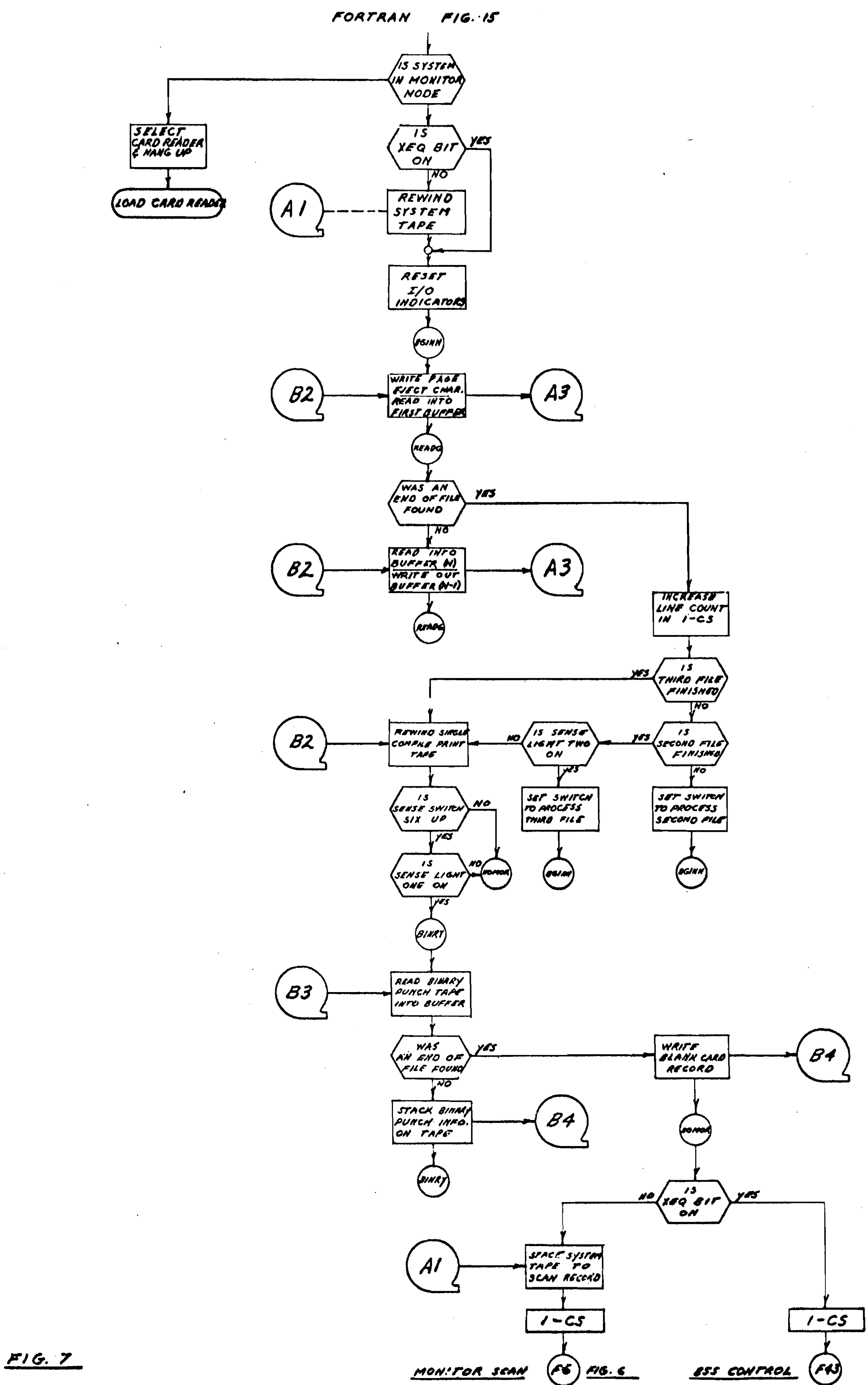


FIG. 7

FORTRAN COMPILER 2.00.00

INTRODUCTION

The FORTRAN Executive routine comprises most of the second file of the System Tape. The exceptions are the two monitor records, Tape Mover and BSS Control. FORTRAN is made up of 32 records (#10 through #41) which are called in one or more at a time. FORTRAN is broken down into six sections each one given a portion of the task of analysis of the source program. There are in addition to the six main sections, four subsections, these are I', I", V' and pre-VI. These subsections are in reality only extensions of the main sections to which they are attached. These sections are operated on sequentially, that is there is never a return to a previous section once control passes to a succeeding section.

FORTRAN can be considered as falling into two divisions, the first comprised by sections I, II and III, the second by sections IV, V and VI. This is due to the fact that by the end of section III, the entire object program is essentially compiled. It is, in fact, compiled except that it exists in the C. I. T. (Compiled Instruction Table) format, and that it has as many symbolic index registers as are required. It is the job of the remaining three sections to correct these two situations. Sections IV and V handle the task of inserting the absolute index registers in place of the symbolic index registers. Since we are reducing a large number of symbolic index registers to the three absolute registers, certain index loading and saving instructions are necessary. This problem is also handled by Sections IV and V.

Section VI, replaces the instructions that are in the CIT format into the proper relocatable binary format.

As for the first three sections, it may be considered that the first two of these do the entire task of source program analysis. This task includes performing most of the instruction (C. I. T.) compilation. With reference to some of the instructions, however, sections I and II simply record information, in tabular form, to pass on to section III, which will use these as a key to insert the proper instructions. Because the analysis of sections I and II are independent, the C. I. T.'s compiled are kept in separate files, which must subsequently be merged. Section III, therefore, has the task of performing this merge as well as a second merge of the C. I. T.'s that it, itself has created. Both section III and the last part of section V, because of their position at the end of necessary primary analysis, perform certain optimizing tasks consisting mostly of removing or inserting certain instructions.

It is well to note that the FORTRAN compiler makes extensive use of tables. These may be considered as of two types: those which are made up directly from the source program statements, and those which result from further analysis. It is the former class of tables which are included in this reference manual. A list of some of these tables and their size limitations will be discussed as they are encountered. The latter class do, in some cases, impose further size limitations. Most tables are passed on from one section to another; some however, are created purely for use within a section. The source program statement, once scanned, are not referred to again. For a more detailed description of some of these tables, see Section 2.04.00.

Section I has the primary output of a file of instructions called the Compail file. The first CIT's that are written in COMPAIL are the Arithmetic Statement Functions. These are labeled in such a manner that the Merge in Section III will recognize and separate these from all other CIT's, and write them as separate files. The arithmetic instructions, of course, refer to symbolic tags in the word four address. Also included in this file are a partial translation of the IF and GO TO Statements, the subprogram definition statements, and input/output statements.

With respect to the IF and GO TO Statements, Section I compiles the necessary test instructions, but it cannot compile the transfer instructions. This is because Section I does not know whether any given IF and GO TO Statement is in the range of a DO and involves a transfer out of the DO. It is not until this is known that it can go directly to the statement indicated in the source program, or go to a set of instructions providing necessary indexing, then the transfer to the specified source program statement. The analysis pertaining to these indexing instructions is left to Section II with the physical instructions being compiled by the second part of Section III. In some cases, a CIT is created containing the transfer instruction, but without the address. The address is filled in Section III.

With respect to subprogram definition statements, information is gathered which is used by section pre-6 in actually filling in the prologue and index-saving instructions.

With respect to I/O statements, all instructions are compiled except those involving DO's implied by I/O statement lists. After Section I has scanned and identified the source program statement, it handles it by transferring to a routine corresponding to it. Then, of course, all information is tabulated and, when possible, compilation performed.

A new internal formula number, initially zero, incremented by one, is assigned to each source statement, whether that statement is executable or non-executable. Where external statement numbers -- i.e., statement numbers assigned by the source programmer -- exist, the TEIFNO table serves to correlate the external and internal statement numbers.

The greatest division in the handling of statements in Section I is between the arithmetic statements and all others. The arithmetic compiler proper constitutes the major portion of Section I in number of instructions. The arithmetic compiler in making its scan of the arithmetic formula makes an enormous number of table entries in addition to doing its statement analysis necessary for compilation.

Among these tables are the TAU tables, recording subscript combination information, the FORVAL and FORVAR tables recording fixed point variables occurring on the left and right hand sides of arithmetic statements, FIXCON and FLOCON, recording the converted fixed and floating point numbers. It should be noted that IF and CALL statements fall onto both sides of this division. They are treated as arithmetic statements, with compilation occurring, that is not due directly to the arithmetic compiler, as well.

The arithmetic compiler is divided into the Scan, Level Analysis, various Optimizing routines, and the Compiler. The Level Analysis sifts out into one group all those algebraic operations which form a unit. A unit is a group that must be performed together and have the same order of binding strength for its operators. "Plus" and "minus" are one order of operators, "multiply" and "divide" are another order. The latter has greater binding strength than the former; consequently, when they occur in the same context the latter are assigned a higher level number. Needless to say, the use of parenthesis in an arithmetic statement is a prime factor in determinging units and, hence, level numbers. Optimization occurs to minimize storage accesses. This means that every attempt is made to link one operation to its successor via the machine registers rather than the storage cells. The compilation then proceeds from highest level number to lowest.

Flow Within Pass I of Section I

The input to Pass I is the source program in BCD form as a single file, on tape B2.

One record at a time is read into a buffer termed FT. All comment cards and blank cards are ignored. A special mode character in card column one is saved.[1] If a statement number (EFN) exists it is converted to a binary number and saved. The FT buffer is now moved to the F region, and a new record is read into the FT buffer. In this manner the program looks ahead one record at a time, to determine if there are any continuation cards, any non-blank, non-zero, character in card column 6). All continuation cards are read for a given statement and assembled in the F region. A word of all ones is written after the last non-blank word in the F region to serve as an end-of-statement marker.

At this point a decision must be made as to whether the statement is arithmetic, if not arithmetic, it is non-arithmetic, some of which are non-executable. The beginning of the non-arithmetic statements are compared to entries in a dictionary of non-arithmetic statement beginnings. If the statement is not identified in the dictionary a diagnostic message is printed.

All executable statements including arithmetic are written on tape B3 with a corresponding label. These records on B3 are essentially the same as the records on B2, except they are in a more compact form and are written in binary. The records on B3 contain all continuation cards of a source statement, less terminal blanks, and certain pre-digested information.

The non-executable statements are processed in Pass I and entries made in the appropriate tables in core. If an external statement number (EFN) appears in the source statement, an entry is made in the TEIFNO table with a corresponding internal formula number (IFN).

[1] For use and operation of these mode characters refer to operating bulletins for the 32K FORTRAN System.

Flow In Pass II of Section I

The input to Pass II is the condensed source program in binary form as a single file on B3.

One record is read in at a time, the first word of each record is a label for the type of statement. This address portion of the label is the transfer address to the appropriate processor.

As the statement is scanned, the various parts are classified and appropriate table entries are made.

When all the statements have been processed, control passes to the next record on the System tape (A1). This record is the Diagnostic for Section I. The diagnostic record can be called earlier if an error is found in the source program or a machine error is encountered. The program consists of:

Program to prepare message
Print program
Table of comments

When an error is found or occurs during Section I control goes to the Diagnostic Program by means of a TSX using IR4. There are several possible cases:

I IR4 ≠ 0 signifies an error call

1) First error: Print "Fortran Diagnostic Program Results" heading and proceed as in 2).

2) Not first error: Construct parameters for printing statement being processed and comment.

   a. If error was source program, return control to Section I for processing next statement.

   b. If error was machine, print "END OF DIAGNOSTIC" message and go to Machine Error Supervisor program (record 8).

II IR4 = 0 signifies control was received at the completion of Section I.

1) No errors had occurred. Go to Section I'.

2) Some source program errors had occurred. Write all diagnostic information which has been printed on tape B2 following source program. Go to Source Program Error supervisor program. (record 9).

TABLES GENERATED BY SECTION I;

1. Generated by Section I and required for reference. These tables, retained in cores are:

NAME	DESCRIPTION
DIM1	one-dimensional arrays
DIM2	two-dimensional arrays
DIM3	three-dimensional arrays
TAU1	one-dimensional subscripts
TAU2	two-dimensional subscripts
TAU3	three-dimensional subscripts
FIXCON	fixed-point constants
FLOCON	floating-point constants
FORSUB	arithmetic statement functions
END	options specified in END statement

2. Generated by Section I and not required for reference. These tables, written on tapes in buffer sized records, with labels where needed are:

   a. Written on tape B2, 100 words per record:

NAME	DESCRIPTION
CIT	COMPILED INSTRUCTION TABLE

   b. Written on tape A4, in buffer sized records with appropriate labels.

LABEL	NAME	DESCRIPTION
0	TEIFNO	corresponding IFNs and EFNs
1	TDO	DO statements
2	TIFGO	IFs, GO TOs, ASSIGN statements
3	TRAD	GO TO statements
4	FORTAG	IFNs - I - TAU tags
5	FORVAR	fixed-point variable usage
6	FORVAL	fixed-point variable definition
7	FRET	FREQUENCY statements
8	EQUIT	EQUIVALENCE statements
9	CLOSUB	names of closed subroutines references
10	FORMAT	FORMAT statements
11	SUBDEF	SUBROUTINE or FUNCTION statements
12	COMMON	COMMON statements
13	HOLARG	Hollerith arguments in CALL statements
14	NONEXC	IFNs on non-executable statements
15	TSTOPS	IFNs of STOP and RETURN statements
16	CALLFN	first and last IFNs of CALL statements
17	FMTEFN	I - O statement references to FORMAT numbers

SECTION I' 2.01.02

This section is a terminal processor for Section I, and is the longest of all secondary sections.

The tables that Section I generated were written on tape A4 as buffer size records, as they became full. They can be many records on tape A4 all of the one table type. These records are not necessarily on the tape consecutively but rather at random intervals, also the buffers in Section I for these tables may have been only partially filled at the end of Section I. These partially filled buffers are left in core for processing by Section I'.

The primary task of Section I' is to collect all like tables from tape A4, combine them, insert the partially filled buffer, determine the word count and write these tables on tape B2, with a label number corresponding to the type of table.

Section I' also makes certain modifications, primarily the replacement of EFN's with corresponding IFN's, using the TIEFNO table. This can only be accomplished when the entire source program has been reduced to tabular form. An example of where the external statement numbers have had to be retained up to this point is in the TDO table. Here, the number referring to the statement number of the DO itself may be an internal formula number because it is readily known due to the constant updating of the current internal formula number. On the other hand, the DO range had to be recorded as an external statement number at the time the TDO table entry was made. This is because it could not then be known how may statements further on in the program the end of the DO range would occur.

The input to Section I' consists of:

1. Various parameters describing tables (in cores).

2. Buffers containing terminal entries in tables (in cores).

3. Tables which Section I require for reference (FORSUB), END, DIM1, DIM2, DIM3, TAU1, TAU2, TAU3, FIXCON, FLOCON in cores.)

4. Tables which Section I did not require for reference. (COMPAIL, on tape B2, TEIFNO, TDO, TIFGO, TRAD, FORTAG, FORVAR, FORVAL FRET, EQUIT, CLOSUB, FORMAT, SUBDEF, COMMON, HOLARG, NONEXC, TSTOPS, CALLFN, FMTEFN, on tape A4.)

The output of Section I' consists of:

1. Tables in cores: TAU1, TAU2, TAU3, FIXCON, FLOCON, FORVAL, TRAD, TIFGO, TEIFNO, NONEXC, TSTOPS.

2. Tables on tape:
   Tape B2: File 1 is Source Program
   File 2 is COMPAIL table
   File 3 Record 1 is FORSUB table except the first word which is the COMPAIL record count.

File 4 Record 1 is FLOCON table.

Record 2 is FORMAT table.

Record 3 is SIZ table.

File 5 Record 1 is END table.

Record 2 is SUBDEF table.

Record 3 is COMMON table.

Record 4 is HOLARG table.

Record 5 is TEIFNO table.

Record 6 is TIFGO table.

Record 7 is TRAD table.

Record 8 is TDO table.

Record 9 is FORVAL table.

Record 10 is CALLNM table.

Record 11 is FORTAG table.

Record 12 is FRET table.

Record 13 is EQUIT table.

Record 14 is CLOSUB table.

The tables are processed in the following order and manner:

32K Version - The contents of the Section I CIT buffer are written as the last record of file 2 on tape B2.

FORSUB - The table of names and degrees of arithmetic statement functions, if any, is written after the COMPAIL record count which is the first word in record 1 of file 3 on tape B2.

FLOCON- The table of floating-point constants and its word count are written as record 1 of file 4 on tape B2.

FORMAT - The table of format statements is assembled from tape A4 and the Section I buffer. It is written as record 2 of file 4 on tape B2; preceded by its identification (10) and word count.

FMTEFN - The table of references to fixed format statements is assembled from tape A4 and the Section I buffer. Each reference to a format is checked against

the FORMAT table. If any referenced statements are missing an error list is developed for Section I''.

DIM1 - The table of one dimensional arrays is renamed SIZ.

DIM2 - Each entry in the table of two dimensional arrays has its two dimensions multiplied to form the size of the array. This table is added to SIZ.

DIM3 - Each entry in the table of three dimensional arrays has its three dimensions multiplied to form the size of the array. This table is added to SIZ.

SIZ - The table is written as record 3 of file 4 on tape B2. It is preceded by the EIFNO table and its word count.

END - The END table is written as record 1 of file 5 on tape B2.

SUBDEF - The table of subprogram definition is assembled from tape A4 and the Section I buffer. It is written as record 2 of file 5 on tape B2; preceded by its identification (11) and word count.

COMMON - The table of common variables is assembled from tape A4 and the Section I buffer. It is written as record 3 of file 5 on tape B2; preceded by its identification (12) and word count.

HOLARG - The table of hollerith arguments is assembled from tape A4 and the Section I buffer. It is written as record 4 of file 5 on tape B2; preceded by its identification (13) and word count.

TEIFNO - The table of corresponding external and internal formula numbers is assembled from tape A4 and the Section I buffer. It is searched for duplicate external formula numbers. If duplicates are found they are flagged as errors for Section I''. Those cases where Section I assigned more than one internal formula number, are not considered as duplicates and the flag is deleted. The table is written as record 5 of file 5 on tape B2; preceded by its identification (0) and word count.

It is also retained in memory for use in processing tables discussed below:

TIFGO - The tables of IFs, GO TOs and ASSIGNs is assembled from tape A4 and the Section I buffer. Each external formula number is searched for in TEIFNO and its corresponding internal number replaces it in TIFGO. Any external formula numbers not found are set equal to 0 as an error signal to Section I''. When all entries have been modified the table is written as record 6 of file 5 on tape B2; preceded by its identification (2) and word count.

TRAD - The table of COMPUTED and ASSIGNED GO TO addresses is assembled from tape A4 and the Section I buffer. Each entry, which is an external formula number, is searched for in TEIFNO. When found it is replaced by the corresponding internal formula number. If not found, it is set equal to 0 as an error signal to Section I''. When all entries have been treated the table is written as

record 7 of file 5 on tape B2; preceded by its identification (3) and word count.

TDO - The table of DO's is assembled from tape A4 and the Section I buffer. Each entry is examined to determine if it originated from a DO or from an Input-Output List. If it originated from a DO the EFN for the end of the DO is searched for in TEIFNO. When it is found the corresponding IFN replaces it in TDO. If not found, it is set equal to 0 as an error signal to Section I''. In those cases where Section I assigned more than one IFN to an external number, the last such IFN is used so that the DO includes all instructions of the terminal statement. When all entries have been treated the table is written as record 8 of file 5 on tape B2; preceded by its identification (1) and word count.

FORVAL - The table of definitions of fixed-point variables is assembled from tape A4 and the Section I buffer.

CALLNM - The table of first and last internal formula numbers of statements containing references to subprograms is assembled from tape A4 and the Section I buffer. Each IFN in FORVAL is searched for as a first IFN in CALLNM. If found, it is replaced by the corresponding last IFN. When all entries have been processed the FORVAL table is written as record 9 of file 5 on tape B2; preceded by its identification (6) and word count. The CALLNM table is dead.

FORVAR - The table of usages of fixed-point variables is assembled from tape A4 and the Section I buffer. It is written as record 10 of file 5 on tape B2 preceded by its identification (5) and word count.

FORTAG - The table of tag usages is assembled from tape A4 and the Section I buffer. It is written as record 11 of file 5 on tape B2; preceded by its identification (4) and word count.

FRET - The table of frequency statements is assembled from tape A4 and the Section I buffer. Each EFN in FRET is searched for in TEIFNO. When found it is replaced with the corresponding IFN. If not found, it is set equal to 0 as an error signal for Section I''. The FRET table is now sorted by IFN to form an ordered list.

TIFGO - The TIFGO table is now re-examined for any entries for COMPUTED GO TO statements. The IFN of each such statement is searched for in FRET. If found, the list of branch frequencies is reversed to correspond to the object program transfer vector.

When all TIFGO entries have been examined, the FRET table is written as record 12 of file 5 on tape B2; preceded by its identification (7) and word count.

EQUIT - The table of equivalence statements is assembled from Tape A4 and the Section I buffer. The table is reformatized to make those variables which are equated into strings of relativelized symbols. Any found to be inconsistent are flagged as errors for Section I''. Any redundancies are deleted. The table is then written as record 13 of file 5 on tape B2; preceded by its identification (8) and word count.

record 7 of file 5 on tape B2; preceded by its identification (3) and word count.

TDO - The table of DO's is assembled from tape A4 and the Section I buffer. Each entry is examined to determine if it originated from a DO or from an Input-Output List. If it originated from a DO the EFN for the end of the DO is searched for in TEIFNO. When it is found the corresponding IFN replaces it in TDO. If not found, it is set equal to 0 as an error signal to Section I". In those cases where Section I assigned more than one IFN to an external number, the last such IFN is used so that the DO includes all instructions of the terminal statement. When all entries have been treated the table is written as record 8 of file 5 on tape B2; preceded by its identification (1) and word count.

FORVAL - The table of definitions of fixed-point variables is assembled from tape A4 and the Section I buffer.

CALLNM - The table of first and last internal formula numbers of statements containing references to subprograms is assembled from tape A4 and the Section I buffer. Each IFN in FORVAL is searched for as a first IFN in CALLNM. If found, it is replaced by the corresponding last IFN. When all entries have been processed the FORVAL table is written as record 9 of file 5 on tape B2; preceded by its identification (6) and word count. The CALLNM table is dead.

FORVAR - The table of usages of fixed-point variables is assembled from tape A4 and the Section I buffer. It is written as record 10 of file 5 on tape B2 preceded by its identification (5) and word count.

FORTAG - The table of tag usages is assembled from tape A4 and the Section I buffer. It is written as record 11 of file 5 on tape B2; preceded by its identification (4) and word count.

FRET - The table of frequency statements is assembled from tape A4 and the Section I buffer. Each EFN in FRET is searched for in TEIFNO. When found it is replaced with the corresponding IFN. If not found, it is set equal to 0 as an error signal for Section I". The FRET table is now sorted by IFN to form an ordered list.

TIFGO - The TIFGO table is now re-examined for any entries for COMPUTED GO TO statements. The IFN of each such statement is searched for in FRET. If found, the list of branch frequencies is reversed to correspond to the object program transfer vector.

When all TIFGO entries have been examined, the FRET table is written as record 12 of file 5 on tape B2; preceded by its identification (7) and word count.

EQUIT - The table of equivalence statements is assembled from Tape A4 and the Section I buffer. The table is reformatized to make those variables which are equated into strings of relativelized symbols. Any found to be inconsistent are flagged as errors for Section I". Any redundancies are deleted. The table is then written as record 13 of file 5 on tape B2; preceded by its identification (8) and word count.

CLOSUB - The table of names of closed (library) subroutines is assembled from tape A4 and the Section I buffer. Duplicates are eliminated. Each name in the CLOSUB table is searched for in the SUBDEF table. If found, it is deleted from CLOSUB as being a dummy name. The table is then written as record 14 of file 5 on tape B2; preceded by its identification (9) and word count.

NONEXC - The table of statement of non-executable statements is assembled from tape A4 and the Section I buffer. It is left in core.

TSTOPS - The table of statement numbers of STOP and RETURN statements is assembled from tape A4 and the Section I buffer. It is left in core.

MISC - One is added to the last IFN used and it is left for Section I''.

SUBROUTINE S - There are two subroutines used by Section I'.

TAP00 - Table assembly Program assembles tables written on tape A4 during Section I. It uses the parameters left by Section I to determine for a given table:

1. number of records on tape A4,

2. number of words in each record,

3. number of words remaining in the core buffer,

4. first location of core buffer.

