Stu Nagy

DIRECTORY

The directory is a table which contains all necessary information about each program or file in the system library. It resides on the disc and may occupy from 1 to 4 disc tracks, depending upon how many discs there are on the system. A core resident table called DIREC contains information on the directory itself.

A directory entry consists of 8 words and has the following format:

WORD		
	0	user id
	1	program or file name — BIT 15 = 1 if protected, 0 if unprotected.
	2	program or file name — BIT 15 = 1 if file, 0 if program
	3	program or file name
	4	unused
	5	date
	6	disc address
	7	-length in words

The directory entries are kept sorted on words 0-3. BIT 15 of words 1 and 2 are not considered in the sorting. Names of fewer than 6 characters are filled out with spaces (40_8). The date is the most recent date on which the program or file was referred to.

The directory contains 2 pseudo entries which are the first and last entries in the table. They have the following form:

	FIRST ENTRY	LAST ENTRY
0	0	177777
1	0	177777
2	0	177777
3	0	177777
4	0	0
5	177777	177777
6	0	0
7	0	0

When the directory occupies more than one track, all the directory tracks appended together form the directory.

1. A. DIREC

DIREC is a core resident table which contains information about the disc directory. It has the following structure:

WORD		
	Ø	-length in words of first directory track
	1-4	same as first 4 words of first directory track
	5	unused
	6	disc address of first directory track
	7-13	same as 0-6 but applied to 2nd directory track
	14-20	same as 0-6 but applied to 3rd directory track
	21-27	same as 0-6 but applied to 4th directory track

A disc address of 0 implies that there is no such directory track. When word Ø is 0, words 1-4 are meaningless.

The disc address of a directory is always sector 0 of a track. Each directory track may contain as many as 5440 words = 85 sectors = 680 directory entries. Directory tracks are allocated as follows:

a.) When the system is initially loaded, or when it is loaded from mag. tape, the number of directory tracks allocated is the maximum of the original number of tracks and the number of discs;

b.) When the DISC command is used to add a disc, a new directory track is allocated unless this would cause there to be more directory tracks than discs.

II. ID TABLE

The ID table (IDT) is a disc resident table which contains one 8-word entry for each ID code on the system. The entries are kept sorted according to the ID codes. An entry has the following format:

WORD			
	Ø	user id	
	1-3	password	(filled with Ø's if fewer than 6 characters
	4	time allowed	(in minutes)
	5	time used	(in minutes)
	6	disc allowed	(in sectors)
	7	disc used	(in sectors)

Words 4-7 are 16 bit quantities with values between 0 and 65535. The following 2 words in core refer to the IDT:

IDLOC = disc address of IDT.

IDLEN = length in words of IDT.

III. AVAILABLE DISC TABLE

The available disc table (ADT) is a disc resident table which contains one two-word entry for each area of the disc which is unallocated. An entry has the following form:

WORD	Ø	disc address
	1	length of area in sectors

Entries are sorted according to word Ø. Each entry may refer to as much as one full track, and no two consecutive entries ever refer to two adjacent disc areas (two tracks are not considered to be adjacent).

Besides the entries for unallocated areas, there is also one ADT entry for each of the five tracks on which the system itself resides, and for each of the sixteen tracks allocated for user swapping. Word 1 of each of these entries is 0 so that they will never be allocated. The purpose of having these entries is to indicate to the system dump that they may be released at that time,and also to indicate to the LOCK and UNLOCK routines that these tracks have special significance.

At the end of the ADT is one additional entry having the form:

0	177777
1	0

Since track 0 is always allocated as a system track, any possible disc address is guaranteed to be bounded by two ADT entries.

The following two memory locations refer to the ADT:

ADLOC = disc address of ADT

ADLEN = -length in words of ADT

The IDT and ADT always reside on the same track. The IDT is at the beginning of the track (sector 0), and the ADT begins at the first sector that is unused by the IDT.

IV. FUSS

The FUSS table is a 128 word table which resides on the disc. Its disc address can be obtained by the instruction

LDA FUSS,I

FUSS is divided into 16 sections of 8 words each. The 8 words in each section are the disc addresses of the user files currently being accessed by the user corresponding to that table. Addresses of 0 indicate no file. Disc addresses with bit 7=1 indicate that the user has read only access. The purpose of maintaining this table is to

1.) prevent simultaneous write access by two users to one file;
2.) prevent KILLing a file when some user has access to it.

A user's FUSS (i.e. his area of the FUSS table) is set by the FILES routine, which is called from BASIC at the beginning of execution of a program containing a FILES statement. It is cleared by BYE,HELLO,KILLID, and sometimes by KILL.

V. COMTABLE

The COMTABLE is a list of all user and system commands containing their ASCII codings and disc locations or core addresses. The structure of the COMTABLE is as follows:

COM1	codes for commands which are executed immediately by the system
COM2	codes for commands which are executed by BASIC
COM3	user commands which are executed by disc resident programs
COM4	system commands - - all are executed by disc resident programs
COM5	starting addresses for those commands which are listed under COM1 and COM2
COM6	disc addresses for those commands which are listed under COM3 and COM4

(this section is filled by the loader)

Since each command is recognized only by its first 3 letters, the scanner converts each letter into a number from 0 to 31_8, and then packs the three codes into one word as three 5-bit bytes. In addition, bit 15 is set for system commands. Codes of -1 in sections 2, 3, and 4 do not correspond to any possible 3-letter code. Their purpose is to generate room in COM6 for disc addresses of routines that are called indirectly, or for tables like FUSS. In the case of CTAPR, the purpose is to generate a status type for printing compiler tape errors without a direct command from the user.

VI: LOGGR

LOGGR is a 32-word queue which contains codes for printing LOGON/OFF messages. Entries are placed on the queue by HELLO, BYE, and SLEEP. Each entry consists of 2 words, with the following format:

WORD Ø: user id (BIT 15=0 for ON, 1 for OFF)
1: bits 15-4 = 60 x hrs + mins
bits 3-0 = terminal number

The representation of a user id is as follows:

BITS 14-10 = letter (A = 1, B = 2, ..., Z = 32_8)

BITS 9-0 = number (0-999)

The following variables are relevant:

LOGCT = # of unprocessed entries in LOGGR
LOGP1 = points to word 1 of last processed entry
LOGP2 = points to word 1 of last unprocessed entry

Note that LOGCT = 0 <=> LOGP1=LOGP2

VII TELETYPE TABLES

This set of 16 tables, one for each user, contains relevant information about the various terminals. The structure of the tables is as follows:

WORD	0	BTIM
	1	CHAR
	2	BCNT
	3	MASK
	4	CCNT
	5	BPNT
	6	BSTR
	7	BHED
	8	BGIN
	9	BEND
	10	LADR
	11	DISC
	12	PROG
	13	ID
	14-16	NAME
	17	PHON
	18-19	TIME
	20	ABCN
	21	CLOC
	22	RSTR
	23	STAT
	24	LINK
	25	PLEV

BTIM, CHAR, BCNT, CCNT, BPNT, BSTR, BHED, BGIN, and BEND are used for I/O and buffer control. BTIM is used by the MPX driver to count interrupts between bits. BCNT is used to count bits within a character. CHAR is used to pack input bits or unpack output bits. The use of the other items varies

depending upon whether input or output is being performed.

During input, the user's buffer acts as a character queue. BHED points to the first character of the first unprocessed input line, i.e. the head of the queue. BSTR points to the first character of the current input line. BSTR = BHED except when the user is in TAPE mode, and there exists the possibility of multiple input lines. At the end of each line of input, BSTR is set to point just beyond the line. After a line is processed by either BASIC or by the system, it advances BHED beyond the line just processed. If it remains unequal to BSTR, a second line must be processed. BPNT points to the tail of the queue, i.e., the location into which the next character is to be deposited. BGIN and BEND are fixed pointers which give the first character of the physical buffer and the first character beyond the physical buffer, respectively. Note that character pointers have the form:

BITS 15-1: WORD ADDRESS
BIT 0 : 0 for left character. 1 for right character

During output, the buffer again acts as a character queue, but lines have no significance. CCNT = -number of characters to be transmitted, including the current one. BPNT points to the character currently being transmitted, and BSTR to the location into which the next output character will be deposited.

The remainder of the words in the table are defined as follows:

LADR: a Pointer to the user's LADDR entry

DISC: disc address of user's swap area

PROG: when user is on the disc, PROG points to the last core location used by the program. When the user is loaded into core, PROG is placed into PBPTR. When he is written back to disc, PBPTR is copied into PROG. BASIC is required to maintain PBPTR as a bound on the core it is using.

ID: user's id, 0 if none

NAME: a three word entry containing the user's program name. It is set by the routines NAME & GET, and cleared by HELLO. When fewer than 6 characters are in the name, blanks are appended.

PHON: when the system telephone routine is timing the user for various things, PHON is set to the value of DATIM+1 necessary to achieve timeout.

TIME: the value of DATIM (0 : 1) when the user logged on.

ABCN: this is used by the abort checker in the MPX driver to count the length of a BREAK. When the abort checker senses a 0 input bit that may be the start of a BREAK, it sets ABCN = -100. If the 0 bits continue for 100 consecutive MPX time periods (~114 ms), then the BREAK key has been pressed.

CLOC: this is the timeout clock used to determine the length of a user's time slice. See the discussion on scheduling for further information.

RSTR: this is set, when a user is placed on the queue, to his starting address in core. When the user is actually initiated, RSTR is set to 0. Whenever RSTR = 0, the transfer address of the user can be found in location PREG.

STAT: indicates user's status. The user's status is as follows:

-2,	system disconnect
-1,	user abort request
0,	idle
1,	system abort
2,	input wait
3,	output wait
4,	syntax processing
> 4,	command processing

When a command is being processed, STAT indicates the command. STAT values are assigned in order of entries in the COMTABLE, so that

RUN = 5

LIST = 6

PUNCH = 7, etc.

LINK: the LINK words in the tables are used to form a queue of active users. All users whose status is ≥ 4 are in the queue. See the discussion on scheduling for further information.

PLEV: this word gives the priority level of the user when he is on the queue. When the user's status is set to 2 or 3, the previous value of STAT is copied into PLEV, and the user removed from the queue. The possible values of PLEV are as follows:

0: highest priority, used for syntax, users returning from I/O suspend, and for disc resident routines once they begin.

1: used for commands RUN,LIST,PUNCH

2: used for disc resident routines until they reach the top of the queue

4: used for long running programs

Associated with each item in these tables is a symbol which is EQUated to the corresponding number of the item. For example:

```
?BTIM   EQU   0
?CHAR   EQU   1
  .
  .
  .
?PLEV   EQU   25
```

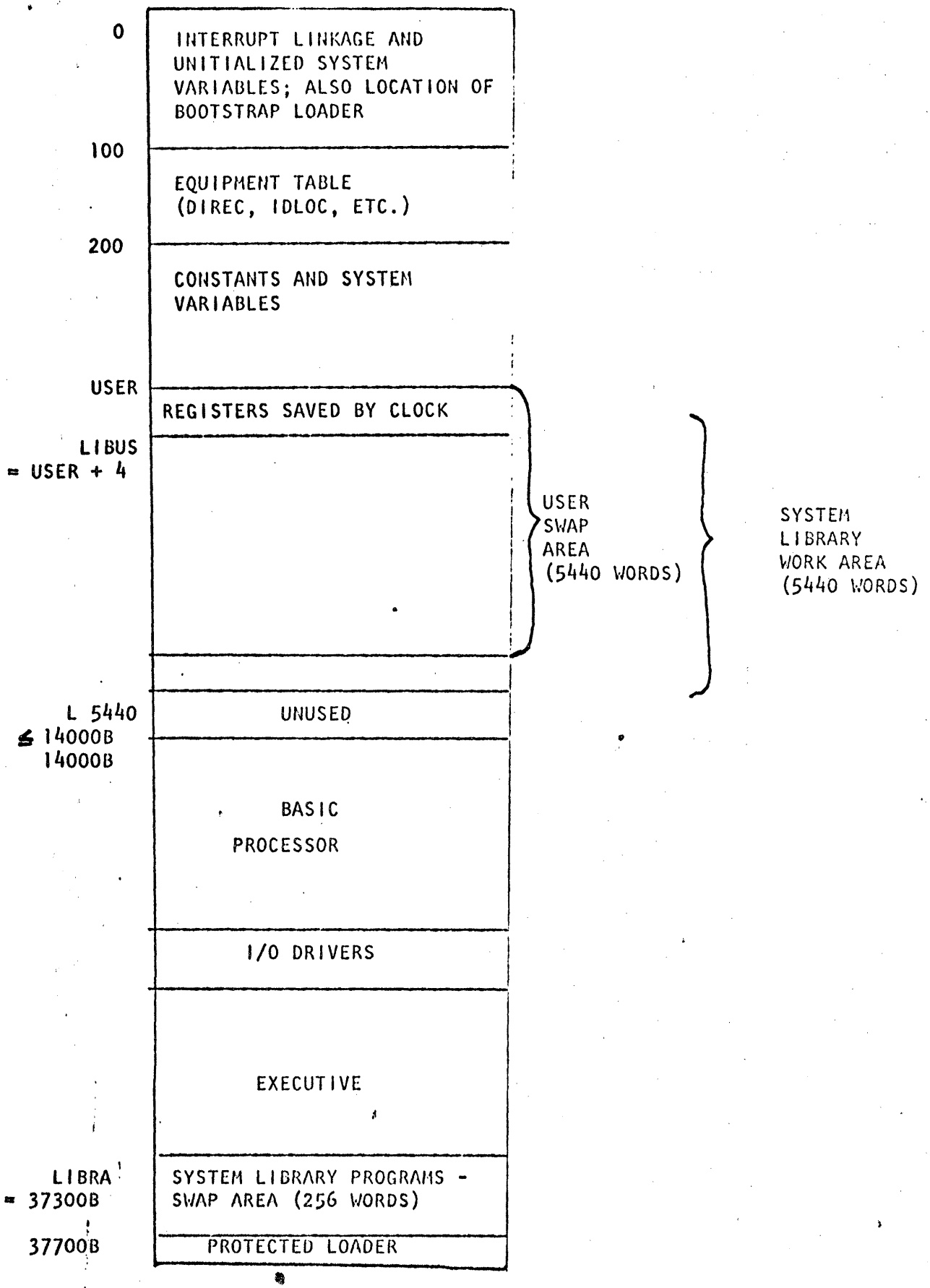
These symbols are primarily used for adjusting pointers to the table. For example, if the B register contains a pointer to the LINK entry of some user, the instruction

```
ADB   .+?  ID - ? LINK
```

will point B to his. ID entry.

. is a symbol located in base page at the 0 entry of a table of constants from -26 to + 49. A word containing the value N, where $-26 \leq .N \leq 49$ can be referenced by .+N.

CORE MAP


0
INTERRUPT LINKAGE AND UNITIALIZED SYSTEM VARIABLES; ALSO LOCATION OF BOOTSTRAP LOADER
100
EQUIPMENT TABLE (DIREC, IDLOC, ETC.)
200
CONSTANTS AND SYSTEM VARIABLES
USER
REGISTERS SAVED BY CLOCK
LIBUS = USER + 4
USER SWAP AREA (5440 WORDS)
SYSTEM LIBRARY WORK AREA (5440 WORDS)
L 5440 ≤ 14000B
14000B
UNUSED
BASIC PROCESSOR
I/O DRIVERS
EXECUTIVE
LIBRA = 37300B
SYSTEM LIBRARY PROGRAMS - SWAP AREA (256 WORDS)
37700B
PROTECTED LOADER

EQUIPMENT TABLE

The equipment table is the area of core which describes the resources available to the system. It resides at locations 100-166, as follows:

100-133:	DIREC	(discussed elsewhere)
134:	IDLOC	"
135:	IDLEN	"
136:	ADLOC	"
137:	ADLEN	"

140-157 TRAX - this is a table of which disc tracks are physically available to the system. Locations 140-143 correspond to disc 0, 144-147 to disc 1, etc. Track 0 of disc 0 is represented by bit 0 of 140, track 1 of disc 0 is represented by bit 1 of 140, etc. A bit is 0 when the track is available, 1 when unavailable.

When a system is initialized, all tracks of disc 0 are made available, all others unavailable. The TRAX table is changed only by the following commands:

DISC - causes all tracks of the specified disc to be made available.

LOCK - all specified tracks are made unavailable.

UNLOCK - all specified tracks are made available.

160-163: ?TBL - there is one word in this area for each of the four discs. Whe the word is zero, the particular disc does not exist. Otherwis bits 15:8 contain the number of sectors / track, bits 7:6 the disc prefix, and bits 5:0 the high priority select code. The prefix is used by the disc driver as the high order 2 bits of the 8-bit track address.

164: MAGSC - high priority select code for mag. tape; if nonexistent, MAGSC = 0.

165: PHSC - select code for autodisconnect board, if nonexistent, PHSC = 0.

166: PHR - 10 x number of seconds allowed for user to log on; applicable only if PHSC≠0.

DISC ORGANIZATION

The disc available to the system consists of from 64 to 256 tracks, depending upon how many discs exist. Each track contains from 90 to 128 sectors of 64 words each, for a total of 5760-8192 words per track. The loader assigns tracks as follows

RESIDENT SYSTEM	(3 tracks, including track 0)
System library routines	(2 tracks)
IDT and ADT	(1 track)
User swap tracks	(16 tracks)
Directory	(1-4 tracks)

All remaining tracks are available for storage of user programs and files. The ADT contains an entry for each available area.

The disc addresses of the individual system library routines are stored into the COMTABLE during loading. Although they are not all the same length, they are limited to 256 words, and so the system reads in exactly 256 words whenever it wants to load such a routine. The loader never assigns a library routine within 3 sectors of the end of a disc track, so that no errors can take place in doing this.

The IDT and ADT are stored on the same track, as described above.

Each directory track is stored beginning at sector 0.

User tracks are initialized to sector 0. The scheduler optimizes swapping, however, by writing user swap areas back at the first possible sector, but on the same track. The SLEEP routine rewrites all user tracks back to track origin (sector 0) so that the system will function correctly when it is reloaded from disc.

During running, each user track contains a copy of the area from core location USER through the core location specified by its ?PROG entry. This includes

all variable data which is relevant to that user's program, and his program itself. The location of various sections in his program is discussed elsewhere.

Programs and files are each required to be stored as contiguous blocks of disc. Since the disc is allocated by sectors, each program may cause part of its last sector to be wasted. When a program is stored (by the SAVE routine), it is first decompiled and is stored in that form. Only the encoded text is stored, so that a program may require as little as 3 words of disc space.

Files always occupy an integral number of sectors (1 - 128), each file occupying a contiguous area on the disc. BASIC does not treat the individual sectors in the same logical sequence as the physical sequence, but rather interleaves the sectors, as follows:

even # of sectors

Physical sequence:	1	2	3	4	...	2n-2	2n-1	2n
Logical sequence:	1	n+1	2	n+2	...	2n-1	n	2n

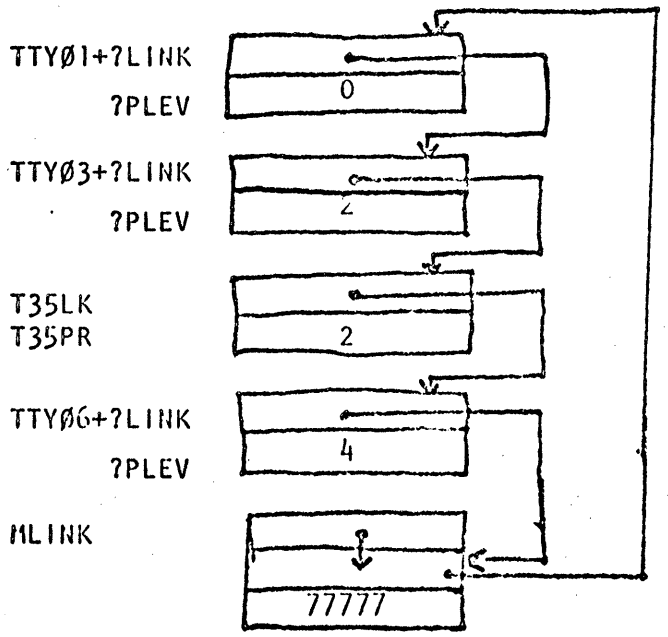
odd # of sectors

Physical sequence:	1	2	3	4	...	2n-2	2n-1
Logical sequence:	1	n+1	2	n+2	...	2n-1	n

This format tends to decrease disc seek time when sectors are accessed in a logically ascending order.

SCHEDULING

[The basic philosophy of the TSB scheduling algorithm is to provide short response times for short, interactive jobs at the possible cost of delays in longer running jobs. The implementation of this involves a queue of jobs to run which is ordered according to a priority scheme.] The queue is a linked list of from 1 to 18 entries, each entry pointing to the next entry, and the last entry pointing back to the first. The 18 possible entries in the queue are the 16 user LINK entries, a LINK word in a truncated TELETYPE table reserved for the system console, and a queue head. The queue head consists of the locations MLINK (0:2), and is always in the queue. The queue head has a priority of 77777_8, which is stored in location MLINK+2, and so it is always the last entry in the queue. As an example of how this works, assume that users 1, 3 and 6 are on the queue in that order and so is the system console, in a position between users 3 and 6. Then the queue will have the following appearance:



Since the MLINK entry is always the last entry on the queue, MLINK+1 is a pointer to the first entry, which in this case is TTYØ1. In the case of an empty queue, MLINK+1 will point to itself, i.e., CONTENTS(MLINK+1) = CONTENTS(MLINK). Each entry on the queue has a priority no greater in numerical value than that of the one it points to. When an entry is added to the queue, this ordering is always preserved by placing the new entry just ahead of the first entry with a larger priority number. Note that when the first entry in the queue has priority 0, it will remain at the head of the queue until it is removed from the queue entirely.

The following rules are used to assign (and reassign) priorities:

1. Upon first entering the queue, jobs are assigned priorities as follows:
   SYNTAX lines and jobs returning from I/O suspend: 0
   BASIC commands (RUN, LIST, PUNCH) : 1
   Commands for disc-resident routines (GET, BYE, etc): 2

2. Priorities of jobs are reassigned in the following way:
   Jobs of priority 2, when they reach the top of the queue, are reassigned priority 0.

   RUN jobs, when they exceed their time slice, are re-assigned priority 4, and repositioned in the queue according to that priority. Each RUN job is assigned a time slice of one second, and if it exhausts that it is assigned another.

The scheduler always chooses to run the job on top of the queue, so that whenever a job is running, MLINK + 1 is pointing to its link word. The two locations MAIN and LIB are control variables which tell what is presently in core. MAIN refers to one of the 16 user programs. It is a pointer to WORD 0 of the TTY table of the user program currently in core. If none is in core, MAIN = 0.

LIB points to the location in the COMTABLE of the disc address of the library routine in core. LIB = 0 when none is present.

The following conditions must exist for the scheduler to permit execution:

A) for Syntax and BASIC commands:
   MAIN set to point to correct user table

B) for disc resident commands:
   MAIN = 0
   LIB set to correct disc resident routines.

The Scheduler routine SWAPR is responsible for creating these conditions, and makes its decisions according to the values of MAIN, LIB, and the entry on top of the queue.

There are six system modules that communicate with each other in various ways: the disc driver, multiplexor (MPX) driver, system console driver, scheduler, BASIC, and system library routines (HELLO, BYE, KILLID, etc.).

1. Disc Driver.

Any section of the system may call the disc driver to perform a disc transfer. Three parameters are passed:

A = disc address	(bits	(15:14)	= disc number
	bits	(13:8)	= track number
	bit	7	= 0
	bits	(6:0)	= sector number)
B = core address	(bits	(14:0)	= core address
	bit	15	= 1 for disc input
			0 for disc output)

WORD = -# of words to be transferred (may be 0, in which case no actual transfer is performed).
Called by JSB DISC,1

It is the responsibility of the caller to insure that the disc is not busy when the call takes place. This is no hardship since while BASIC or a system library routine is running, no other module ever initiates disc transfers. As a result, the disc will appear to be busy only if the module itself has initiated the transfer.

Upon initiation of a disc transfer, the variable ENDSK is set to 1, and it is cleared upon completion. A complete transfer can be performed by:

```
JSB DISC,1
LDA ENDSK
SZA
JMP *-2
```

The system never suspends a program for a disc transfer because the high speed of the disc does not cause any great overhead.

The value of WORD is not modified by the driver.

II. Output to MPX

Output to the MPX driver is performed on a character by character basis via the routine OUTCH. The calling sequence is as follows:

A = character to be output (in bits (6:0); bits (15:7) may be anything)

B = address of WORD Ø of user's teletype table.

```
JSB OUTCH,I
```

The OUTCH routine places characters into the user's buffer until it is filled (99 characters), at which point the user is suspended by OUTCH. This is no problem for BASIC, but due to reentrancy problems must not be allowed by other modules. The buffer is always empty when a library routine is initiated, so they normally do not have to worry about it.