The calling sequence in Section I' supplies the:

1. table identification (which also serves to locate the parameters left by Section I).

2. first location of buffer into which the table is to be assembled,

The routine tests each table for overflow against a table of permissible maximums.

Tables Assembled by TAP00 are shown on the following page.

NAME	IDENTIFICATION (First Word)	MAXIMUM WORD COUNT
TEIFNO	0	3000
TDO	1	3000
TIFGO	2	2400
TRAD	3	1000
FORTAG	4	6000
FORVAR	5	6000
FORVAL	6	4000
FRET	7	3000
EQUIT	8	6000
CLOSUB	9	6000
FORMT	10	6000
SUBDEF	11	180
COMMON	12	2400
HOLARG	13	3600
NONEXC	14	1200
TSTOPS	15	1200
CALFN	16	2400
FMTEFN	17	2000
END	19	15

WAT00 - Writes assembled table on tape B2: preceded by its identification and word count. Calling sequence supplies identification and first location of buffer in which table has been assembled.

SECTION I''   2.01.03

Section I made a determined effort to eliminate the errors in any one statement. No effort was made in Section I toward relating a particular statement to the rest of the program. It would not have been convenient to do so since the tables were not complete nor in order. It was the job of Section I' to complete the tables and get them in order.

Section I'' therefore, can be considered nothing more than a continuation of Section I' in the form of a diagnostic. It attempts to find as many source program errors as possible arising from an interrelationship of the statements.

The errors that Section I'' is able to find are mainly errors in program flow. Such as transfers to non-executable or even non-existent statements, and conversely, no transfers to executable statements which is not in the direct path of flow. These and other errors, are found through a scan of the various tables of information which comprise the 5th file on tape B2. These tables are of such rigid format that it is easy to examine them for correct ordering and content. All errors found by Section I'' are accummulated in an error list by several different error routines. The table scan is only discontinued by table overflow or a machine error. Section I'' uses the general diagnostic in the 4th file of the System tape.

In general then, it is true that by the end of Section I'' very nearly all source program errors have been found. Such things as overlapping DO ranges and certain rare cases of faulty flow still may not be found until Sections II, IV or V. Also some table overflow errors may be found after Section I'', however, most of the tables are tested prior to this point and any overflow discovered. An understanding of the conventions described in the examples below will be necessary for the description of the tables that follow:

5 GO TO 10   5 IF (----) 10, 20, 30

α GO TO β α IF (----) β_1, β_2, β_3

(alpha) is the symbolic location

(beta) is the symbolic address

alpha and beta are in the form of internal formula numbers (IFN).

Section I'' first initializes the error list with the count of missing format statements. The EFN's of missing format statements are left in the error list by Section I'.

TEIFNO

The TEIFNO table is scanned for duplicate statement numbers. Duplicate statement numbers are flagged minus by Section I' when it assembles the TEIFNO table. If any minus entries are found, they are entered in the error

list by the ERROR routine. TEIFNO was retained in core from Section I'.

TIFGO

Each of the 2 word TIFGO entries is examined for references to non-existent statement numbers, i.e., that there are not any zeros except those peculiar to the particular TIFGO format. Section I' gives non-existent EFN an IFN of zero. Further, each reference β must be to an executable statement. Therefore, a β cannot be in the table of non-executable statements, the NONEXC table. Each of the eight different types of TIFGO entries is checked by a specific subroutine within the TIFGO processor. This scan of the TIFGO table will result in the checking of the TRAD table, if one exists. If any errors are found, they are entered in the error list by either the ERROR routine if β is non-executable or the NOBETA routine if β is non-existent.

In order to do a quick flow analysis the IFN α of a TIFGO statement is entered in the ALPHA table, and the references (IFN β's) are entered in the BETA table. The number of branches associated with a particular TIFGO entry is also entered in the ALPHA table with the IFN α. All TIFGO entries, except ASSIGNS, are entered into these tables. The position of an ASSIGN in the source program does not effect the path of flow in the program.

The ALPHA and BETA tables are internal to Section I'' and have the following format:

ALPHA

DECREMENT,	TAG,	ADDRESS	
N	0	IFN α	N:: Number of branches.

The table of STOP and RETURN statements, TSTOPS, is a part of the ALPHA table.

BETA

DECREMENT,	TAG,	ADDRESS
0 or 1*	0	IFN β

*Decrement will be 1 if β is non-executable.

The BETA table consists of the β's from TIFGO, the entire TRAD table, and the last IFN α+1 in the program. In the 704, the inclusion of machine language necessitated the building of a second BETA table, the BETA2 table. This second BETA table is an extension of the BETA table and has the same format. BETA2 consists of the TSKIPS table, table of skip type instructions such as CPY,

CAS, LBT, etc., and the α+1 of conditional transfers from TIFGO. Conditional transfers are TXH, TIX, TMI, etc.

FLOW ANALYSIS

Example 1

α	GO TO β
α+1	DIMENSION X (5)
α+2	FORMAT (F8. 3) More non-executable statements.
α+M	A = B + C

A brief flow analysis is performed using the information in the ALPHA, BETA, and NONEXC tables. Each α in the ALPHA table is the termination of a path of flow in the source program. Therefore, there must be a transfer to the first executable statement following each α in the ALPHA table. That is, that the IFN α+M in Example 1 must be in the BETA table, since β's are statements transferred to. In reference to Example 1, the flow analysis processor will first search the BETA table for α+1. Not finding α+1 in the BETA table, it will then search for α+1 in the NONEXC table, and a match will be found, Upon finding α+1 in the NONEXC table, the processor will then follow the same procedure for α+2, α+3,......, α+M. In searching for α+M, if the processor finds it in the BETA table, the processor will then proceed to execute a flow analysis for the next α in the ALPHA table. However, if α+M is not in the BETA table, and since it is an executable statement, α+M will not be in the NONEXC table.. Therefore, if α+M is not in either the BETA or NONEXC tables, it is a part of the program not reached, i.e., an executable statement with no path of flow to it. If any errors are found, they are entered in the error list by the NOBETA routine. TIFGO was retained in core from Section I'.

TDO

The TDO table is examined for DO statements that specify an illegal β. The three legal references checked for by Section I'' are:

1. That the IFN β exists, i.e., that the reference β is not zero.

2. That the IFN β is executable, i.e., that the reference β is not in the NON-EXC table.

3. That the IFN β is not a transfer, STOP, or RETURN statement, i.e., that the reference β is not in the ALPHA table.

If any errors are found, they are entered in the error list by both NOBETA and αDO β routines, in that order. TDO is read from the 5th file on tape B2.

FRET

The number of branches for a TIFGO statement is saved in the ALPHA table with the IFN during the scan of TIFGO. Section I'' ignores statement numbers in the FRET table which are not in the ALPHA table, but saves any statement number where the count of branches in FRET is greater than the count of branches shown in the ALPHA table. Section IV ignores extra frequencies given for statements other than TIFGO statements, but would be confused by misinformation generated when there are move frequencies given than there are branches. If any errors are ound, they are entered in the error list by the NOBETA routine. FRET is read from the 5th file on tape B2.

EQUIT

If Section I' has found any inconsistent equivalences when assembling the EQUIT table, it sets an error flag at the beginning of the table and only enters those variable names which are erroneous, and sets another flag at the end of the list. The errors are entered in the error list by the ERROR routine. The EQUIT TABLE is read from the 5th file on tape B2.

If any errors have been found in Section I'', it spaces the System Tape to the diagnostic and reads in D001. This is the only section of FORTRAN that does not use the usual diagnostic caller. If no errors have been found, tape B2 is spaced over the 5th end of file mark and control is transferred to 1-CS to continue compilation.

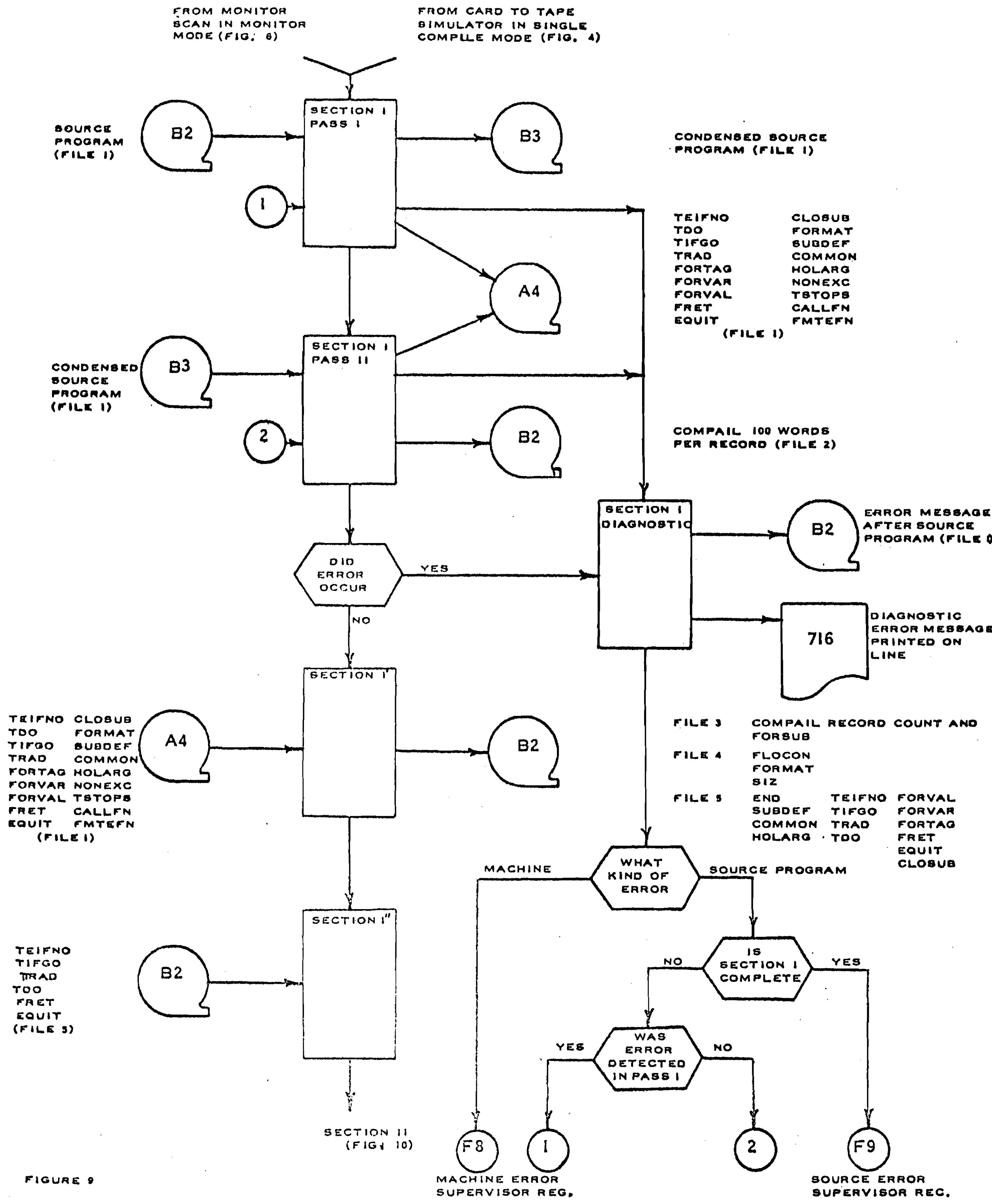


FIGURE 9

It is the task of Section II to process DO statements and subscripted variables appearing in arithmetic and input output statements. The primary output of this section is a file of CIT's called the COMPDO file. It was not convenient for Section I to do this, since the tables were not in order. As you will recall it was the task of Section I' to sort these tables into an ordered form. The tables that Section II will use are in the 5th file on tape B2 and in memory. In addition, Section II will create a secondary file of closed subroutines for the computation of, relative constants, subscript combination load values. Also information about transfers out of DO loops is recorded in the TRASTO and TRALEV tables. These two tables will be used by Section III to produce the TIFGO file, this is not to be confused with the TIFGO table created in Section I.

Preliminary Description of the Problem

A. Tags created by Section I

Section II compiles the instructions necessary to compute and index so that the symbolic index registers, (tags), set up in Section I for tagged instructions will contain their proper values. The tagged instructions compiled by Section I refer to subscripted variables. The symbolic tag is in fact, a subscript combination with given dimensions and coefficients. The tags are divided into three classes, 1, 2, and 3 dimensional, and are recorded in separate tables, TAU1, TAU2 and TAU3

B. DO's

The COMPDO file of instructions contains the computing andindexing instructions for the various subscript combinations contained within DO ranges and any necessary additional tags. These instructions are associated with the beginnings and ends of DO's. At the beginning of DO's they will contain the computing instructions necessary to determine the load value for a tag(subscript combination index register) and the load instructions In addition, index saving instructions may occur At the end of DO's these instructions refer to the indexing required to increment subscript combinations values for the next DO loop execution, to test whether or not control may pass out of the DO range and in the latter case, to reset the DO's subscript combinations to their lowest values if control is still in a DO containing the first DO (nest).[1]

All of these instructions result from the configuration of the combination of DO-nest structure on the one hand and subscript combinations within the DO-nest on the other. A DO nest is defined as any set of DO's all of which are bounded contained within a single DO. Figuratively, this means that the outside single DO is on level one, the next DO which it contains, on level two. and so forth Of course, in a single nest there may be more than one DO on any one level greater than level 1.

[1] The instructions performing these three functions are TXI, TXL, and TIX respectively.

Further complications may result from the transfers out of DO's where additional indexing and saving instructions will be required. The TIFGO file that will be compiled in Section III will contain the instructions necessary for doing this. The TIFGO file will be made up from information passed on to it in the TRASTO and TRALEV tables created in this Section.

C. Relative Constants

A considerable portion of the work of Section II is devoted to the proper handling of subscript combinations which are called relative constants. A Relative constant is a subscript symbol not under control of a DO on that symbol.

That is, it receives its definition in some fashion other than the indexing normally associated with a DO. A subscript combination may, therefore, be a pure relative constant (where none of its symbols is under control of a DO), a mixed relative constant (where at least one is not under control of a DO while the others are), or a normal DO-subscript combination (where all subscript symbols are under control of a DO).. Each of these three types requires its own mode of treatment by Section II.

The FORVAL table is the key in determining the point of definition of relative constants.

The DOFILE (C) file generated in Block IV contains the instructions necessary for the COMPUTATION of the subscripted variable load value.

FLOW WITHIN SECTION II

To carry out the analysis and to deal with the various complexitites involved, there are six logical blocks in Section II.

BLOCK 1	Nest analysis, flow analysis.
BLOCK 2	Subscript combination analysis.
BLOCK 3	Relative constant subscript analysis.
BLOCK 4	Compilation of subroutines for computing relative constant index values.
BLOCK 5	Compilation of loop initialization, incrementing and testing instructions.
BLOCK 6	Reordering the DO file for input to Section III.

BLOCK I

The task of this block is to examine the DO nesting structure and the flow of the program. This information which Section I extracted from DO statements and Input-Output lists is contained in the tape table TDO, which on being read in, is further expanded into the 9 word table DOTAG to accommodate

the results of analysis. The DO is scanned to determine if it contains other DO's (DO Nest) and if any of the rules for DO nesting have been violated by the source program. The TIFGO table is then searched to determine if there are transfers within a DO loop. This is done by searching all DO loops for corresponding TIFGO table entries. If one is found an entry is made in the TRALEV table to indicate to Section III to compile indexing and saving instructions for the transfer. An entry is also made in the DOTAG table indicating a transfer exits in its range. The DOTAG and TRALEV tables are written out on tape at the end of Block I.

BLOCK II

The block II analysis is carried out for each subscript combination occurrance, at least one of whose subscripts is under control of a DO. Only the areas within DO's need therefore, be examined. The search for tags is carried out nest by nest, and within the nest DO by DO. The order in which the analysis is carried out is by selecting the inermost DO of a nest first and working toward the outermost DO of the nest. Any FORTAG (SECTION I table) entry being within and controlled by this DO is analyzed. If such a controlling DO is not found for a subscript, it is called a relative constant. The relative constant will be dealt with by Block III. If a transfer out of the range of a DO exists, a search is made within the DO for an equlavent subscript combination. If such a tag is found, the required value would be in an index register at the time of the transfer. A TRASTO entry must be made to indicate to Section III that instructions would be compiled at the point of transfer to save the index register value.

When all possibilities have been dealt with, the results of the whole analysis of subscript combinations are written out as the TAGTAG table entry on tape A4. This provides Block 5 with information so that it can compile the appropriate initializing and indexing instructions at the appropriate points. The DOTAG table compiled by BLOCK II is then written out on tape B2 as the 6th file with its record count as the 7th file. This DOTAG table is essentially the same as the one output by Block I with additional entries created by Block II.

BLOCK III

This block completes the subscript analysis by dealing with those subscript combinations not already analyzed in Block II, namely, pure relative constants. A pure relative constant is a subscript combination none of whose subscripts is under control of a DO. A relative constant can be defined in two different ways:

1. By appearing on the left hand side of a arithmetic statement or in an input-output statement, both which are recorded in the FORVAL table.

2. By a transfer out of a DO for that subscript combination.

Both of these situations were examined in Block II but were left for Block III to process.

To process the first situation above the FORTAG table entries are selected one at a time. It is only necessary to look at the ones not processed by Block II. The FORVAL table is then searched for the occurrence of the particular FORTAG entry. If one is found an entry is made in a table called TSXCOM. This table enables Section III to compile a TSX to a subroutine which will compute the current index value for the tag. The subroutine is compiled by Block IV.

The remainder of this Block is devoted to the second method of defining a relative constant. All FORTAG table entries not processed by Block II or the previous part of this Block, are now selected one at a time. The DOTAG table is then searched for a DO for one of the FORTAG symbols. If such a DO is found a search is then made to find a transfer out of this DO. When such a transfer is found an entry is made in the TRASTO table to indicate to Block IV what kind of subroutines will be necessary for the above conditions.

The tables generated by Block III are carried over to Block IV as memory tables.

BLOCK IV

This Block will process the tables generated in Block III and compile the subroutines necessary for computing relative constant index values. The subroutines are now written out on tape B2 files 8 and 9 as the DOFILE (C) and its record count.

BLOCK V

This Block compiles the necessary indexing instructions for the tags, using the results of the subscript and flow analysis provided by Blocks I and II. The information necessary for this compilation is contained in the TAGTAG and DOTAG tables which are on tape A4 and B2 respectively.

The process is broken down into two phases. The first being the alpha which provides the loading and initializing instructions at the beginning of a DO. The second is the Beta phase which compiles the incrementing, testing and resetting instructions at the end of a DO, (i. e., TXI, TXL and TIX respectively.)

The CITs are now complete as far as DO's are concerned and are now written out onto tape B3 as the first file. These are in reverse order, and are left this way for Block VI to take care of.

BLOCK VI

The order in which Block 5 compiles DO instructions for a nest is the backward sequence of α and β of the nest, although within each α and β block, the instructions are in the natural order. The α and β blocks of CIT's must therefore be inverted, so that Section III can merge the DO file with the COMPAIL file, output

by Section I. The beginning of each block is marked by an all one's CIT entry, and after reading a nest of CIT's (the end of a nest being marked by zeros), Block VI searches from the end of a nest until an all one's fence is found. The instructions just scanned are output as the DO file, and would correspond to the first α of the nest. Block VI then looks for another fence, and so on, until the whole nest is output. When the DO file is complete, control passes to Section III.

FLOW IN SECTION II

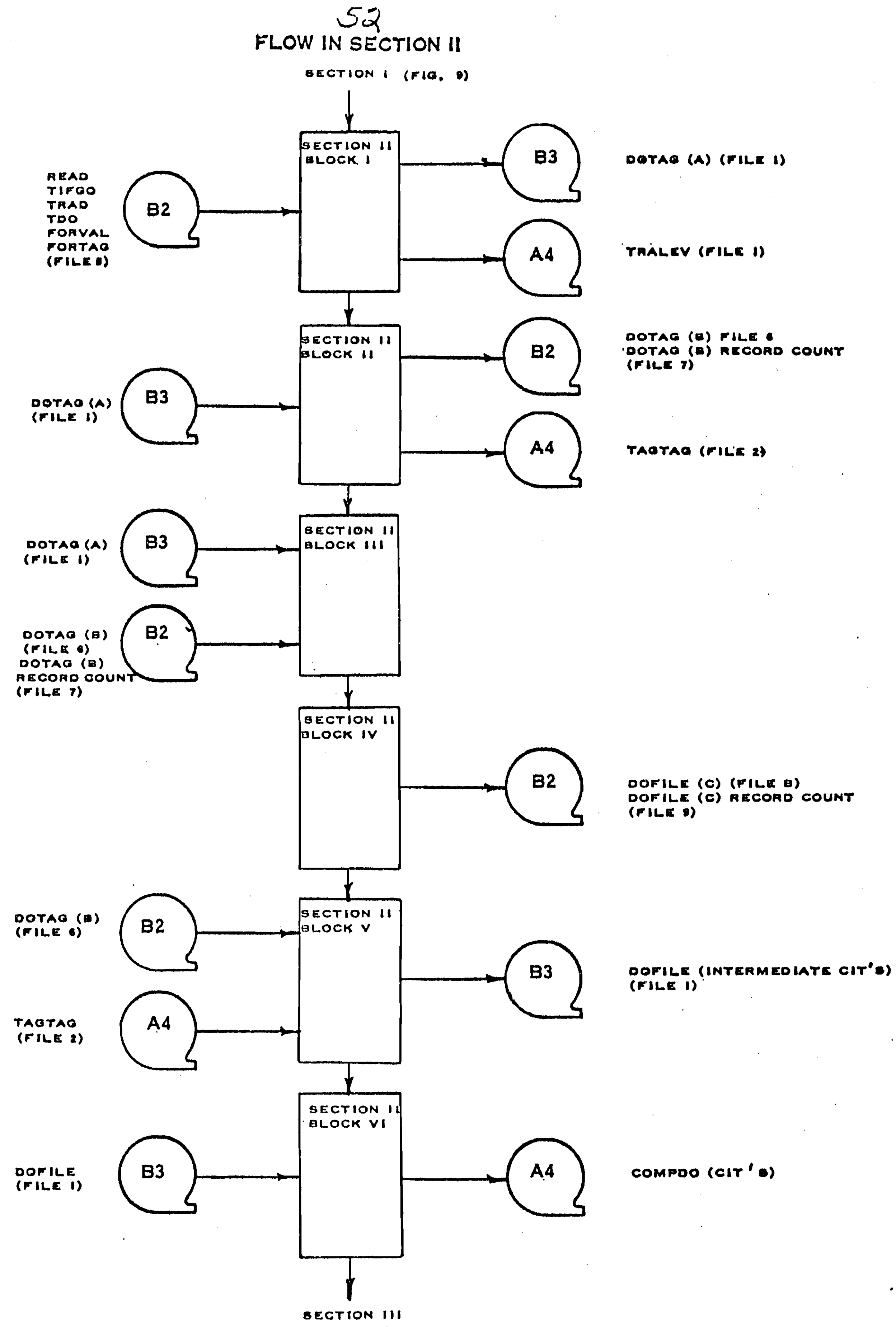


FIGURE 10

The MERGE has the primary function that its name implies. That is, it must merge or collate the different files of compiled instructions (CIT's) that are available to it. There is, however, an important additional function which the MERGE serves. This is the creation of an additional file of instructions. This additional file is based on information gathered by Section II and passed on to the MERGE in the form of tables.

The MERGE, therefore, falls naturally into three main divisions: Merge I merges the two files passed on to it by Sections I and II; MERGE II creates the additional file of instructions; MERGE III merges the two files of instructions now existent. The two files of instructions compiled by Sections I and II are the COMPAIL and the COMPDO files. The file created in MERGE II is called the TIFGO file. The results of the MERGE I file is called simply the FIRSTFILE. MERGE III, of course, merges the FIRSTFILE with the TIFGO file.

At the end of MERGE III, then a single file of CIT's exists and is passed on to Section IV. This single file is essentially the completed compiled program. That is, it contains all the instructions necessary for the translation of the source program, on the assumption that the object machine contains as many index registers as there are symbolic tags in the single file of instructions. Therefore, the remainder of the FORTRAN Executive Program is devoted to two main tasks:

a. Substituting absolute index registers for the symbolic index registers assumed up to this point.

b. Inserting the load and save index instructions required by the limited number of absolute tags.

It is important, further, to point out that the additional file of instructions created in MERGE II (TIFGO) does not result from any further analysis of the FORTRAN Source Program as such. Rather, it is compiled from tables which are themselves the result of such analysis. The MERGE, therefore, does no analytical work of its own; it simply stands at the crucial crossing point between the first part of the compiler which does the basic analysis and the latter part which handles the index register problem and the assembling problems.

Partially as a result of this critical position of the MERGE in the over-all flow of the FORTRAN Compiler, the MERGE is given certain additional subsidiary tasks to perform as it does its primary merging tasks. In this description, these subsidiary tasks will be listed and described in their appropriate place. It is only worth noting here that many of these tasks could theoretically have been done earlier; that they were not done earlier and were, instead, left to the MERGE is to a greate extent, a matter of convenience for the earlier analysis. The fact that the MERGE must make several

complete passes over all the CIT's makes it simple for the MERGE to make the insertions required by these subsidiary tasks.

FLOW WITHIN SECTION III

MERGE I

A. The merge of the COMPAIL and the COMPDO files occurs by simple numerical collation. The two files are on two separate tapes, COMPAIL as the second file on tape B2 and COMPDO as the second file on tape A4. These files exist in 100 word records, maximum that is 25 instructions per record. The first word of each instruction contains the internal formula number. The internal formula number is physically present only for the first instruction of the translation belonging to any unique source statement. The exception to this is where an input-output statement gave rise to more than one internal formula number. The remaining CIT's for any one source statement will have the first word containing all zeros. Therefore, the instructions exist in blocks, each of which is headed by an instruction with an explicityly stated internal formula number.

B. ADDITIONAL MERGE I FUNCTIONS

1. As a result of the Section II analysis, it may be found that certain tag (subscript combination) names must be changed. As you will recall the tag names were nothing more than a subscript combination. An arithmetic statement containing these tags may be used in several different DO loops. With this in mind it is quite obvious that the same tags really cannot be used, therefore the names are changed to overcome this problem. All subscript information is still retained with the addition of a flag indicating that the name is different. The name changes are recorded in the Change Tag table.

   Therefore, the first task that MERGE I performs is the editing of the Change Tag table. If the Change Tag table were unedited, it would be necessary for the MERGE to scan and test every tag field of every CIT appearing within the given ranges. In order to avoid this extended testing, the table is edited. This editing enables the exact location of the tags requiring changed names to be localized from the range of several statements to a single statement. The editing occurs with the aid of the FORTAG table which contains an association of tag names with specific internal formula numbers. The edited Change Tag tables, are the same as the unedited table with the exception that the range of the statement has been reduced down to a single statement number. While scanning the CIT's during the merging process a test is made on statement numbers to see if they match the number in the edited change tag table. If they do, the new names are inserted in the tag field.

2. Open Subroutines. Whenever an open subroutine reference is encountered, during the compilation of the arithmetic instructions, a CIT is compiled

which is merely a signal to the MERGE. This signal tells the MERGE not merely that an open subroutine is necessary at this point, but it also designates which open subroutine is given information about where the argument is to be found. The results of an open subroutine are left in the accumulator. Encountering this signal CIT or CIT's, the MERGE inserts the appropriate open subroutine. The designations referring to input arguments and output results, of course, pertain to the problem of arithmetic instruction linkage. With the compilation of these functions the MERGE has produced a single file of instructions called the FIRST-FILE.

MERGE II

MERGE II of Section III does not do any merging; it produces a new file of instructions. The tables used in producing this TIFGO file of instructions are the TIFGO and TRAD tables from Section I and TRALEV and TRASTO tables from Section II.

The need for the TIFGO file of instructions arises in the following way. The main body of computing and indexing instructions, included in the COMPDO file, are associated with the beginning and end of DO's. That is, the internal formula numbers of their CIT's have the internal formula numbers belonging to the DO statements within the range of DO statements. The entire Section II mechanism is set up to do compiling of the beginning and end of DO indexing instructions. Merge II only does the analysis necessary for the indexing instructions, required within the range of the DO's but does not compile the instructions. Instead it prepares the two tables TRALEV and TRASTO, which summarize this information.

All of these types of indexing instructions arise from the fact that transfers occur within DO's, specifically transfers going out of the range of a DO. In considering this problem, an entire DO nest, involving possibly many levels of DO's as well as many DO's on any given level, must be considered. Consequently, a transfer out of a DO within any DO nest may be a transfer entirely outside the nest (that is, to level zero) or to another DO within the nest (that is from level 1 to level n). Specific TIFGO instructions are caused by the fact that some indexing must occur before a transfer out of the DO is made, provided that the configuration of the DO within the nest, subscript combinations within the nest, and the uses of the DO indicates that indexing instructions which may precede any individual transfer. These six sets account for six different types of TRASTO entries. Either one or a combination of these sets may be required before any transfer. The TRASTO tables are numbered: this means that the instructions corresponding to each type of TRASTO entry must occur in the sequence indicated by the number. In setting up the TRASTO table entries, Section II determines the relevant facts with respect to both the location of the transfer instruction itself and the transfer addresses of any single source program instruction. No detailed explanation will be given for the specific reason for each of the six types of TRASTO entries.

The MERGE II analysis proceeds in this general manner. It uses the TIFGO table as its guide. In this connection, it must be remembered that the TRAD table is simply an extension of the TIFGO table. It simply supplements those TIFGO entries arising from computed GO TO and ASSIGN GO TO statements. When it comes across a TIFGO entry it checks to see if it is also in the TRALEV table. If it is not, there is no further concern for possible TRASTO instructions and the direct transfer addresses are compiled into the relevant transfer instruction. By direct, we mean here the number given in the source program, translated into its internal formula number. When the transfer or TIFGO entry is in TRALEV, there arises the further possibility of TRASTO entries for any of its addresses. The TRALEV table, it must be remembered, lists the levels of each of the transfer addresses. Consequently, a search is made through the TRASTO tables, first for those entries indicated by the TRALEV entry. If these conditions are met, then MERGE II compiles the indexing instructions corresponding to this type of TRASTO.

The most important subsidiary task performed by this part of the MERGE is the putting together of the ASSIGN CONSTANT table. This comes about as a by-product of the scan of the TIFGO table, which contains the ASSIGN GO TO entries. The ASSIGN CONSTANT table appears subsequently as the 5) block, containing the transfer instructions to each of the possible ASSIGN GO TO addresses.

With the completion of MERGE II, a new file of instructions exists, containing the computing and indexing instructions arising from transfers within DO's. This is the TIFGO File which is output as the 8th file on tape B2.

Control now passes to Merge III for the final Merge.

MERGE III

The task of MERGE III is comparatively simple. It simply does a direct merge on the FIRSTFILE and the TIFGO file. These two files are brought together and written as the first file on tape A4. Here too, the principles of the numerical collation apply. It might be noted that in some cases, MERGE II will simply have supplied transfer addresses for instructions which were partially compiled in Section I. That is, the Section I instructions will be complete except for addresses. In this case, the new instructions are brought together by "oring" one over the other.

A subsidiary task here is the insertion of the instructions necessary to branch to the subroutines for the computation of relative constant.

All that remains for the MERGE to do is follow the main file of instructions with the two secondary files of instructions compiled by Section I and Section II. These are the arithmetic statement function instructions and the A) subroutines, respectively. Updating of the ASSIGN CONSTANT, and FIXCON tables was required in Section III. These tables are now complete and are written as the eighth and ninth files on tape B2.

These are written as the second file on tape A4.

At the end of the MERGE all instructions resulting from an ahalysis of touring the source program are complete, except for the existence of symbolic tags, rather than at the absolute tags. This provides the main task of Sections IV and V.

FLOW IN SECTION III

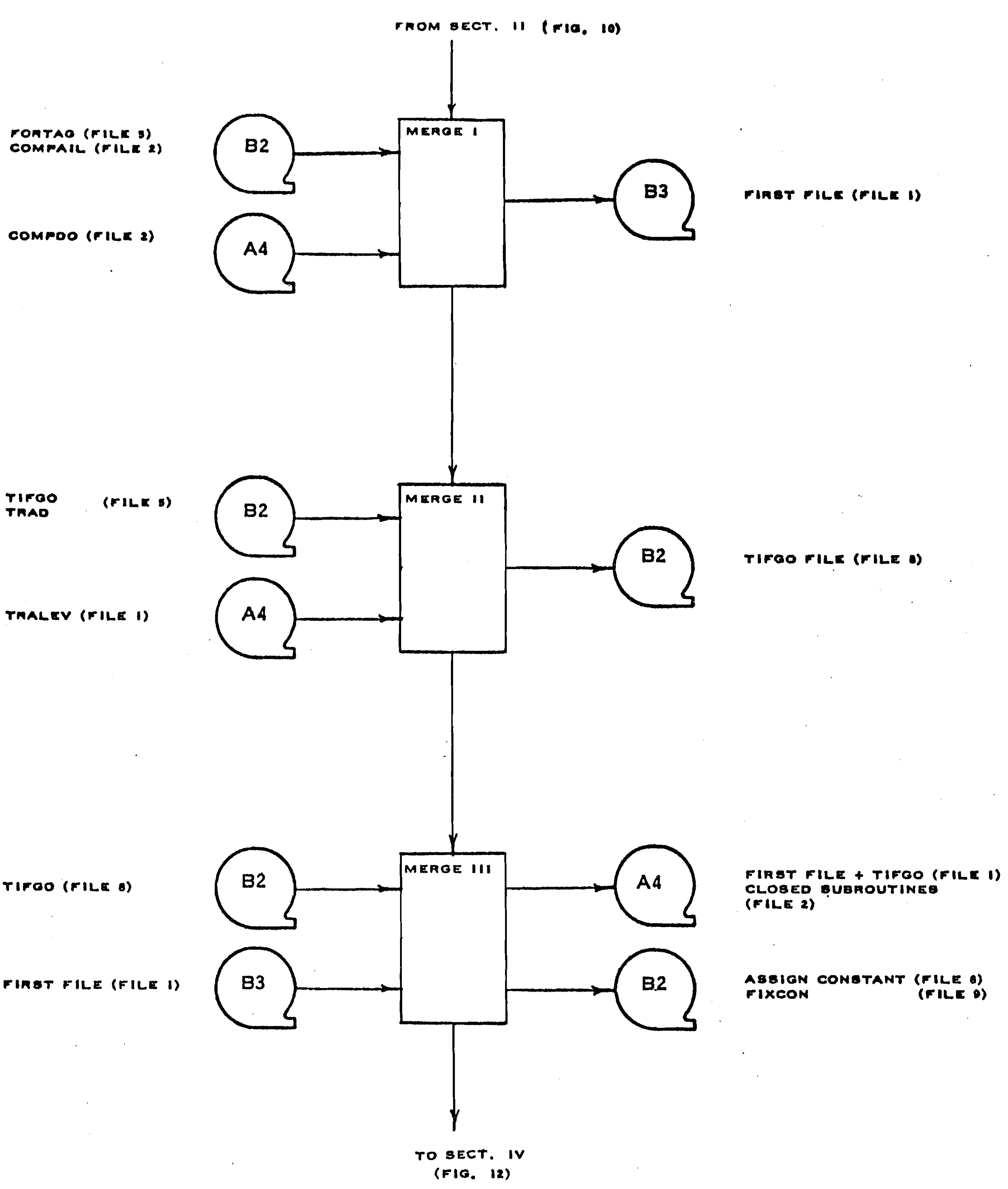


FIGURE 11

PREPARATORY NOTES FOR SECTIONS IV AND V 2.01.06

It was mentioned in the Introduction that the Fortran program fell into two divisions; the first, comprised by Section I, II and III; and the second by Sections IV, V and VI. At the end of Section III the program is essentially compiled. It is, in fact, compiled except that it exists in the CIT format and the program assumes an object machine with as many index registers as symbolic tags. Since, however, the machine will have three index registers, it is necessary to substitute assignments of these three for the indefinitely high number of symbolic tags. The object here will be to minimize the number of LXD's and SXD's -- load and save instructions -- required by this fact. By "number" here, we mean not only the number of separate physical instructions, but also the number of executions of them. That is, optimization with respect to time takes precedence over optimization with respect to space. For example, if a tag is used in a very high frequency, part of the program (such as the inner DO of a DO-next three levels deep), and a branch transfer is made to four different areas, each of which requires saving of the tag before it is reused, a single save instruction before transferring out of the high frequency area is logically sufficient. However, our method is to place four separate save instructions at the point of entry to each of the four branch points, thus eliminating the instruction from the path which would require most frequent executions of it.

This case also serves to illustrate some of the problems confornting Section IV and V -- the two sections whose concern this task is. It shows that there is a linkage, with respect to index registers, of different parts of the program and that details of the linkage must be known for efficient insertion of load and save instructions. For example, in the above case, the SXD will not be used on any of the four paths where it is not required. Furhtermore, a comprehensive knowledge of areas and their expected frequencies of object time flow is necessary. As a corollary to these problems, there is the one of avoiding the SXD instruction for a tag which is no longer to be used. That is, the tag can be efficiently killed by over-loading it in its index register when the next use of the tag in it requires a load instruction. If the last reference to this tag is one that changed its value, it must be saved; if the last references did not change its value but merely used its earlier established value, it is not necessary to save. Here, a distinction between active and passive references to tags is necessary.

This entire complex of problems comprise the task of Sections IV and V. The work required of these sections falls naturally into two divisions, allowing the division of labor between them. Sections IV informs Section V of the divisions of the object program for purposes of flow analysis and the relative frequency of paths of flow over these divisions. Its task is much the lesser of the two sections. Section V then uses this information along with a knowledge of the specific tags required by each of the "divisions" to assign absolute index registers and compile necessary indexing instructions.

Before giving the general discussion of the work of these two sections, it is well to note how this work was presupposed in the handling of symbolic index registers by the earlier sections of Fortran. Essentially, this can be stated very simply: the earlier sections simply ignored the problem and acted as if as many index registers as were needed were available. That is, load instructions may appear in sequence up to any number. The assumption is the "saves" necessary to make the "loads" effective will be added later. The important thing to note here is that SXD's and LXD's are not always coupled as the previous discussion might imply. There is an asymetry between them; the earlier sections have complete freedom with respect o LXD's, but very rarely compiling an SXD. On the object program level this difference is reflected in the cells which the SXD's and LXD's address. Section II instructions for example, mostly refer to the subscript symbol cells in the regular data area of core storage. On the other hand, section V's instructions always refer to the specially designated eraseable area for storage of index registers. These eraseable storage cells are referred toa s the C) cells. The actual designation is C)i, where i is an increment resulting from the conversion of the symbolic tag name. By means of this device there is co-ordination between Section V references to such tag storage cells and whatever section II references are necessary.

SECTION IV 2.01.07

Section IV has for its main task the assembling of four different tables. These are the BBB table, the Predecessor, the Successor table, and the Tag List table. The primary input to section IV is the single file of merged CIT's; section IV also uses other tables created earlier. The BBB table is a list of the Basic Blocks of the object program, plus indices referring to each Basic Block's Successors and Predecessors. A Basic Block is a stretch of program into which there is only one entrance and from which there is one exit. "Exit" must here be interpreted in the logical sense; that is; it may consist of more than one transfer instruction, going to a variety of Basic Blocks. Each of these Basic Blocks, then, is a Successor Basic Block. As implied by this, section IV must mark off the Basic Blocks of the program and determine the Successor and Predecessor Basic Blocks for any one Basic Block. A BBB entry corresponds to each Basic Block; it has references to the Predecessor and Successor tables denoting its Predecessor and Successor Basic Blocks. But section IV work goes beyond this. It must provide the information to section V concerning frequency of paths of flow. Therefore, the form of the Predecessor and Successor table entries which section IV passes on to section V will contain, in addition to the Basic Block reference number, a number denoting relative frequency of transition between the two Basic Blocks. Here, the two Basic Blocks refer to the BBB Basic Block and the Basic Block or Blocks of the Predecessor and Successor table that it designates. In order to achieve these relative frequency numbers, section IV performs a simulated flow over the program going from Basic Block to Basic Block.

The major problem here is in determining which Successor Basic Block to go to when, as a result of conditional transfer, a possibility of more than one Successor Basic Block exists. At this point a "Monte Carlo" technique is used. A random number is generated and, in accordance with the numeric possibilities of succession indicated by the frequence statement entries for that conditional transfer, a particular Successor Basic Block is chosen. The random number is meant to assure that over the long run of the entire simulated flow, the possible Successors will be chosen in the proportions indicated by the Frequency entries. Where no frequency entry is made by the source programmer, the assumption is that of equal probability for all paths of succession.

Some of the special problems encountered during the performing of this simulated flow are those given by conditional transfers where the conditions are set directly in the source program (such as ASSIGN GO TO's and Sense Light Tests) and DO's involving variable parameters. For both of these additional intermeditate tables are necessary. In the case of DO-nests, three general circumstances, involving flow analysis probelms, may occur. One is a DO-nest whose DO's all have constant parameters and contain no transfers, another is constant parameters with transfers, and the third is a DO-nest at least one of whose DO's have variable parameters. For the last

mentioned circumstance, either the frequency entry for the DO must be used or barring that, a frequency of five is chosen for the number of times of repetition of the DO range.

For purposes of the simulated flow, a large number is chosen, which is a function of the number of Basic Blocks and distinct transfer branches occurring in the problem. For every transition between a Basic Block and its Successor that is made during that simulation, this number is ticked off by one. The flow ends when this number equals zero.

It should be pointed out, finally that this simulated flow has nothing whatever to do with the individuals instructions of the problem. It is concerned only with Basic Blocks as units and not with the contents of a Basic Block. As far as section IV is concerned a Basic Block may actually contain one hundred instructions or two instructions, and these instructions may contain many tags or no tags: section II treatment of it is the same. It may also be mentioned here that the division into Basic Blocks is based on an examination of the compiled instructions. Of course, the recognition of transfers -- beginning with the letter "T" -- is vital. For this reason, section I finds it necessary to use pseudo-names in the CIT's of some of its instruction. It does not wish section IV to think that these end Basic Blocks when actually they do not.

After the flow analysis is completed, section IV assembles the BBB, Predecessor, and Successor tables. These are a summary of the Basic Block flow and relative frequency of this flow. The BBB entries also contain a designation of the type of ending for each Basic Block; absolute transfer, preset transfer, conditional transfer, and so forth. The last significant item that each BBB entry contains is an index to the Tag List entries belonging to it. The Tag List table is made up at the end of section IV it is a list of all symbolic tags contained in the CIT's of the program together with a code designating the type of instruction referring to the tag. The index to this table that is placed in the BBB entry, then tells which tags occur in each Basic Block of the program and how they are used.

FLOW WITHIN SECTION IV

Section IV is logically broken down into three parts called blocks.

BLOCK I

The first task of Block I is to divide the object program into basic blocks, a basic block being a stretch of program with but one entry point, and one exit point. In order to do this the merged CIT's are read in from tape A4 file one. A pass is then made over the CIT's looking for transfers, tests and skip type instructions. Absolute and conditional transfer addresses, and the location of instructions following skip type instruction, or TXL's (end tests of DO's) these are all entered in the BBLIST table once, in algebraic order, by means of a binary search technique.

During this pass, when a TXL is encountered, both its location and address are entered in the new table DOLIST, thus providing a list of the beginning and end locations of all DO's in end location order. The DOLIST table will be used later in this block for analysis of the flow of the program.

The TIFGO, TRAD and FRET tables are read in from the fifth file of tape B2. A new table TIFRD is now formed from the assign and assigned GO TO entries in the TIFGO table, together with the associated entries in the TIFGO table, together with the associated entries in the TRAD tables. (TIFGO entries are of fixed word length, and the TRAD table was therefore created to accommodate all possible Assign GO TO and transfer addresses.) At the same time, all the transfer addresses are entered in BBLIST table. When all information is extracted from TIFGO and TRAD and entered in the TIFRD table, TIFGO and TRAD are of no more use to Section IV. The BBLIST table now constitutes a list of the beginnings of all basic blocks in the program, in the order in which they occur. The basic block number which is referred to in block II, is the relative address of the particular basic block within the BBLIST table.

The FRET table that was read in from tape earlier is now examined, all frequency entries that correspond to DO statements are extracted and placed in the new DOFRET table. This is done in preparation for the simulated flow that will follow. The remaining FRET entries are now moved up to occupy the vacated positions.

The table DOLIST, created earlier and ordered on the ends of the DO's, is now sorted into the order of beginnings of DO's. When these are equal, the DO with the largest remaining location takes precedence. The table is now compared with BBLIST, and the internal formula number in DOLIST are replaced by basic block numbers. The DOTAG table is now read from file 6 on tape B2 and scanned. Each time a DO is encountered which has a transfer in its range (a DO with an IF) an indicator is set in the appropriate DOLIST table indicating this.

The loop count is now computed for DO's with constant parameters. If any of the parameters are variable the loop count is taken from the DOFRET table providing the frequency was given in the source program. If no DOFRET entry exists for this DO's an arbitrary loop count of 5 is given.

This terminates the work of block I, control now passes to block II.

BLOCK II

Tape A4 is rewound and the complete CIT is again read.

A second pass is now made over CIT, producing the three principal tables with which simulation is accomplished, namely BBTABL, SET and TRATABLE. There is one 1-word entry in BBTABL for each basic block in the program,

but there may be several SET and TRATBL entries corresponding to this one BBTABL entry. At the beginning of each basic block, the next available location in SET and TRATBL are entered in the BBTBL, thus providing a key to information which will be accumulated during the pass, for the basic block.

The TRATBL table contains for each basic block in the object program the basic block numbers of it successor basic blocks (those to which a transfer is made). Associated with each of these successor basic block numbers is a counter which during simulation will keep count of the number of times the path between the predecessor and successor in question has been traversed. The SET table contains information pertinent to the three types of setting that must be done during simulation. The three types are:

1. The setting of the assign GO TO addresses.

2. The setting of sense lights. (Dummy lights only are maintained during the program simulation pass, in block III, not actual machine lights).

3. The resetting of DO indexs for DO's which have transfers out of their range.

The remainder of the analysis during this block is concerned with obtaining information about basic block endings. If the instruction following the current one begins a new block, (its location is in BBLIST), then the current instruction should end one. If the following instruction is a skip type instruction or a conditional transfer these also constitute basic block endings. It can be seen then that a basic block can have several types of endings. These endings are coded and entered in the BBTABL which also contain the TRATBL and SET addresses for the basic block.

BLOCK III

The object program is now simulated many times in order to obtain statistical information concerning relative frequencies of flow paths taken. The number of simulations is equal to the total number of transfers in the program multiplied by 128, which means that the more complex the program, the greater the number of simulations. The program is stepped through basic block by basic block, using the BBTABL as a guide, starting with the first basic block. No reference is made to the actual compiled instructions.

A BBTABL entry is selected, and corresponding SET table entries are obtained. Settings are made according to these entries, that is, the SET address, or setting, is stored in the location given in the decrement. For instance, a SET entry turning a sense light on would cause a 1 to be stored in the dummy sense light address. A Set entry to reset the loopcount to be stored in the TRATBL of the DO. (This is where the iterations are counted during the simulation.)

Depending on how the basic block ends the proper successor basic block is chosen. This process continues until the simulation is over.

After the simulation has been dealt with, a process is begun to adjust the flow counts of basic blocks which lie within DO's without IF's.

It may be recalled that during the simulation, these DO's where not simulated as were the DOs with IFs and therefore the flow counts of the basic blocks within them have not used the loopcounts of the DO's.

The third and final pass is made over CIT to collect tag information for Section V, and during this pass, two new tables are built. These are the TAG table, and the BBTAG table. For each occurrence of a tagged instruction, an entry is made in the TAG table. This entry contains the symbolic tag name, together with a code which tells what kind of instruction it is used with.

Each time the beginning of a basic block is encountered, a BBTAG entry is made, containing the number of entries so far in TAG. The last BBTAG table entry is a dummy and contains the total number of TAG entries.

The BBTBL table and BBTAG table are now combined to form the BBB table which will be used in Section V.

The TAG table is now written as the third file on tape B3. The BBLIST table that was created in block I is now written out as the fourth file on tape B3. The BBLIST table was not written sooner because it will be used after the TAG Table in Section V, this saves time in moving tape.

Control now passes to Section V.