After the user has typed a carriage return, the MPX driver does not permit him to abort until one character is output. Therefore, those routines which do not wish to be terminated do not output anything until they are willing to be. Each routine must output at least one character (usually a line feed) to allow the user to type again.

III Input from MPX

Input from a user teletype is allowed only when he is in idle or input status, or when he is entering a program tape. Upon completion of input (CR), the MPX sets his MPCOM bit, and the scheduler, upon seeing the bit set, takes the appropriate action. BASIC can ask for input from a user terminal by performing

```
JSB  SCHIN,I
```

Although no other routine ever does this, the only possible problems involved would be those of reentrancy.

IV System Console Driver

The system console driver maintains two flags, T35F1 and T35F2, which determine its status. The meanings of these flags are as follows:

T35F1: = -1 during output, 0 otherwise
T35F2: Normally 0, it is set to -1 by the driver at the conclusion of input, and cleared to 0 externally. The combined values of these is more significant:

F1	F2	
0	0	driver is accepting input
0	-1	1) input command received and being processed, or 2) output terminated from a system command which is to be reinitiated
-1	0	outputting
-1	-1	outputting, at the end of which the current system command will be reinitiated.

When F2 = -1, the driver will not accept any input. This guarantees system library programs that they will not be interfered with. These routines are responsible for clearing F2 when they call the driver for the last time.

The calling sequence is:

A: bit 15 = 0 if CRLF is to be appended, bits (14:0) = # of chars.
B: Bit 15 = 1 if punching is to take place in addition to printing, bits (14:0) = core address of output buffer.

```
JSB TTY35,I
```

The driver uses the 36 word buffer T35BF as an input buffer. Most of the library routines use it for output, and occasionally for temporary storage between lines of output.

V. Input and Termination Requests

BASIC may obtain input from a user console by performing the instruction

```
JSB SCHIN,I
```

Upon return, the input will be in the users buffer as indicated by the pointers ?BHED and ?BEND.

Either BASIC or a system library routine terminates by:

```
JSB SCHEN,I
```

It is possible for BASIC to call a system library routine directly by executing:

```
JSB SCHLB,I
DEF <location in COMTABLE of disc address of program>
```

In the initial system, this is done only with the FILES routine. It is necessary that the library routine cooperate with BASIC, i.e., not any program can be so called.

FILES

The FILES routine is used by BASIC to process a FILES statement in a user's program. The function of the FILES routine is to translate the file names in the user's program into a table for use during execution. This table contains a 7-word entry for each file. Its format is:

1. physical length in sectors (BIT 15 = 1 if read only)
2. disc address of last logical sector
3. not set by FILES routine
4. disc address of first sector

5-7. not set by FILES routine

During operation of the FILES routine, the user's buffer is used as a table to store intermediate data. Three words of the buffer are used for each file. The operation is as follows:

1. Translate characters in FILES statement into the buffer table. Filenames are extended to six characters, if necessary, and those which are specified to be public files are marked by setting Bit 15 of their first word to 1. Possible errors found in this step are:
   a. file name of 0 or > 6 characters
   b. more than 8 files requested
2. Perform directory search for each file. Change the last two words of its entry in the buffer table to the disc address and length in sectors. The read-only bit is set if the file is a public file and the user is not A000. An error occurs if the file is nonexistent or protected. Update the date word in the directory entry for this file.

3. Test to make sure that there is sufficient room in the user area for the file table.

4. Scan the FUSS table to see if any other user has write capability on the files requested. Mark any such files as read-only. Copy the disc addresses of the requested files into the user's portion of FUSS. Indicate read-only files by marking bit 7 in FUSS.

5. Build the table specified above. FILTB is a pointer to the beginning of the table. Upon exit, VALTB and PBPTR both point to the first word following the table.

SAVE

The SAVE routine is called by a user to save a program in the library. Its operation is as follows:

1. Test for the existence of a program name and a non-null program.
2. If the user's program is in compiled form (CFLAG bit = 1), call DCMPL to put it into the form in which we will save it.
3. Test to see that the user has sufficient disc space allocated to save the program. The test to be satisfied is:

   (disc currently in use) + (length of program in sectors) $\leq$ (disc allowed).
4. Search the ADT for the first entry large enough to hold the program. Remember the address of the entry in SAVA.
5. Perform a directory search on the program to be saved. Fail if such an entry already exists.
6. If the directory track is full, call the SUPERSAVE routine to attempt to reallocate the directory. SUPERSAVE will perform step 7 itself and proceed to step 8.
7. Insert a new directory entry into the directory.
8. Update the IDT and ADT.
9. Copy the user's program to its library area.

SUPERSAVE

The SUPERSAVE routine is called by the SAVE and OPEN routines when they want to make a directory entry on a track that is already full. SUPERSAVE assumes that the following words are set properly:

(LTEMP:LTEMP+3) = first 4 words of entry.

(LTEMP+4) = pointer to DIREC entry for appropriate directory track

(LTEMP+5) = core address of entry which is to precede the new entry

(LTEMP+6) = disc address of entry

(LTEMP+7) = length of entry

Note that (LTEMP+4) and (LTEMP+5) are set correctly by DLOOK.

SUPERSAVE attempts to redistribute the directory tracks so that they will be as equal in length as possible. This will generally prevent it from being called very frequently. The operation is as follows:

1. Scan through DIREC and determine the total length of all directory tracks, and add 8 for the new entry. If all directory tracks are full, exit through failure location.
2. Divide total directory length by number of available disc tracks to determine their new individual lengths. Insert these in the table at (DEFNN+1:DEFNN+4) as negative.
3. Now redistribute the directory tracks. The basic idea of the algorithm is to fill the swap area with as much of the directory information as we can, reading from the beginning, and then to write out as much as we can, always making sure that when writing we don't overlay any portion that hasn't been read yet. The following variables are used:

   (SUP) K1 points to the DIREC entry for track being read (initially DIRECØ).

   L1 points to the DIREC entry for track being written (initially DIRECØ).

K2 = # of words read so far from track K1 (initially 0)

L2 = # of words written so far on track L1 (initially 0)

P = # of words in core (initially 0)

PP points to DEFNN entry, telling how many are to be written on L1.

TG = 1 if we have already inserted the new entry.

4. If L2 =-(PP), we have completely written track L1, so check for L1 = DIRD3. If it is, we've written all the tracks, so go to step 10. Otherwise, advance L1 to the next directory track advance PP, set L2 = 0, and repeat this step. If L2 ≠ -(PP), go to step 5.

5. If P ≥ 5432, we have read as much as we can, so go to step 7. If K1 = DIRD4, there is nothing left to read, so go to step 7. If K2 = # of words on track K1, we've read the entire track, so advance K1 to the next track, set K2 = 0, and repeat this step. Otherwise, compute the number of words we can read. If there is room to read the balance of the track, we will, otherwise we will read the maximum number of full sectors possible. If this is zero, go to step 7. If it is not zero, read from sector K2/64 into core location LIBUS + P. Add the number of words read to P and to K2.

6. If TG = 0, determine if we can insert the new entry. This will be so if K1 = (LTEMP + 4) and (LTEMP + 5) - LIBD < K2. If this is not the case, go back to step 5. Otherwise, set TG to 1 and insert the new entry in core. Set P to P + 8 and go back to step 5.

7. Write section. Set S = 0. This is the number of words written.

8. Compute number of words we can write on track L1. First set A = - number of words left to write on the track. If L1 = K1, we haven't finished reading everything from track L1, so if L2-A > K2 change A to L2-K2, which is the number of words we can write without destroying any unread directory information. If P-S < -A, we don't have as much in core as we are capable of writing, so set A = -(((S-P)÷64) x 64), an exact number of sectors.

9. If A = 0, we can't write anything, so if S ≠ D slide the remaining P-S words in core up to location LIBUS, set S = 0 and P = P-S. Then go back to step 4.
If A ≠ 0, write -A words to sector L2 ÷ 64 of track L1. If L2 = 0, set the first 4 words of the L1 DIREC entry to the first 4 words written. Set L2 to L2-A, S to S-A, and go back to step 8.

10. Set the new directory lengths into DIREC and go back to the calling program.

GET

The GET routine is called by a user to load a program from the library. The operation is as follows:

1. Translate name of program from user's input. If preceded by a $, set up for A000 search; otherwise set for searching on user's id.
2. Perform directory search. Print error if not found.
3. Fail if entry is a file (BIT 15 of word 2 of entry is 1). Check that the program will fit into the user area. This is necessary in case a program which was saved under an old version of the system can no longer fit with the current version.
4. Set the date into word 5 of the directory entry and write it back. Copy the program name into the user's table, and if this is a run-only program, set the run-only bit, unless the user is A000.
5. Scratch any previous user program, read in the basic portion of his user area, and then append the library program on. Set PBPTR correctly, set his CFLAG bit to 0, set SYMTB to 0, and exit.

APPEND

The APPEND routine is called by a user to append a library program onto his current program. The operation is the same as GET for steps 1-4, except that the old name is preserved, and then continues as follows:

5. Load user's current program and call DCMPL. Read in the program to be appended at the end of the current program.
6. If the current program is not null, search it for the sequence number of the last statement, and insist that it be smaller than the sequence number of the first statement of the appended program. If o.k, update PBPTR and exit.

HELLO

The HELLO command is used to log a user on to the system. Its operation is as follows:

1. If the current id is 0, there is no user to log off, so go to step 2. Otherwise, clear the user's section of FUSS, set the PHON entry in his table to (DATIM+1) + PHR, and set his PHT bit. This will force the user to be disconnected if he does not successfully log on.
2. Read the IDT. If there is no user to be logged off, go to step 3. Find the old user's IDT entry and update his total time used. Add an entry to LOGGR to be printed on the system console. Set the user's ID word to 0.
3. Translate the new idcode and search for it in the IDT. If not found, print an error message and terminate. Compare the password typed to the correct one, and fail if they disagree. Also, check that the time used to date is less than the time allowed.
4. Add a LOGON entry to LOGGR, and set the starting time into the user's table. Also insert the idcode, clear the name, clear the program and clear the PHT bit.
5. Search the directory for a public library HELLO program. If not found, or if it is a file, or if it won't fit in core, print READY and terminate.
6. Read in the fixed user area and append $HELLO. Set PBPTR, clear the user's CFLAG, set HFLAG, and clear SYMTB. Change the user's status to RUN, set TIMEF, and transfer to BASIC.

BYE

This command is used to log a user off. It operates as follows:

1. Set the user's PLEX bit to full duplex. If the user id is 0, set his disconnect bit, clear his PHT bit and terminate.
2. Clear the user's FUSS table and read in the IDT. Compute the time used and update his IDT entry. Create a LOGOFF entry in LOGGR. Clear the user's id entry and output a message. Set his PHON entry and set PHT so that he will be disconnected in 4 seconds (the time required to print the message). Then terminate.

KILL

The KILL routine is called by a user to delete a program or a file from the library. Files which are being accessed by another user are not allowed to be killed. The operation is as follows:

1. Translate the program or file name and perform a directory search. Fail if illegal name or the search fails.
2. If the entry is a file, search the FUSS table to see if any other user has access to the file. If so, print a message and terminate. If not, clear the user's section of FUSS.
3. Delete the entry from the directory and adjust DIREC. Subtract the program length from the user's IDT entry, and restore the space to the ADT.
4. If a file was killed, read the user's program in and decompile it. This guarantees that any old references to the file will disappear.

RENUMBER

The function of RENUMBER is to assign a new set of sequence numbers to a user program. The user may specify the sequence number of the first statement and the increment between statements. If unspecified, these are set to 10.

There are actually two sets of numbers that must be modified. One set is the sequence numbers themselves, each of which occupies the first word of its statement. The other is the set of references, which are labels in GO TO, GOSUB, RESTORE, and IF statements. Each of these also occupies one word. For programs in compiled mode, they are pointers to the statement they reference; in decompiled mode they are the actual statement number.

The primary technique used is to change all the references to absolute pointers (if in decompiled mode), then to change all the sequence numbers, and then (if in decompiled mode) to change the references to the new statement numbers. References to nonexistent labels are left unchanged.

Because the process of changing all the references to absolute pointers can become quite time consuming (due to the search that must be performed for each reference), a table is built in advance essentially dividing the program into 32 parts, each containing the same number of statements. For large programs with many references, this effectively cuts the time down by a factor of close to 32.

The subroutine RENSK is used to scan for references. It maintains two pointers, P and Q. Whenever it is called, it moves P to the next reference, and sets Q to point at the statement following the one that

P is pointing at. It takes advantage of the fact that any references within a statement are always the last word or words of the statement. Before calling RENSK for the first time, Q is set to point at the beginning of the program, and P is set to Q-1.

The operation of RENUMBER is as follows:

1. If null program, terminate immediately. Otherwise, read in user program.
2. Translate and check parameters M and N.
3. Scan through program and make sure that the new sequence numbers will not exceed 9999.
4. If program is in compiled mode, go to step 7. Otherwise, set up a table in ERSEC which divides the program into 32 parts. The result is that for each I from 0 to 31

   ERSEC [I] = sequence number of first statement in part I,

   ERSEC [I+32] = Absolute address of that statement

   If there are 32K + L statements ($0 \leq L < 31$) in the program, ERSEC [I] is the sequence number of statement

   $(K + 1)\, I + 1$, if $I < L$

   $KI + L + 1$, if $I \geq L$, $K > 0$

   L if $I \geq L$, $K = 0$

   Set Q = PBUFF, P = Q-1. (PBUFF points to the first statement).
5. Call RENSK to find the next statement reference. If there are none left, go to step 7. Find the largest I for which ERSEC [I] $\leq$ (RENP). If there is none, the statement referenced does not exist, so go to step 6. Otherwise, test all statements from (ERSEC [I + 32]) to either (ERSEC [I + 33]) or PBPTR,

depending upon whether $I < 31$ or $I = 31$. If found, set (RENP) to the location of the statement referred to, and repeat this step. Otherwise, go to step 6.

6. Set (RENP) = (RENP) + 100000_8 and go back to step 5.

7. Change the sequence numbers of all statements, according to the M and N parameters. If compiled mode, terminate. Otherwise, set Q = PBUFF, P = Q-1, and go to step 8.

8. Call RENSK to find the next statement reference. If none left, terminate. If (RENP) < 0, the reference was undefined, so set (RENP) = (RENP)-100000_8, and repeat this step. Otherwise, set RENP = ((RENP)) and repeat this step.

NAME

The NAME routine is called by a user when he wants to assign a name to his program. The program name is placed in his teletype table. The operation is as follows:

1. Get an input character. If a carriage return change it to a blank. If a control character, ignore it and repeat this step. If a "$", and this is the first character, print an error message and terminate.
2. Add the character to the user's name area. If < 6 characters, go back to step 1. Otherwise, restore the RUN-ONLY bit, and get one more character. If not a blank, print an error message. Then terminate.

CATALOG

The CATALOG routine prints a list of all programs and files in the user library. The operation is as follows:

1. Perform directory search on the program with all nulls. Get first directory entry following the one sought.
2. If the entry does not belong to this user, output a CRLF and terminate. Otherwise, output the 6 characters of the name one at a time, then a blank, then the 4 digits comprising the length of the program or file, and then another blank.
3. If < 6 names have been printed on the line, advance to the next directory entry and return to step 2. Otherwise, output a carriage return and suspend until the buffer is almost empty. Note that during step 2, the user's BHED word was set to point to the beginning of the last program name printed. This will insure that step 4 will work.
4. Read the name of the last program printed from the user's buffer and perform a directory search. The reason for doing this in this way rather than saving a pointer to the directory is that during the time CATALOG was suspended, the directory may have been changed in any way. Get the first directory entry following and go back to step 2.

LIBRARY

The LIBRARY routine prints a list of all programs and files in the public library. Its operation is identical to that of CATALOG except that A000 is used for directory searches instead of the user's id.

DELETE

The DELETE command allows a user to delete a section of his program. He can specify two parameters, M and N. M refers to the first line to be deleted, N to the last. If N is not specified, the entire program is deleted, starting at line M. The operation is as follows:

1. Translate and check parameters. If N is not specified, set it to 9999.
2. Decompile program.
3. Locate range of statements to be deleted.
4. Move portion of program following deleted area up against portion preceding.
5. Reset PBPTR and exit.

TIME

The TIME command prints the user's console time and total time. The operation is as follows:

1. Print "CONSOLE TIME ="
2. Read IDT.
3. Compute console time and print it.
4. Print "TOTAL TIME="
5. Find user's IDT entry. Add the time in there to the console time and print it.
6. Exit.

PROTECT

The PROTECT command allows user A000 to protect a program or file. Program protection means that no other user may list or save the program. File protection means that no other user may access the file. A000 files are always protected against other users writing on them. The operation is as follows:

1. Check for A000.
2. Translate and check the program or file name.
3. Perform a directory search on the specified program. Fail if not found.
4. Set the protect bit (BIT 15 of word 1 of the directory entry), write the directory back to the disc, and terminate.

UNPROTECT

This is identical to PROTECT except that it clears the protect bit.

OPEN

The OPEN command is used to open data files. The user must specify the filename and file length in sectors (1 to 128). The operation is as follows:

1. Translate and check the file name and length.
2. Check the IDT and ADT to see if a) the user has enough disc allocated to him to satisfy the command; and b) there is an area on the disc which is large enough to accommodate the file. Save the location of the ADT entry and its information, but don't update it until we know that there is room in the directory.
3. Perform a directory search on the file name. If found, this is a duplicate entry, so terminate. Otherwise, if the directory track is not full, insert the new entry. If it is full, call in SUPERSAVE to restructure the directory and insert the entry.
4. Update the IDT and ADT appropriately.
5. Initialize the file so that a -1 (end-of - file) is at the beginning of every sector. Write the file to the disc and then terminate.

LENGTH

The LENGTH command prints the length of the user's program, as it would be if saved. This is only the length of the source area of the program, and includes neither the fixed portion nor any of the tables used at run time. The length is determined in one of two ways:

1. if the user is in decompiled mode, length = PROG-PBUFF. PROG is just a copy of PBPTR, which points to the last word +1 of the program. PBUFF points to the first word.
2. if the user is in compiled mode, length = SPTR-PBUFF. It is necessary to read in the user's program to obtain SPTR.

ECHO

The ECHO command is used to control the computer echo of teletype input. Echoing is determined by the user's bit in the word PLEX. Bit = 0 implies no echo, 1 implies echo. The user will want echoing if and only if his teletype is full duplex. The command format is:

ECHO-ON for full duplex.

ECHO-OFF for half duplex.

REPORT

The REPORT command prints IDT information on the system console. From each IDT entry, the user id, time consumed, and disc consumed are printed. The entries are printed three per line. Note that the time printed on the console does not include any time for currently active users, since these are not added to the IDT until the user logs off. The operation of REPORT is as follows:

1. Print heading and suspend.
2. Read portion of IDT containing next three IDT entries.
3. Translate id, time, and disc of next three entries into output buffer. If less than three left, only do those.
4. Print and suspend if necessary, otherwise terminate.
5. Go back to step 2.

RESET

The RESET command modifies the time to date of a user's IDT entry. There are 3 cases:

a) all users set to zero;

b) one user set to zero;

c) one user set to specified amount.

The operation is as follows:

1. Read IDT.
2. Set ID = T = 0.
3. If no parameters, all users are to be set to zero, so go to step 5.
4. If no time specified, go to step 5. Otherwise, set T = specified time.
5. If ID = 0, clear word 5 of all IDT entries. Otherwise, locate specified id and set word 5 to T.
6. Write IDT back to disc and terminate.

CHANGEID

The CHANGEID command is used to modify any or all of the parameters in an IDT entry. The parameters that can be specified are: password, time allowed, disc allowed. The operation is as follows:

1. Translate id specified. Read IDT and locate the specified id. Fail if not found.
2. If password specified, insert into IDT entry. If followed by comma, go to step 3, otherwise to step 5.
3. If time specified, insert into entry. If followed by comma, go to step 4, otherwise to step 5.
4. Insert new disc value.
5. Write IDT back to disc and terminate.

DIRECTORY

The DIRECTORY routine prints a list of all directory entries. The entries are printed one per line. The items printed are: id, name, date, disc address, length. The operation is as follows:

1. Print heading and suspend.
2. Set up parameters for directory search for null program.
3. Perform directory search.
4. Get first directory entry following the one sought. If pseudo entry, terminate.
5. If id of entry is different from that of the preceding entry, place the ascii representation of the idcode in the output buffer. Otherwise, place blanks in the buffer. Save the idcode in location 35 of the buffer.
6. Convert the name, date, disc address, and length into the buffer.
7. Print line and suspend.
8. Set up parameters for directory search. These can be gotten from locations 35, 3,4, and 5 of the buffer. Go to step 3.

STATUS

The STATUS routine prints a summary of the various system resources. The only noteworthy thing about it is that the subroutine STAPR, which forces printing of a line and suspends, is only called from top level code. This is because any other subroutine entry points will be lost by overlays while the STATUS routine is suspended. STAPR fools T355P into thinking it was called from the location which STAPR was actually called from. The operation of STATUS is as follows:

1. Print IDLOC, IDLEN, ADLOC, ADLEN
2. Print disc addresses and lengths for each of the four directory tracks.
3. Search the ADT for the first five entries with length 0. These are the five system tracks. Print their disc addresses.
4. Print disc addresses of users 0-7.
5. Print disc addresses of users 8-15.
6. Print select codes for magtape, phones, and discs.
7. Print TRAX in 4 lines of 64 digits each.
8. Terminate.

SLEEP

The SLEEP command is used for system shutdown. It operates as follows:

1. Remove all users from the queue and make sure they can't get back by:
   a) clearing MPCOM,
   b) setting all status words to -2
   c) setting T35LK to point to MLINK+1
2. Output the sleep message to all active users, preceded and followed by a CRLF.
3. When all terminals are done outputting (IOTOG = -1), disconnect the telephones.
4. Update the IDT entry for each active user and create a logoff entry in LOGGR.
5. Clear FUSS to zeroes.
6. Set all user swap areas at track origin. This corresponds with the copy of the system that is on the disc.
7. Wait for the console to finish any output and then read the overlay.

The SLEEP overlay packs each library track so that the only unused area is at the end of the track. It also builds a table at TLTAB, which is of length 255. TLTAB [T] = - length of track T. This is used by the magtape dump routine. The operation is as follows:

8. Read in ADT. Set T = 1. T is the track number.
9. If track T is locked or is a system track, or has an ADT entry with length = that of the disc, there are no programs on it, so set TLTAB [T] = 0 and go to step 15.
10. Write the ADT back to disc. Set S = R = the disc address <T, 0>. Set P = Q = STAB. P and Q point to a table which will serve as

a subdirectory. Each program on track T will cause a two word entry to be created, the first of which is the old disc address of the program, and the second of which is the new disc address of the program following.

11. Search the directory for the next program on track T. If none left, go to step 12. Otherwise, set MEM[P] = old disc address of program, set disc address in directory entry to SLES, P=P+1, MEM[P] = SLES = SLES + length in sectors of program, P = P+1, and repeat this step.

12. Read in programs. If Q = P, we have read in all the programs, so go to step 13. Otherwise read in MEM [Q + 1] -R sectors from disc address MEM [Q] to core address LIBUS + sector (R) x 64, set Q = Q+1, R = MEM[Q], Q = Q + 1, and repeat this step.

13. Write R-T sectors to disc address T from core address LIBUS, set TLTAB [T] = 64 x T-R .

14. Read in the ADT, and replace all entries referring to track T by either no entries if the track is full, or by one entry with values R and # of sectors/track + T-R

15. Set T=T+1. If T < 256 go back to step 9.

16. Write the ADT back to the disc, write the equipment table (100-177) to track 0, sector 4, read in the dump routine, turn off all the IO and interrupt system, and jump to the dump.

NEWID

The NEWID routine adds an entry to the IDT. The operation is as follows:

1. If the IDT is at full capacity, print an error message and terminate.
2. Read in the IDT.
3. Translate the parameters.
4. Search the IDT for the specified id. Fail if found. Otherwise insert the new entry in its appropriate position, update IDLEN, write the IDT back to disc, and terminate.

KILLID

The KILLID routine removes a specified id from the system. The operation is as follows:

1. Get the id. If the id is A000, fail. This is because the files belonging to A000 may be accessed by other users, and removing them would be almost impossible.
2. Search the IDT for the specified id. If not found, terminate. Otherwise, delete the entry from the IDT and write it back to the disc.
3. If any user with the specified id is currently on the system, set the id item of his TTYTABLE to 0, set his status to -2 and his MPCOM bit to force him to be disconnected, and remove him from the queue if he is on it. Also, zero out his section of the FUSS table.
4. Load the overlay section. This section will remove from the directory any entries belonging to the user being killed, and will release the space occupied to the system.
5. Remove all directory entires belonging to this user, and build a table which will be used to patch the ADT. For each directory entry, two words are placed in the table, the disc address and length of the released area.
6. Update the ADT, using the patch table information.

UNLOCK

The UNLOCK command is used to restore disc tracks to the system. The operation is as follows:

1. Interpret parameters, setting F and L to the first and last tracks to be unlocked.
2. Scan the TRAX table to determine the number of tracks to be unlocked. Set CN to this number.
3. Set CN = min {CN, (5440 + IDLEN ∧ M64 + ADLEN)/2} . The parenthesized expression is the number of words that can be added to the ADT.
4. Read the ADT into core location LIBUS + 2 CN.
5. Set MOVED = LIBD, MOVES = LIBD + 2CN.
6. If track F is unlocked go to step 8. Otherwise, unlock it by clearing its bit in TRAX. If MOVED = MOVES, we can't insert an ADT entry, so go to step 8.
7. If MEM [MOVES] < F, move 2 words and repeat this step. Set MEM [MOVED] = F, MEM [MOVED + 1] = # Sectors/track, MOVED=MOVED + 2.
8. If F ≠ L, set F = F + 1 and go to step 6. Set ADLEN = ADLEN - 2CN. Write the ADT back to disc and terminate.

LOCK

The LOCK routine is used to tell the system that certain disc tracks are not to be used. Only tracks which are part of the program library are lockable, but tracks which contain active files are not. Any programs or files on tracks being locked are removed from the system. The operation is as follows:

1. Interpret the parameters and set F and L to the first and last tracks to be locked. Check that none of these tracks are mentioned in FUSS, is a directory track, id track, system track, or user track.
2. Delete from the ADT all entries with disc addresses on the tracks being locked.
3. For each track being locked, set its TRAX bit to 1.
4. Read in the LOCK overlay. The overlay will delete all directory entries for programs on the locked tracks, and also update the IDT appropriately. To do this, it maintains a table of IDT updates, each entry containing an id and a sector count, which is -# of sectors removed from that id.
5. Set ID = LIBUS - IDLEN, P = LIBUS + 5440, I = DIRDO. ID is a bound on the IDT, P a pointer to the update table, I a pointer to the DIREC entry for the directory being scanned.
6. If P < LIBUS - MEM [I] , we can't read the directory without clobbering the update table, so call LOCFX to remedy the situation.
7. Read the directory. Set MOVES = MOVED = LIBD, D = LIBD - MEM [I].
8. If MOVES = D, we're done with this directory, so go to step 9. Otherwise, if the entry pointed to by MOVES is not on a track being locked, perform an 8-word move and repeat this step. If it is on a

track being locked, we want to delete the entry. Set T = id of entry, T1 = - sector length of entry, MOVES = MOVES + 8. If P < LIBUS + 5440 and MEM [P] = T, set MEM [P + 1] = MEM [P + 1] + T1, and repeat this step. If P = D, set N = MOVED, perform a move of length D - MOVES, set D = MOVED, MOVED = MOVES = N. If P = ID, write out all words from LIBUS to D-1 to the directory track, call LOCFX, and read the stuff back in. Set P = P-2, MEM [P] = T, MEM [P + 1] = T1, and repeat this step.

9. Set MEM [I] = LIBD - MOVED, write out the new directory track, and update direc. If I ≠ DIRD3, set I = I + 7 and go back to step 7.

10. Call LOCFX and terminate.

The LOCFX subroutine is used by LOCK to update the IDT from the update table. It operates as follows:

1. If P = LIBUS + 5440 then exit immediately. Otherwise, read the IDT and set B = LIBUS - IDLEN.

2. Set B = B - 8. If MEM [B] ≠ MEM [P], repeat this step. Otherwise set MEM [B + 7] = MEM [B + 7] + MEM [P + 1], P = P + 2. If P ≠ LIBUS + 5440 repeat this step. Otherwise, write the IDT back to disc and exit.

PURGE

The PURGE routine is used to delete from the library all programs or files which have not been referenced since a certain date. The operation is as follows:

1. If HELLO program exists, assign it today's date. This is because the HELLO routine does not perform this function.
2. Interpret parameters and set DT to the purge date. Make sure that DT ≤ today's date.
3. Make sure that FUSS is empty. This is to avoid killing any active files.
4. Set ID = - max (LIBUS-IDLEN, LIBUS-ADLEN)-4. This is used to determine when the update table described below has reached the point when the updates must be made.
5. Set P = LIBUS + 5440, I = DIRDO. P is a pointer to the update table. Each entry in the update table contains 3 words:
   a) id
   b) disc address
   c) length in sectors
6. Read directory. If LIBUS-MEM [I]>P, the directory won't fit, so call PURFX to remedy the situation. Then read the directory. Set MOVED=MOVES-LIBUS, D = LIBUS-MEM [I].
7. Test next entry. If MOVES = D, we're done with this directory track, so go to step 11. If MEM [MOVES + 5] ≥ DT, we don't want to delete the entry, so perform an 8 word move and repeat this step.
8. Entry deletion. Set T = MEM [MOVES], T1 = MEM [MOVES + 6], T2 = (- MEM [MOVES + 7]+ 63) ÷ 64, MOVES = MOVES +8. If P-3 ≥ D, we have room for another update entry, so go to step 9. Otherwise, set N = MOVED, perform a move of D-MOVES words, set D = MOVED, MOVED = MOVES = N.

PURGE (contd)

9. If P + ID ≥ o, we can add a new update and still be able to load the IDT and ADT, so go to step 10. Otherwise, write LIBUS through D-1 to the disc, call PURFX, and read back LIBUS through D-1.

10. Make entry in update table. Set MEM [P-1] = T2, MEM [P-2] = T1, MEM [P-3]= T, P = P - 3, and go back to step 7.

11. End of directory track. Set MEM [I] = LIBUS-MOVED, update DIREC and write the directory back to the disc. If I ≠ DIRD3, set I = I + 7 and go to step 6. Otherwise, call PURFX once more and then terminate.

The PURFX routine is brought in as an overlay. It operates as follows:

1. Save MOVED and MOVES in M and M1.
2. Read the IDT, set B = LIBUS-IDLEN-8, set PP=P.
3. If PP=LIBUS+5440, write back the IDT, read in the ADT, and go to step 5.
4. Search for ID. If MEM [PP] ≠ MEM [B], set B = B-8 and repeat this step. Otherwise, set MEM [B + 7] = MEM [B + 7] - MEM [PP+2], set PP=PP+3, and go back to step 3.
5. Update ADT. If P = LIBUS + 5440, set MOVED = M, MOVES = M1, write the ADT back to disc, set ID = -max(LIBUS-IDLEN,LIBUS-ADLEN)-4, and exit. Otherwise, insert into the ADT the entry specified by MEM [P + 1] and MEM [P + 2], set P = P + 3, and repeat this step.

ROSTER

The ROSTER routine prints a listing of the id codes of all active users. These are obtained from the ID word in the 16 TTYTABLES. The absence of a user is indicated by the word being zero.

DISC

The DISC routine is used to add discs to the system or to remove discs. It operates as follows:

1. Interpret parameters.

2. If specified select code is 0, make sure all disc tracks are locked, set 0 into the TBL entry, and terminate.

3. Otherwise, make sure that the TBL entry is 0.

4. Search TBL and determine the lowest numbered prefix which has not been assigned to any other disc with the same select code. Create a TBL entry containing the number of sectors/track, the prefix, and the select code.

5. Unlock the disc tracks by clearing the four words in TRAX corresponding to the specified disc.

6. Insert one entry into the ADT for each disc track. If fewer than 64 entries can be made, some tracks will not be used.

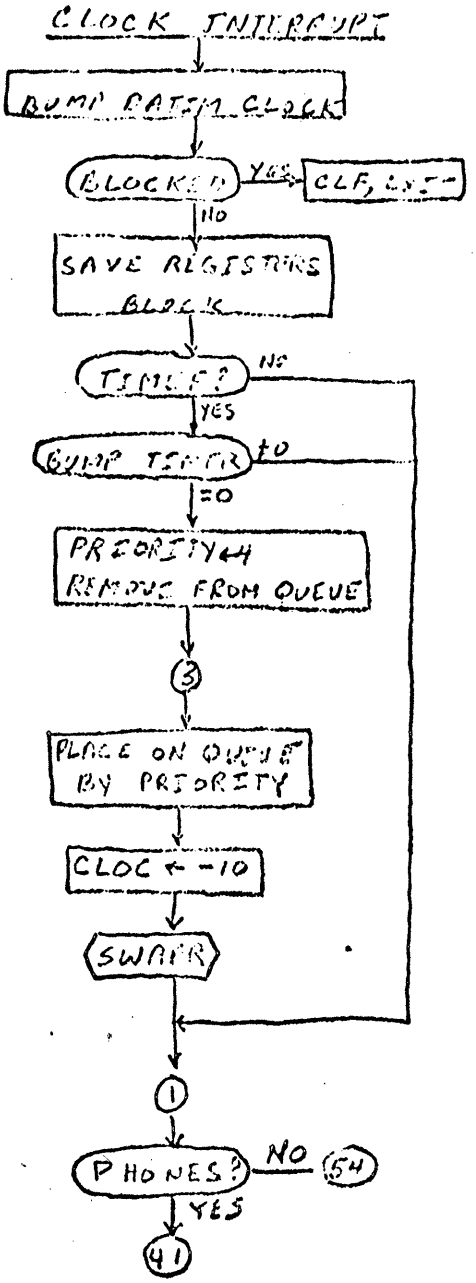
7. If there are more discs than directory tracks, allocate a new DIREC entry of length 0, and exit.

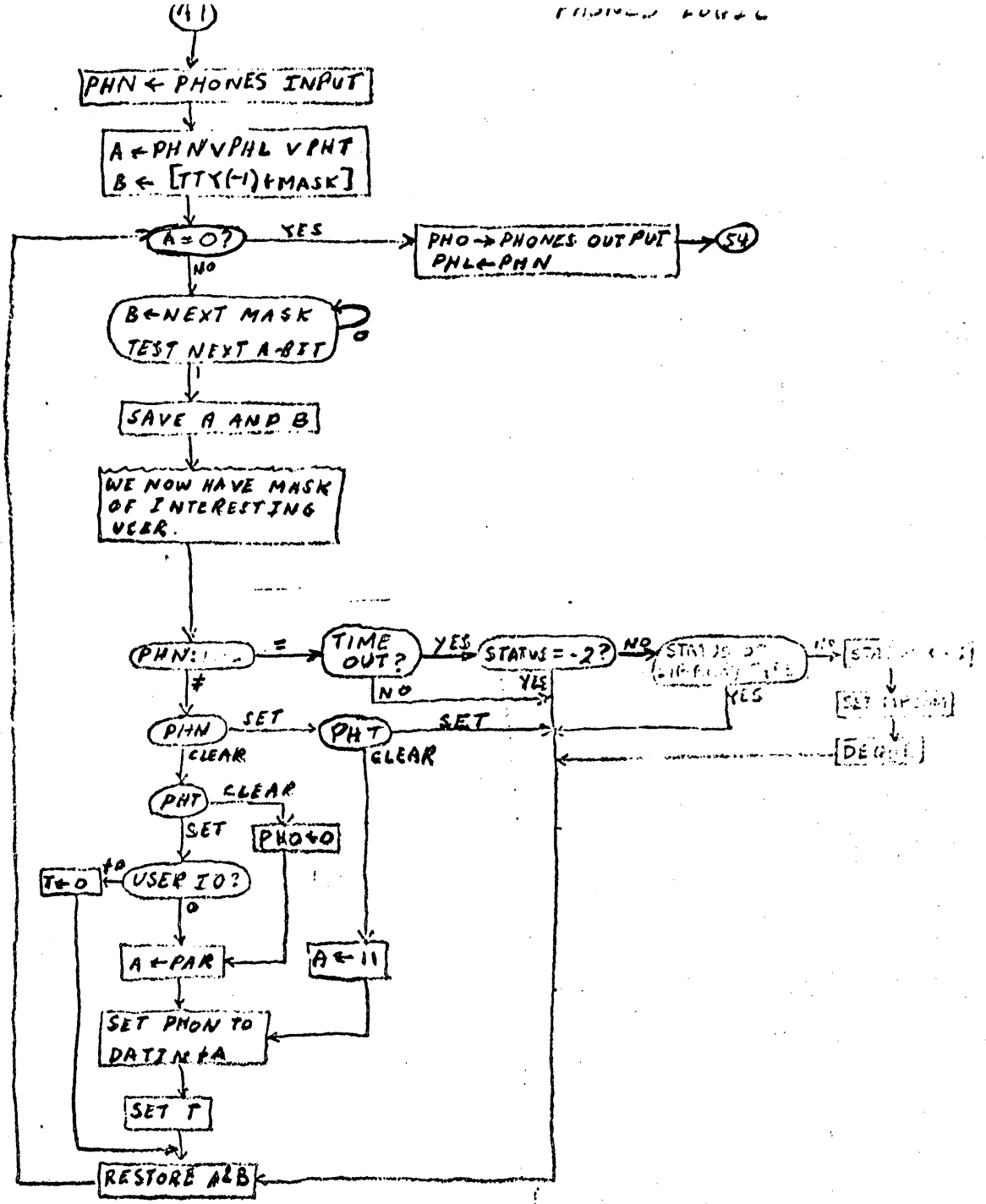
MAGTAPE

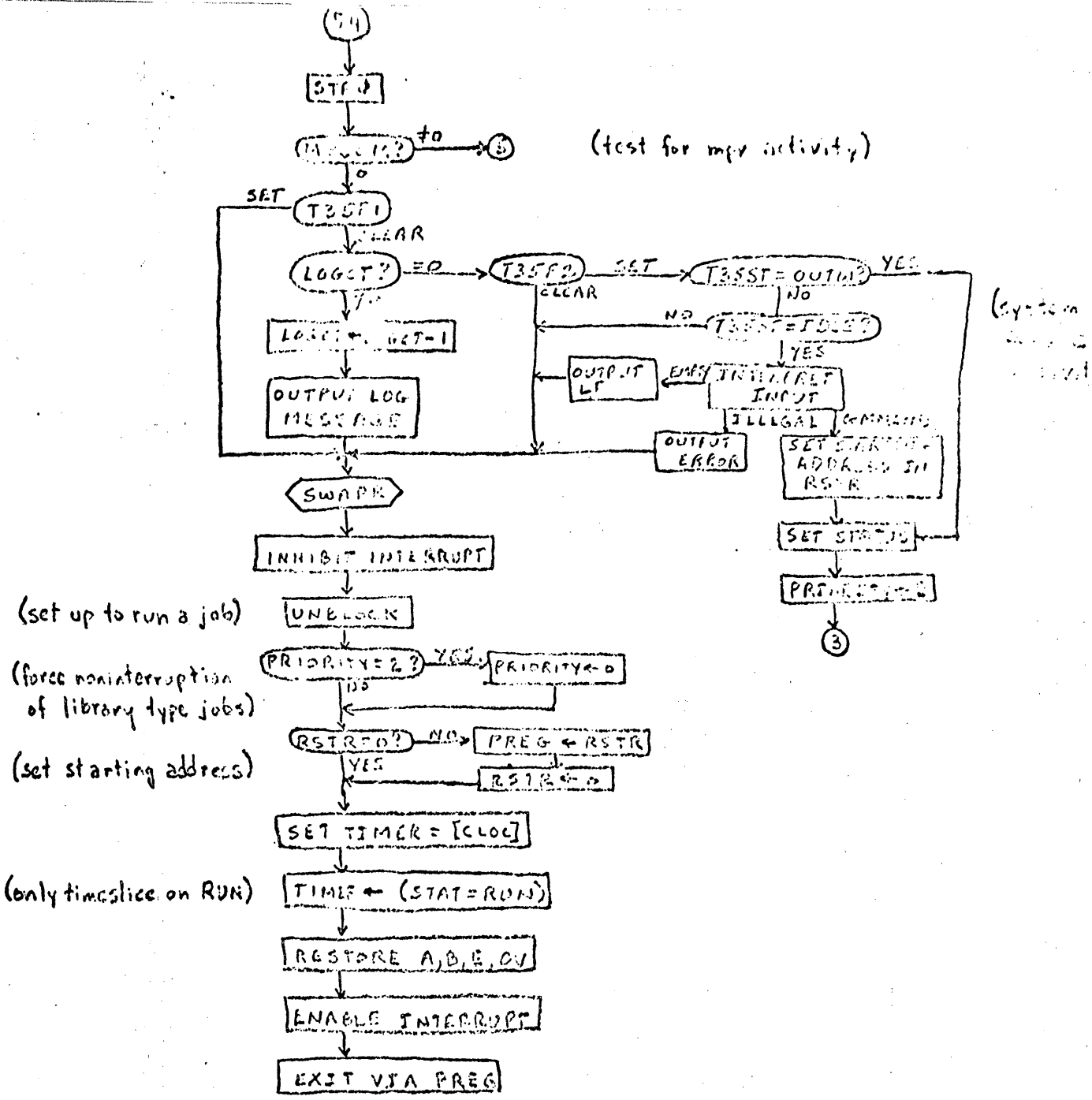
The MAGTAPE routine is used to set a select code into the location MAGSC.

PHONES

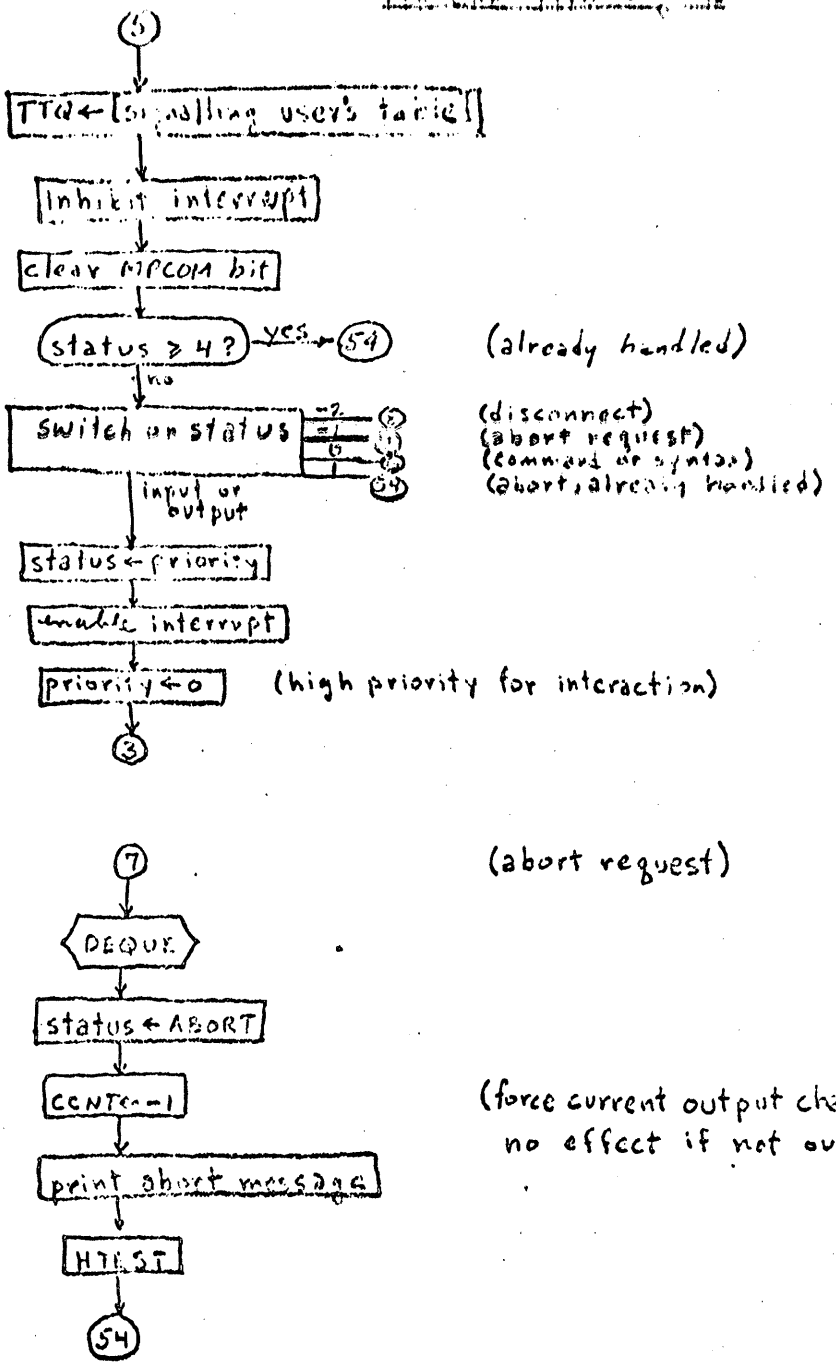
The PHONES command is used to specify the disconnect parameters. If the select code given is 0, it sets PHSC = 0 and forces the scheduler to skip around the disconnect logic. If the select code is non zero, it sets PHSC, constructs the LIA and OTA instructions in the phone logic, allows the scheduler to enter the phone logic, and initializes PHL to the current state of the disconnect input. If a logon time is specified, it multiplies it by 10 and sets it into PHR. If none is specified, it uses 120 seconds. The specified time can be no greater than 323 in order that the various checking algorithms work properly.

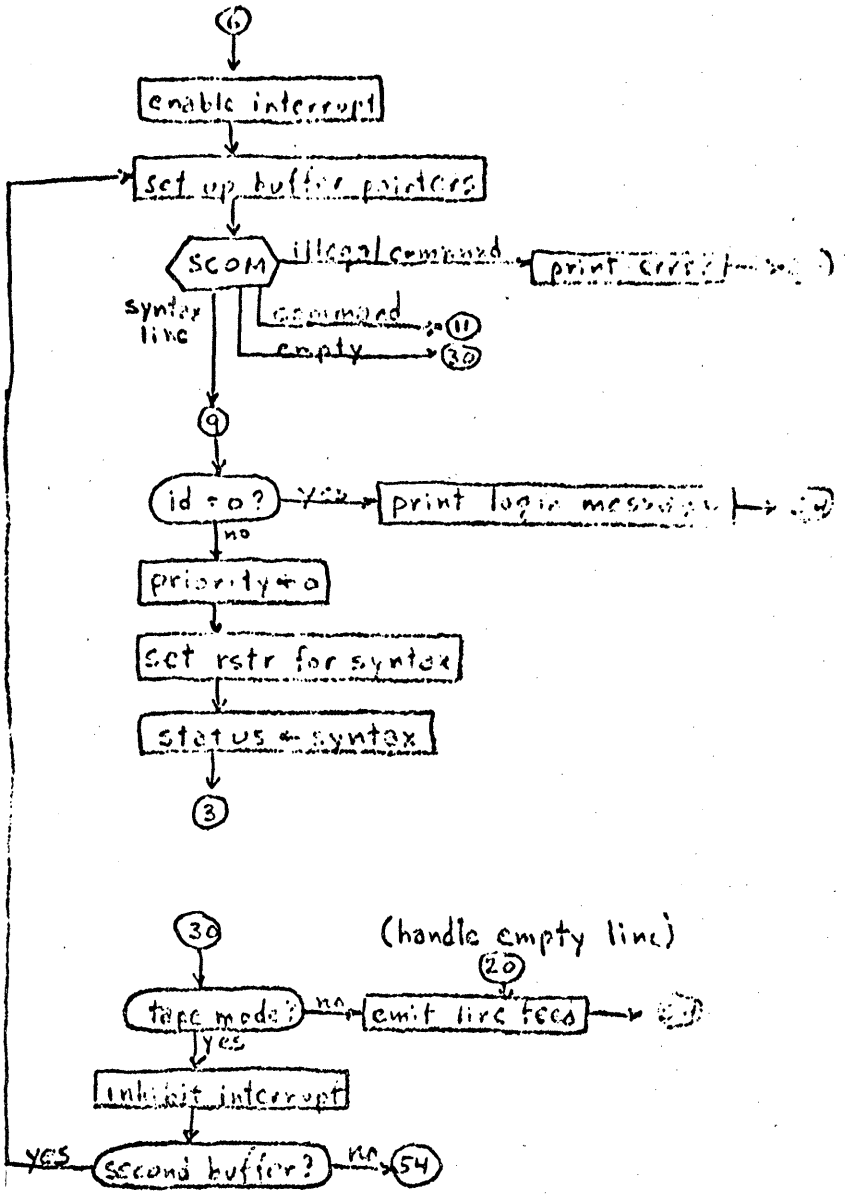

CLOCK INTERRUPT
BUMP DATE/TIME CLOCK
BLOCKED
YES
CLR, EXIT
NO
SAVE REGISTERS
BLOCK
TIMER?
NO
YES
BUMP TIMER
TO
=0
PRIORITY←4
REMOVE FROM QUEUE
(test for end of quantum)
(move to end of queue)
3
PLACE ON QUEUE
BY PRIORITY
CLOC ← -10
(set timing to 1 sec)
SWAPR
1
PHONES?
NO
54
YES
41