FLOW IN SECTION IV

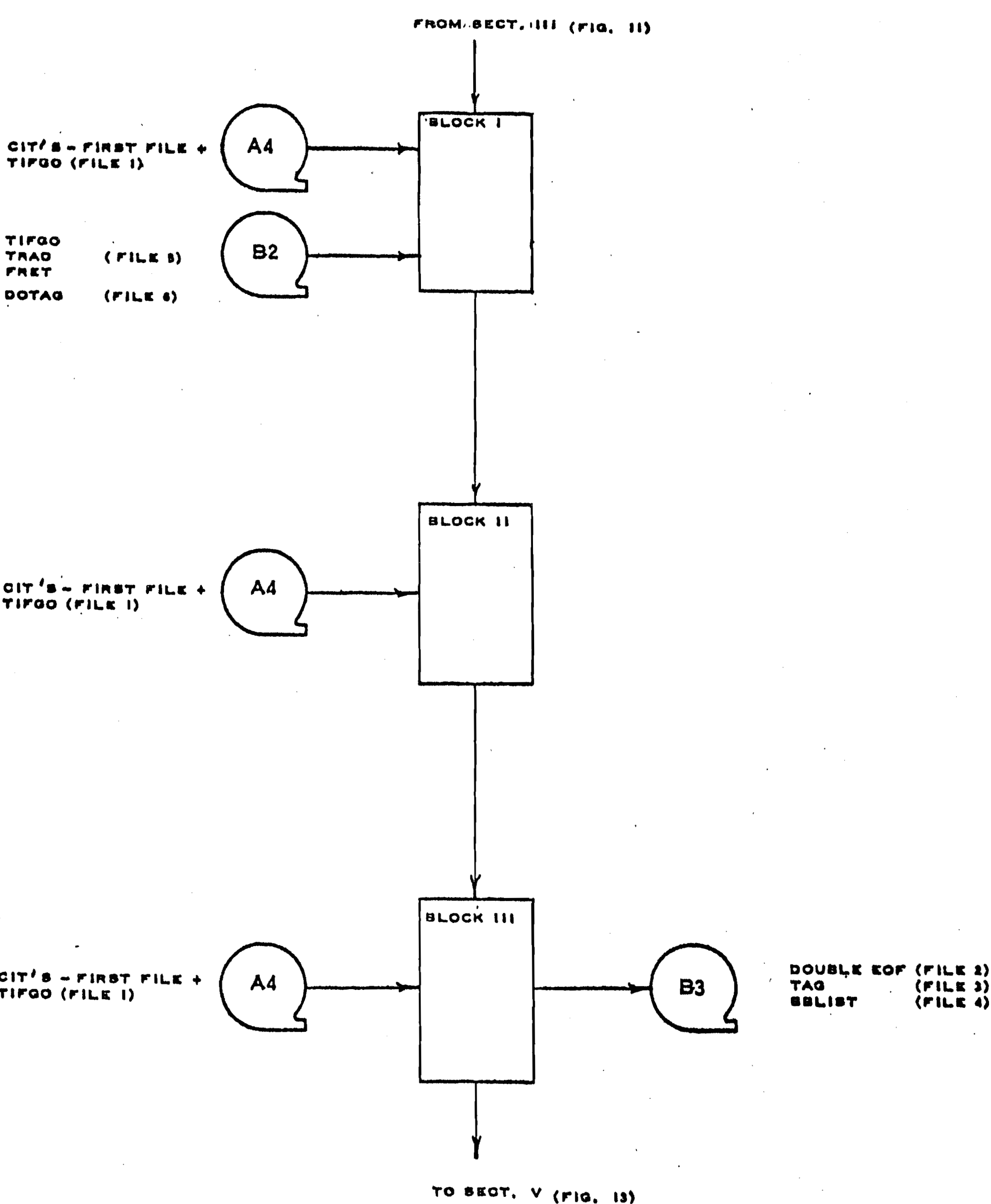


FIGURE 12

Section V must now substitute references to tags 1, 2 and 4 for the symbolic tags which occupy the address portion of word 4 of the CIT's. As a corollary to this, the loading and saving instructions would be inserted for the appropriate index registers. These will load from and save in the group of cells designated as C) - cells. The information contained in the four tables created for it by section IV are sufficient to do this.

To perform this main task, section V operations fall logically into two broad divisions. These are Region Generation and LXD and SXD assignment.

Region Generation is the method of setting aside a portion of the program, consisting of one or more basic blocks, for independent treatment with respect to index register assignment. After a set of basic blocks have been set aside as a region and treated, it then, as a region, becomes a separate unit liable to be incorporated in a new region along with other basic blocks. The flow configuration of a problem determines when a region itself becomes part of another region. When it does it loses its identify for the new region is an independent and separate unit. Ultimately, of course, all regions and basic blocks become absorbed into a single region which is the entire program. At this point the section V analysis is complete. In referring to "treatment" above, we mean the LXD and SXD assignment.

There is, then, an interweaving of the operations of the two main divisions of section V, Region Generation and SXD and LXD assignment. (The second of these divisions is often referred to as the LXing Pass.) The regions grow recursively until the entire problem is one region. At any given time during this recursive treatment, several regions may exist independently or one only may exist.

The "treatment" of a region is based on another type of simulated flow through it. This simulated flow affects the symbolic index register usage occurring in the region. In cells representing the three index registers, the symbolic tags are loaded, then comparisons made with successive symbolic tags, as these are revealed in TAG list. When it becomes necessary to save one of the three index registers, a look ahead through Tag list is made to determine which it is preferable to save; that is, which is the last index register used further ahead in the program. It should be noted that the two fundamental problems are involved here. One is simply the problem of assignment of index registers; this involves the compilation of LXD's and the choice of an index register. The other is the problem determining when to save an index register when the quantity is subsequently going to be overwritten by a load into that index register.

With respect to the second of these two problems, a tag must be saved to initialize the appropriate C) cell for later loading, and to handle "active" index registers. "Activity" is denoted by the type of reference made to the

tag in the tag instruction. The TAG List code referring to the tagged instruction tells essentially whether that instruction is active or passive. An active instruction is simply one that changes the value of an index register (such as TXI or LXD) and a passive instruction is one that uses the tag only (such as CLA). Where "activity" is present and a subsequent load will over-write the index register, an SXD is inserted following the last use of the symbolic tag. Activity has meaning applying beyond the context of the immediate region in which it is discovered. It may subsequently be found that in succeeding regions a new tag value is required. Activity for regions, then, must be carefully noted.

As a result of this simulation within a region, the index registers upon entry into a region and upon exit from it are assigned certain symbolic tags. These are noted in the BBB entry for the basic block as its entrance and exit conditions. When a region -- which, of course, has been previously treated -- is encountered, a match must be made of the exit conditions of the last basic block with the entrance conditions of the basic block by which that region is entered. Where necessary, permutation of the index registers within the already treated region takes place to force compliance. If a match cannot be made, LXD's are called for at the head of the region. These LXD's are called inter-block LXD's because they concern the linkage between regions as distinct from basic blocks. There are also inter-block SXD's. These result from activity within a region already treated. The SXD is placed at the head of the region using the active tag. In this way, incidentally, the deployment of save instructions among different low frequencies paths rather than the single save instruction with the high frequency path occurs.

Continuing to work in this way, from region to region, the high frequency paths of flow naturally receive priority in the assignment of index registers. The SXD's and LXD's are inserted enforcing conformity of the low frequency paths with the already assigned high frequency paths.

During this entire analysis, Section V records within tables the information needed to make the actual compilation and insertions of the LXD and SXD instructions. The compilation itself occurs later. A new table, the STAG table, is created for recording these instructions as needed within a region. The necessity for inter-block instructions is recorded in the Predecessor table.

The inter-block instructions, because they are at the head of a region, must take their own location symbols so that transfers may occur to the block. These location symbols are: D), when the instruction is an LXD, and E), when it is an SXD. A TRA instruction may have to be added to bypass these instructions when entry to the block occurs from the part of the program immediately preceding it.

Section V, also because it makes a pass over the entire program, performs certain small optimizing operations on the compiled program.

FLOW WITHIN PART I

Section V uses the information about basic blocks (which has been passed on from Section IV) to combine these basic blocks into larger groups called regions. The flow within a region is simulated in order to determine which symbolic tags are required and which index registers should be assigned to them.. During the course of simulation, flags are set to indicate where an SXD or LXD is required. When a region has been treated it may be combined with other regions. Eventually all basic blocks will have been combined into a single region, and the complete object program will have been treated.

The most frequent paths of flow between basic blocks are handled first. Since an SXD or LXD is not inserted until necessary, this results in the most frequent paths having the least of them, and therefore a faster object program.

The first step of this treatment is the formation of the Looplist table showing the path of flow through a new region. The starting point in Looplist formation is the most frequent link between basic blocks which has not yet been considered. (The PRED and SUCC tables have frequency counts which are examined to find most frequent predecessor or successor basic blocks. When a link has been treated, the entry which refers to it is marked with a minus sign so it will not be considered again.) Looplist is expanded by including as many of the most frequent unconsidered predecessors as possible and then as many of the most frequent successors as possible. If the most frequent link is to a basic block which is in a region previously treated, this whole region is included in the Looplist. Thus a Looplist may consist of a combination of untreated basic block and regions (or basic blocks which have already been treated).

Regions are classified as either opaque or transparent. An opaque region is one in which all three index registers are used. A transparent region has one or more index registers still available. When an opaque region is encountered while forming Looplist, no more links are added to it. However, a transparent region may still be added to, since there are index registers available within it to which tags can be assigned.

The Looplist table consists of one word entries for each basic block or region. A code in the prefix of the word indicates whether it refers to a basic block, a transparent region, or an opaque region. If the entry is a basic block it contains the BB number, and if the entry is a region it contains the numbers of the basic blocks at the entry and exit points of the region. The end of Looplist is indicated by a word of all sevens.

From the starting point in Looplist, the most frequent predecessors are added one at a time until one of the following conditions have been encountered. If an entry is already in the current Looplist, this makes Looplist a loop and prohibits further building. If an entry is an opaque region or if there

are no unconsidered predecessors, then additions are made at the other end, and the most frequent successors are looked for. Again the same conditions apply. Basic blocks or regions are added until a loop or an opaque region is encountered, or there are no unconsidered successors to the last entry. When a Looplist has been completed, it will reflect the flow in a section of the object program. It may have a loop, reflecting a loop in the object program. In such a case, if there is an end of Looplist not included in the loop, that section is eliminated from Looplist. Only the loop itself will remain in Looplist for further treatment in this Looplist. On the other hand, the Looplist may be a string with no loops, having been stopped in both directions by encountering an opaque region or by finding no unconsidered links to it.

After the Looplist has been formed, the path of flow indicated is ready for treatment. Then next stop is to prepare for simulation which is done in the 2nd LXing pass. If the Looplist is a string, then the only preparation necessary is to mark the initial conditions of the IRs. If the Looplist is a loop, however, the 1st LXing pass is entered.

The index registers used by the object program are simulated in Section V by three storage locations which are continually updated. These cells are referred to as IRs. During simulation they will contain the symbolic tags needed by the corresponding part of the object program.

The 1st LXing pass simulates the loop in order to find out the condition of the IRs when the 2nd LXing pass is begun. Each basic block in the Looplist is examined to see which tags are necessary. This is done by referring to TAGLIST (which was read in from tape B3 file three and contains a list of all tagged instructions in the object program). Tags are placed in the IR cells as required. When a region is met in Looplist, the previously determined exit conditions from the region are placed in the IRs. After the whole looplist has been done the IR cells contain the initial conditions for the 2nd LXing pass.

Simulation in the 2nd LXing pass is much more complex than the treatment of the 1st LXing pass. Entries are made in tables when a tag must be loaded into or displaced from an IR. STAG is used to record LXs and SXs within a basic block, and PRED is used for those between BBs. When a tag is displaced, its value is saved if necessary in a cell set aside for that purpose. These tag cells are thus kept up to date so that the next time a tag not already in an IR is required, an LX fromthe corresponding cell will be correct.

In order to determine when an SX is necessary, the concept of activity is used. When the initial value of a symbolic tag is set, or when that value is changed by an indexing instruction such as TXI, the IR becomes active. This means that the value in the storage cell corresponding to that tag is outdated. This fact is recorded in cells referred to as AC 1, 2 and 3, one for each IR. If this tag must be displaced while treating the same Looplist, an SX will be introduced immediately after the active instruction, thus updating the tag cell and ending the activity. But if the tag has not been displaced from the IR after the treatment of the Looplist, the section of Looplist is marked active

from the point of the active instruction. This is done by placing activity bits in the BBB entry for each BB in that section of Looplist. When flow goes through such a BB in a subsequent Looplist, the activity will be noted, and if a future SX is necessary it will be placed in the link from the region containing the BB.

A tagged instruction that does not change the value of the tag, does not require this treatment. Such an instruction is called passive. A passive instruction, such as CLA or TXL, only makes it necessary to have the appropriate tag in an IR. When a tag is required that is not already in an IR, an LX from the appropriate tag cell is called for. Because of the way activity is handled, the tag cells may always be considered up to date. All that is necessary is a determination of the most desirable IR to use. If they all contain tags, this is done by searching ahead to find out which of the tags presently in the IRs will be needed last.

Treatment in the 2nd LXing pass begins with the first entry in LPLST and proceeds in sequence to the last. The three types of entries, 1) BBs, 2) transparent regions, and 3) opaque regions, are distinguished by a code number and each is treated differently.

If there are still active IR's remaining, just as the 1st LXing pass was required, another pass, the active pass, is executed. LPLST entries are examined and treated again in a manner similar to that of the 2nd LXing pass, with SXs called for where necessary. After each LPLST entry has been dealt with, a test is made to see if there is still an active IR. Eventually they will have all been taken care of and the active pass finished.

It only remains to bring the appropriate tables up to date. The PRED and SUCC entries that have been treated are flagged negative. BBB has the new region references entered. And finally the region table is updated by wiping out obsolete entries (regions absorbed into the new one) and making the entry for the new region.

Part 1 repeats the cycle of looplist formation and treatment, with new, large regions absorbing old ones, until all links between basic blocks have been treated and the object program consists of a single, all encompassing region.

FLOW WITHIN PART II

In part I, tags were continually reassigned to index registers on the basis of the optimal match that could be achieved. This reassignment was done by changing the permutation numbers in the 2nd word of the BBB table. Part 2 makes the actual changes in the appropriate tables on the basis of the final permutation numbers. It also combines BBLIST (read in from tape B3 file four), with BBB for convenience later on.

Each basic block is examined in sequence. The location word of CIT for the

first instruction in each BB (which has been put in BBLIST by Section IV is placed in word 6 of BBB. Then the LX and SX bits in the PRED entries are changed according to the permutation numbers. Next, the STAG entries are similarly updated. Then, for each BB which ends with an Assigned GO TO, the BB number of the last assigned GO TO is stored in word 2 of BBB. This is done in order that part 3 may find all GO TO N BBs easily. Finally, the entrance and exit conditions in words 3, 4 and 5 of BBB are reentered in accordance with the permutation numbers.

FLOW WITHIN PART III

Section Vmay insert SXDs and LXDs at points in the object program which are transferred to by an Assigned GO TO. It may therefore happen that the transfer should no longer go to it's original address, but to one of the SX's or LX's. Part 3 handles this by making the necessary changes in the assign constants.

The Assign Constants are read in from tape B2 file eight. The Basic Blocks are examined one at a time to determine which ends with an Assigned GO TO. For each one that does the appropriate PRED entry is found. From the SX and LX bits in PRED, the correct transfer address is then prepared. The assign constants are then compared to the first instruction of each successor BB, and when a match is found the Assign is replaced by the new symbol. The SX bits are also stored in the prefix of word 2 in BBB for use in Block IV. When all the Assigned GO TO BB's have been found and treated, the altered Assigning Constants are written back on tape B2 as file ten for use by Section VI.

FLOW WITHIN PART IV

Part 4 does the actual compilation of instructions on the basis of the information passed on by the previous parts of Section V. The bits in PRED indicate when inter-block SX and LX instructions are required. STAG has the necessary information about when to compile an LX or SX immediately preceeding or following a tagged instruction.in CIT. The real index register assignment for each tag is also indicated by bits in STAG. Part 4 follows these directions while compiling. In addition, some minor optimizing is done.

A pass over CIT is made, and the method used to bring in blocks of instructions and scan them for tagged instructions and endings of BBs is similar to that used by Section IV. This is the only time that Section V looks at the CIT. The instructions are brought in from tape A4 and examined in groups, and when the necessary modifications have been made, they are rewritten on tape B3 for Section VI.

First, part 4 considers a basic block as a whole. By referring to the BBB and PRED entries for the BB, a list of the necessary LXs in the links to the BB is formed. Then a list of the necessary SXs in the link is formed. When the SX lists are compiled for the various PREDs, it may happen that two or more of these are the same. The symbolic locations of these SX lists will be different however, because of the number of the PRED entry is contained in the location symbol. A SYN pseudo instruction is compiled in this case.

A "sequential transfer", which is one from the last instruction in the previous BB to the first instruction in this BB, is compiled if necessary. The transfer may be around one or more lists of LXs and SXs associated with other PREDs for this BB. On the other hand, the transfer may be deleted if no instructions had to be inserted between the BBs.

After the inter-block SXs and LXs have been taken care of for each BB, all the instructions within the BB are handled. All CIT entries without tags are, of course, kept. A CIT entry which already has a real tag of 4 is checked to see if it is an SXD or LXD, which has been placed around a subroutine calling sequence. If such is the case and if IR4 is not necessary for Section V assignment of a symbolic tag at this point, the SXD or LXD will be deleted. The SXD location will be compiled as a BSS 0 since it may be referred to elsewhere in the program. When an LXD after a subroutine calling sequence cannot be deleted because IR4 is necessary, if the following instruction is a similar SXD, both are deleted. As a result, a series of TSX instructions will have the unnecessary SXDs and LXDs removed.

When an instruction with a symbolic tag is encountered in CIT, the STAG entry referring to it is examined. If STAG requests it, in LX from the tag cell will now be compiled. Then the instruction itself is compiled and next an SX to the tag cell if so indicated. Each of these instructions will have had the real tag assigned also on the basis of the STAG entry. The LXP pseudo instruction is deleted when it occurs, as is a DED. These instructions were put in as signals to part 1 and are no longer required.

After an instruction has gone through the foregoing treatment, it is checked to see if this is the end of the BB. If it is not, the next CIT entry is examined and treated. When the ending is found, any transfer addresses are examined to see if the transfer is to a BB with SXs or LXs in the PRED link. If it is, the address is changed to the location of the proper SX or LX. Any "sequential transfers" are not compiled at this time, however. An indicator is stored if there is one, and the deletion or insertion of this transfer is left up to the analysis of the PRED link when the next BB is treated. The case of an Assigned GO TO ending is treated differently. The SX bits placed in word 2 of BBB by part 3 are examined and SXs compiled where necessary. Then the transfer to N is compiled.
When all the instructions in the BB have been treated and the ending taken care of, the next BB is dealt with as before. The process continues until the end of CIT is reached. Finally the relative constant routines are copies at the
end of CIT is reached. Finally the relative constant routines are copies at the
end of CIT and control passes to the Section V'.

SECTION V' 2.01.09

The purpose of Section V' is to add to the CIT file all constants and source program data appearing in the symbolic listing, except for the B) and 9) constants, for use in Section VI.

At the end of Section V the CIT file contains the entire working program, the arithmetic statement function definition subroutines, and the relative constant computation subroutines A). Available to Section V' are tables on tape B2, containing the assign constants 5), fixed point constants 2), floating point constants 3), format BCD words 8).

Assign constants are in the ASSIGN table, one record of file ten, on tape B2.

Fixed point constants are in the FIXCON table, one record of file nine, on tape B2.

Floating point constants are in the FLOCON table, record one of file four, on tape B2.

Format BCD word are in the FORMAT table, record two of file four, on tape B2.

Universal constants 6), are compiled for all programs, as certain subroutines assume that they be present.

To initialize V', the last CIT record previously compiled is read from tape B2. To this record, and to additional records as they become full, are added the four word CIT for each word in each of the tables. The records are written out on tape B3 as they become full (100 words per record).

When all constants have been compiled, the partially filled final CIT record, if any, is written off on tape B3, an end of file is added to mark the end of the CIT file. Control now passes to Section Pre VI.

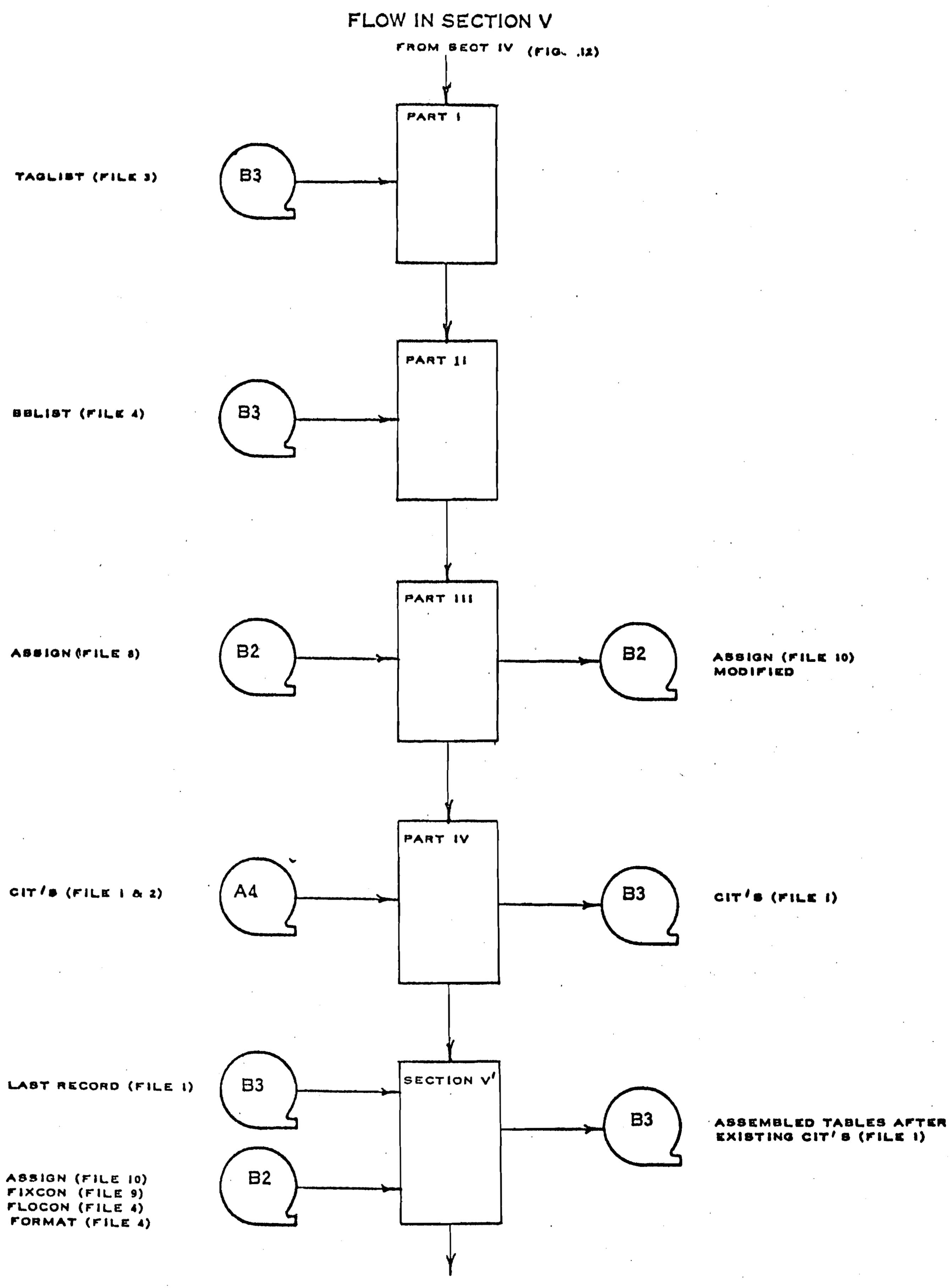


FIGURE 13

Since the object program is symbolically complete, all that remains is to assemble the compiled instructions, producing a relocatable binary program ready for loading and running, and a listing of certain information concerning the program being compiled. Section VI is primarily an assembler, differing little from any standard assembler. It builds a table of symbol names and relocatable locations, translates BCD operation codes to binary instructions, replaces symbolic locations with relocatable locations, and assembles the binary operation code, decrement, tag, and address into one word which shall occupy one location in memory during object time. In addition, options are available to include in the binary deck library subroutines for use at object time; to punch on line a row-binary deck, preceded by the BSS loader if a main program; to punch on line a column-binary deck; to produce a third file of SAP-like symbolic listing of the compiled program; to produce a binary symbol table.

FLOW IN SECTION VI

Record 34

Record 34 (also called Pre VI) completes the CIT file. It uses information in the CLOSUB, SUBDEF, and HOLARG tables, and scans during prefirst pass the entire CIT file on tape B3 for those instructions referring to arguments which require initialization. It writes the transfer vector, and if a subprogram, prolog, and initialization on tape A4; copies during presecond pass the CIT file from tape B3 to tape A4 changing certain pseudo op codes used internally in FORTRAN to machine op-codes; and adds to the end of the CIT file Hollerith arguments, and initialization addend constants. It also reads into memory tables required by section VI.