(41)
PHN ← PHONES INPUT
A ← PHN∨PHL∨PHT
B ← [TTY(-1)∧MASK]
A=0?
YES
NO
PHO → PHONES OUTPUT
PHL ← PHN
54
B ← NEXT MASK
TEST NEXT A-BIT
0
1
SAVE A AND B
WE NOW HAVE MASK OF INTERESTING USER.
PHN:
=
≠
TIME OUT?
YES
NO
STATUS = -2?
NO
YES
SET
CLEAR
PHN
PHT
PHT
CLEAR
SET
PHO←0
USER IO?
T←0
0
A ← PAR
A ← 11
SET PHON TO DATIN∧A
SET T
RESTORE A&B

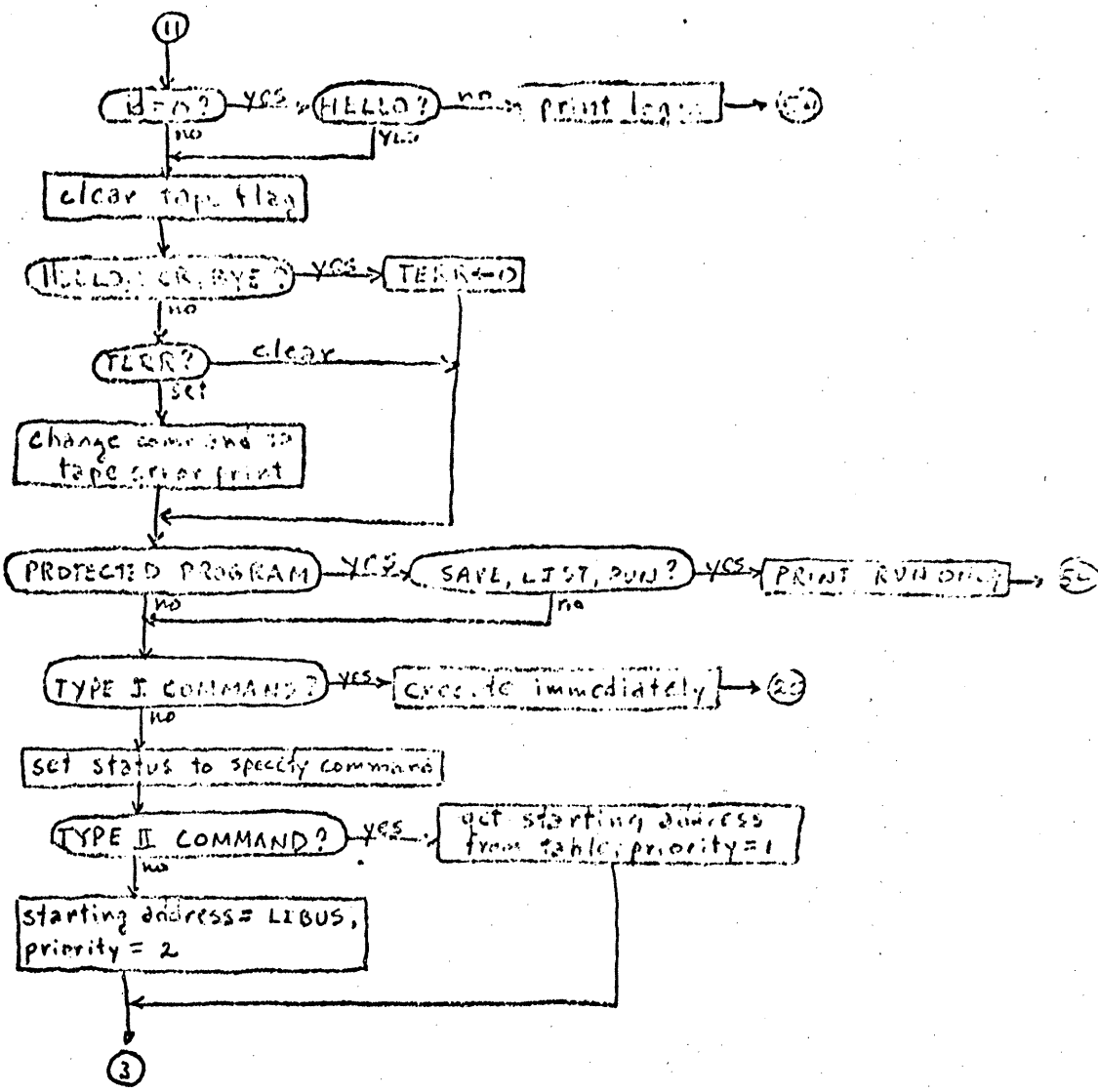


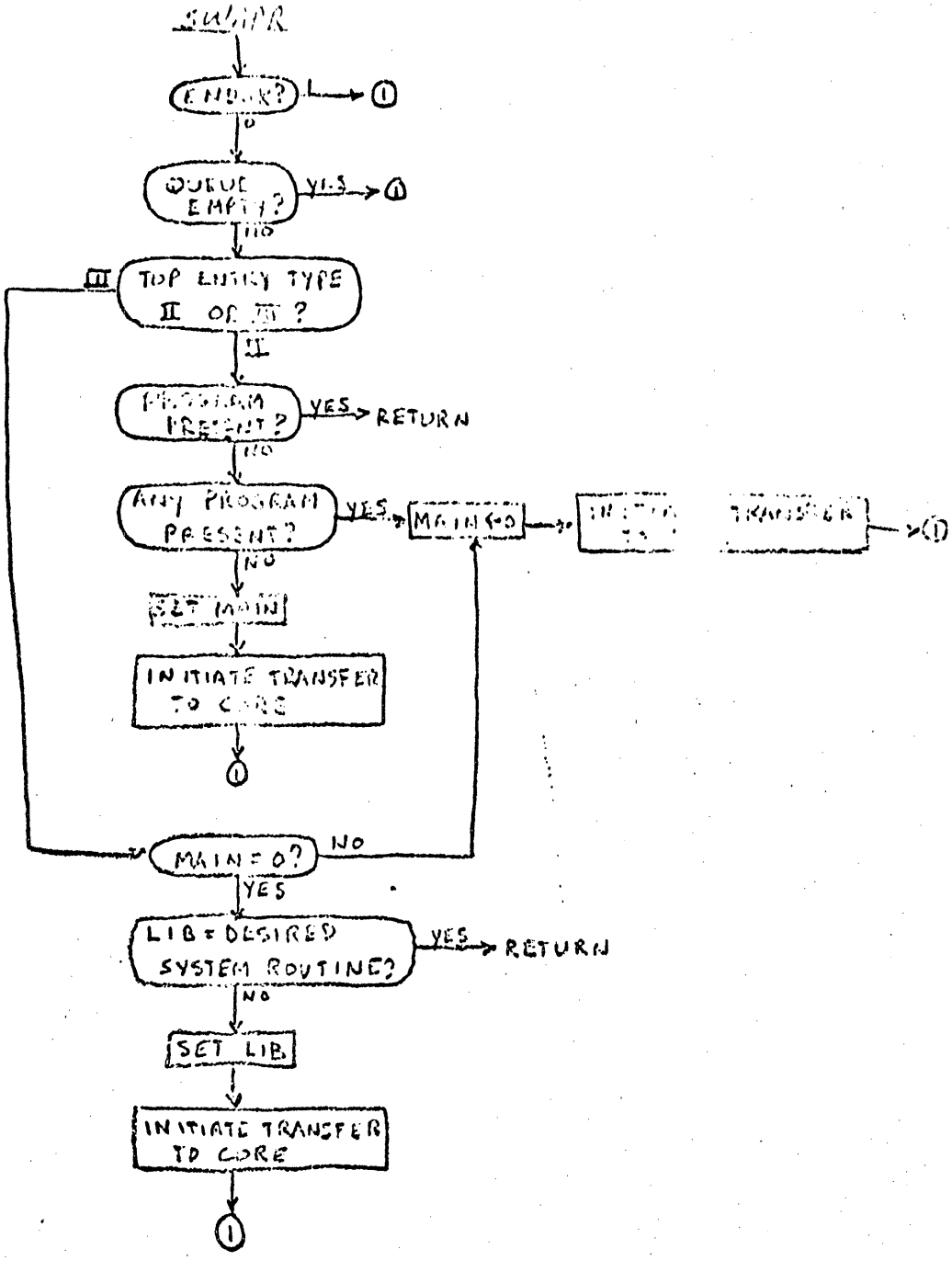
(test for mpr activity)
SET
CLEAR
LOGCT?
=0
T35F2
SET
CLEAR
T35ST = OUTW?
YES
NO
NO
YES
OUTPUT LOG MESSAGE
OUTPUT LF
EMPTY
INTERPRET INPUT
ILLEGAL
OUTPUT ERROR
SWAPR
INHIBIT INTERRUPT
(set up to run a job)
UNBLOCK
(force noninterruption of library type jobs)
PRIORITY = 2 ?
YES
PRIORITY ← 0
NO
(set starting address)
RSTR = 0?
NO
PREG ← RSTR
YES
RSTR ← 0
SET TIMER = [CLOC]
(only timeslice on RUN)
TIMER ← (STAT = RUN)
RESTORE A,B,E,OV
ENABLE INTERRUPT
EXIT VIA PREG

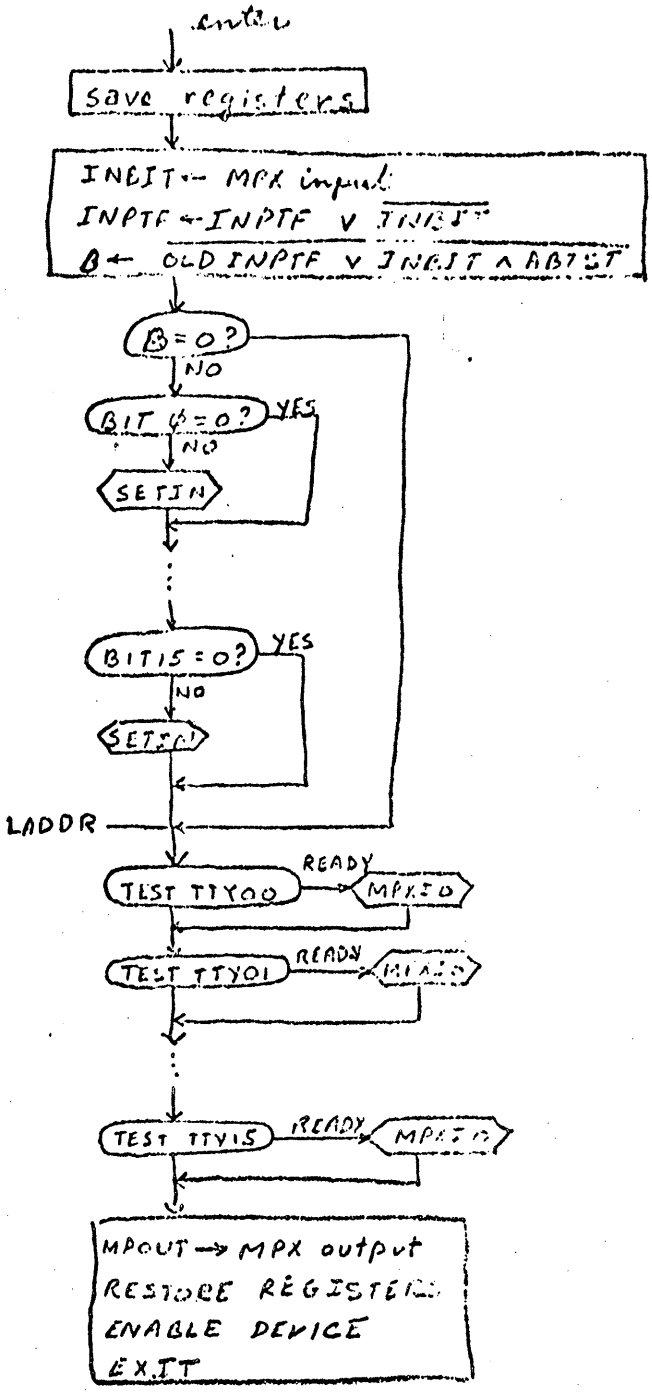

5
TTA ← [signalling user's table]
inhibit interrupt
clear MPCOM bit
status ≥ 4 ?
yes
54
(already handled)
no
switch on status
-2
-1
0
7
(disconnect)
3
(abort request)
(command or syntax)
54
(abort, already handled)
input or output
status ← priority
enable interrupt
priority ← 0
(high priority for interaction)
3
7
(abort request)
DEQUE
status ← ABORT
CCNT ← -1
(force current output char. to be the last, no effect if not outputting)
print abort message
HTEST
54

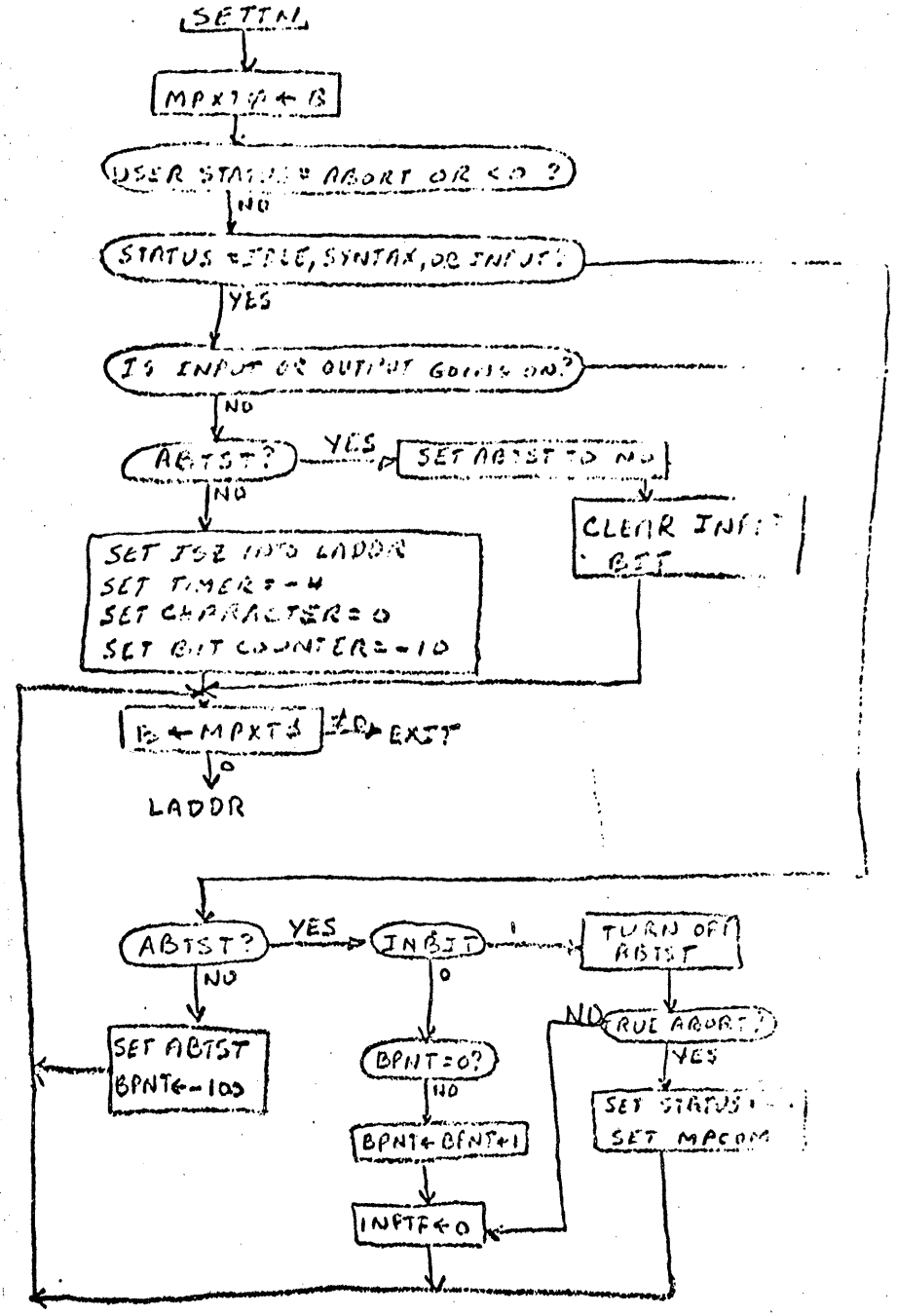

6
enable interrupt
set up buffer pointers
scom
illegal command
print error
syntax line
command
empty
11
30
9
id = 0?
yes
print login message
no
priority = 0
set rstr for syntax
status = syntax
3
30
(handle empty line)
20
tape mode?
no
emit line feed
yes
inhibit interrupt
yes
second buffer?
no
54

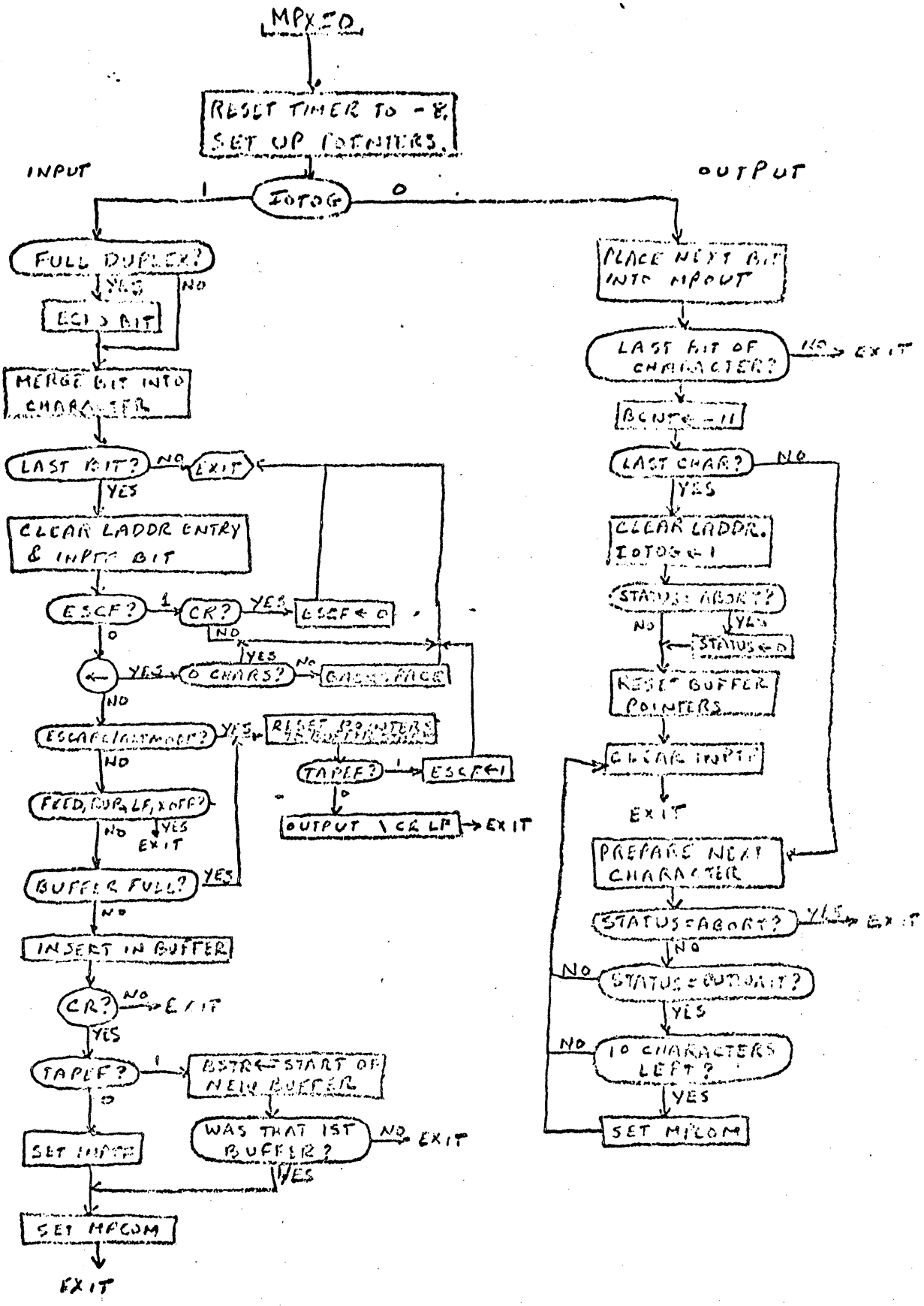
command input

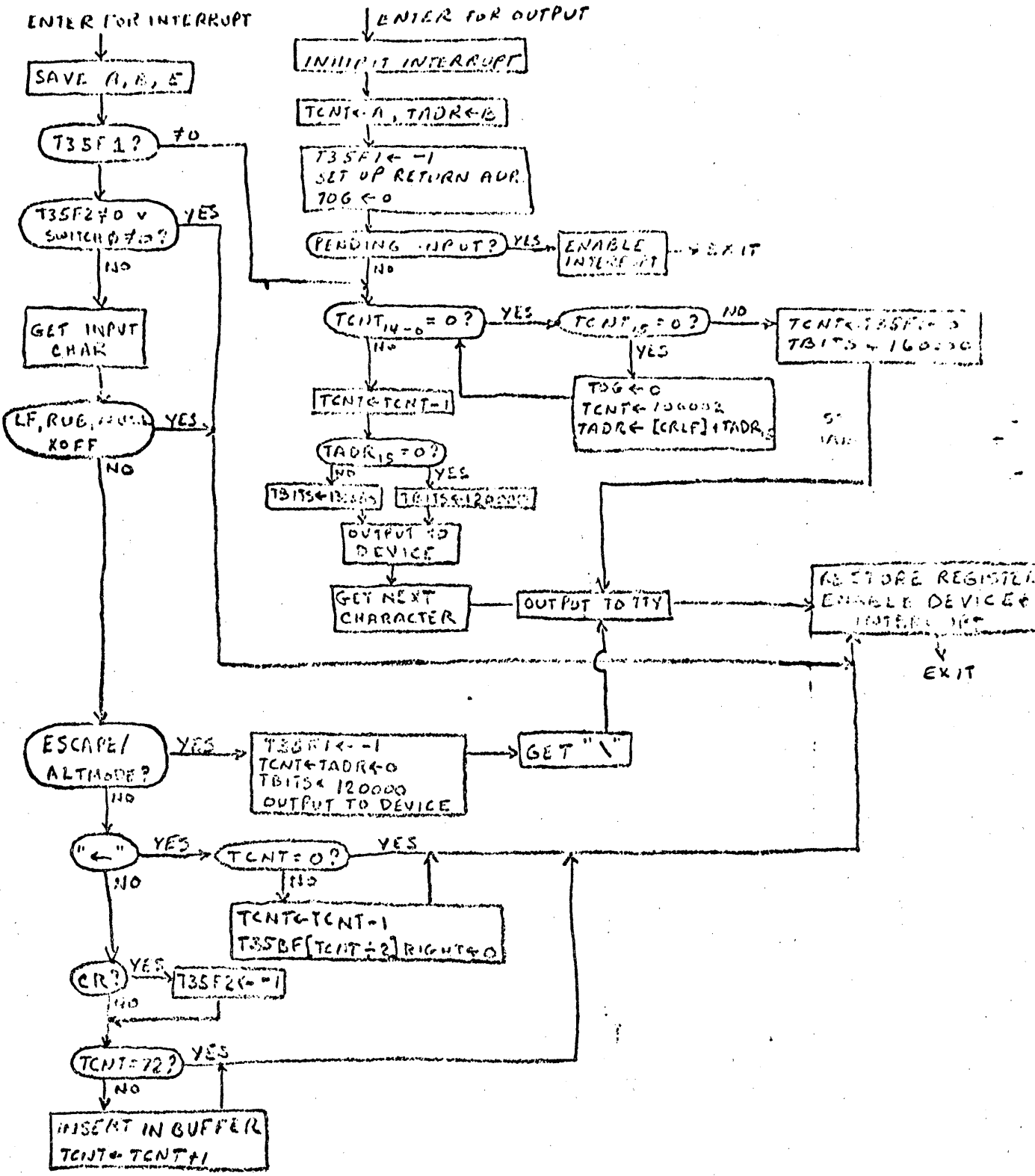


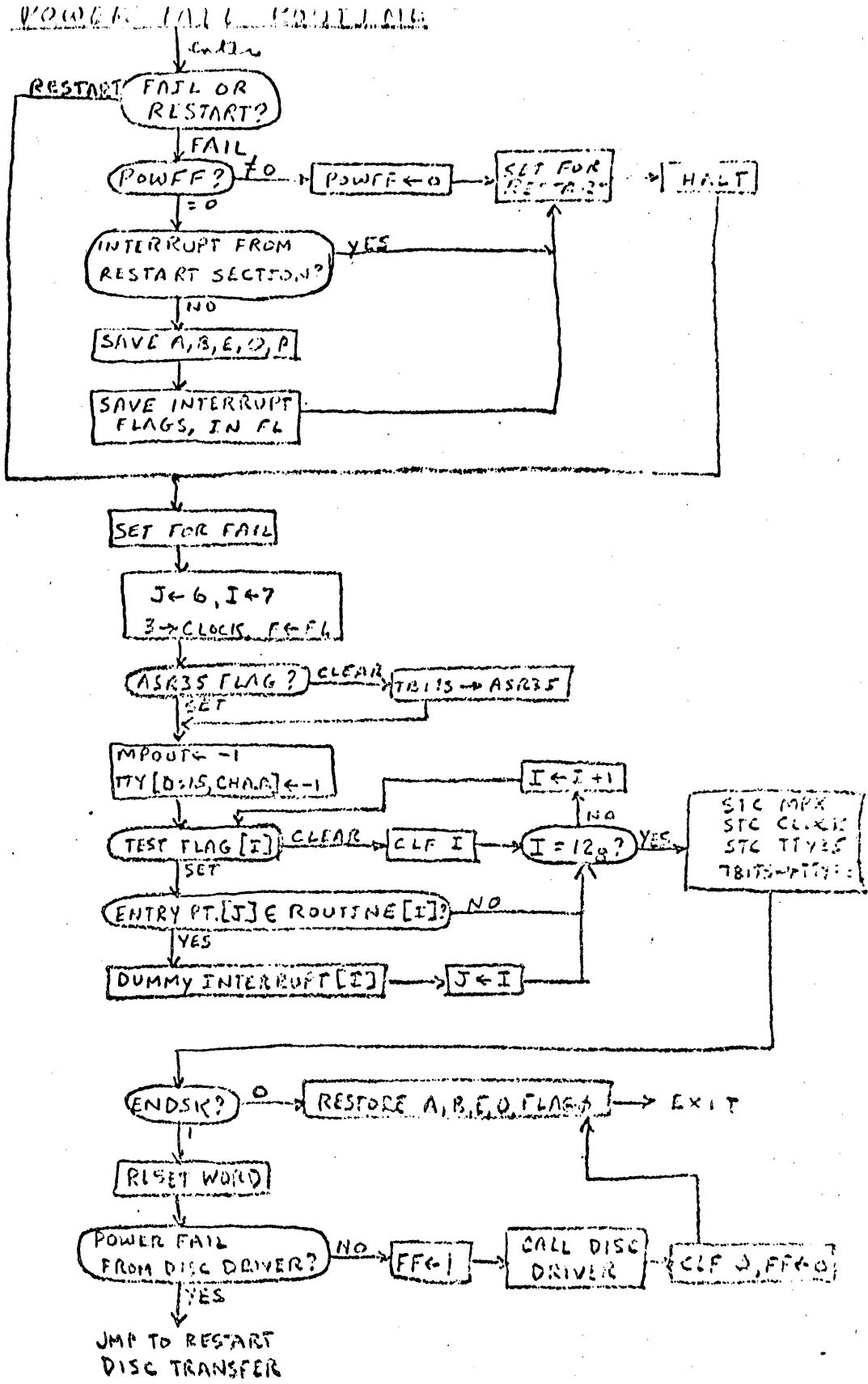

SWAPR
ENDOK?
1
0
QUEUE EMPTY?
YES
1
NO
III
TOP ENTRY TYPE II OR III ?
II
PROGRAM PRESENT?
YES
RETURN
NO
ANY PROGRAM PRESENT?
YES
MAIN ← 0
TRANSFER
1
NO
SET MAIN
INITIATE TRANSFER TO CORE
1
MAIN = 0?
NO
YES
LIB = DESIRED SYSTEM ROUTINE?
YES
RETURN
NO
SET LIB
INITIATE TRANSFER TO CORE
1