Included in this record is a common binary search routine which remains in memory for use by subsequent parts. The maximum table length which can be searched by this routine is 16383 words, which is the effective limit to the length of any table which must be searched.

Record 35

Record 35 builds that portion of the dictionary which is defined by COMMON, DIMENSION, EQUIVALENCE, CALL, SUBROUTINE, and FUNCTION statements, and any statement referring to a library subprogram, such as PRINT or X = SQRTF(B). The names of variables, dummy variables (arguments), or subroutine or subprogram entry points are entered into the DEV table, while the relocatable address assigned to each is entered into the associated DEA table.

First processed are variable names appearing both in COMMON and EQUIVALENCE statements. A variable name is selected from EQUIT table. It is compared with the names appearing in the COMMON table. If it appears in both, the entire sentence in the EQUIT table in which this variable appears is assigned to upper memory.

An equivalence sentence, assembled by Section I', contains all variable names, the relative locations of which have been fixed by EQUIVALENCE statements. The sentence contains no redundancies or inconsistancies. The sentence is made up of two-word entries, the BCD variable name, and the relative location (subscript) to each other. The end of each sentence is marked by a flág (negative sign) in the final subscript.

The equivalence sentence is scanned for the greatest subscript. The current value of the location counter, initially at -207 in the 709, is reduced by the greatest subscript. This is the base from which the location assigned to each of the variable names is computed. The equivalence sentence is scanned again for any variables which are names of arrays. It a variable appears in the SIZ table, the overhang of the array length over the base location (array length - subscript) is computed, and the maximum of these is found. The equivalence sentence is scanned again. Each subscript is added to the base address, in effect creating an array stored backwards in memory, and the variable or array name is entered into DEV with its corresponding location in DEA, flagged minus. The array name with the greatest subscript will be assigned the value of the location counter before it was reduced, in effect locating the most precedent array name in the first available memory location. The value of the location counter is then reserving memory for the overlapping array extending farthest into memory, and reserving for the next variable name the next lower cell.

Suppose there are common symbols E, D, X, which are related by EQUIVALENCE (E (5), D(2), X), and the E and D occur in dimension statements giving their total size as E(6) and D(5), X being a non-subscripted variable. The first variable to be defined is the one with the largest element number in the equivalence group, E in this case, and the 1st element of E is given the highest free location, i. e., LCTR. D and X are immediately defined by their equiaalence relationship with E:

$$E(5) = D(2) = X$$

$$D = E-3$$

or $$= LCTR-3$$

and $$X = E-4$$

$$= LCTR-4$$

It must also be determined how much space these variables occupy. Since the array E has 6 elements, thelast of these would be in LCTR-5, and similarly D has 5 elements, the last of which would be in LCTR-7. Clearly then, the first free location is the one following array D, namely LCTR-3, which then becomes the new LCTR for the next set of assignments. This maximum overhang would be (5-2) = 3, the base address LCTR-5 being so reduced to determine the cell LCTR-8.

After all equivalence sentences in common have been assigned, storage is assigned for all other variables appearing in COMMON statements. The COMMON table, assembled by Section I is made up of one-word entries, the BCD name of a variable appearing in a COMMON statement. Each variable name is checked against DEV to determine if it had appeared in an equivalence sentence. If it is not so redundant, it is entered into DEV with the contents of the location counter as the corresponding location in DEA. The SIZ table is checked to determine if this is an array name, and the value of the location counter is reduced by the length of this array; or if not an array, by 1. Every variable is assumed to be an array with a length of at least 1. This, in effect, creates the array stored backwards in memory, reserving for the next variable name the next lower cell.

When all of common has been assigned, the current value of the location counter, the cell next below the last cell in common, is entered into the program card 8R address (common break).

Next to be processed are equivalence sentences not assigned to upper storage. The first symbol of each equivalence sentence is checked against DEV to determine if any symbol in this sentence had appeared in a COMMON statement. If it is not so redundant, the entire sentence is assigned storage locations, identically as described above, again flagged minus in DEA. The array name with the greatest subscript in the first equivalence sentence will be assigned the location stored in the common break, the cell next below the last cell in common. This, and all subsequent storage assignments later will be relocated downwards in memory.

Next to be processed is the SUBDEF table. If this program is a FORTRAN subprogram, defined by a SUBROUTINE or FUNCTION statement, the name of the program and the argument list are assembled into the SUBDEF table by Section I. Each entry is a one-word BCD name of a dummy variable used as an argument. Each argument name is compared with the subprogram name. If it is multiply defined, a diagnostic message results. Entry is made into DEV to prevent assignment of a storage location for this dummy variable if it appears in a DIMENSION statement not in common, or as the symbolic address (word 3) of a CIT. The corresponding address in DEA is the flag 77777. If this dummy name is already in DEV, it has appeared in a EQUIVALENCE statement and a diagnostic message results. If the flag in DEA is not minus, the dummy name appeared in a COMMON statement. This is permissible, and storage is reserved, but the DEA entry is altered to that dummy variable will not appear on the COMMON storage map.

The SIZ table, assembled by Section I', is made up of two word entries, the BCD name of the array, and the length of the array (the product of its dimensions as stated in a DIMENSION statement). Each array name in the SIZ table is checked against the DEV table to determine if it has appeared in a COMMON or EQUIVALENCE sentence or is a dummy variable name of an argument. If it is not so redundant, it is entered into DEV with the current contents

of the location counter as the corresponding location in DEA. The value of the location counter is reduced by the length of this array. This, in effect, creates an array stored backwards in memory, reserving for the next variable name the next lower cell. A dummy variable name of an argument may appear in a DIMENSION statement in order that a proper relative address may be computed for reference to a specific element in an array, but no storage will be allocated to this dummy variable.

The storage for variables appearing in COMMON statements is now mapped. The variable name, right adjusted, is inserted in the second word of a tetrad; the decimal location, right adjusted with leading zeros suppressed, inserted in the third word; and the octal location, right adjusted with leading zeros included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line if any, are written on tape B2 immediately following theinternal end of file, marking the end of the source program listing.

Next to be processed is the transfer vector. If the source program refers to other subprograms through a CALL statement or an arithmetic statement in which a function name appears, or if a library subroutine is called, Section I assembles the BCD name of the entry point to each such subprogram as one-word entries in the CLOSUB table. The transfer vector, made up of N such names, occupies (relocatable) storage locations 0 thru N-1 of the object program. Each subprogram name is entered into DEV with the corresponding lower storage locations entered into DEA. If the name is already in DEV, it has appeared in a COMMON, EQUIVALENCE, or DIMENSION statement, and a diagnostic message results. If the name of the subprogram is a dummy name, to be initialized, it does not appear in CLOSUB, and no conflict exists.

Finally, the names of arithmetic statement functions are processed. If such a statement appears in the source program, Section I assembles the BCD name of the function so defined, and the internal formula number assigned to the subroutine, in a two word FORSUB entry. Each name is entered into the DEV table with location zero (to be entered later) entered into DEA. If the name is already in DEV, it has appeared in a COMMON, EQUIVALENCE, or DIMENSION statement, or has been referred to in a CALL statement or an arithmetic statement including an argument list with the terminal F omitted from the name, or as a dummy variable name, and a diagnostic message results. The improper use of the name with the terminal F omitted and with no argument list will compile, however improperly. In each case, the improper use of the name of the subprogram being compiled as an argument in the argument list, as a name in the transfer vector, or the name of an arithmetic statement function, is checked, and if so used, a diagnostic message results.

The DEV and DEA tables are now complete. All other variable names in the source program are nonsubscripted, requiring one storage location each.

Record 36

Record 36 includes the first pass over the complete CIT file, to define all internal formula numbers, source program symbols not in DEV, and internal symbols.

The DEA table is moved up in memory and packed against the end of DEV. The IFN table will share memory with the DEA table, the former occupying the decrement portion of each word, while the latter occupies the address. The TEV table will follow the longer of the two.

CIT records are brought into memory from tape A4, and are replaced with the next subsequent record when completely scanned. Each CIT is scanned first for its op code. I. it is OCT or BCD the address portion is ignored.

For other codes the symbolic address is scanned next. If the address is an internal formula number, the address is ignored. A SYN to an IFN is undefined. If the address is a subsidiary internal formula number (nAm), the symbol is assembled into TIV form and TIV is searched to define a possible SYN to this symbol. If it is not in TIV, it is entered, undefined. If the address is *, the contents of the program counter are used to define a possible SYN to this symbol.

If the address is

2)	Fixed point constant
3)	Floating point constant
5)	Assign constant
6)	Universal constant
8) N	Format specification word
9)	Initialization addend constant
B)	Hollerith subroutine argument

It is in the symbolic listing, and the address is ignored. A SYN to one of these symbols is undefined.

If the address is

1) N	Arithmetic eraseable
4) N	Arithmetic statement function argument storage
7) N	Arithmetic statement function index register eraseable
C) N	Index register eraseable.

It is not in the symbolic listing, and is entered into TIV with greatest level of storage (decrement of word 4 CIT) as the address. A SYN to one of these symbols is undefined.

If the address is

A) N	Location symbol for subroutine to compute relative constants
D) N	Location symbol for a section 5 LXD instruction
E) N	Location symbol for a section 5 SXD instruction

It is in the symbolic listing, but TIV is searched to define a possible SYN to one of these symbols.

If the address is $ or $$, the location assigned to each of these is used to define a possible SYN. If the address is an external variable, DEV and TEV are searched to define a possible SYN to one of these symbols. If this variable name is not in DEV or TEV, it is entered into TEV, the location to be defined later.

The op code again is scanned for SYN. The symbol D)N or E)N in the symbolic location can be synonymous with another symbol D)N or E)N, compiled by Section V. If the SYN is undefined, a diagnostic message results. For all op-coes other than BSS or SYN, the location counter is bumped by 1. If it is BSS, the length of block reserved as assumed to be zero. If it is SYN, no location is reserved.

Next to be scanned is the symbolic location. If the location is an internal formula number, the contents of the program counter are entered into the IFN table (decrement portion of the joint IFN-DEA table), ordered as to internal formula numbers. The test for an internal formula number is such that it may not extend over more than 12 bits in the decrement field, a maximum of 4095. If any internal number is greater, it will appear to be an internal symbol, and will miscompile. No diagnostic message results. If the location symbol is a subsidiary internal formula number (nAm), TIV is searched to determine if there had been a prior reference to the symbol. If such a reference had been made, the contents of the program counter are entered into TIV to define this symbol. If no prior reference had been made, the symbol remains undefined. This is to optimize entries into the TIV table. If the reference to the subsidiary internal formula number is prior to the appearance of the number in the location field, it will have been entered into TIV, and defined in pass 1. If such reference is subsequent to such appearance, the TIV entry will be made, but the symbol will remain undefined until pass 2. During pass 2, this symbol will be defined prior to such subsequent reference. Hence, any subsidiary internal formula number to which a reference is made will eventually appear defined in TIV, while no reference is made will not be entered

into TIV. If the location symbol is *, it is a flag set by section III that this CIT results froma TIFGOentry for TRASTO transfer address, for consideration by section IV, and it is ignored. For all other internal symbols appearing in the symbolic listing, a TIV entry is made, the contents of the program counter defining this symbol. If the location symbol is $ or $$, each of these is defined by the contents of the program counter. If the location symbol is an external symbol (transfer vector name), it is ignored.

At the end of the first pass over the complete CIT tape, all symbols appearing in compiled instructions have been entered into one of the tables, DEV, IFN, TEV or TIV. The upper location counter is one cell below the lowest cell reserved for a DEV entry. The location counter is reduced by the length of the TEV table, and each variable in TEV is implicitly defined as the current contents of the location counter plus its ordered location in the TEV table. Later, these locations will be relocated downwards in memory.

Assignment of storage locations for eraseable cells in TIV is made next. Each TIV entry is examined to determine if it is an eraseable cell (1)N, 4)N, 7)N, C)N.) If it is, the location counter is reduced by the largest value of the block required, the address portion of the TIV entry, and this location defines the symbol. This, in effect, creates an array stored forwards in memory. The location counter is reduced by one more to reserve the next lower cell for the next symbol. The symbol 4), eraseable for library subroutines, is defined as the location at the top of memory, 77777.

The storage assignments at this point are as in the following diagram.

Location Symbol		Table Entries	
NAME	TRANSFER VECTOR	DEV	relocatable zero
$	PROLOG		subprograms only
	INITIALIZATION		
$$ nA nAm D)N E)N	OBJECT PROGRAM	IFN TIV TIV TIV	
nA	ARITHMETIC SUBROUTINES	(Name in DEV) IFN	
A)N	RELCON SUBROUTINES	TIV	
5) 2) 3) 6) 8)N B)	PROGRAM CONSTANTS	TIV	end of symbolic listing.
	NOT ASSIGNED		program counter contents of location counter
7)N 4)N 1)N C)N	ERASEABLE STORAGE	TIV	
NAME	NON SUBSCRIPTED VARIABLES	TEV	
NAME	DIMENSION VARIABLES	DEV	
NAME	DIMENSION EQUIV VARIABLES	DEV	common break
NAME	COMMON DIMENSION VARIABLES	DEV	
NAME	COMMON DIM. EQUIV. VARIABLES	DEV	-207 (709)
4)	LIBRARY SUBROUTINE ERASEABLE	TIV	top of memory

Note: argument dummy names (in subprograms) are entered into DEV, flagged 77777 in DEA

Record 37

Record 37 assigns locations for arithmetic statement function subroutines and maps them. The DEV table is scanned for the name of each subroutine (word 1 of each FORSUB table entry). If it is not found, a machine error has occurred, and a diagnostic message results. The location of the internal formula number assigned to this subroutine name (decrement of word 2 of FORSUB table entry) is found in the IFN table, and inserted in the address of word 2 of the FORSUB table entry, and to define this symbolic location, in the DEA table.

The location of each subroutine is now mapped. The subroutine name, right adjusted, is inserted in the second word of a tetrad; the decimal internal formula number, right adjusted, inserted in the third word; and the octal location of this internal formula number, right adjusted with leading zeros included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line if any are written on tape B2, following the mapping of common storage assignment, if any.

Record 38

Record 38 maps external formula numbers with corresponding internal formula numbers and relative locations.

Each decimal external formula number (address portion of one-word entry) in EIFN table, right adjusted, is inserted in the second word of a tetrad; the decimal internal formula number (decrement portion of entry), ritht adjusted is inserted in the third word; and the octal location of this internal formula number, found in the IFN table, right adjusted with leading zeros included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line if any are written on tape B2, following the mapping of arithmetic statement function subroutines, if any.

Record 39

Record 39 relocates storage not in common downwards packed against program constants.

The length of unassigned memory is computed (contents of location counter less contents of program counter, plus one), and is the extent of relocation. The position of the program break is computed (location of common break less contents of the location counter, number of variables to be relocated, added to contents of program counter), and inserted in the program card 8L address.

Each location in DEA is compared against the common break (highest cell in storage to be relocated). If it is not in common, a transfer vector name, a

subprogram argument dummy variable (flagged 77777), or an arithmetic subroutine, the location is reduced by the extent of relocation. The base location for TEV is so relocated; in effect, relocating each variable in TEV. Each location in TIV is compared against the common break and the program break. If it is not 4) (location 77777), program data in the symbolic listing, or an instruction location symbol, it is an eraseable cell and is so relocated.

The final storage assignments are as in the following diagram.

	Storage	
	TRANSFER VECTOR	relocatable zero
entry point (subprogram)	PROLOG	} subprograms only
entry point (main program)	INITIALIZATION	
	OBJECT PROGRAM	
	ARITHMETIC SUBROUTINES	
	RELCON SUBROUTINES	
	PROGRAM CONSTANTS	
	ERASEABLE STORAGE	
	NON SUBSCRIPTED VAR.	
	DIMENSION VARIABLES	
	DIMENSION EQUIV. VAR.	
		program break
	NOT ASSIGNED	
		common break
	COMMON DIMENSION VAR.	
	COM. DIMEN. EQUIV. VAR.	-207 (709)
	LIBRARY SUB. ERASEABLE	

The limits of storage not used by program (program break and common break), converted to decimal, right adjusted with leading zeros suppressed, are inserted in the third word of a tetrad; converted to octal, right adjusted with leading zeros included, inserted in the fourth word. The first and second word of this tetrad are blank. The title, column headings, and this line are written on tape B2, following the mapping of external-internal formula numbers, if any.

Next are mapped the transfer vector, program variables not in common, and internal symbols. The number of entries in the transfer vecotr is one location greater than that of the last name in the transfer vector. Each location in DEA is compared against the location of the first instruction following the transfer vector, and if in the transfer vector, the corresponding transfer vecotr name in DEV, right adjusted, is inserted in the second word of a tetrad; the decimal location, right adjusted with leading zeros suppressed, inserted in the third word; and the octal location, right adjusted with leading zeros included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partical line if any, are written on tape B2 following the mapping of the storage limits.

If any arithmetic subroutines exist, the location following them is the first location in which a variable may appear. If not, the location following the transfer vector is this location. Each location in DEA is compared against the first location following either the transfer vector or arithmetic subroutines, and against the common break. If it has not been listed previously as a transfer vector or arithmetic subroutine name, as a variable in common, or is not a subprogram argument dummy variable name, it is a subscripted variable not in common, and the corresponding name in DEV, right adjusted, is inserted in the second word of a tetrad; the decimal location, right adjusted with leading zeros suppressed, inserted in the third word; and the octal location right adjusted with leading zeros included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line, if any, are written on tape B2 following the mapping of the transfer vector, if any.

Each entry in TEV (nonsubscripted variable not in common), right adjusted, is inserted in the second word of a tetrad; the decimal location, the sum of base location for TEV and the relative location of this variable in TEV, right adjusted with leading zeros suppressed, inserted in the third word; and the octal location, right adjusted with leading zeros included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line, if any, are written on tape B2 following the mapping of the subscripted variables not in common , if any.

Each entry in TIV is then mapped. A TIV entry consists of a symbol in bits, S, 1, 2, 3; bits 4 and 5 zero; sub symbol, if any, in bits 6-20; and the location in bits 21-35. A subsidiary internal formula number consists of bits S, 1, 2, 3, 20 zero, the intemal formula number in bits 4-14 (maximum size 2047); the subsidiary number in bits 15-19; and the location in bits 21-35.

If the TIV entry is a sub internal formula number, it is ignored. If it is an internal symbol for a storage cell, an alpha numeric character from the set 1 through 9, A through E is assigned to the 4-bit pseudo symbol, followed by a right parenthesis. The 15 bit subsymbol, if any, is converted five bits at a timeto 3 alpha numeric characters from the set 1 through 9, A through W. The pseudo symbol, left adjusted, is inserted in the second word of a tetrad; the decimal location, right adjusted with leading zeros suppressed, inserted in the third word; and octal location, right adjusted with leading zeros included, inserted in the fourth word. The first word of every tetrad is blank. The title, column headings, each line as completed, and the final partial line, if any, are written on tape B2 following the mapping of nonsubscripted variables not in common, if any.

Next the program card is written as the first binary output on tape B3. Program card 9L includes a 4 punch in the prefix and a word count of 4 in the decrement. 8L contains the length of transfer vector in the decrement and program break in the address. 8R contains the common break in the address. 7L contains the BCD subprogram name, if any. 7R contains the entry point, relative to zero in the

address. The computed checksum of the card is inserted in 9R.

Column binary bits, 7-9 punch in column 1, not included in the checksum, are inserted in 9L, and the program card is written as the first record on binary output tape B3.

Record 40

Record 40 includes the second pass over the CIT tape to define each of the symbols used in each CIT, construct a binary instruction for each CIT, and write the compiled program on binary output tape # B3.

CIT records are brought into memory from tape A4, and are replaced with the next subsequent record when completely scanned.

Relocation bit patterns are of three types. Type 00 indicates that address portion of the instruction is not relocatable. Type 010 indicates that the address portion is relocatable as data on the proper side of the program break. Type 011 indicates that the instruction is complement relocatable; the address refers to a cell in an array the base symbol of which is on the opposite side of the program break, and should be relocated as its base symbol would be. The decrement of an instruction is not relocatable in a FORTRAN object program.

The relocation bits are initially reset to not relocatable. First to be scanned is the opcode. If it is OCT or BCD, the address portion is not relocatable. For all other opcodes, the symbolic address is scanned next. If the symbolic address is zero, it is not relocatable. If the symbolic address is an internal formula number, the location is obtained from the TIV table. If the symbolic address is *, the location is the current contents of the program counter. If the symbolic address is $ or $$, the location is as assigned to either of these. If the symbolic address is an external symbol, the location is obtained from TEV or DEA. If any symbol has as yet not been defined, a diagnostic message results. For each of these, the address is tentatively set directly relocatable.

The opcode is again scanned. If it is SYN, the definition is saved to be checked. No binary output results. If it is BSS, the length of the block reserved is assumed to be zero. No binary output results. For all opcodes other than BCD, OCT, BSS or SYN, the binary machine code is found in the SOPR table. If the opcode is not found in the table, a diagnostic message results.

The relative address is added to the location for the symbolic address to determine the absolute address for the symbol. If negative, it is complemented. The base symbol (symbolic address) is examined to determine if both the base symbol and the absolute address are on the same side of the program break. If they are not, the address is set complement relocatable. The binary decrement, absolute tag, and absolute address are combined with the operation code. For BCD or OCT, the binary word (symbolic address) is used. The program counter is bumped one location.

The relocation bits are packed left adjusted against any prior relocation bits already in the 8 row of the card image. The binary instruction is inserted in the next available half row of the card image. When the card image is full, the word count is inserted in 9L decrement, the load address is inserted in 9L address, the checksum is computed and inserted in 9R, column binary bits added to 9L, the card is written on tape B3, and the load address is updated to the program counter for the next instruction. Column binary bits are not included in the checksum 9R, and tape B3 is unusable for off line output.

For all CIT's the symbolic location is scanned. If it is a subsidiary internal formula number and is not in TIV, it has been omitted as no reference to it was made in the symbolic address, and it is ignored. If it is in TIV and is not yet defined, the reference to it was later in the CIT file than its appearance in the location field. It is here defined, the location assigned to this symbol is checked against the program counter. If it is inconsistent, a diagnostic message results. If the symbolic location is an internal formula number, it is checked for inconsistent definition. If the symbolic location is an internal symbol, it is the symbol for program data appearing in the symbolic listing, or the symbol assigned to a Section V LXD or SXD instruction or a relcon subroutine. If the symbol appears in TIV it is checked for inconsistent definition. If it does not appear in TIV, it is a machine error, but no diagnostic message will result. If it is $, $$, *, or an external symbol in DEV (transfer vector name) it is ignored. No other external symbol in DEV or any in TEV should appear in a location field.

After the entrie CIT file has been scanned, the final partial card image, if any, is written on tape B3. Processing is now complete, except for the transfer card, and for options.

Record 41

Record 41 processes the options which the programmer has instructed the FORTRAN compiler to provide. Available are the following options:

Sense Switch 1	UP	Binary cards for the object program are punched on line. Tape B4 also contains the stacked binary program if row binary has been requested on line.
	DOWN	Binary cards for the output program are not punched. Tape B4 contains the stacked binary program for the source program compiled.
	Note:	Binary card images on tape B3 omit the column binary bits from the checksum and are unusable.
Sense Switch 2	UP	Produces, on tape B2, two files for source program compiled, consisting of the source program and a map of object program storage.

	DOWN	Adds a third file for each program compiled (see above) containing the object program in symbolic machine language on tape.
	Note:	This listing will be stacked on tape A3 as one file for the entire monitor run.
Sense Switch 3	UP	No on line listings are produced.
	DOWN	Lists on line the first two or three files of tape B2, depending on the setting of Sense Switch 2.
Sense Switch 4	UP	Relocatable row binary cards for the object program are punched on line, if Sense Switch 1 is up.
	DOWN	Relocatable columnar binary cards for the object program are punched on line if Sense Switch 1 is up. These will not be stacked on tape B4. Also these appear on tape B3, see note under Switch 1.
Sense Switch 5	UP	Library routines are not punched out on line or written on tape B4.
	DOWN	Causes library routines to be punched on line and written on tape B4, depending on whether Sense Switch 1 and 4 are in the Up or Down position. See note under Switch 4 down, and Switch 1.
Sense Switch 6		Not applicable to Section 6.

The transfer vector, which has been stored as one record following the CIT file on tape A4, is brought back into memory. End card setting and/or physical sense switch 5 is tested to determine if a library search is required. If the transfer vector is not empty and a library search is requested, a flag for subroutines found in each pass over the library file on the FORTRAN system tape is reset.

The next record in the library file is read into memory. If 9L prefix has a 4 punch, it is a program card; if not the next record is read in.

After a program card has been found, the next record is brought into memory with rows 8 through 12 packed against the earlier card image. Row 9L prefix is again tested to determine if the program card continues over more than one card. When a card other than a program is encountered, the tape is backspaced over the card image, and a consolidated program card exists in memory. The word count of the consolidated program card is inserted in the decrement of 8L.

Each right row (entry point relative to zero corresponding to entry point name) is scanned to determine if it is flagged by a sign bit punch as a secondary entry point. I it is not so flagged, the left row (name of primary entry point) is compared against the transfer vector to determine if this subroutine is required to complete the object program. If no such name is found, the remainder of the subroutine in the library file is passed over to find the first program card of the following subroutine.

If a primary entry point to a subroutine is found in the transfer vector, the name is transferred from the transfer vector to a list of entry points to subroutines output from the library. A flag is set that at least one subroutine has been found on this pass over the library file.

The names of all entry points to subroutines output are added to the found list, and if any of these are in the transfer vector, they are deleted from the transfer vector.

The consolidated program card is converted back into individual card images, and writte on tape B3 following the object program, or the last library subroutine output, and the next record read fromthe library file. If the library subroutine includes a transfer vector, each name in the subroutine transfer vector is compared against the found list and the object program transfer vector. If it is in neither, it is a new requirement and it is added to the object program transfer vector look for list. The card image is written on tape B3 followingthe library program card, or last subroutine card. If the subroutine transfer vector is not exhausted, the next card image is read in and processing continues as above. If a program card or end of file is sensed before the subroutine transfer vector is exhausted, a diagnostic message results.

After the subroutine transfer vector is exhausted, the remaining cards in the library subroutine are copies from the library file to tape B3, until the next program card is encountered. If the object program transfer vector is exhausted, the search is completed. If not, the search continues until the end of the library file is sensed.

After the end of the library file, the flag for subroutines found is examined. If any subroutines have been found on this pass, the subroutine transfer vector may require another pass over the library file, as these new subroutines may have added names to the look for list. If the look for list is not exhausted, and subroutines have been added this pass, the system tape is backspaced to the beginning of the library file, the subroutines found flag reset, and another pass over the library file is made.

After the library search is completed, the system tape is repositioned at the end of this record, and if any names of entry points to library subroutines are on the found list, these names are written on the storage map. Each BCD name is right adjusted and inserted in the second word of a pair. The first word is blank. The title, each line as completed, and the final partial line, if any, are written on tape B2 following the mapping of internal symbols.

After this mapping, or if the library search was not required, the transfer vector is examined to determine if any subprograms exist which are not library subroutines. Each BCD name remaining in the transfer vector is right adjusted and inserted in the second word of a pair. The first word is blank.

The title, each line as completed, and the final partial line, if any, are written on tape B2 following the mapping of names of entry points to library subroutines found, if any.

The storage map is now complete and marked with an end of file.

If the object program is not a subprogram, a transfer card is written on tape B3. The end of binary output is marked with an end of file, and the tape is rewound.

End card setting and/or physical sense switch 1 is tested to determine if cards are required on line. If cards are required on line, end card setting and/or physical sense switch 4 is tested to determine if cards should be row binary or column binary.

If switch 4 is up, cards are to be row binary, and if the object program is not a subprogram, the BSS loader is punched on line. The column binary bits are deleted from 9L of each card image and the card punched on line.

If switch 4 is down, cards are to be column binary, the column binary bits are added into the checksum, 9R, or each card image, the row binary rotated to a column binary image, and the card punched on line.

If no column binary cards have been punched on line, sense light 1 is turned on to so flag the monitor and the binary output will be stacked on tape B4. The binary output on B3 does not include column binary bits in the checksum, and is unuseable.

End card setting and/or physical sense switch 2 is tested to determine if a machine language listing is required. If it is so requested , sense light 2 is turned on to flag monitor that a third file exists on the BCD output tape. An additional pass is made over the CIT tape toaccomplish this.

CIT records are brought into memory from tape A4, and are replaced with the next subsequent record when completely scanned. First the symbolic location is processed, If the symbolic location is an internal formula number or a subsidiary internal formula number, the main number is converted to decimal, the character A appended, and the subsidiary number converted to decimal. The largest internal formula number which can be stored in TIV is 2047, and a subsidiary number can be one character only, Hence this symbol cannot exceed six characters. If the symbolic location is *, it is deleted. If the symbolic location is an internal symbol, a pseudo symbol is constructed which cannot exceed five characters. If the symbol is $, $$ or a transfer vector name, these characters are used. The BCD symbol, so constructed, right adjusted, is inserted in the third word of a hexad.

The BCD opcode, preceded and followed by a blank, is inserted in the first five characters of the fourth word.

If the opcode is BCD, and the symbolic address is a 777777777777 flag, the code is replaced by OCT, and processing continues as an octal symbolic address.

If the symbolic address is not a flag, the numeral 1 is inserted in the sixth character of the fourth word, and the six character BCD word in the fifth. The sixth word is blank.

If the opcode is OCT, the first bit is interpreted as a sign, inserted in the sixth character of the fourth word, and the 35 bit binary number, converted to 12 BCD octal digits, is inserted in fifth and sixth words.

For all opcodes other than BCD or OCT, the symbolic address is processed as follows. If the symbolic address is an internal formula number or a subsidiary internal formula number, the main number is converted to decimal, the caracter A appended, and the subsidiary number, if any, converted to decimal. The first BCD character of the internal formula number is inserted in the sixth character of the fourth word. The remaining BCD characters (five or fewer) of the symbol, followed by blanks, are saved. If the symbol address is an internal symbol, a pseudo symbol is constructed. The first BCD character of the pseudo symbol is inserted in the sixth character of the fourth word. The remaining BCD characters (four or fewer) of the pseudo symbol, followed by blanks are saved. If the symbolic address is an *, $, $$, or any external symbol, the first BCD character of the symbol is inserted in the sixth character of the fourth word. The remaining BCD characters (five or fewer) of the symbol followed by blanks are saved.

The remaining characters in the symbol (five or fewer) followed by blanks are examined one at a time for the first blank character. The non-blank characters are packed left adjusted into the fifth character of the fifth word. The relative address is isolated from the decrement of the CIT word 4. If it exists, it is converted to five or fewer BCD decimal digits. The BCD sign is inserted packed against the symbol, no farther than the sixth character of the fifth word. The BCD relative address is packed against the sign, extending no farther than the fifth character of the sixth word. The tag is isolated from the address of the CIT word 4. If the tag is greater than four, the flag T is inserted in following the tag. No diagnostic message results. A comma is inserted packed against the symbol or relative address, no farther thant he sixth character of the sixth word, followed by the tag, no farther than the first character of the seventh word. The CIT decrement is isolated from the address portion of CIT word 1. If it exists, it is converted to 5 or fewer BCD decimal digits. A comma is inserted packed against the symbol or relative address, no farther than the second character of the seventh word, followed by the decrement, packed against the comma, no farther than the second character of the eigth word.

If following the symbolic address (and, if it exists, the relative address) no tag exists, the CIT decrement is isolated from the address portion of CIT word 1. If it exists, a zero is selected as the tag field, and processing continues as before.

If no symbolic address exists, the CIT four word is tested for a relative address and/or tag. If either or both exists, the relative address is isolated. If it exists, it is converted to a 5 or fewer BCD decimal digits. If it is negative, the sign is inserted in the sixth character of the fourth word. If positive, the first BCD numeral is inserted in the sixth character of the fourth word. The remaining characters are inserted left adjusted in the fifth word, and the tag and decrement are processed as before. If no symbolic address or relative address exist, a zero is inserted in the sixth character of the fourth word, and the tag and decrement are porcessed as before.

If no symbolic address, relative address or tag exist, the CIT decrement is isolated from the address portion of CIT word 1. If it exists, a zero is inserted in the sixth character of the fourth word, and the tag and decrement processed as before. Processing of the full tag is necessary to insert the nonredundant comma and zero tag field.

After the variable field has been processed, the final word is filled with blanks. If no variable field exists, a blank is inserted in the sixth character of the fourth word.

All processing converges at this point. If the opcode is not SYN or BSS, the relative counter is converted to 5 BCD octal digits, left adjusted, followed by a blank, and inserted in the second word of the hexad. The relative counter is bumped by one. If the opcode is BSS, the block length is assumed to be zero, hence for either BSS or SYN word two is blank, and the relative counter is unchanged. Word one of every hexad is blank. The CIT is now in the standard form.

SYMBOL OPC ADDRESS+RA, TAG, DECREMENT

The six words of every hexad are transferred to a page image buffer. In this process, overflow of the machine language image to the seventh and first two characters of the eighth word are truncated. As the FORTRAN processor compiles TIX, TXI and TXL only in a DO loop, the only machine language instructions which may contain decrement fields are TXI*+1, 4, 32767 TIX*+1, 4, 32767 and TXL 4095A, 4, 32767. None of these will overflow.

A count is kept of the hexad entries made in the page image buffer. The first 58 entries are made in column one, the next 58 entries in column 2, and the next 58 entries in column 3. When 174 entries have been made, the page image is followed by a page restore.

Whenthe end of the CIT file is sensed, the buffer is checked for a partial page image. If a partial image exists, it is written on tape B2. An end of file mark is written following the machine language listing, and tapes B2 and A4 are rewound. The information on tape A4 is no longer of significance.

End card setting and/or physical sense switch 3 is tested to determine if on-line output of the source program, storage map, and machine language listing (if any) is required. If it is so requested, the page is restored so that each file begins on a new page.

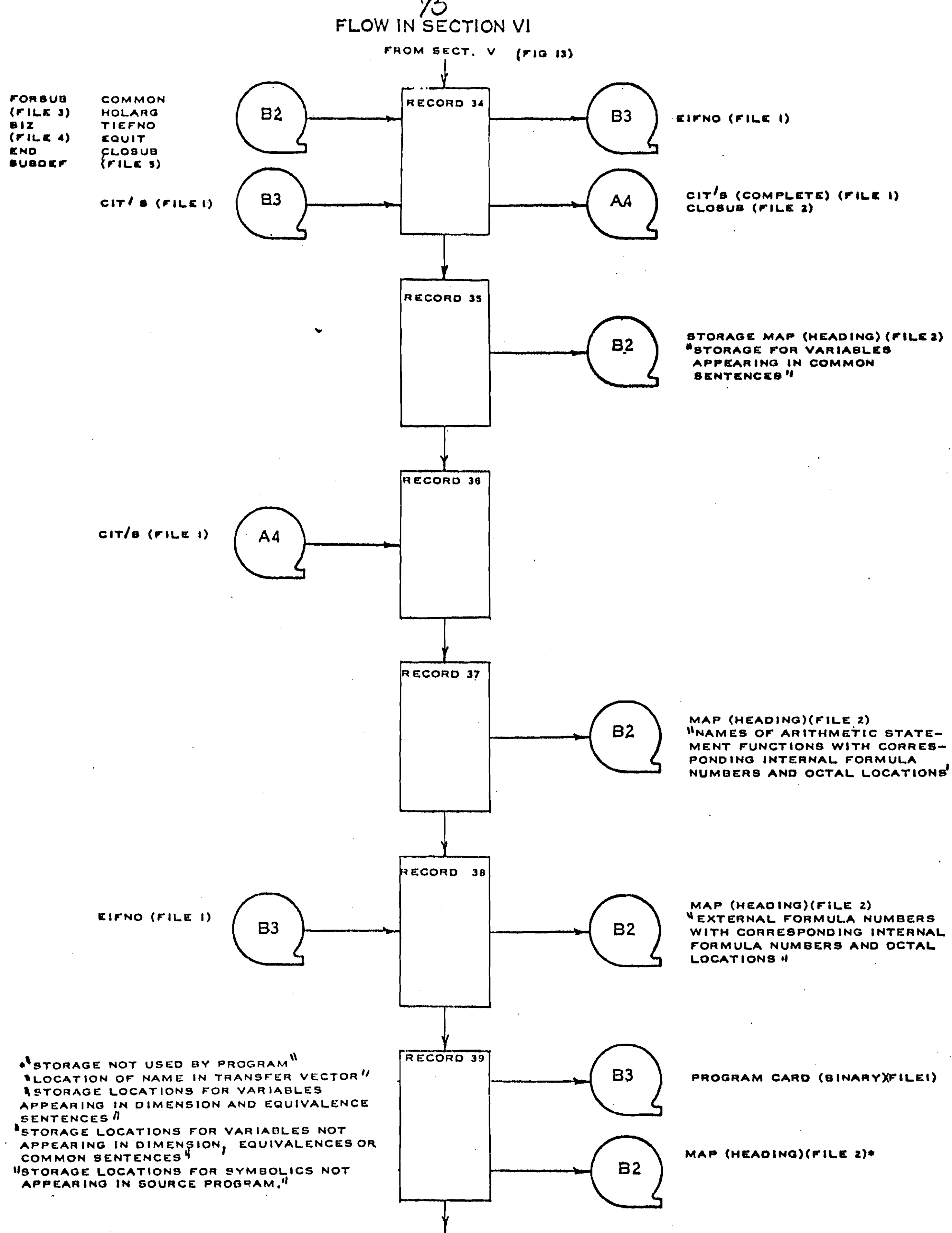
One record (one printed line) is read from tape B2, is converted to a card image, and the line is printed.

When the end of the source program file is sensed, the page is restored and the map is printed line by line

When the end of the storage map file is sensed, sense light 2 is tested to determine if a third file, the machine language listing, exists on tape B2. If it does, this flag is restored for monitor, the page is restored and the listing is printed line by line.

When the end of the listing file is sensed, tape B2 is rewound. The FORTRAN compiler has completed processing of the source program. The results of the FORTRAN compilation are on two tapes: tape B2, the BCD source program, storage map, and symbolic listing if requested; and tape B3, the binary program card, the object program, library subroutines including their program cards if requested, and transfer card if a main program. If on line output has been requested, and transfer card if a main program. If on line output has been requrested, cards have been punched and listings have been printed.

Control is passed to the monitor.



NOTE • HEADINGS WRITTEN ON TAPE B2 (FILE 2) ONLY AS PERTINANT. ONLY HEADINGS MARKED + WILL ALWAYS APPEAR.

FIGURE 14.

FLOW IN SECTION VI CON'T

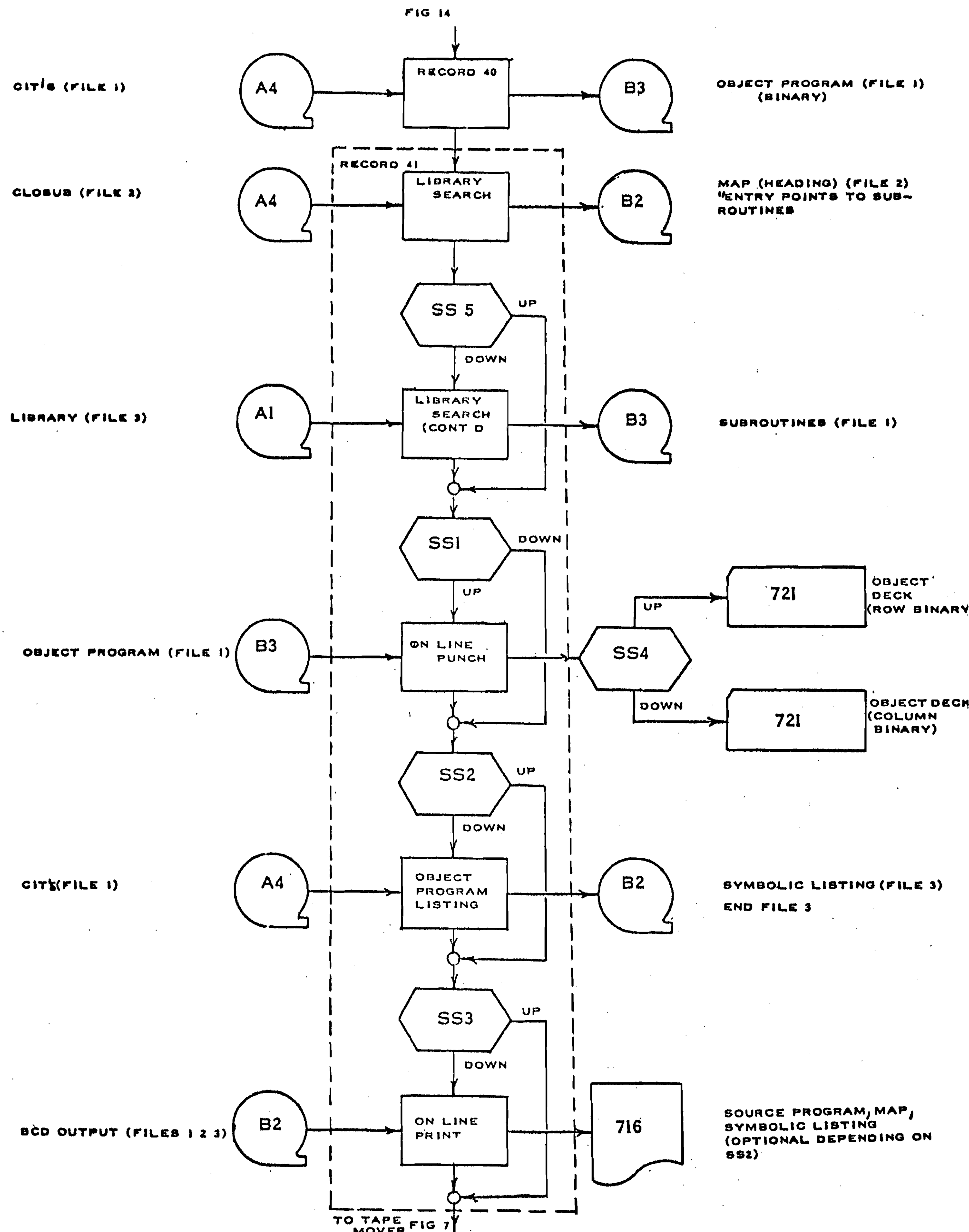


FIGURE 15

The library is contained in the third file on the System Tape. The library contains the input, output, math, monitor, various control routines and the librarian, required for the execution of any object program compiled by FORTRAN.

The math and control routines will not be discussed in detail. The logic and coding of these routines is straight forward, and reference to the specific listings should be made if any question should arise.

The input/output routines are not as straight forward as the aforementioned routines, therefore, a more complete write-up will be given in 2.02.01.

The library contains the following routines:

INPUT-OUTPUT LIBRARY

Control Routines

IOS/ Input-Output Supervisor

IOU/ Input-Output Channel-Unit Table

SLO/ Short-List Output

SLI/ Short-List Input

WER/ Tape Write Error

RER/ Tape Read Error

Hollerith Input-Output

IOH/ Input-Output Hollerith

STH/ Storage to Tape Hollerith

TSH/ Tape to Storage Hollerith

CSH/ Card to Storage Hollerith

SCH/ Storage to Card Hollerith

SPH/ Storage to Printer Hollerith

Binary Input-Output

- IOB/ Input-Output Binary
- STB/ Storage to Tape Binary
- TSB/ Tape to Storage Binary
- DRM/ Write Drum and Read Drum

Tape Non-Transmission

- BST/ Backspace Tape
- EFT/ Endfile Tape
- RWT/ Rewind Tape

MATH LIBRARY

- XP1/ Exponential - FXPT Base - FXPT Exp.
- XP2/ Exponential - FLPT Base - FXPT Exp.
- XP3/ Exponential - FLPT Base - FLPT Exp.
- ATN/ Floating Point Arctangent
- XPF/ Floating Point Expenential Function
- LOG/ Floating Point Natural Logarithm
- SCN/ Floating Point Sine and Cosine
- SQR/ Floating Point Square Root
- TNH/ Floating Point Hyperbolic Tangent

MONITOR LIBRARY

- CHN/ Chain
- DMP/ Dump
- XIT/ Exit

OTHER LIBRARY ROUTINES

FPT/ Floating Point Trap

TES/ Test Last Write

XLO/ Relocated Location Function

EXE/ Execution Error

THE LIBRARY EDITOR

LIB/ Librarian

The 709 Fortran I/O Library was designed as a simple, generalized and flexible method for handling the input-output and conversion of data required by Fortran-compiled programs at object time under Monitor or non-monitor operation. The I/O Library (IOL) consists of hand coded, Fap-assembled, relocatable subroutines, which communicate with Fortran programs by means of linkage compiled by the I/O Translator (IOT) in Section I.

Most of the analysis done by the IOT concerns the items in the List. When an item in the List specifies an array (i. e., used in a Dimension statement) but is not subscripted in the List, the Short List subroutines (SLI, SLO) are used to communicate between the array and the Mode subroutine. If, however, the item in the List is subscripted, indexing instructions will be necessary. IOT will make entries in the TDO table, which cause Section II to compile the necessary instructions for the treatment of arrays conforming to standard Fortran usage, i. e.; the first element is assigned the highest location of the array. The remainder of IOT's task is simple: the communication of the minimum amount of information necessary to the IOL. This could be: the unit designation, type designation, location of Format specification, and the termination of the List.

The simplicity of this scheme will become apparent during the following description. Its flexibility and generality provide advantages of easy modification, and a continuing opportunity for improvement. This partly expalins the reason for the fragmentation of the IOL into about twenty different routines. Generally, in systems design, the linkage cost of keeping functions separate and distinct, is repaid both in memory space and in the ease with which additions and improvements may be made.

The IOL contains four types of routines:

1) for initialization and control: IOS, IOU, SLO, SLI, WER, RER;

2) for the transmission of information to and from each TYPE of I/O unit: STH, TSH, CSH, SCH, SPH, STB, TSB, DRM;

3) for the conversion of data, and/or its transmission to and from the data area, according to MODE: IOH, IOB;

4) and for non-transmission TYPE tape handling; BST, EFT, RWT.

In the following write-up, the mode routines (IOH, IOB) will be described in conjunction with the unit routines.

The general overall flow can be outlined as follows:

1) The logical unit designation, if necessary, is picked up, and control exits from the calling sequence to the indicated TYPE routine.

2) If this is a non-transmission TYPE routine, control passes directly to the control routine, IOS, for initialization. If a transmission type, except DRM, and TYPE routine furnishes the correct switch setting for input or output to the

appropriate MODE routine. Then the MODE routine conveys the logical unit designation, along with the correct mode indication, to IOS.

3) IOS turns to the IOU table for the logical-actual unit correspondence, after having checked for the correct completion of a previous write statement. When all I/O commands have been initialized, control returns to the MODE routine (or to the non-transmission caller).

4) The MODE routine now controls transmission, and/or conversion, of data according to the Format specification and the List of items indicated by the calling sequence. A return is made to the TYPE unit routine for each record of input or output.

5) When the List is satisfied a final return is made to the MODE routine to make sure the last record is read or written, and to restore conditions.