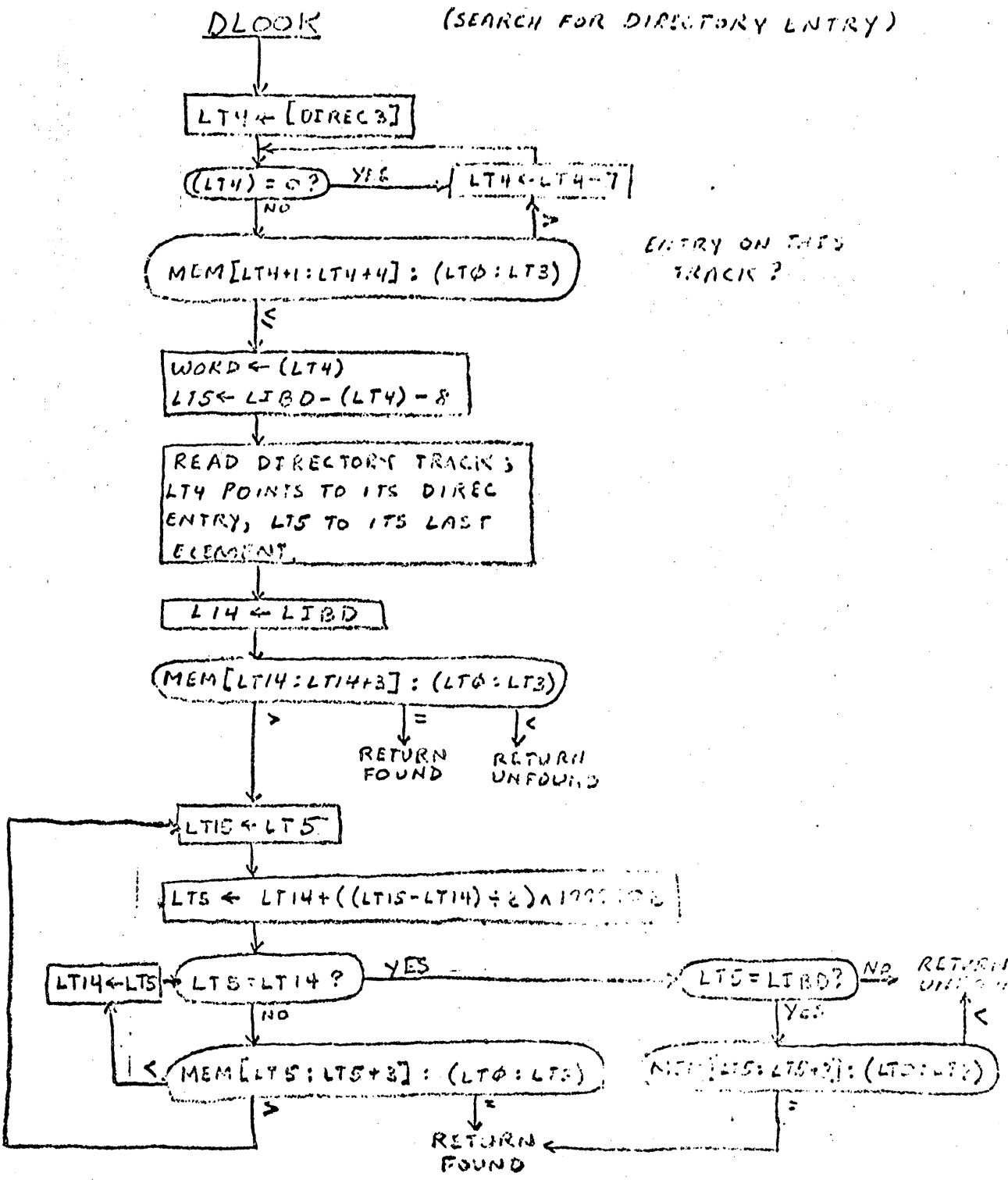

enter
save registers
INBIT ← MPX input
INPTF ← INPTF ∨ INBIT
B ← OLD INPTF ∨ INBIT ∧ ABTST
B = 0?
NO
BIT 0 = 0?
YES
NO
SETIN
BIT 15 = 0?
YES
NO
SETIN
LADDR
TEST TTY00
READY
MPXID
TEST TTY01
READY
TEST TTY15
READY
MPOUT → MPX output
RESTORE REGISTERS
ENABLE DEVICE
EXIT


SETTNL
MPXT# ← B
USER STATUS = ABORT OR <0 ?
NO
STATUS = IDLE, SYNTAX, OR INPUT
YES
IS INPUT OR OUTPUT GOING ON?
NO
ABTST?
YES
SET ABTST TO NO
NO
CLEAR INPUT BIT
SET ISR INTO LADDR
SET TIMER = -4
SET CHARACTER = 0
SET BIT COUNTER = -10
B ← MPXT#
≠0
EXIT
0
LADDR
ABTST?
YES
INBIT
1
TURN OFF ABTST
NO
0
SET ABTST
BPNT ← -10
BPNT=0?
NO
TRUE ABORT?
NO
YES
BPNT ← BPNT+1
INPTF ← 0
SET STATUS
SET MPCOM

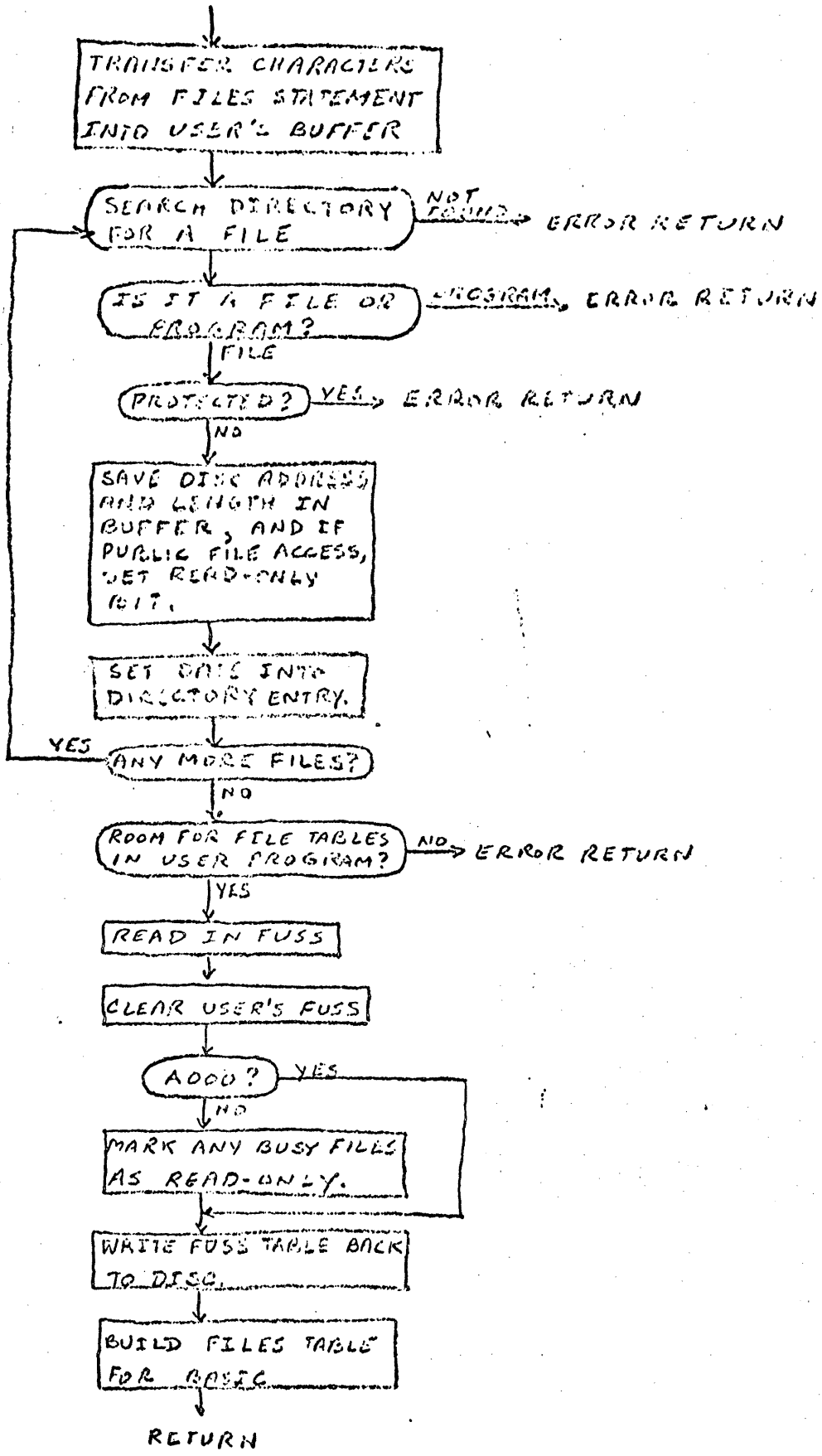

MPXFD
RESET TIMER TO -8.
SET UP POINTERS.
INPUT
OUTPUT
IOTOG
1
0
FULL BUFFER?
YES
NO
ECI > BIT
MERGE BIT INTO CHARACTER
LAST BIT?
NO
EXIT
YES
CLEAR LADDR ENTRY & INPUT BIT
ESCF?
1
CR?
YES
ESCF ← 0
NO
0
YES
0 CHARS?
NO
YES
NO
ESCAPE/LASTMODE?
YES
RESET POINTERS
NO
TAPEF?
1
ESCF ← 1
0
FEED, OUR, LF, XOFF?
NO
YES
EXIT
OUTPUT \ CR LF
EXIT
BUFFER FULL?
YES
NO
INSERT IN BUFFER
CR?
NO
EXIT
YES
TAPEF?
1
BSTR ← START OF NEW BUFFER
0
WAS THAT 1ST BUFFER?
NO
EXIT
YES
SET INPUT
SET MPCOM
EXIT
PLACE NEXT BIT INTO MPOUT
LAST BIT OF CHARACTER?
NO
EXIT
BCNT ← -11
LAST CHAR?
NO
YES
CLEAR LADDR.
IOTOG ← 1
STATUS = ABORT?
NO
YES
STATUS ← 0
RESET BUFFER POINTERS
CLEAR INPUT
EXIT
PREPARE NEXT CHARACTER
STATUS = ABORT?
YES
EXIT
NO
NO
STATUS = OUTPUT?
YES
NO
10 CHARACTERS LEFT?
YES
SET MPCOM

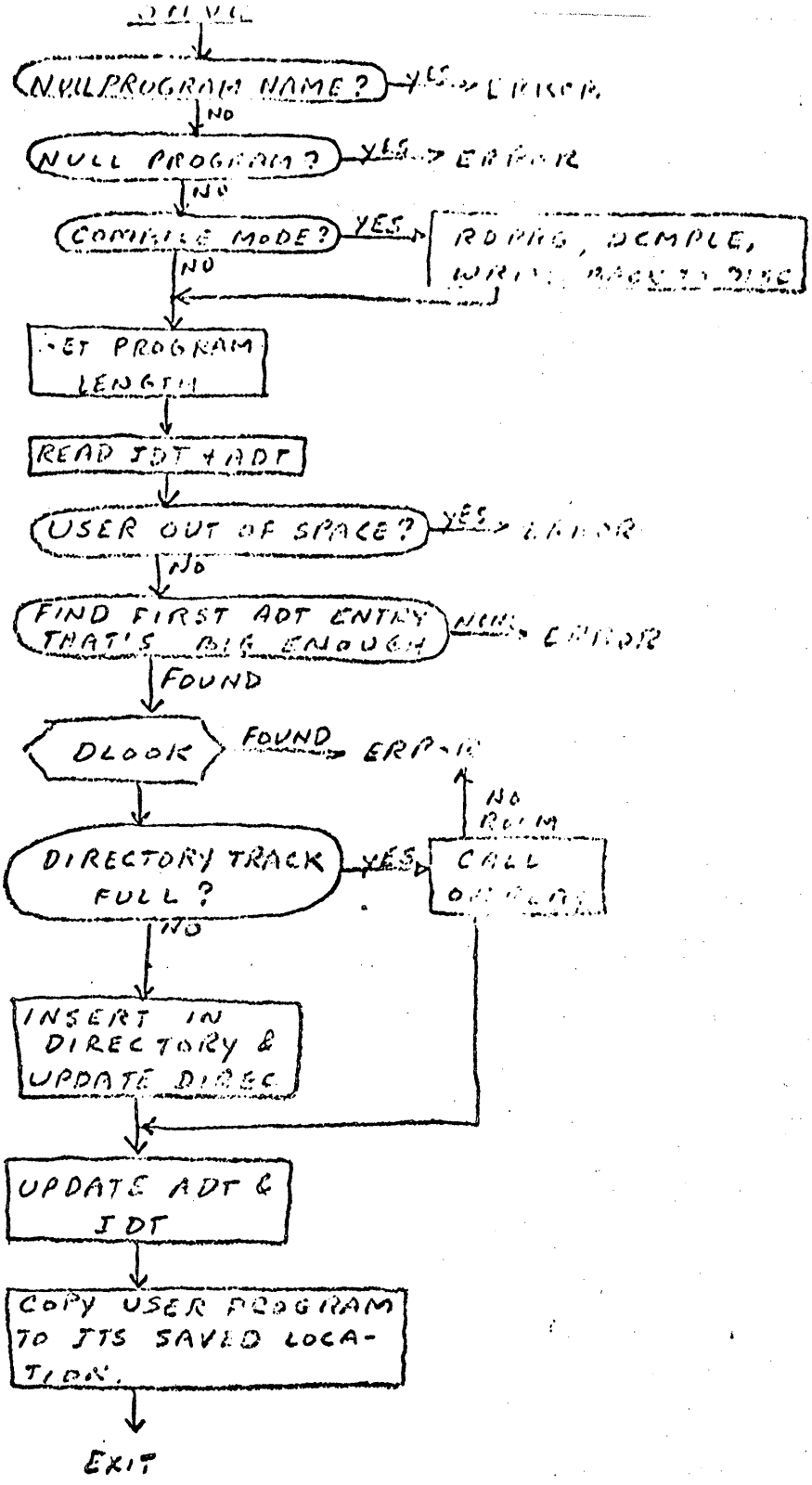

ENTER FOR INTERRUPT
ENTER FOR OUTPUT
SAVE A, B, E
INHIBIT INTERRUPT
TCNT←A, TADR←B
T35F1?
TO
T35F1←-1
SET UP RETURN ADR
TOG←0
T35F2≠0 v SWITCH 0≠0?
YES
PENDING INPUT?
YES
ENABLE INTERRUPT
EXIT
NO
GET INPUT CHAR
TCNT14-0 = 0?
YES
TCNT15 = 0?
NO
YES
TCNT←T35F1←-1
TBITS←160000
TOG←0
TCNT←100002
TADR←[CRLF]+TADR
TCNT←TCNT-1
TADR15=0?
NO
YES
OUTPUT TO DEVICE
GET NEXT CHARACTER
OUTPUT TO TTY
RESTORE REGISTERS
ENABLE DEVICE INTERRUPT
EXIT
LF, RUB, XOFF
YES
NO
ESCAPE/ ALTMODE?
YES
T35F1←-1
TCNT←TADR←0
TBITS←120000
OUTPUT TO DEVICE
GET "\"
"←"
YES
TCNT=0?
YES
NO
NO
TCNT←TCNT-1
T35BF[TCNT÷2] RIGHT←0
CR?
YES
T35F2←-1
NO
TCNT=72?
YES
NO
INSERT IN BUFFER
TCNT←TCNT+1


POWER FAIL / RESTART
CALL
RESTART
FAIL OR RESTART?
FAIL
POWFF?
≠0
POWFF←0
SET FOR RESTART
HALT
=0
INTERRUPT FROM RESTART SECTION?
YES
NO
SAVE A,B,E,O,P
SAVE INTERRUPT FLAGS, IN FL
SET FOR FAIL
J←6, I←7
3→CLOCKS, FF←FL
ASR33 FLAG?
CLEAR
TBITS→ASR33
SET
MPOUT←-1
TTY[0:15,CHAR]←-1
I←I+1
TEST FLAG[I]
CLEAR
CLF I
I=12₈?
NO
YES
STC MPX
STC CLOCK
STC TTY33
TBITS→TTY33
SET
ENTRY PT.[J] ∈ ROUTINE[I]?
NO
YES
DUMMY INTERRUPT[I]
J←I
ENDSK?
0
RESTORE A,B,E,O,FLAGS
EXIT
1
RESET WORD
POWER FAIL FROM DISC DRIVER?
NO
FF←1
CALL DISC DRIVER
CLF J, FF←0
YES
JMP TO RESTART DISC TRANSFER

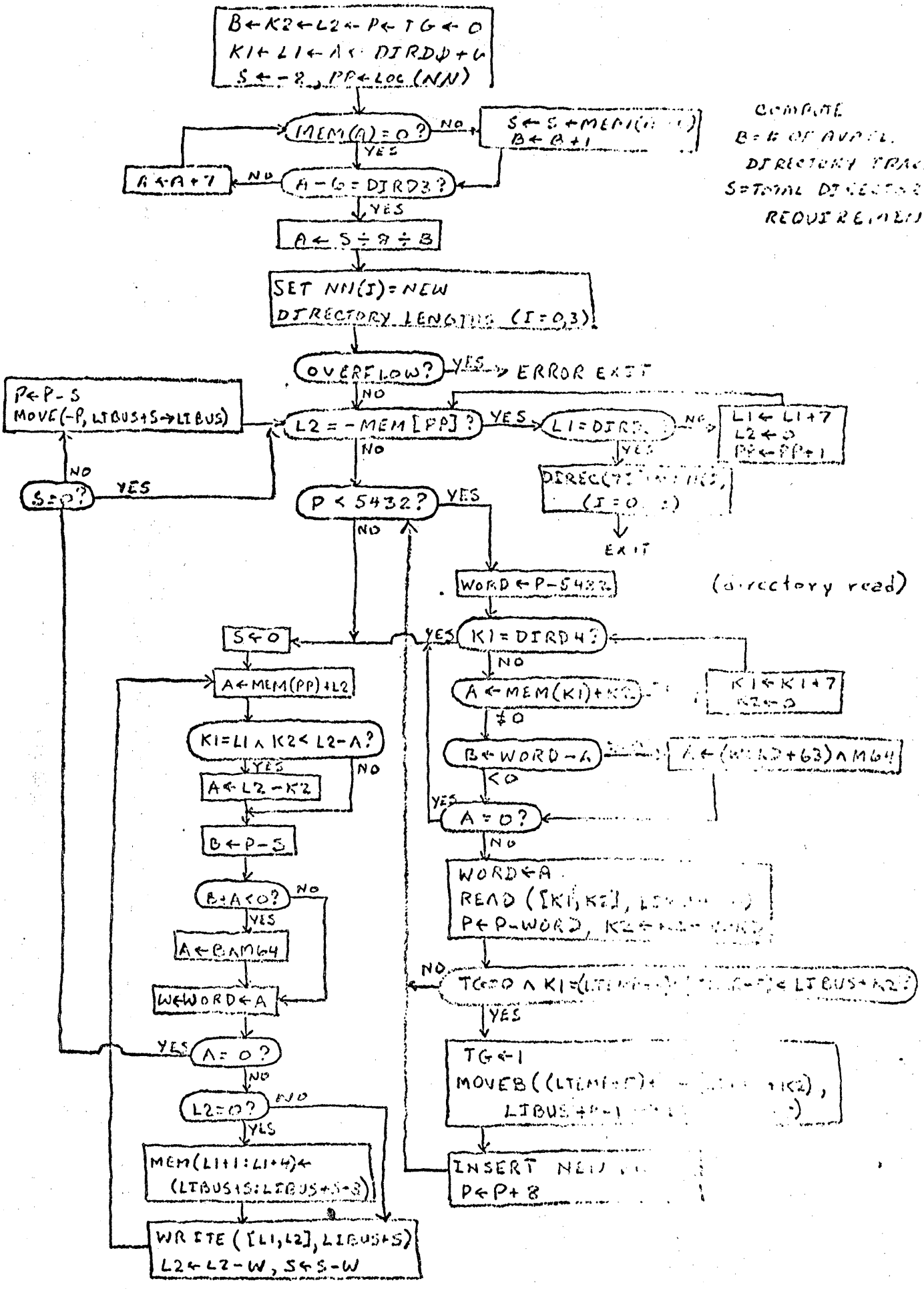

DLOOK
(SEARCH FOR DIRECTORY ENTRY)
LT4 ← [DIREC3]
((LT4) = 0 ?
YES
LT4 ← LT4 - 7
NO
MEM[LT4+1 : LT4+4] : (LTØ : LT3)
ENTRY ON THIS TRACK ?
WORD ← (LT4)
LT5 ← LIBD - (LT4) - 8
READ DIRECTORY TRACK; LT4 POINTS TO ITS DIREC ENTRY, LT5 TO ITS LAST ELEMENT
LT4 ← LIBD
MEM[LT14 : LT14+3] : (LTØ : LT3)
RETURN FOUND
LT15 ← LT5
LT5 ← LT14 + ((LT15 - LT14) ÷ 2)
LT14 ← LT5
LT5 = LT14 ?
YES
LT5 = LIBD?
NO
YES
MEM[LT5 : LT5+3] : (LTØ : LT3)
RETURN FOUND

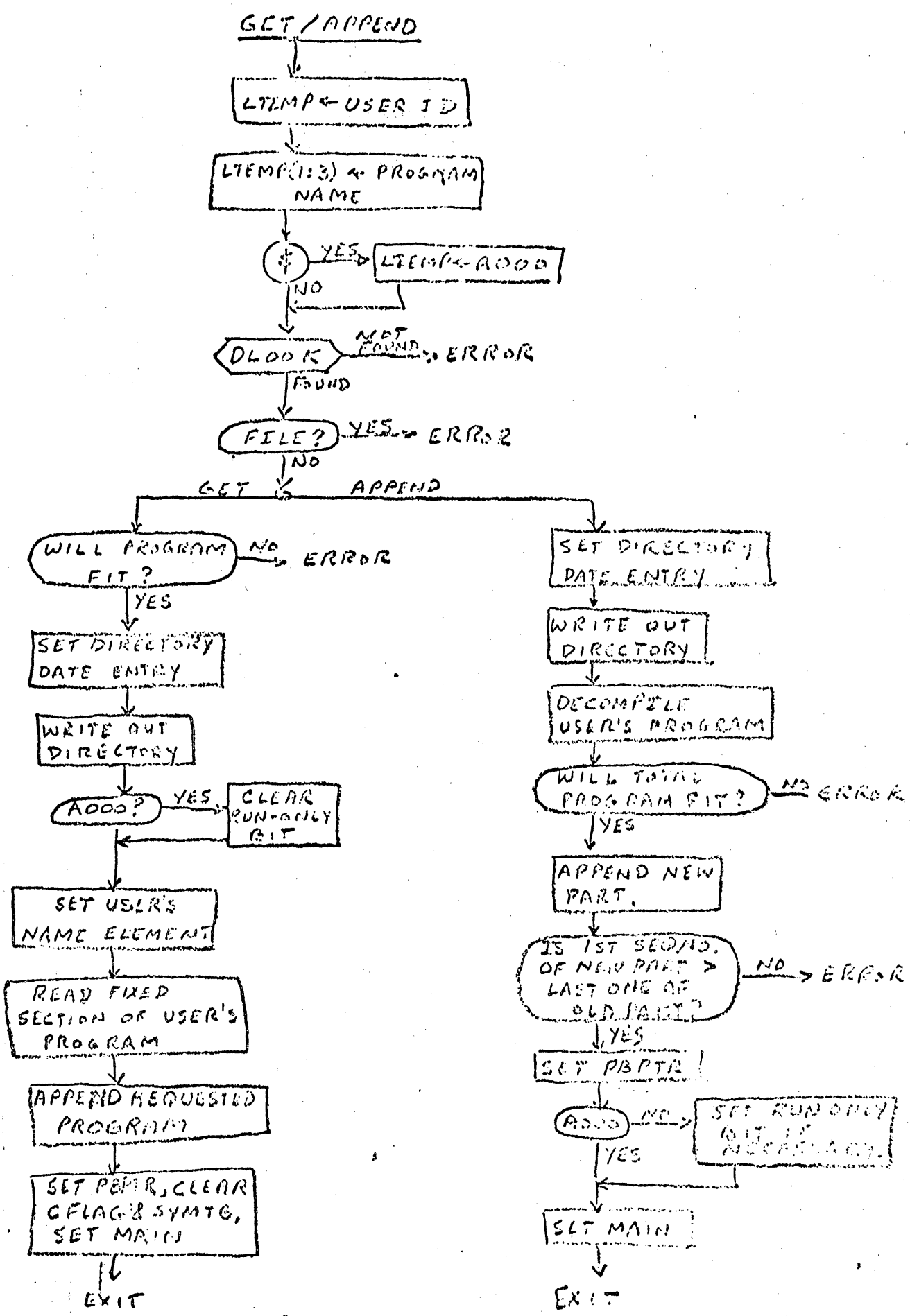
FILES

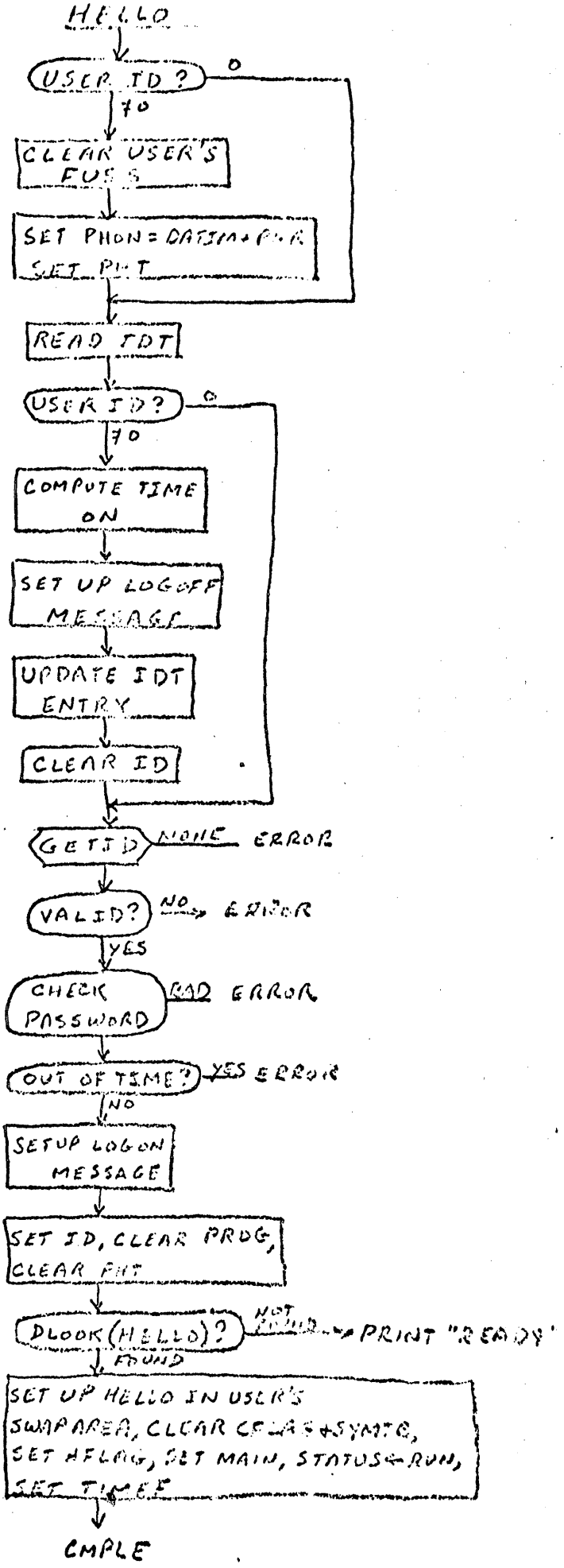

TRANSFER CHARACTERS FROM FILES STATEMENT INTO USER'S BUFFER
SEARCH DIRECTORY FOR A FILE
NOT FOUND
ERROR RETURN
IS IT A FILE OR PROGRAM?
PROGRAM
ERROR RETURN
FILE
PROTECTED?
YES
ERROR RETURN
NO
SAVE DISC ADDRESS AND LENGTH IN BUFFER, AND IF PUBLIC FILE ACCESS, SET READ-ONLY BIT.
SET DATE INTO DIRECTORY ENTRY.
YES
ANY MORE FILES?
NO
ROOM FOR FILE TABLES IN USER PROGRAM?
NO
ERROR RETURN
YES
READ IN FUSS
CLEAR USER'S FUSS
ADDD?
YES
NO
MARK ANY BUSY FILES AS READ-ONLY.
WRITE FUSS TABLE BACK TO DISC.
BUILD FILES TABLE FOR BASIC
RETURN

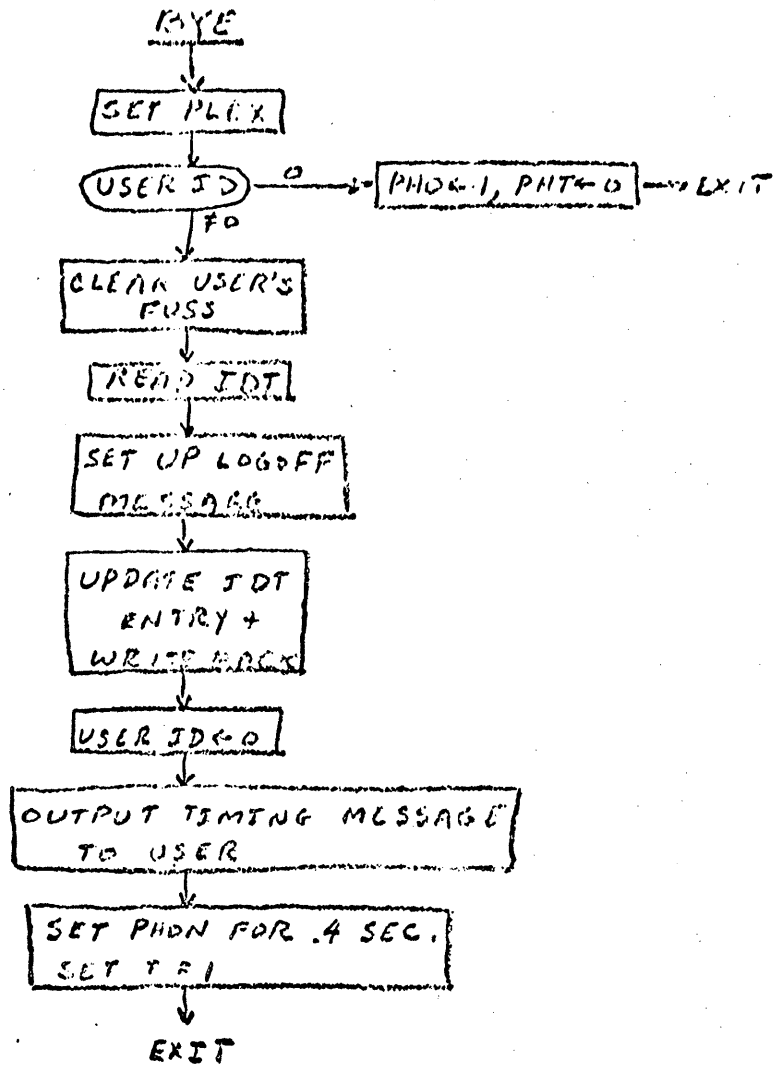

NULL PROGRAM NAME?
YES
NO
NULL PROGRAM?
YES
NO
COMPILE MODE?
YES
NO
GET PROGRAM LENGTH
READ JDT + ADT
USER OUT OF SPACE?
YES
NO
FIND FIRST ADT ENTRY THAT'S BIG ENOUGH
FOUND
DLOOK
FOUND
NO ROOM
DIRECTORY TRACK FULL ?
YES
NO
INSERT IN DIRECTORY & UPDATE DIREC
UPDATE ADT & JDT
COPY USER PROGRAM TO JTS SAVED LOCATION.
EXIT

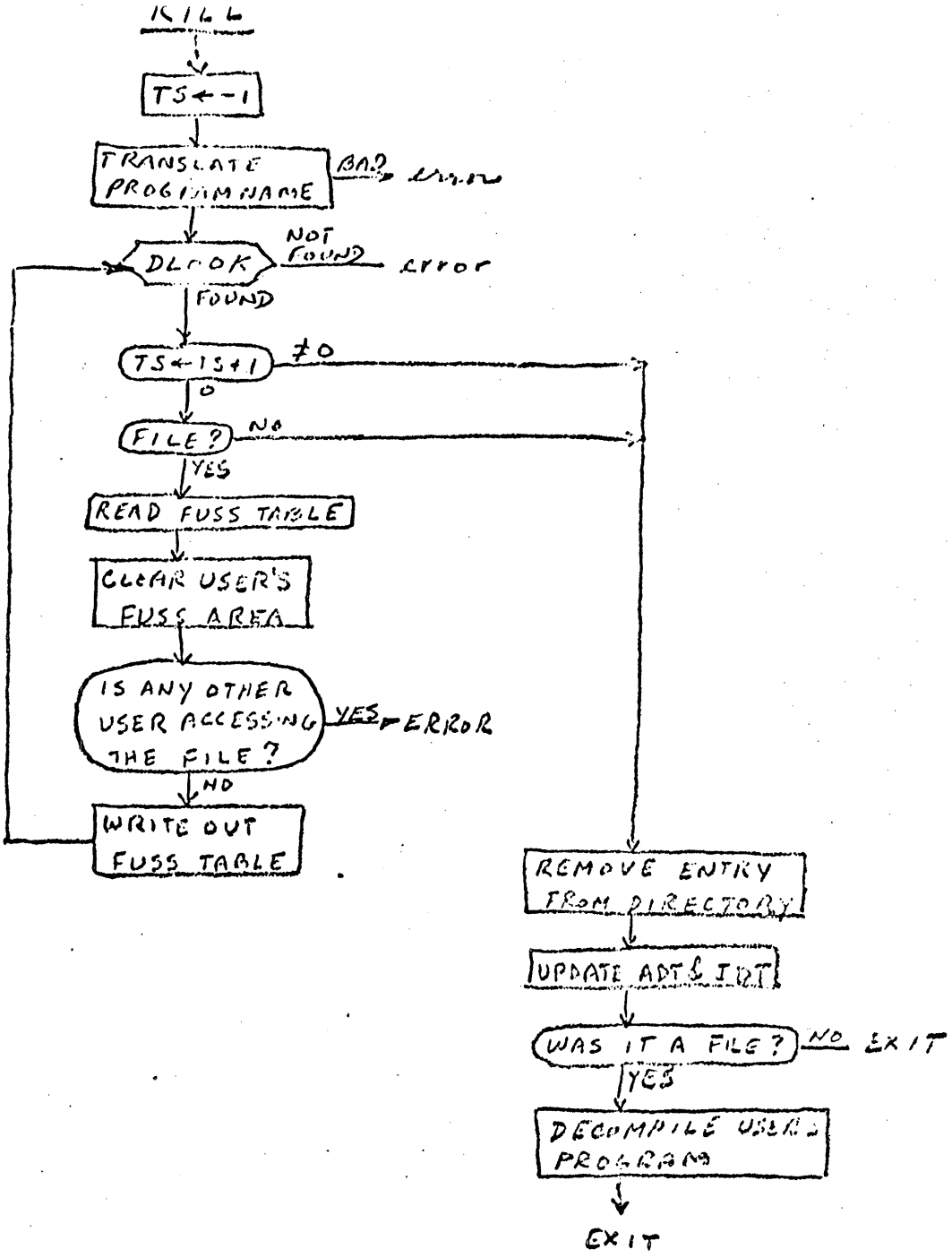
SUPERSAVE

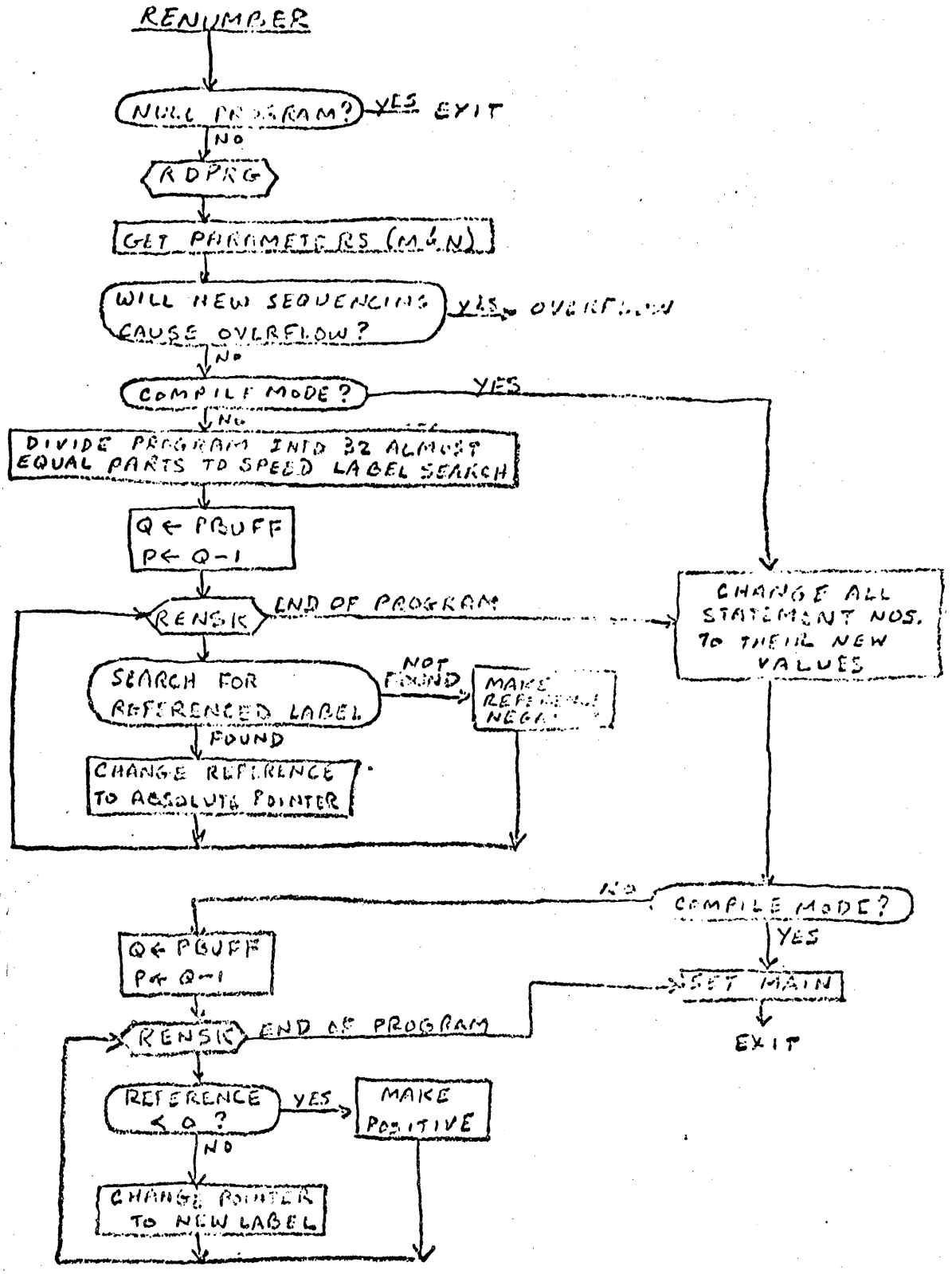

B←K2←L2←P←TG←0
K1←L1←A←DIRD0+6
S←-8, PP←LOC(NN)
MEM(A)=0?
NO
S←S+MEM(A+6)
B←B+1
YES
A←A+7
NO
A-6=DIRD3?
YES
A←S÷8÷B
SET NN(I)=NEW DIRECTORY LENGTHS (I=0,3)
OVERFLOW?
YES
ERROR EXIT
NO
P←P-S
MOVE(-P, LIBUS+S→LIBUS)
L2=-MEM[PP]?
YES
L1=DIRD.
NO
L1←L1+7
L2←0
PP←PP+1
YES
DIREC(?
(I=0
EXIT
NO
S=0?
YES
P<5432?
YES
NO
WORD←P-5432
(directory read)
S←0
YES
K1=DIRD4?
NO
A←MEM(K1)+K2
K1←K1+7
K2←0
≠0
B←WORD-A
A←(
+63)∧M64
<0
YES
A=0?
NO
WORD←A
READ([K1,K2],
P←P-WORD2, K2←
A←MEM(PP)+L2
K1=L1∧K2<L2-A?
YES
NO
A←L2-K2
B←P-S
B+A<0?
NO
YES
A←BAM64
W←WORD←A
YES
A=0?
NO
L2=0?
NO
YES
MEM(L1+1:L1+4)←
(LTBUS+S:LTBUS+S-8)
WRITE([L1,L2], LTBUS+S)
L2←L2-W, S←S-W
NO
TG=0∧K1=(
<LTBUS+K2?
YES
TG←1
MOVEB((LTEM
+K2),
LTBUS+P-1
INSERT NEW
P←P+8
COMPUTE
B=# OF AVAIL.
DIRECTORY TRACKS
S=TOTAL DIRECTORY
REQUIREMENT


GET / APPEND
LTEMP ← USER ID
LTEMP(1:3) ← PROGRAM NAME
YES
LTEMP ← ADOO
NO
DLOOK
NOT FOUND
ERROR
FOUND
FILE?
YES
ERROR
NO
GET
APPEND
WILL PROGRAM FIT ?
NO
ERROR
YES
SET DIRECTORY DATE ENTRY
WRITE OUT DIRECTORY
ADOO?
YES
CLEAR RUN-ONLY BIT
SET USLR'S NAME ELEMENT
READ FIXED SECTION OF USER'S PROGRAM
APPEND REQUESTED PROGRAM
SET PBPTR, CLEAR CFLAG & SYNTG, SET MAIN
EXIT
SET DIRECTORY DATE ENTRY
WRITE OUT DIRECTORY
DECOMPILE USER'S PROGRAM
WILL TOTAL PROGRAM FIT?
NO
ERROR
YES
APPEND NEW PART.
IS 1ST SEG/D. OF NEW PART > LAST ONE OF OLD PART?
NO
ERROR
YES
SET PBPTR
ADOO
NO
SET RUN ONLY BIT IF NECESSARY.
YES
SET MAIN
EXIT

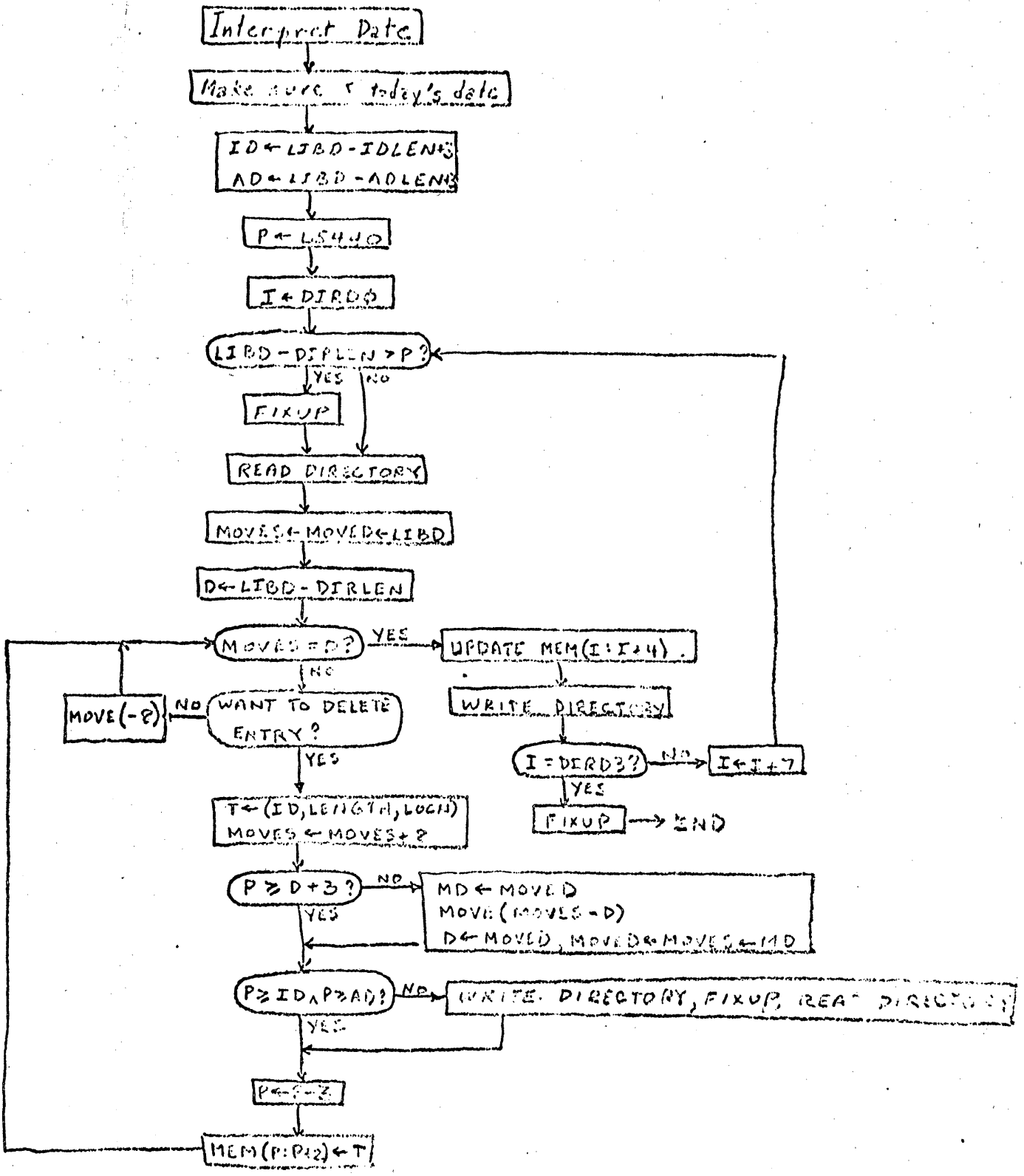

HELLO
USER ID ?
0
70
CLEAR USER'S FUSS
SET PHON = DATIM + PWR
SET PHT
READ IDT
USER ID?
0
70
COMPUTE TIME ON
SET UP LOGOFF MESSAGE
UPDATE IDT ENTRY
CLEAR ID
GETID
NONE
ERROR
VALID?
NO
ERROR
YES
CHECK PASSWORD
BAD
ERROR
OUT OF TIME?
YES
ERROR
NO
SETUP LOGON MESSAGE
SET ID, CLEAR PROG, CLEAR PHT
DLOOK (HELLO)?
NOT FOUND
PRINT "READY"
FOUND
SET UP HELLO IN USER'S SWAPAREA, CLEAR CFLAG + SYMTB, SET HFLAG, SET MAIN, STATUS ← RUN, SET TIMEF
CMPLE