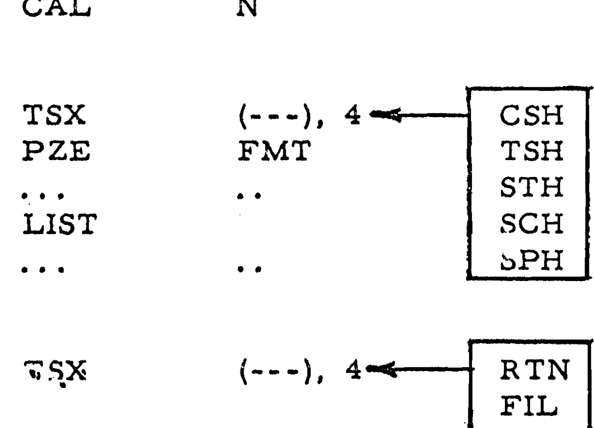
TABLE OF USAGE

TYPE UNIT	MODE	SHORT-LIST CONTROL	TAPE ERROR CONTROL	CHANNEL-UNIT CONTROL
TSH	IOH	SLI	RER	IOS
STH	IOH	SLO	WER	IOS
CSH	IOH	SLI	-	IOS
SCH	IOH	SLO	-	
SPH	IOH	SLO	-	IOS
STB	IOB	SLO	WER	IOS
TSB	IOB	SLI	RER	IOS
BST	-	-	-	IOS
EFT	-	-	-	IOS
RWT	-	-	-	IOS
DRM	-	-	-	-

HOLLERITH INPUT/OUTPUT

READ FMT, LIST
READ INPUT TAPE N, FMT, LIST
WRITE OUTPUT TAPE N, FMT, LIST
PUNCH FMT, LIST
PRINT FMT, LIST

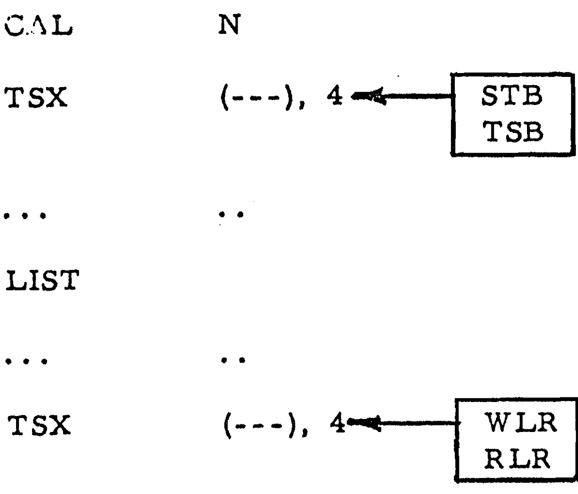
709 CALLING SEQUENCE



BINARY TAPE INPUT/OUTPUT

WRITE TAPE N, LIST
READ TAPE N, LIST

709 CALLING SEQUENCE



BINARY DRUM INPUT/OUTPUT

WRITE DRUM N, J, LIST
READ DRUM N, J, LIST

709
CALLING SEQUENCE

```
CAL     N
TSX     (---), 4 <---- [SDR / DRS]

CAL     J
LDA     ..
...     ..
LIST
...     ..
```

TAPE NON-TRANSMISSION

BACKSPACE	N
ENDFILE	N
REWIND	N

709
CALLING SEQUENCE

```
CAL     N
TSX     (---), 4 <---- [BST / EFT / RWT]
```

SUBSCRIPTED ARRAY LISTS

709

INPUT

```
...
STR
STQ ARRAY + 1, TAG
...
```

OUTPUT

```
...
LDQ ARRAY + 1, TAG
STR
...
```

INDEXING

```
TXI
TXL
```

NON-SUBSCRIPTED ARRAY LISTS

709

INPUT

```
TSX (SLI), 4
PZE ARRAY + 1
PZE SIZE
```

OUTPUT

```
TSX (SLO), 4
PZE ARRAY + 1
PZE SIZE
```

NOTE

709 DRUM LISTS COMPILE
THE SAME AS 704 BINARY

GENERAL DIAGNOSTIC

The general diagnostic for the FORTRAN system covers machine and source program errors revealed by Section I' through VI. When a machine or source program error is encountered in any one of these executive system records, a TSX DIAG, 4 transfers control to the diagnostic caller. In the 709 FORTRAN system the diagnostic caller remains in lower memory with 1 to CS. The caller then dumps a buffer of 2500 words onto tape A3. The diagnostic caller then spaces the system tape to the 4th file and proceeds to read in the main diagnostic record.

The main diagnostic record of the 4th file (record 1) contains all the information necessary to call one of the subroutines needed for converting and printing error comments, and for returning to the proper record in the FORTRAN Monitor. The main record converts the contents of index register 4 back to the location number of the TSX, and uses this constant as the error number. It is in the main record that the heading (FORTRAN DIAGNOSTICS RESULTS), Section number, record number and the location of the TSX in that Section is printed. Also, upon return from the appropriate error processing subroutine the message END OF DIAGNOSTIC PROGRAM RESULTS----PROGRAM CANNOT BE CONTINUED will print.

The main record performs a table search in order to determine which of the fourth file records contains information pertinent to the error. The error number (2's complement of IR4) is compared to a list of errors. This list has 2-word entries. The first word is an error number corresponding to the location of a TSX in the executive system. The second word is the record number in the fourth file which contains the pertinent comments and coding to print information about the error. If the second word is minus, it will also contain the FORTRAN record number of the TSX. The minus indicates that the error number may be duplicated in the error list and if the FORTRAN record number does not match the one picked up by the diagnostic from 1 to CS, the comparison with the error list continues, until a match is found.

When the match has been found, the diagnostic record number is used to space the system tape to the correct record in the 4th file. If a match is not found in the error list, the main record will then read in D002 which concerns unlisted error calls. The pertinent diagnostic record is then read in over the error list and the main record transfers control to it.

Each of these records is set up to handle information about one error, or one specific type of error, only. Usually, this is done by straight forward coding which makes use of the print subroutines in the main record. The program instructions executed may obtain further information to be inserted into the error comment from tapes, cores, or the core dump. The error comment, which is contained in that particular diagnostic record in BCD, is then printed

After all error comments have been printed, control is always returned to one of two points in the main diagnostic record. This will depend upon whether the error encountered was a machine error or a source program error.

The main diagnostic record spaces the System tape to either the machine error or the source program error record in the FORTRAN Monitor, depending upon the aforementioned error return. The diagnostic then prints the end comment and transfers control to 1-CS to read in the proper Monitor error record.

Operator options, if any, are printed by the Monitor error record on the 709. The options will vary depending upon whether the system is operating in the Monitor mode or single compile mode.

THE DIAGNOSTIC RECORD FOR SECTION I"

A few diagnostic records obtain information from an error list left in upper memory by the system record that has called the diagnostic. The diagnostic record for Section I" is such a record. D003 is unique in that it contains all of the error comments for Section I", rather than just one comment. In the case of D003, the information for a particular error is preceded by a flag. The format of the error list is described in the write-up for Section I'andI".

D003 performs a table search in order to determine which subroutine within itself is to process the error currently being treated in the error list. This table search is done by comparing the flag in the error list with the first word of a two word entry in an error table.

The first word in the error table entry is the location of a TSX to the error routine in Section I". The second word is the location of the subroutine in D003 for processing that type of error. When a match has been found, the table search routine transfers control to the proper subroutine. The subroutine extracts whatever information it may need from the error list and, like other diagnostics, uses the subroutines in the main diagnostic record for producing an error comment. When the subroutine has finished its task, control is returned to the table search routine. At this point the subroutine will have correctly incremented the index register that references the error list so that the table search routine will examine the next flag in the error list.

D003 is also given a word count of the number of words in the error list by Section I". The table search routine tests against this word count for an exhausted error list. If the error list has not been fully treated, the process of table searching, transferring to a subroutine, and returning to the table search routine continues. When all accumulated errors have been treated, D003 then returns control to the main diagnostic record.

The main diagnostic record will space the System tape to the source program error record in the FORTRAN Monitor, this is due to the fact that all errors found by Section I'' are assumed to be source program errors.

DIAGNOSTIC LINKAGE

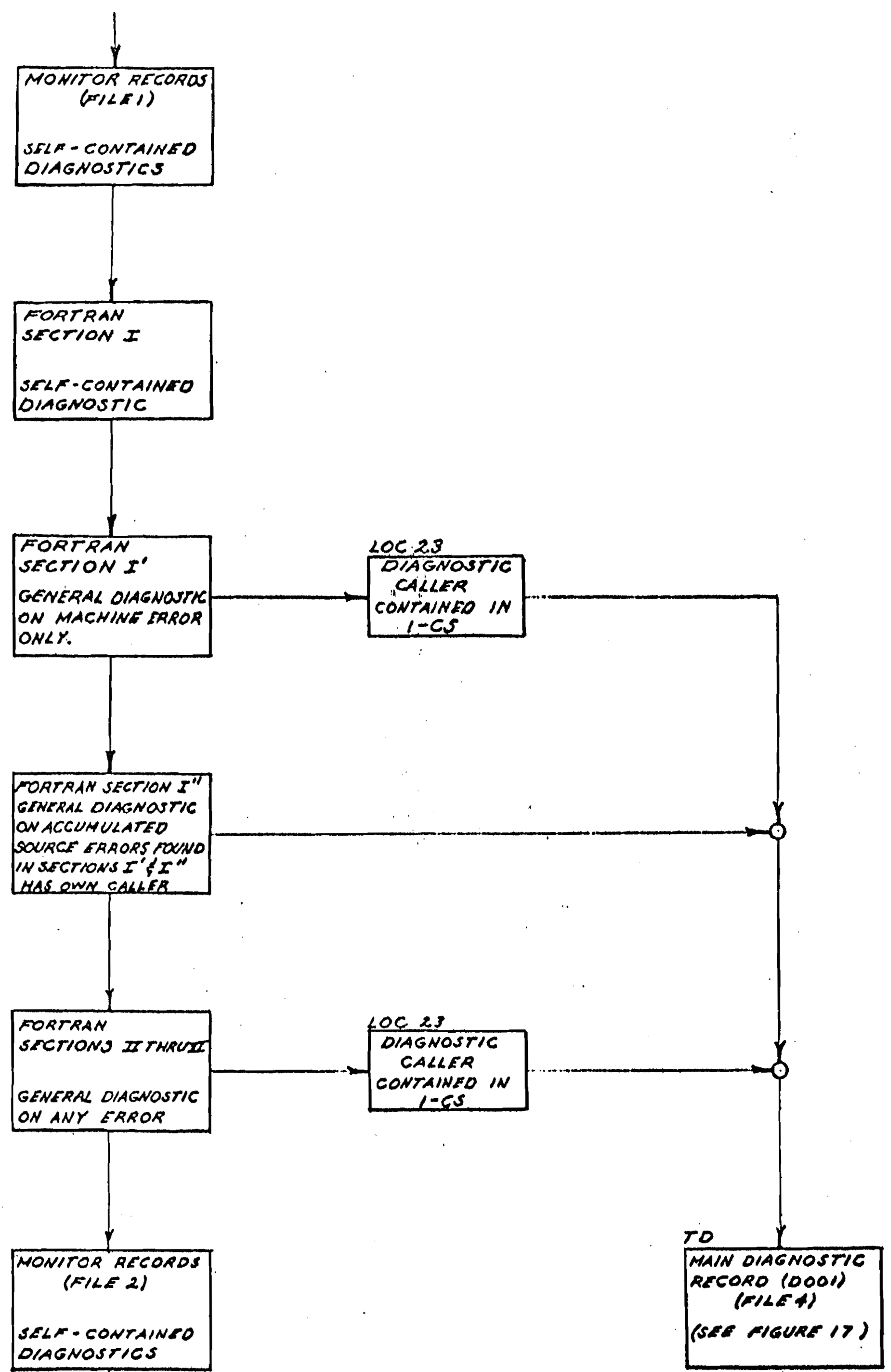


Figure 16

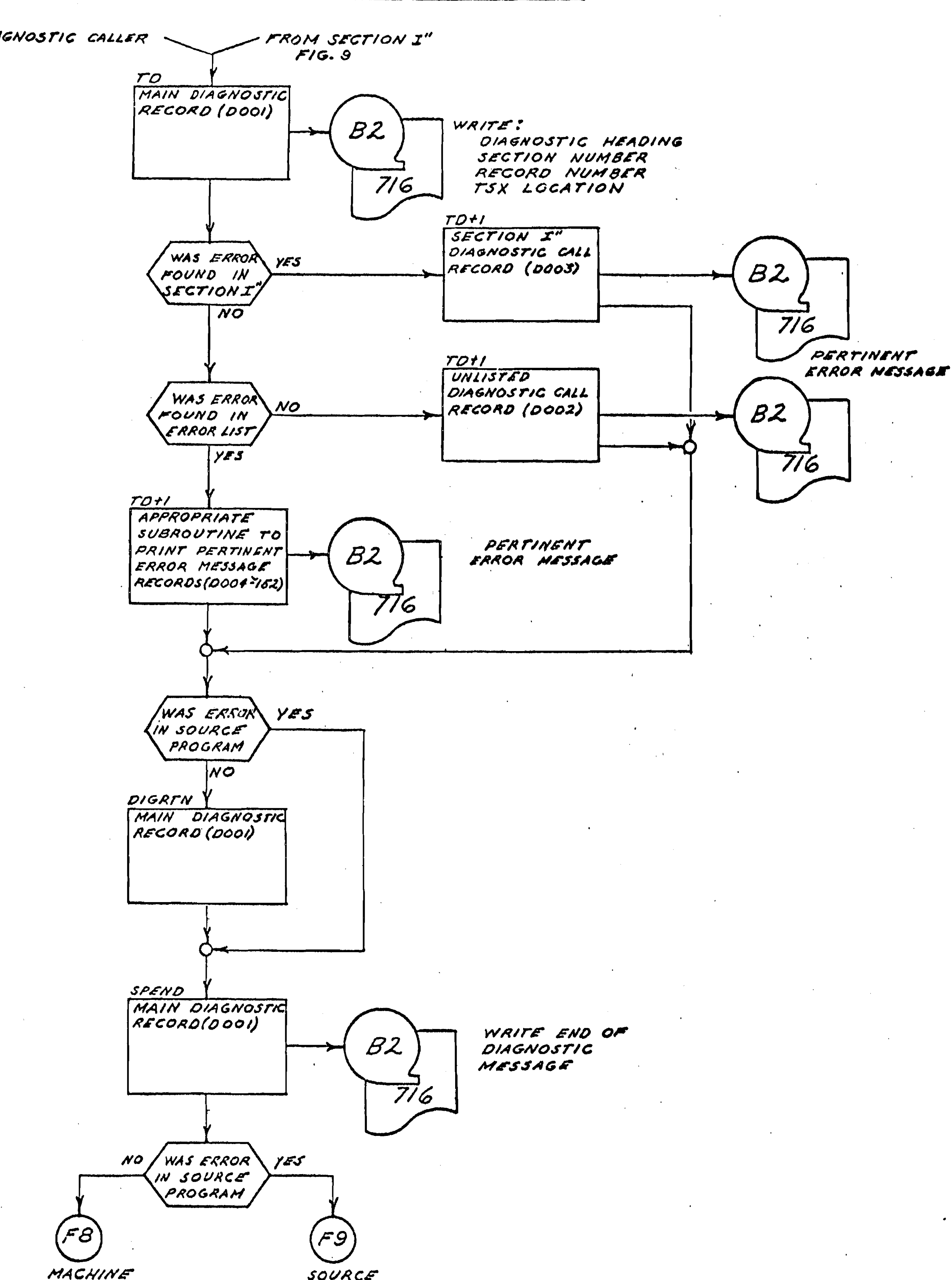


Figure 17

TABLES GENERATED BY FORTRAN 2.04.00

During the execution of the Fortran Executive program the source program is examined and broken down into two principle parts. These two forms are CIT's (compiled instruction tables) and tables.

The objective of the Fortran executive routine is o present to Section VI, the source program in CIT form. Section VI will examine these CIT's,produce the relocatable binary deck map of storage, and the symbolic listing.

When CIT's cannot be completed in a given section, tables are generated to be passed on to subsequent sections to supply the necessary information to complete the CIT's. A good many of these tables are generated by Section I and edited by Section I'. Section I' writes these tables on tape B2 in files 3, 4 and 5. The first word of each of these records is an identification number and the second word is the word count of the record.

Following will be a brief description of some of the principably used tables generated by Fortran.

ASSIGNED CONSTANT 2.04.01

The Assigned Constant table is generated by Section III during the scan of the TIFGO table, which contains the ASSIGNED GO TO entries.

During Section V, if any index saving instructions are necessary to the result of the GO TO, the assigned constant table will be updated so as to reflect this condition.

The table is preceded by an identification number and the table. Each assigned constant is a one-word binary number in the decrement field corresponding to some internal formula number used in the source program.

IFN

During the Section V' all Assigned Constant entries will be placed in CIT form for the 5) region.

Section III will write the Assigned Constants as the eighth file on Tape B2. Section V after updating will write them as the tenth file on Tape B2.

CALLFN (also called CALLNM) 2.04.02

The CALLFN record is a table of internal formula numbers IFN's presented in CALL statements. Each entry into the table requires only one full word. The decrement word contains the internal formula number IFN of the first variable in the CALL statement.

Each IFN in FORVAL is searched for as a first IFN in CALLFN. If found, it is replaced by the corresponding last IFN. When all entries have been processed the CALLFN table is dead.

This table was written during Section I onto tape A4 as buffer size records. It will be read in by Section I and processed.

	IFN (1st VARIABLE)		IFN (LAST VARIABLE)

CLOSUB 2.04.03

The CLOSUB record is a table of closed subroutines called for in the source program. A closed subroutine is one with a single entrance and single exit. Entries are made in the CLOSUB record when the source program refers to a system subroutine or function type subprogram, a subroutine or function defined by the programmer, or by INPUT/OUTPUT statements and by any use of any Fortran function that are defined as closed subroutines. The functions ABSF, XABSF, INTE, XINTF, MODF, XMOOF, MINOF, MINIF, XMINOF, XMINIF, FLOATE, XIFXF, SIGNF, XIGNF, DIMF, XDIMF, are open subroutines. One entry is made in the CLOSUB table for each subprogram called for in the source program, this one word contains the BCD name of the function or subroutine.

The CLOSUB table is stored on Tape A4 during Section I, Section I' will then edit these buffer size records and rewrite this table on Tape B2 in the fifth file with the first word being the identification number. The second word being the count of the number of words in the table.

(BCD) Name of Subroutine

COMPILED INSTRUCTION TABLES 2.04.04

By the end of Section III, the object program is completely compiled in symbolic form (with the exception of library subroutines and some constants).

Ultimately most source information must appear in Compiled Instruction Table, (CIT), form.

Most CIT's are stored on tape during the run of the executive routine. There is only one standard four-word format for CIT's:

WORD 1	LOCATION SYMBOL (BCD)	SECONDARY LOCATION SYMBOL (BCD)
WORD 2	OPERATION CODE (BCD)	DECREMENT OCT

WORD 3	SYMBOLIC ADDRESS (BCD)			
WORD 4	ADDEND OCT	I	TAU	TAG

The decrement of word one contains the IFN. (location symbol), of the instruction in word 2. The address of word 1 is the secondary location symbol if needed.

The decrement of word 2 contains a BCD memonic representation of the instruction for which this entry is made (i. e. , CLA, OCT, ADD, etc.). From this entry the reader can appreciate the sophisication of the Fortran translator, the executive program has written a symbolic instruction which will be subsequently assembled by an assembly program similar to most symbolic assemblers. The address of word 2 contains, if any, the decrement of a type A instruction.

Word 3 contains a BCD representation the symbolic address assigned to the instruction by the executive routine. As in coding by hand, the executive routine uses symbolic addressing in writing its instructions in symbolic form. It is interesting to note that the symbols used by the machine have no mnemonic value to the human reader but of course a one bit difference in configuration is accurate enough discrimination for the machine. A typical symbolic address is 6). The decrement of word 4 is the addend to the symbolic address in word 3 or the relative absolute part of the address of the instruction.

For example, in CLA N+3, the +3 would be entered into this field. The address of word 4 contains the symbolic tag of the FORTAG table for this instruction.

The COMPAIL file generated by Section I and the COMDO file generated by Section II are typical CIT tables. The COMPAIL file is contained in file 2 on tape B2, the COMPDO file that is generated by Section II is contained in file 2 of tape A4.

COMMON 2. 04 05

Normally data and instructions are compiled adjacent to each other in order to preserve high order storage cells.

The COMMON statement permits the programmer to assign specific core storage areas to the storage of data. The COMMON statement is the following form: COMMONX, ANGLE, MATA, and MATB.

The items listed after COMMON statements will be assigned to core storage starting at location 77461_8 and continuing downwards. Entire arrays may be shifted to upper storage through the use of the COMMON statement.

The COMMON table is a compilationof all COMMON statements and is recorded on tape A4 in Section I and edited by Section I' and written on tape B2 in the fifth file.

Each entry into the table requires as many words as there items in the list following the word COMMON. For example, the COMMON statement: COMMON X, ANGLE, MATA, MATB requires the use of four full words and is recorded in the following format in BCD:

WORD 1	X	bl	bl	bl	bl	bl
WORD 2	A	N	G	L	E	bl
WORD 3	M	A	T	A	bl	bl
WORD 4	M	A	T	B	bl	bl

DIM 2.04.06

The DIM record is generated during the arithmetic processing in Section I storage, as a result of encountering DIMENSION statements, and is left in for Section I'. Recall that the DIMENSION statement consists of a list of variables with an integer in parenthesis following the variables. Integer represents the greatest number of elements in an array. During Section I' the DIM table is converted to the SIZE table.

The DIMENSION statement is not executed (no instructions will appear in the object program for this statement) but will preserve blocks of storage for subscripted dimensions and four words for three dimensions. The entries are made according to the following format:

One-dimensional array: Example: DIMENSION A (7)

	Decrement	Address	
WORD 1	A		Subscripted Variable
WORD 2		7	Dimensions
WORD 3	A	+7	Check -sum of entry

Two-dimensional array: Example: DIEMSNION A (7, 12)

	Decrement	Address	
WORD 1	A		Subscripted Variable
WORD 2	7	12	Dimensions 1 & 2
WORD 3	A+7	+12	Check sum of entry

Three-dimensional array: Example: DIMENSION A (7, 12, 6)

	Decrement	Address	
WORD 1	A		Subscripted Variable
WORD 2	7	12	Dimension 1 & 2

WORD 3		6	Dimension 3
WORD 4	A+7	+12+6	Check sum of entry

DOFILE (C) 2.04.07

The DOFILE (C) is a file of CIT entries of subroutines, necessary to complete relative constants. These are generated by Section II after examination of all the DO statements in the source program.

The DOFILE (C) is recorded, in standard CIT format, on tape B2 as the eighth file. A count of the number of records is recorded as a one word record on the tape B2 file nine.

DOTAG (B) 2.04.08

The DOTAG (B) table is the result of an analysis of priority of interlocking DO statements (nests). In this analysis an entry is made in the DOTAG table for every entry in the TDO table. Each table consists of nine words, the first five are identical to the corresponding entry in the TDO table. The last four words are a result of an analysis, by Section II, of the nests of DO statements. The last four words have the following format:

	Level Number of this DO		X=(n2-n1+n3/n3) n3
	N		Level of definition of n1
	*		" " " " n2
	Eras- name of tag which able will be used for test		" " " " n3

* Contains bits, the rightmost of which determines the highest level of transfer from this DO.

EIFNO 2.04.09

After the analysis is done by Section I the last IFN number +1 is left in a cell called EIFNO. The EIFNO table consisting of 1 word only will be carried over to Section I' as a memory table. This will be used during the scan of TIFGO by Section I''. The scan looks at the EIFNO table to see if any IF or GO TO entries are outside the range of the source program.

The EIFNO table will be written as the first word in the SIZ table by Section I'. This will be used by Section VI to form the base location for lower storage variables.

END 2.04.10

The END statement permits the programmer to compile several programs

within one job. The Monitor Scan record, upon sensing an End card, will pass control to the Fortran routine. After compilation of the first job Fortran will turn control back to Scan, to initialize the second program This process will continue until an end of file is sensed on the input tape This indicates an end of job to Scan

In addition to the batch compiling feature of the End statement, it also is used to indicate Sense Switch settings. This will permit the programmer to use separate sense switch control for each program.

The END table consists of 15 entries, one for each Sense Switch option At present, the first five are operative. These correspond to the first five Sense Switches on the operators console The remaining 10 will be added later as additional system features are needed.

The options of each Sense Switch, 0, 1 or 2, are stored in the address portion of each entry, in the END table.

This is generated during Section I and held in storage until Section I' when it is written on tape B2 is the fifth file It will be used by Section VI, as outlined in the Fortran Monitor Operations Manual*, to process the programs options

* (C28-6065)

EQUIT 2 04 11

The EQUIT record is a table containing all the information included in equivalence statements Each item in the pairs of parenthesis requires two full 709 words for storage. The first word contains the variable name in BCD form The constant, if present, is converted to binary, and stored in the second word. If no constant is present, it is taken to be one, and a one is stored in the second word. This process continued for each specification in the Equivalence statement. A minus sign in the last entry indicates the end of a series of equivalent storage locations

WORD 1	VARIABLE NAME (BCD)
WORD 2	N
WORD n	VARIABLE NAME (BCD)
WORD n+1	N

The EQUIT record is generated during Section I and edited in Section I' and written as the 12th record in the 5th file on tape B2.

FIXCON 2.04.12

The FIXCON record is a table of fixed point constant specified by the program. These constants are entered in fixed point form as data or are subsequently computed from other fixed point constants. These numbers, entered without decimal points during READ statements are defined according to some FORMAT statement as fixed point constants, are one of the types entered into FIXCON table. Numbers appearing as constants in statements of the form A = 3 + B are entries in the FIXCON table; in this example 3 is the entry.

The FIXCON table is generated during Section I and retained in core until Section III. Each entry requires two full words, the first being the fixed point constant in binary, the second its check sum.

WORD 1	FIXED POINT CONSTANT
WORD 2	CHECK SUM

FLOCON 2.04.13

The FLOCON record is a table of floating point constants occurring in the source program. They may be entered from an input source such as cards or tape, computed from combinations of floating point constants, or appearing as coefficients with decimal points in Fortran source statements.