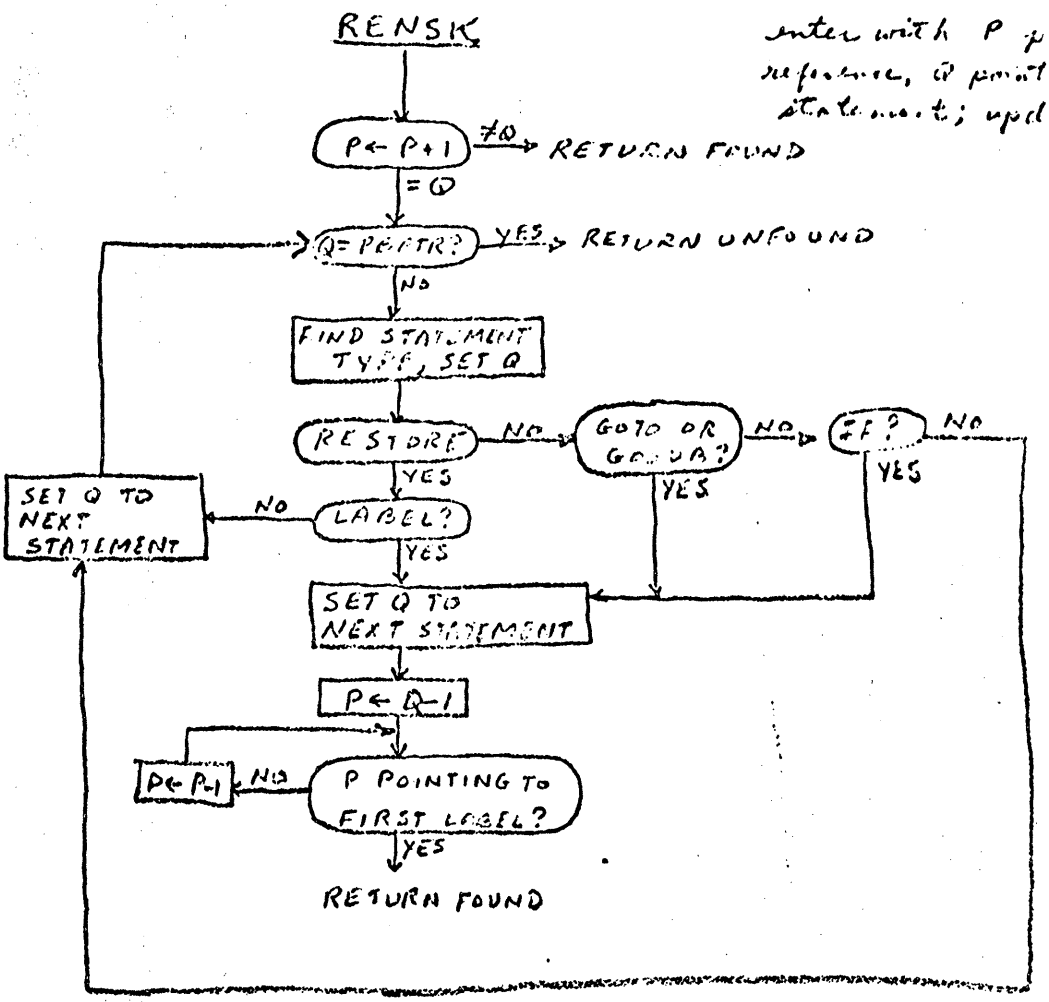

BYE
SET PLEX
USER ID
0
PHON←1, PHT←0
EXIT
FO
CLEAR USER'S FUSS
READ IDT
SET UP LOGOFF MESSAGE
UPDATE IDT ENTRY + WRITE BACK
USER ID←0
OUTPUT TIMING MESSAGE TO USER
SET PHON FOR .4 SEC.
SET T=1
EXIT

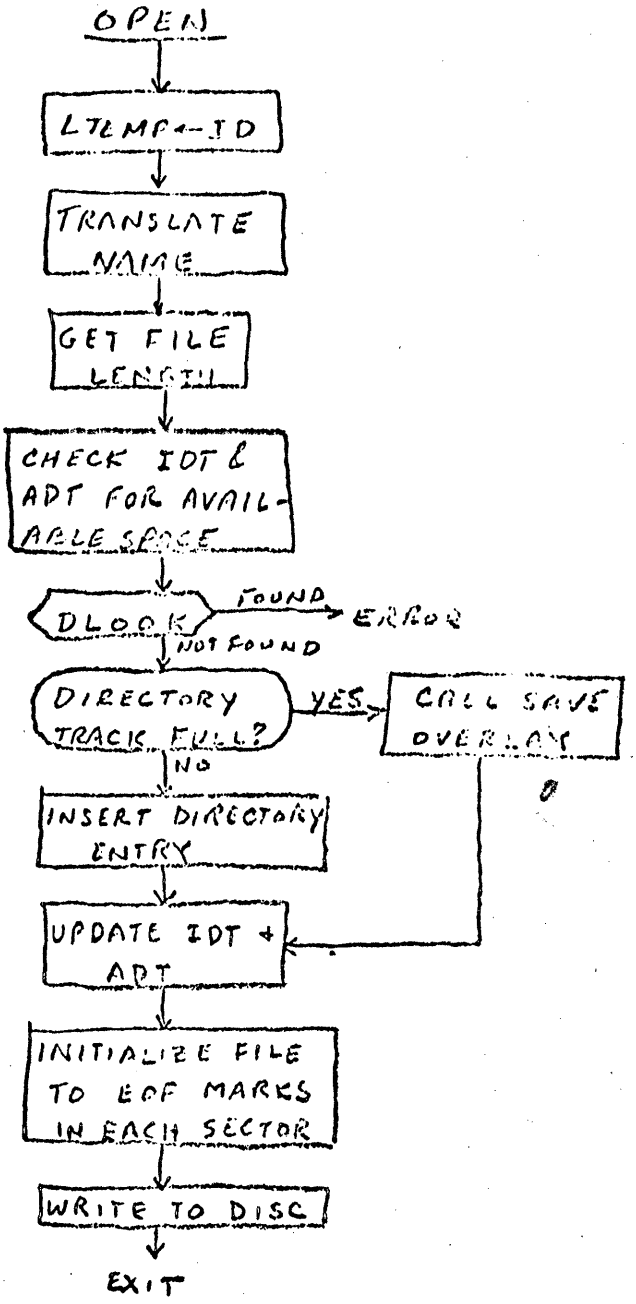

KILL
TS←-1
TRANSLATE PROGRAM NAME
BAD → ERROR
DLOOK
NOT FOUND
ERROR
FOUND
TS←TS+1
≠0
0
FILE?
NO
YES
READ FUSS TABLE
CLEAR USER'S FUSS AREA
IS ANY OTHER USER ACCESSING THE FILE?
YES → ERROR
NO
WRITE OUT FUSS TABLE
REMOVE ENTRY FROM DIRECTORY
UPDATE ADT & IRT
WAS IT A FILE?
NO
EXIT
YES
DECOMPILE USER'S PROGRAM
EXIT

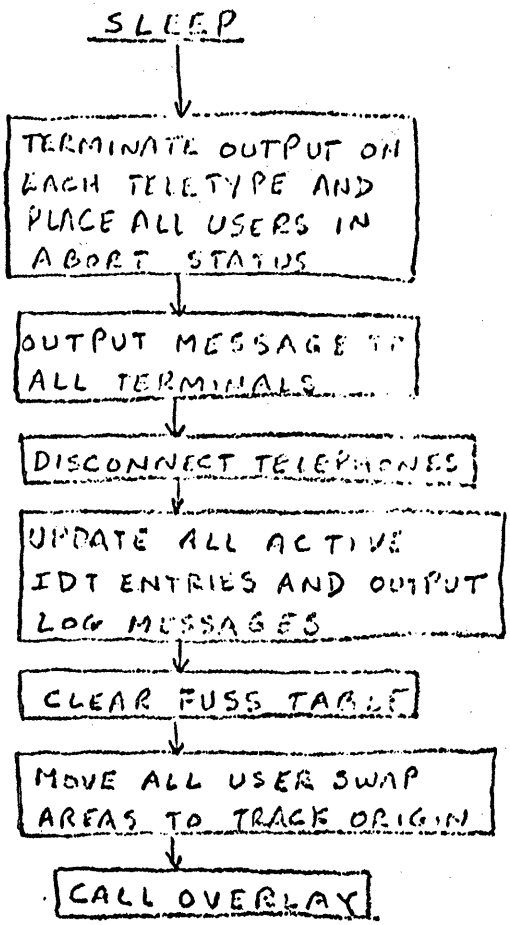

RENUMBER
NULL PROGRAM?
YES
EXIT
NO
RDPRG
GET PARAMETERS (M,N)
WILL NEW SEQUENCING CAUSE OVERFLOW?
YES
OVERFLOW
NO
COMPILE MODE?
YES
NO
DIVIDE PROGRAM INTO 32 ALMOST EQUAL PARTS TO SPEED LABEL SEARCH
Q ← PBUFF
P ← Q-1
RENSIK
END OF PROGRAM
CHANGE ALL STATEMENT NOS. TO THEIR NEW VALUES
SEARCH FOR REFERENCED LABEL
NOT FOUND
MAKE REFERENCE NEGATIVE
FOUND
CHANGE REFERENCE TO ABSOLUTE POINTER
NO
COMPILE MODE?
YES
Q ← PBUFF
P ← Q-1
SET MAIN
EXIT
RENSIK
END OF PROGRAM
REFERENCE < 0 ?
YES
MAKE POSITIVE
NO
CHANGE POINTER TO NEW LABEL

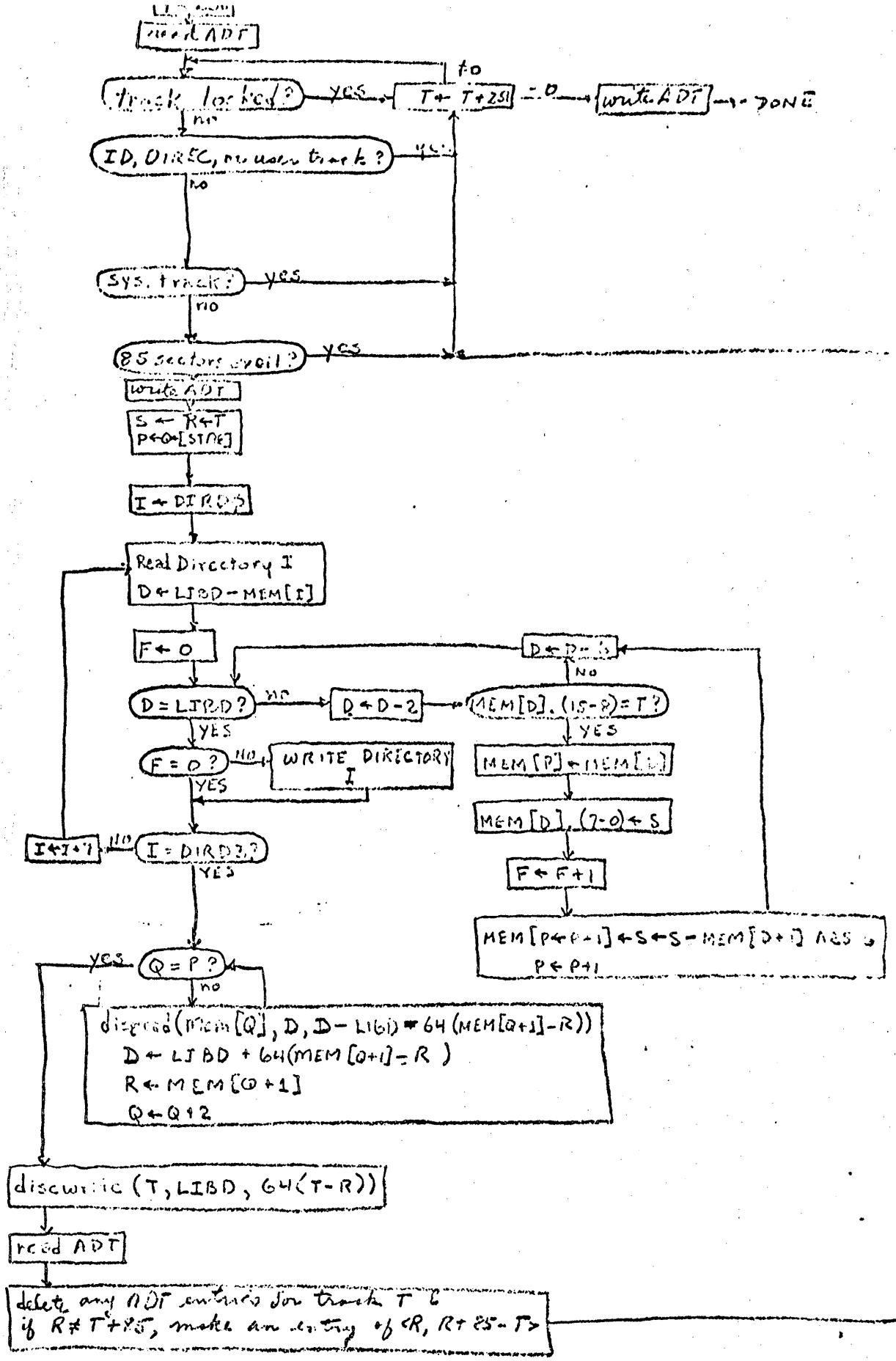
Purge


Interpret Date
Make sure ≤ today's date
ID ← LIBD - IDLEN3
AD ← LIBD - ADLEN3
P ← LS4U0
I ← DIRD0
LIBD - DIRLEN > P?
YES
NO
FIXUP
READ DIRECTORY
MOVES ← MOVED ← LIBD
D ← LIBD - DIRLEN
MOVES = D?
YES
UPDATE MEM(I:I+4)
NO
WRITE DIRECTORY
MOVE(-8)
NO
WANT TO DELETE ENTRY?
I = DIRD3?
NO
I ← I + 7
YES
FIXUP
END
YES
T ← (ID, LENGTH, LOCN)
MOVES ← MOVES + 8
P ≥ D + 3?
NO
MD ← MOVED
MOVE(MOVES - D)
D ← MOVED, MOVED ← MOVES ← MD
YES
P ≥ ID ∧ P ≥ AD?
NO
WRITE DIRECTORY, FIXUP, READ DIRECTORY
YES
P ← P - 2
MEM(P:P+2) ← T


RENSK
enter with P pointing to last reference, Q pointing to next statement; update P to point to first
P ← P+1
≠Q
RETURN FOUND
=Q
Q=PTR?
YES
RETURN UNFOUND
NO
FIND STATEMENT TYPE, SET Q
RESTORE
NO
GOTO OR GOSUB?
NO
IF?
NO
YES
YES
YES
SET Q TO NEXT STATEMENT
NO
LABEL?
YES
SET Q TO NEXT STATEMENT
P ← Q-1
P ← P-1
NO
P POINTING TO FIRST LABEL?
YES
RETURN FOUND

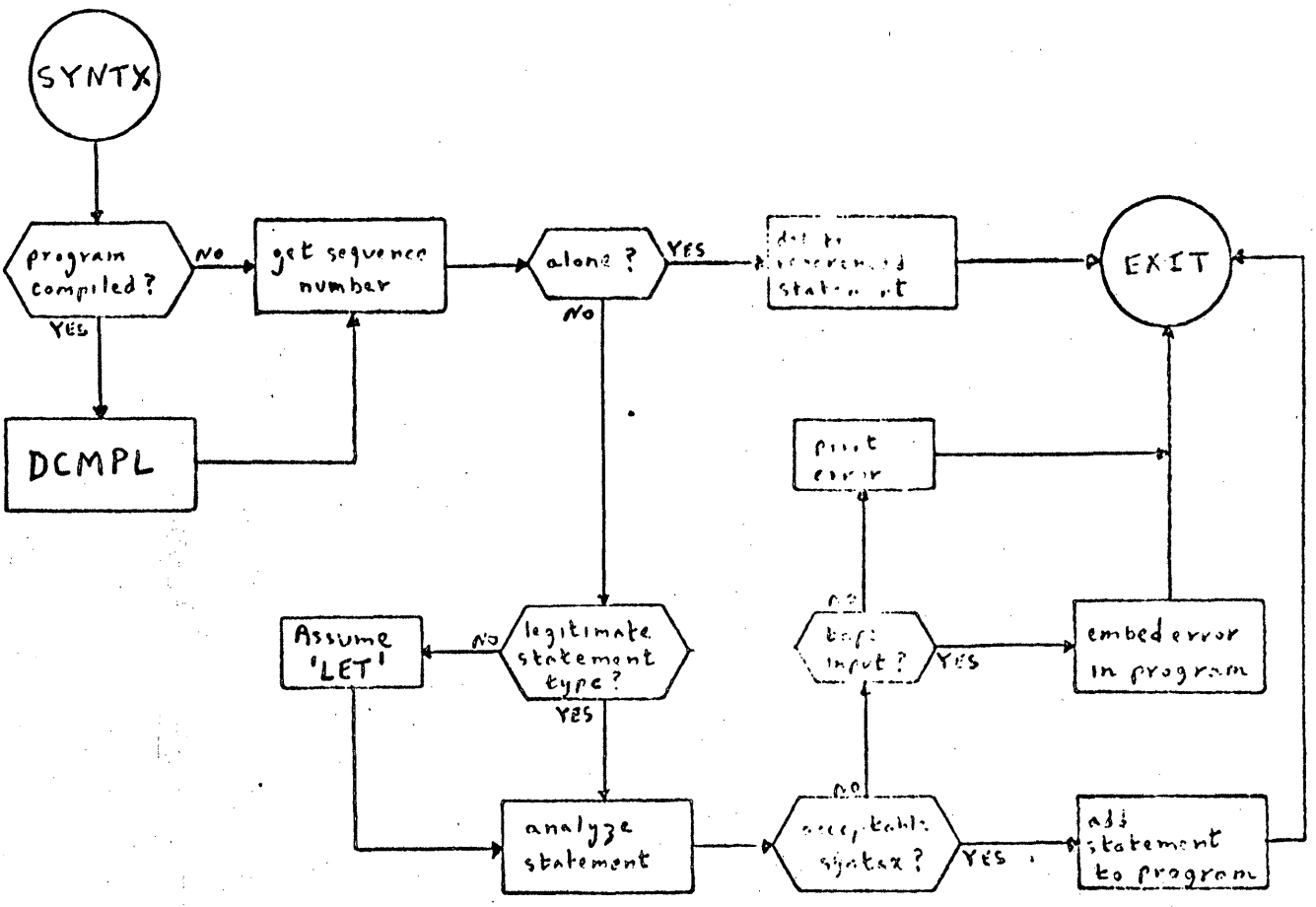

OPEN
LTEMP←ID
TRANSLATE NAME
GET FILE LENGTH
CHECK IDT & ADT FOR AVAILABLE SPACE
DLOOK
FOUND
ERROR
NOT FOUND
DIRECTORY TRACK FULL?
YES
CALL SAVE OVERLAY
NO
INSERT DIRECTORY ENTRY
UPDATE IDT + ADT
INITIALIZE FILE TO EOF MARKS IN EACH SECTOR
WRITE TO DISC
EXIT


SLEEP
TERMINATE OUTPUT ON EACH TELETYPE AND PLACE ALL USERS IN ABORT STATUS
OUTPUT MESSAGE TO ALL TERMINALS
DISCONNECT TELEPHONES
UPDATE ALL ACTIVE IDT ENTRIES AND OUTPUT LOG MESSAGES
CLEAR FUSS TABLE
MOVE ALL USER SWAP AREAS TO TRACK ORIGIN
CALL OVERLAY


read ADT
track locked?
yes
no
T ← T+251
= 0
write ADT
DONE
ID, DIREC, or user track?
yes
no
Sys. track?
yes
no
85 sectors avail?
yes
write ADT
S ← R←T
P←Q←[STAGE]
I ← DIRD3
Read Directory I
D ← LIBD − MEM[I]
F ← 0
D = LIBD?
no
D ← D − 2
MEM[D].(15-8) = T?
YES
F = 0?
NO
WRITE DIRECTORY I
YES
MEM[P] ← MEM[D]
MEM[D].(7-0) ← S
F ← F+1
MEM[P←P+1] ← S ← S − MEM[D+1] ABS
P ← P+1
I ← I+7
NO
I = DIRD3?
YES
Q = P?
yes
no
disgrad(Mem[Q], D, D − LIBD − 64(MEM[Q+1] − R))
D ← LIBD + 64(MEM[Q+1] − R)
R ← MEM[Q+1]
Q ← Q+2
discwrite(T, LIBD, 64(T−R))
read ADT
delete any ADT entries for track T &
if R ≠ T+85, make an entry of <R, R+85−T>