The FLOCON table is developed during Section I and is stored in core, in the same format as the FIXCON table. That is, there are two words required for each entry, the first containing the floating point constant and the second the check sum of this one word.

The FLOCON table is written on tape B2, file 4, the first record by Section I'.

FMTEFN 2.04.14

The FMTEFN is a table of external formula numbers associated with the format numbers in read-write statements. For each read-write statement a one word entry is made in FMTEFN containing the binary equivalent of the EFN.

The FMTEFN table was written as buffer size records, during Section I, on tape A4. During Section I' as the table is being assembled, the entries are compared with the FORMAT table. If any statements in the FORMAT table are missing, (no match i FMTEFN,, an error list is developed for Section I'. The table will not be needed again and is considered dead.

FORMAT RECORD 2.04.15

The FORMAT Record is a table of arguments presented in FORMAT statements. The arguments are stored in BCD form in sequential storage loca-

tions. Since the length of arguments is a variable, the number of words required to store all the argument must be variable. Each entry into the table is separated from succeeding entries by a word filled with bits.

WORD 1	EFN of Format statement
WORD 2	FORMAT SPECIFICATION (BCD)
WORD 3	
WORD 4	ALL ONES

This is recorded on tape A4 in buffer size records by Section I. During Section I' it is edited and written as the second record of file 4 on tape B2. The FORMAT table will be used in Section V'.

FORSUB 2.04.16

A FORSUB entry is made for each arithmetic function definition appearing in the source program. The function name appears on the left side of the equal sign and the parameters appear on the right.

Each entry in FORSUB requires 2 words. The first word is the function name, the second word is the internal formula number of the statement. For example: FIRSTF (X) = A*X+B. The table appears:

WORD 1	FIRST (BCD)	
WORD 2	IFN	

The FORSUB record is retained in memory until Section I', when it is written as the only record in file 3 on tape B2. The first word of the FORSUB record is the COMPAIL record count, and the second word is the FORSUB word count.

FORTAG 2.04.17

The FORTAG record is a table that represents an index to the TAU table. It has a one-word entry of the following format:

IFN		*XR INFO	**INDEX TO TAU TABLE
1 17		24 26	27 35

*XR INFO. This field indicates whether or not the FORTAG entry use an absolute or symbolic index register. If there are no entries, a symbolic XR is inferred. If there is an entry the field is treated like the tag field of an instruction (i.e., 24 = XRA, 25 = XRB, 26 = XRC).

** Index to TAU table. The bit configuration in this field indicates which TAU table entry has the associated IFN.

The table is generated during Section I and written in buffer size records on tape A4. During Section I' the table is edited and written at the 11th record in the 5th file on tape B2 with its identification label.

FORVAL AND FORVAR 2. 04. 18

The FORVAL and FORVAR records are tables of the fixed point non-subscripted variable, appearing to the left of (FORVAL), and the right of (FORVAR), the equality sign in a statement. A fixed point non-subscripted variable must satisfy the following conditions:

1. Must be six or less characters.

2. The first character must be alphabetic.

3. I an integer, it must start with I, J, K, L, M or N.

4. Must not read like a function name.

5. Must not have a left parenthesis following it.

6. Must be entered as data in fixed point form.

For example, if A and B are fixed point form, the statement, "ARG - BRAND +6" contains "ARG" as an entry in the FORVAL table and "BRAND" as an entry in the FORVAR table.

For example, the statement ARG = BRAND +6 would be written:
FORVAL TABLE (BCD)

A R G

FORVAR TABLE (BCD)

B R A N D

The tables were generated during Section I and temporarily stored on tape A4 in buffer sized records. The tables are edited during Section I' and written as the 9th and 10th records in file 5 on tape B2.

FRET 2. 04. 19

The FRET table is a table generated from the FREQUENCY statement given

in the source program. This is a variable length entry table; that is, each entry occupies an indeterminate number of words, dependent on the number of branch points described by frequency statements. Each FREQUENCY statement permits the programmer to specify the number of times a particular branching point will be utilized by the source program. For instance, a particular IF statement may appear in a program as:

38 IF N (10, 20, 30)

The programmer can best use index registers in the program by informing the program that branch 10 will be used five times, branch 20 will be used three times and branch 30 will be used six times, by entering the following frequency statement:

FREQUENCY 38 (5, 3, 6)

The general form is

FREQUENCY N (i, j, k....)

Where N = EFM of branch point

i, j, k = frequency of each branch

Entries into the FRET table are made according to the following format:

	Decrement	Address
WORD 1		38
WORD 2		5
WORD 3		3
WORD 4		6

The length of each entry will be determined by the number of branches.

During Section I the FRET table was written as buffer size records on tape A4. During the editing by Section I', each EFN in FRET is searched for in TEIFNO. When found it is replaced with the corresponding IFN. If not found, it is set equal to zero as an error signal for Section I'. The FRET table is then sorted by IFN to form an ordered list. It is then written as the 12th record of the 5th file on tape B2.

HOLARG 2.04.20

The HOLARG record is a collection of Hollerith arguments in CALL statements. In a CALL statement, the Hollerith argument, is not describing an argument of some subprogram, but is itself the data to be operated on. An example of this kind of CALL statement is:

CALL (9HEND POINT)

In this example the name of the subprogram is "SUBP" which is a hardcoded program. In the argument 9 specifies the number of Hollerith characters that follow and the H specifies that the data is Hollerith. "The HOLARG" table will contain only the "END POINT" the 9H will be dropped. Since any number of characters may be specified, "the HOLARG" table can be a variable number of words per entry. For this reason a word of all ones is written at the last word in the table.

As in the above example the table would look like:

WORD 1	E	N	D	bl	P	O
WORD 2	1	N	T	bl	bl	bl
WORD 3	77	77	77	77	77	77

The HOCARGE table is written during Section I on tape A4 as buffer size records. During Section I' it is written as the 4th record in file 5 on tape B2 with its label number and word count. This table will be used during Section VI to generate the 5) region.

NONEXC 2.04.21

The NONEXC record is a table of IFN's and associated EFN's for non-executable Fortran statements. The following statements are non-exeuutable:

PAUSE, FORMAT, DIMENSION, EQUIVALENCE, FREQUENCY

Each entry into the NONEXC table requires only one word. The decrement of this entry contains the IFN, and the address contains the EFN of the non-executable instruction.

The HOLARGE table is generated during Section I. It is retained in core until Section I', when it will be used to detect transfers to non-executable statements, and flag these as errors. The NONEXC table will not be used again.

PREDESSOR 2.04.22

During the flow analysis by Section IV the source program is broken down into what is termed Basic Blocks. A Basic Block is a stretch of source program into which there is only one entrance and from which there is only one exit. Exit must here be interpreted in the logical sense, that is, it may consist of more than one transfer instruction, going to a variety of Basic Blocks. Each of these Basic Blocks, then is a Successor Basic Block. As implied by this, Section IV must mark off the basic blocks of the program and determine the Successor and Predessor Basic Blocks for any one Basic Block. During the flow analysis by Section IV a count of the number of times that a Basic Block is entered, this is called the flow count. The PREDESSOR table is made up of one word entries for each Basic Block.

3	17	21	35
FLOW COUNT		BASIC BLOCK NUMBER	

The decrement portion of the Predessor table entry contains the flow count, the address portion contains the Basic Block number. The PREDESSOR table is passed on from Section IV as a memory table to Section V to be used for a further analysis of the flow of the source program.

SIZ 2.04.23

The SIZ table contains the variable and maximum dimensions of arrays. This table is made up of the product of the dimensions contained in the DIM1, DIM2 and DIM3 tables generated by Section I.

The SIZ record requires two full words for each entry. The entry is of the following format:

WORD 1 VARIABLE NAME (BCD)

WORD 2 TOTAL SIZE OF ARRAY

For example: given the DIMENSION statement:

DIMENSION C (3, 4, 5)

The table entry would appear as:

WORD 1 C(BCD)

WORD 2 *60 (BIN)
*3x4x5 = 60

The SIZ table is written as the 3rd record in file 4 on tape B2 by Section I'.

SUBDEF 2.04.24

Fortran can also call in subroutines described by the programmer in the source program. For example, the subroutine introduced by the statement SUBROUTINE MATMPY (A, N, M, B, L, C) could be called into the main program by the statement:

CALL MATMPY (X, 5, 10, 4, 7, Z).

Essentially, what happens is that the previously described MATMPY subroutine is brought into the compilation with the arguments of the SUBROUTINE statement. Naturally the arguments of the SUBROUTINE statement should correspond in mode, number, and order to those of the original MATMPY subroutine.

Each entry into the SUBDEF record requires one full word for the name of the subroutine (i.e., MATMPY) and one full word for each of the arguments included (A, N, M, B, L, C) is recorded as:

	Decrement	Address	
WORD 1		MATMPY	(BCD)
WORD 2		A	
WORD 3		N	
WORD 4		M	
WORD 5		B	
WORD 6		L	
WORD 7		C	

The SUBDEF table is recorded in buffer size records, during Section I, on tape A4. During Section II it is written as the second record in file 5 on tape B2 with its table number and word count.

SUCCESSOR 2.04.25

The SUCCESSOR table is identical to the PREDESSOR table described in Section 2.04.22. The main difference is that the address portion of the one word entry contains the SUCCESSOR Basic Block number instead of the PREDESSOR basic block number. This table is passed on to Section V as a memory table.

TAU 2.04.26

The TAU table is a collection of the subscript information used in I/O lists or in Arithmetic expressions. The TAU table may be one, two, or three dimen-

sional, recorded in the following format:

Subscript is one-dimensional:

TAU1

WORD 1	C1*	
WORD 2	VARIABLE NAME (BCD)	

C1 is coefficient

Subscript is two-dimensional:

TAU2

WORD 1	C1	C2
WORD 2	VARIABLE NAME 1 (BCD)	
WORD 3	VARIABLE NAME 2 (BCD)	
WORD 4	d1	

Subscript is three-dimensional:

TAU3

WORD 1	C1	C2
WORD 2	C3	
WORD 3	VARIABLE NAME 1 (BCD)	
WORD 4	VARIABLE NAME 2 (BCD)	
WORD 5	VARIABLE NAME 3 (BCD)	
WORD 6	d1	d2

C1 is first coefficient
C2 is second coefficient
C3 is third coefficient
d1 is first dimension from DIM3
d2 is second dimension from DIM3

TDO 2.04.27

The TDO record is a table which results from DO statements in the symbolic program. Each entry requires five full words. The five words are written according to the following format:

WORD 1	Decrement	Internal formula number (IFN of the DO Statement (α)
	Address	The EFN of the last statement executed under control of the DO statement (β)
WORD 2	Decrement	The BCD symbol for the integer variable of the DO statement (I, J, K, L, M or N)
WORD 3	Address	First value of variable (n_1)
WORD 4	Address	Final value of variable (n_2)
WORD 5	Address	Increment of the variable (n_3)

The following DO statement would result in the table entry shown:

5 DO 8 I = 1, 25, 2

	Decrement	Address
WORD 1	5	8
WORD 2	I	
WORD 3		1
WORD 4		25
WORD 5		2

TIEFNO 2.04.28

Two reference numbers are associated with Fortran statements, the internal IFN and external EFN formula numbers. All statements in the source program have internal formula numbers (IFN). These numbers are assigned to the statement sequentially starting with 1. The external formula number (EFN), on the other hand, is an arbitrary integer assigned to the statement by the programmer, generally to permit reference to the particular statement by the source program. There is no need to assign an external formula number to any statement to which reference is never made. Therefore, all statements have IFN and some have both IFN and EFN.

The EFN's and their corresponding IFN's are stored in the TEIFNO record by the translator during the run of Section I. Each statement requires the use of one full word for storage. Then entry is made as follows:

IFN	EFN

For example, if the following statement is the 28th statement in the program, the indicated table entry is made.

STATEMENT

15 DO 6 I = 1, 8

TEIFNO Entry

28	15

The TEIFNO table is written as buffer size records on tape A4 during Section I. During Section I' it is written as the fifth record in file 5 on tape B2 with its corresponding table number and word count.

TIFGO 2.04.29

The TIFGO record is a table of the IF, ASSIGN, and GO TO statement in the source program. Each statement in the program demands the use of two full 709 words for storage. This section describes entries that result from each type of statement. The first word of the first record in the TIFGO table is the label number.

IF Statement Entry. Example: IF (E) n_1, n_2, n_3

The entry for this statement is shown below:

WORD 1	(IFN)	n_1
WORD 2	n_2	n_3

Unconditional GO TO Entry. Example: GO TO n

The Entry for this statement is shown below:

WORD 1	(IFN)	0
WORD 2	0	n

Assigned Go To Entry

In this type of statement the GO TO destination is determined by a previous ASSIGN statement. The list of alternatives following in parenthesis are merely a list of all the possible GO TO destinations.

For example, consider the following statement:

GO TO N (B_1, B_2, B_3, B_4..............B_N)

The GO TO destination will be the Ith statement. The TIFGO table entry for this statement would be:

WORD 1	IFN	2
WORD 2	*$CTRAD_1$	**$CTRAD_N$

*$CTRAD_1$ - The number of the entry in the TRAD record corresponding to the first possible transfer address given in the GO TO argument.
**$CTRAD_N$ - The number of the last possible transfer address.

Computed GO TO Statement. Example: 26 GO TO (B_1, B_2, B_3....B_N) I.

In this type of statement a transfer will take place in the object program dependent on the current value of I. I is a variable which is assigned some computed integer by the source program. The transfer takes place to the Ith term of the GO TO list of B's. For example, if the value of I is computed as 3, then the program will transfer to the third location in the list of locations which follow GO TO (B_3 in the example). The entry for the computed GO TO takes the following form:

WORD 1	(IFN)	2*
WORD 2	$CTRAD_1$	$CTRAN_N$

*Unconditional

Assigned Statement Entry. Example: ASSIGN 30 to N

This statement is used in conjunction with a GO TO statement, as described under the computed GO TO Statement.

The table entry of the above example takes the following form:

WORD 1	(IFN)	6*
WORD 2		Assigned value 30

Indicator-Controlled IF Statement. Example: IF(indiaator type) A_1, A_2

This statement is used in conjunction with:

1. Sense switches
2. Sense lights
3. Divide check indicator
4. Accumulator overflow light
5. Quotient overflow light

If the corresponding indicator is on or switch is down, transfer of the program proceeds to the statement specified by the first number following the parenthesis. If the corresponding indicator is off or switch is up, transfer of the program proceeds to the statement specified by the second number following the parenthesis.

The table entry takes the following format, for the example given:

	Decrement	Address
WORD 1	(IFN)α	3, 4, or 5*
WORD 2	n_1	n_2

*3 = Sense Switch or Sense Light
4 = Divide Check
5 = ACC or MQ overflow

The TIFGO FILE table is written as buffer size records on tape A4 during Section I. During Section I' it is written as the sixth record in file 5 on tape B2, with its corresponding table number and word count.

TIFGO FILE 2.04.30

The TIFGO FILE table is generated in Section III. It is produced from the TIFGO and TRAD table from Section I, and the TRALEV and TRASTO table from Section II.

The need for the TIFGO FILE of instructions arises in the following manner. The main body of computing an indexing instruction, included in the COMPDO file, are associated with the beginning and end of DO's. However, if a transfer should exist within DO certain indexing and saving instructions will be necessary if entry is made back into the DO from the transferred point. The TIFGO FILE contains the CIT's necessary to produce these index saving instructions.

The TIFGO FILE is recorded as the eighth file on tape B2 by Section III. During the last pass of Section III theTIFGO FILE is read in from tape B2 and merged with the COMPDO and COMPAIL files. This merge constitutes the first file on tape A4, at the end of Section III.

TRAD 2.04.31

The TRAD table contains all of the possible transfer addresses listed in assigned and computed GO TO statements.

As many words are used as there are transferred to, addresses in the GO TO statement. The transfer address (EFN) is entered in binary form into the address field of consecutive words in the TRAD table.

The following GO TO statement would cause the following table entry to be made:

GO TO (n_1, n_2, n_3, ... n_m), I

TRAD

WORD 1	n_1 (EFN)
WORD 2	n2 (EFN)
WORD 3	n_3 (EFN)
WORD m	n_m (EFN)

The TRAD table is written as buffer size records on tape A4 during Section I. During Section I' all external formula numbers (EFN) are searched for in TEIFNO and when found are replaced by their corresponding internal formula number (IFN). TDO is then written as the eighth record of file 5 on tape B2.

TSTOPS 2. 04. 32

The TSTOPS table contains the external and internal formula numbers associated with the STOP statements in the source program. Each entry into the table requires only one full word. The decrement of the word contains the IFN and the address contains the EFN of the STOP statements.

An entry is also made in the TSTOP table for return statements. The TSTOP table will be left in core for processing by Section I''. At the end of Section I'' the table will no longer be used.

APPENDIX A

FORTRAN TAPE B2 STATUS BY SECTION

(This configuration holds only at the end of the given section)

File	CONTENTS			Written by Section	Overwritten by Section *
1	SOURCE PROGRAM (BCD) - 1 FORTRAN Statement card/record			Peripheral Reader or card-to tape simulator	
2	COMPAIL - 100 words/record			I-Pass II	VI
3	COMPAIL RECORD COUNT and FORSUB (if it exists)			I'	VI
4	Table Label	Table Name (In order as on tape)	Maximum No. of Words	I'	
	10	FLOCON	1800		
		FORMAT	6000		
		SIZ	2320		
5		END	15	I'	
	11	SUBDEF	180		
	12	COMMON	2400		
	13	HOLARG	3600		
	0	TEIFNO	3000		
	2	TIFGO	2400		
	3	TRAD	1000		
	1	TDO	3000		
	6	FORVAL	4000		
	5	FORVAR	6000		
	4	FORTAG	6000		
	7	FRET	3000		
	8	EQUIT	6000		
	9	CLOSUB	6000		
6	DOTAG B - variable number of records - -variable number of entries /record - -9 words /entry			II - Block 2	
7	DOTAG B RECORD COUNT			II - Block 2	
8	DOFILE C - CIT's for A) subroutines			II - Block 4	III - Merge II
9	DOFILE C RECORD COUNT			III - Block 4	III - Merge III
8	TIFGO FILE			III - Merge III	III - Merge III
8	ASSIGN CONSTANT			III - Merge III	
9	FIXCON			III - Merge III	
10	ASSIGN CONSTANT			V - Part 3	
2	STORAGE MAP (BCD) FOR PROGRAM			VI	
3	SYMBOLIC LISTING FOR PROGRAM			VI	

* Any overwriting of file(s) obsoletes all information previously following it on the tape.

APPENDIX A

FORTRAN TAPE B3 STATUS BY SECTION

(This configuration holds only at the end of the given Section)

File	Contents	Written by Section	Overwritten by Section *
1	CONDENSED SOURCE PROGRAM	I - PASS 1	II - BLOCK 1
1	DOTAG A - Variable number of records; variable number of entries/record; 9 word/entry; maximum of 1350 words DOFILE - INTERMEDIATE CIT's for DO STATEMENTS 400 words/record	II - Block 1 II - Block 5	III - Block 1 III - Block 1
1	FIRSTFILE Merged CIT's of COMPAIL and COMPDO) 100 words/record	III - Block 1	V - Block A
2	CIT's FOR FORTRAN FUNCTIONS - 100 words/record	III - Block 1	IV - Block 3
2	DOUBLE END OF FILE MARK	Block 3	
3	TAGLIST - 15 words/record	Block 3	
4	BBLIST - 6 words/entry	Block 3	
1	CIT's	V - PART IV	V'
1	ASSEMBLED TABLES	V'	VI - PART A
1	EIFNO	VI - PART A	VI
1	BINARY OUTPUT (card image form) a. Program Card b. Binary Object Program c. Library Routines (if requested) d. EOF	VI	

* Any overwriting of file (s) obsoletes all information previously following it on the tape.

APPENDIX A

FORTRAN TAPE A4 STATUS BY SECTION

(This configuration holds only at the end of the given Section)

File	Contents	Written by Section	Overwritten by Section *
1	Various table buffers written in the order in which filled. Each table is preceded by an identification label. (See tape B2, files 4 and 5).	I	II
1	TRALEV - maximum 2400 words/record	II - Block 1	III MERGE
2	TAGTAG - 1 record/nest of DO's with tags; 4 words/ tag entry	II - Block 2	II- Block 6
2	COMPDO - 100 words/record	II - Block 6	III- MERGE 3
1	MERGED CIT's OF COMPAIL, COMPDO, TIFGO	III - MERGE3	VI- PART A
2	CIT's for CLOSED SUBROUTINES FOR DOFILEC and FORTRAN FUNCTIONS	III - MERGE3	VI- PART A
1	CIT's (complete)	VI - PART A	
2	CLOSUB (1 record)	VI - PART A	

* Any overwriting of file(s) obsoletes all information previously following it on the tape.

APPENDIX B

FORTRAN EDIT RECORD CHART - 32K SYSTEM

Record Number	Description	Transfer Address	Initial Address	Final Address	#2 Word (Loc 56_8) Fortran Records
File # 1	MONITOR				
9F00	1-CS	50	23	56	xxx xxx xxx xxx
01	Card-to-tape	145	144	1026	100 012 000 145
02	Dump	145	144	3312	100 024 000 145
03	Sign - on	145	144	1311	100 036 000 145
04	FAP-Pass 1	232	144	10435	100 050 000 232
05	FAP-Pass 2	232	232	7163	100 062 000 232
06	Monitor Scan	145	144	2076	100 074 000 145
07	BSS Control	74457	74454	77777	100 106 074 457
08	Machine Error	145	144	1474	100 120 000 145
09	Source Error	145	144	352	100 132 000 145
09. 1	Dummy Record	144	144	145	100 133 000 144
File # 2	SECTION I				

Record Number	Description	Transfer Address	Initial Address	Final Address	#2 Word (Loc 56_8) Fortran Records
9F10	Pass 1	1014	144	4100	100 144 001 014
11	Pass 2	1062	3000	17515	100 156 001 062
12	Diagnostic	21451	21451	24251	100 170 021 451
	SECTION I'				
9F13		1014	1014	3502	100 202 001 014
	SECTION I"				
9F14		315	315	1477	100 214 000 315
	SECTION II				
9F15	Block One	357	144	2344	100 226 000 357
16	Block Two	453	310	4151	100 240 000 453
17	Block Three-A	421	310	21471	100 252 000 421
18	Block Three-B	421	421	1513	100 264 000 421
19	Block Four	15341	15321	16202	100 276 015 341
20	Block Five	14147	14147	34476	100 310 014 147
21	Block Six	310	310	412	100 322 000 310
	SECTION III				

Record Number	Description	Transfer Address	Initial Address	Final Address	#2 Word (Loc 56_8) Fortran Records
9F22	Block One-A	50	27340	27506	100 334 000 050
23	Block One-B	310	144	2547	100 346 000 310
24	Block Two	310	310	2633	100 360 000 310
25	Block Three	310	310	2633	100 372 000 310

Record Number	Description	Transfer Address	Initial Address	Final Address	#2 Word Fortran Records
	SECTION IV				
9F26	Block One	242	144	1225	100 404 000 242
27	Block Two	247	202	1350	100 416 000 247
28	Block Three	250	207	1232	100 430 000 250
	SECTION V				
9F29	Part One	4617	144	11511	100 442 004 617
30	Part Two	203	202	526	100 454 000 203
31	Part Three	203	203	523	100 466 000 203
32	Part Four	3367	202	4346	100 500 003 367
	SECTION V'				
9F33		366	144	702	100 512 000 366
	SECTION VI				
9F34	A	460	144	1676	100 524 000 460
35	C	632	460	1434	100 536 000 632
36	G	460	460	1357	100 550 000 460
37	H	460	460	1022	100 562 000 460
38	I	460	460	747	100 574 000 460
39	J	460	460	1626	100 606 000 460
40	N	460	460	6654	100 620 000 460
41	P	460	460	3540	100 632 000 460
	MONITOR				
9F42	Tape Mover	270	144	1461	100 644 000 270
43	BSS Control	74457	74454	77777	100 656 074 457

File # 3 LIBRARY

9FPT	9CSH	9RER
9XP1	9SCH	9IOH
9XP2	9SPH	9IOB
9XP3	9STB	9IOS
9ATN	9TSB	9IOU
9XPF	9BST	9CHN
9LOG	9EFT	9DMP
9SCN	9RWT	9XIT
9SQR	9DRM	9XLO
9TNH	9SLO	9TES
9TSH	9SLI	
9STH	9WER	

File # 4 GENERAL DIAGNOSTICS