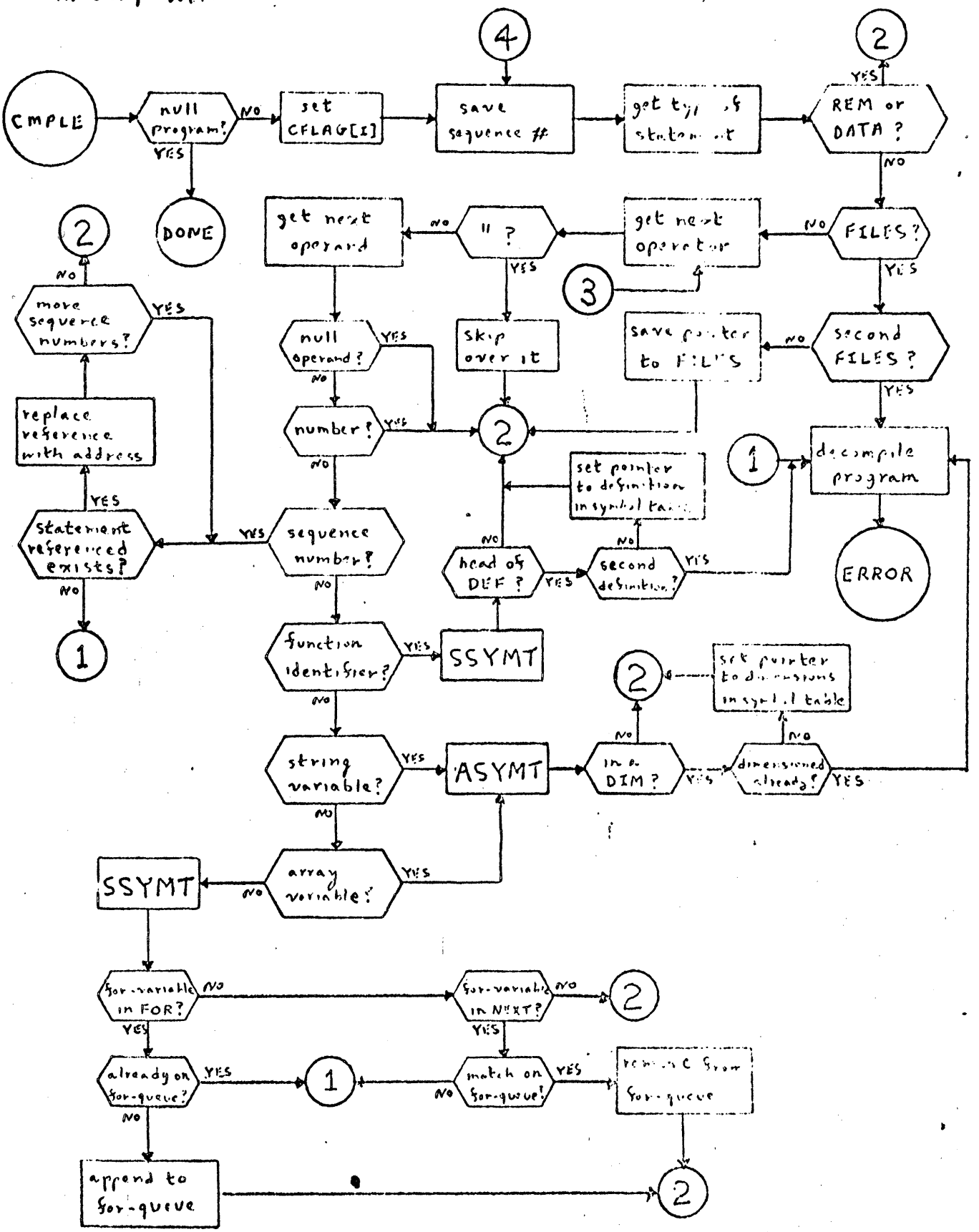
BASIC FLOW CHARTS

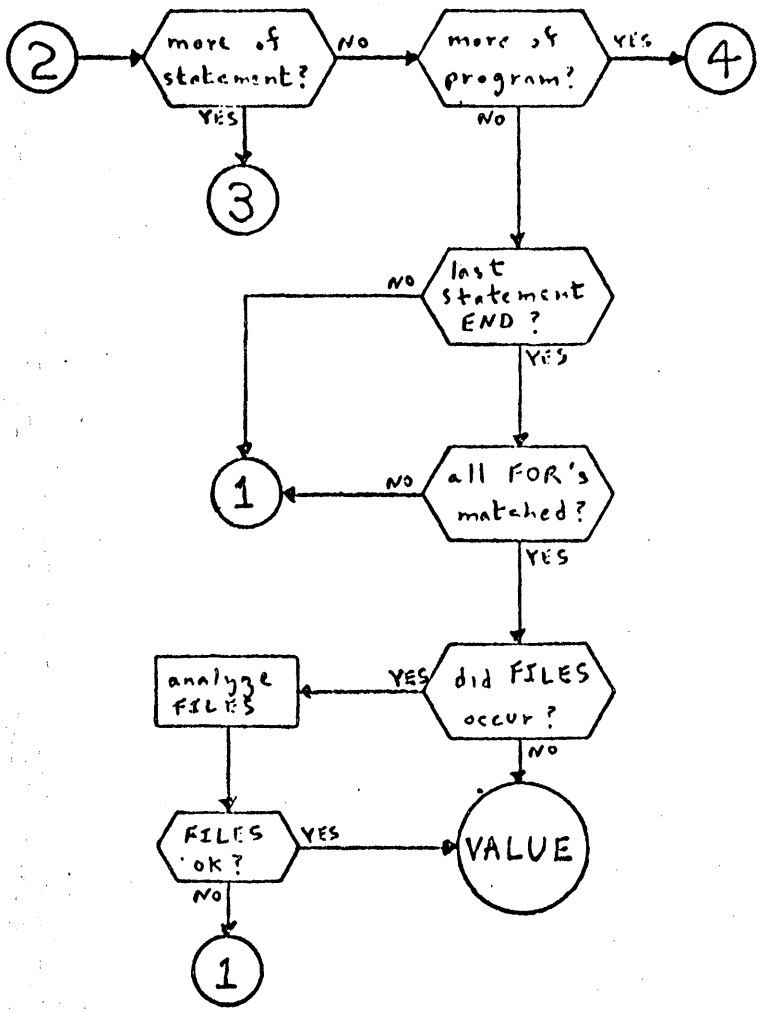
I. Syntax

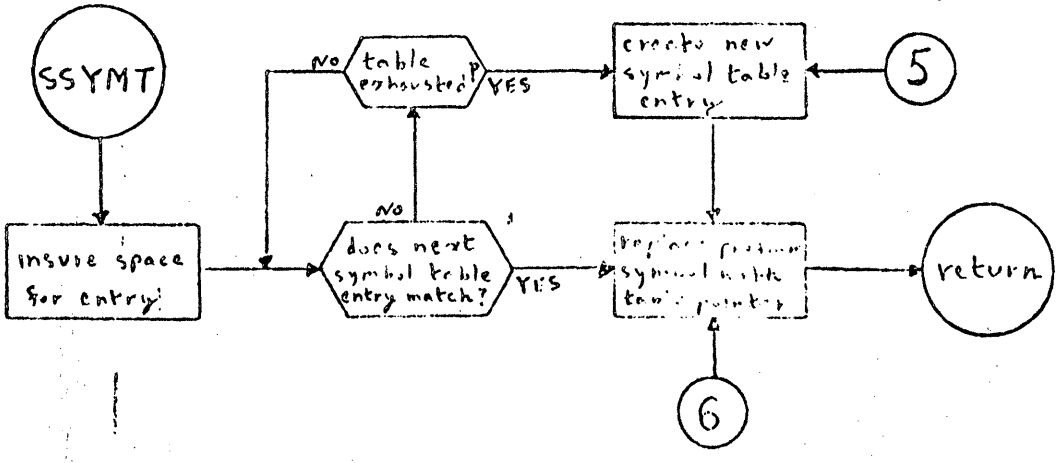


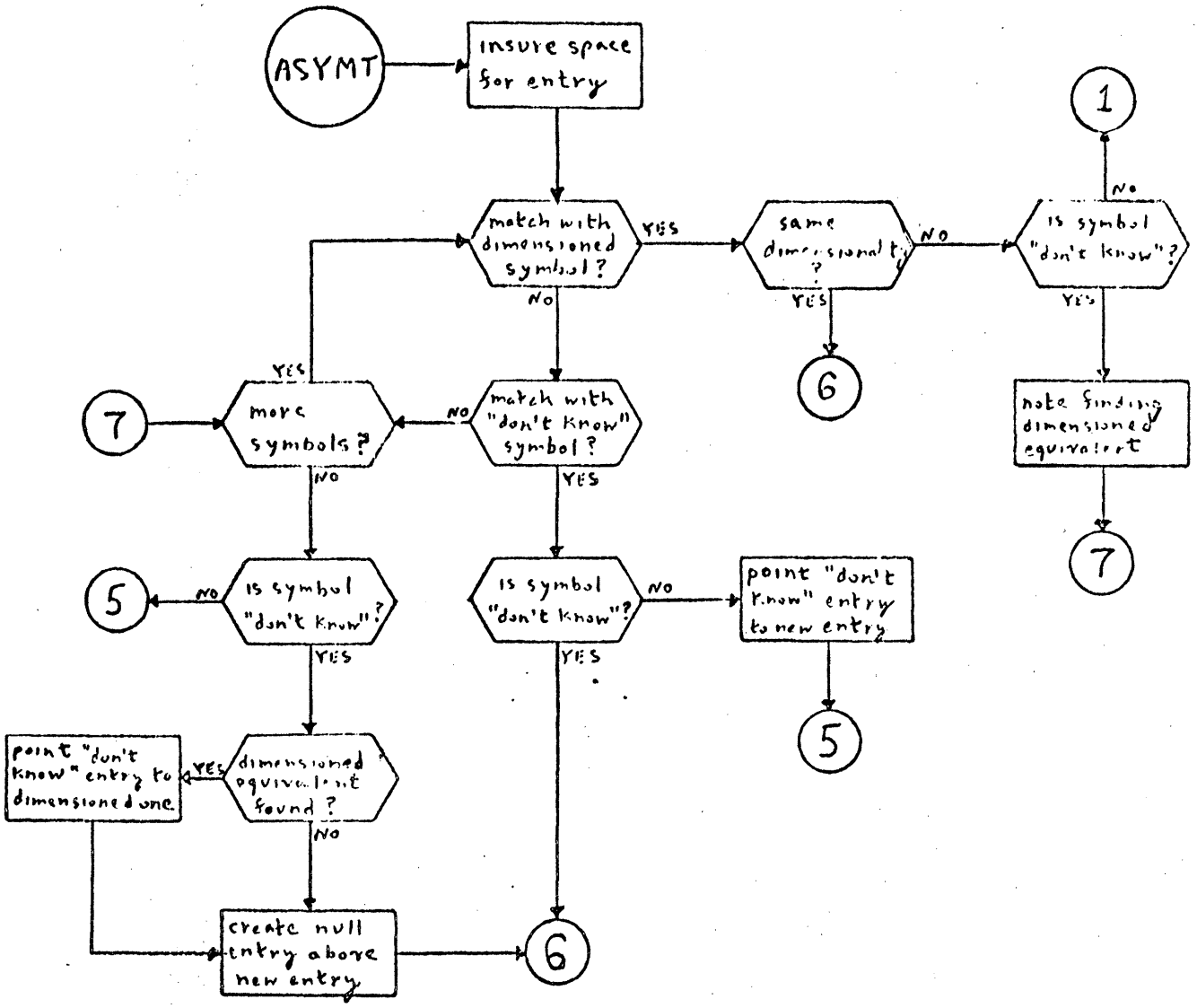
II. Phase 2

A. Compilation

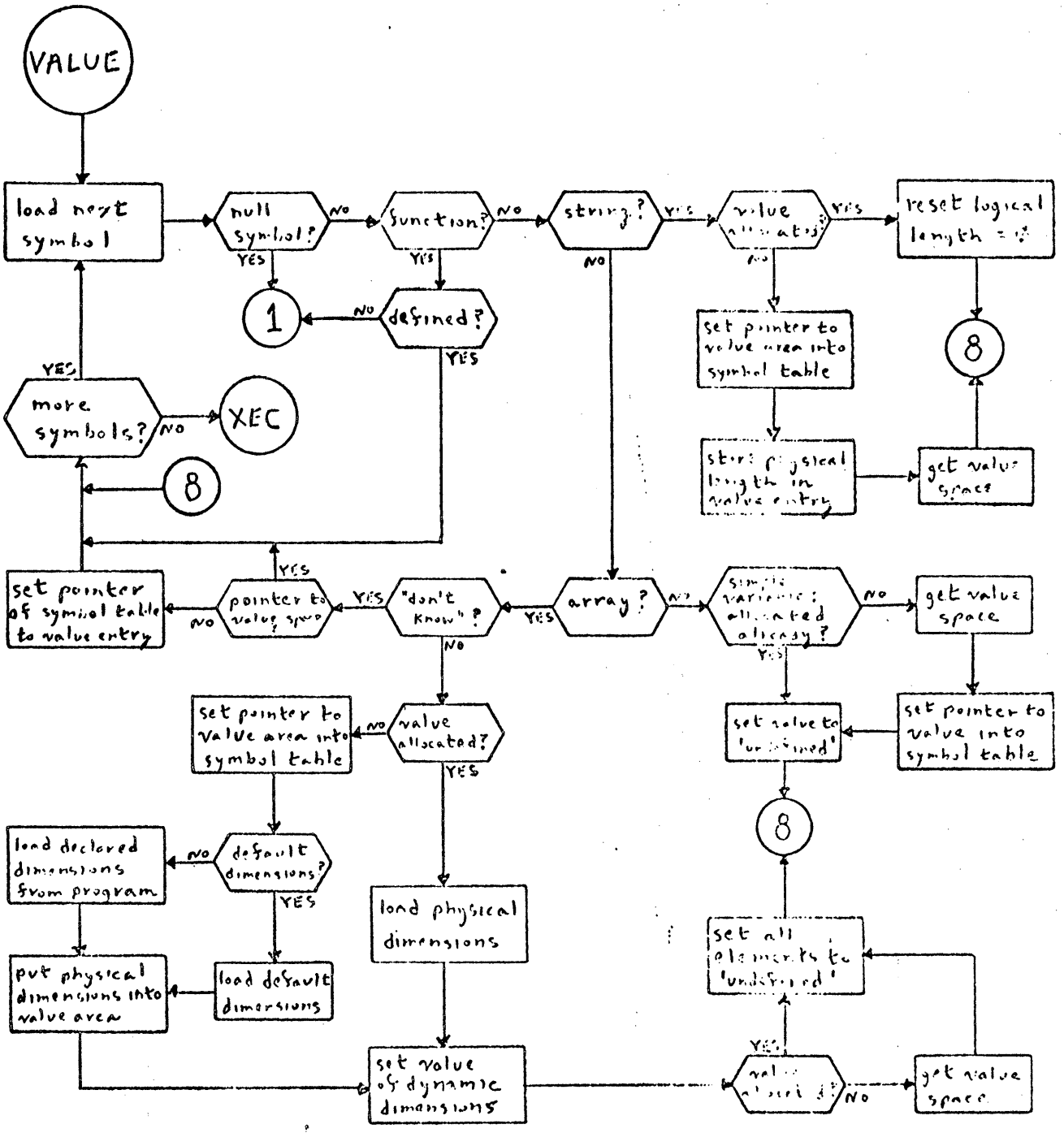



2
more of statement?
NO
more of program?
YES
4
YES
3
NO
last statement END ?
NO
YES
1
NO
all FOR's matched?
YES
analyze FILES
YES
did FILES occur ?
NO
FILES OK ?
YES
VALUE
NO
1

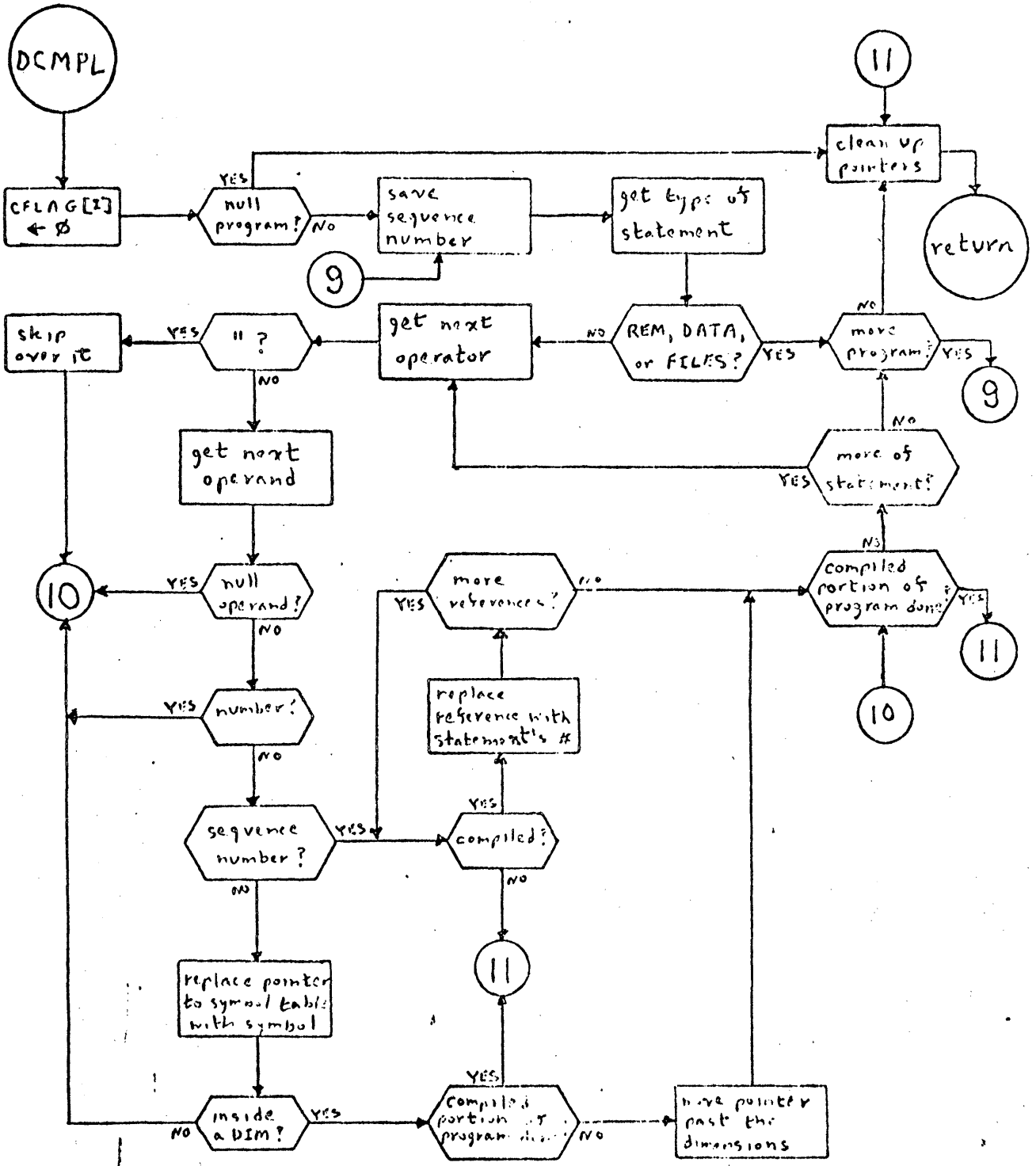

SSYMT
NO
table exhausted?
YES
create new symbol table entry
5
insure space for entry
NO
does next symbol table entry match?
YES
return
6


ASYMT
insure space for entry
match with dimensioned symbol?
YES
same dimensionality?
NO
is symbol "don't know"?
1
NO
YES
YES
6
note finding dimensioned equivalent
7
NO
match with "don't know" symbol?
YES
7
more symbols?
YES
NO
5
NO
is symbol "don't know"?
YES
is symbol "don't know"?
NO
point "don't know" entry to new entry
YES
5
point "don't know" entry to dimensioned one
YES
dimensioned equivalent found?
NO
create null entry above new entry
6

B. Value

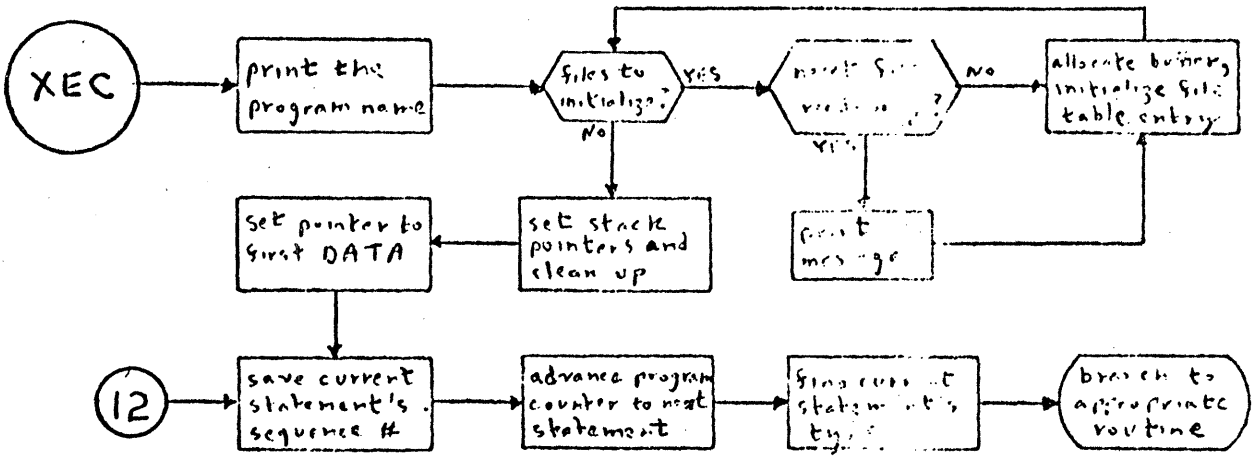

VALUE
load next symbol
null symbol?
NO
YES
function?
NO
YES
string?
YES
NO
value allocated?
YES
NO
reset logical length = 0
1
NO
defined?
YES
set pointer to value area into symbol table
8
YES
more symbols?
NO
XEC
8
store physical length in value entry
get value space
YES
set pointer of symbol table to value entry
NO
pointer to value space
YES
"don't Know"?
YES
NO
array?
NO
YES
simple variable allocated already?
NO
get value space
set pointer to value into symbol table
set value to 'undefined'
8
set pointer to value area into symbol table
NO
value allocated?
YES
load declared dimensions from program
NO
default dimensions?
YES
load physical dimensions
load default dimensions
put physical dimensions into value area
set value of dynamic dimensions
set all elements to 'undefined'
YES
NO
get value space

C. Decompilation

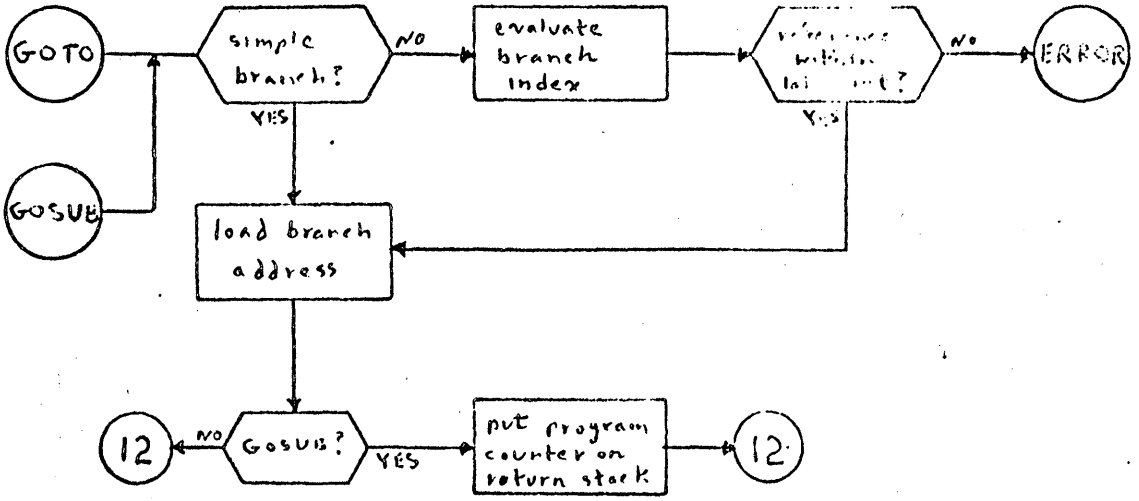


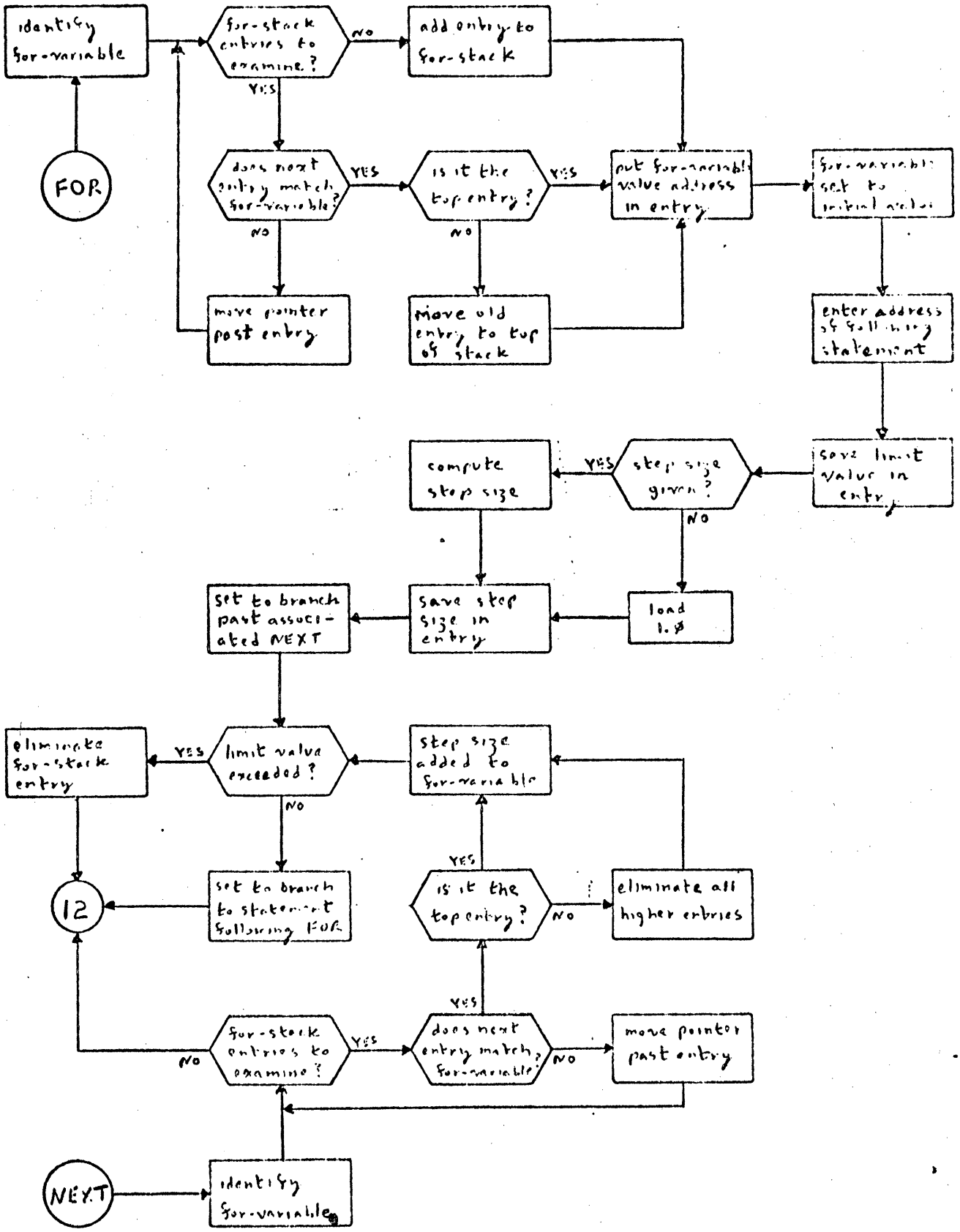
III. Execution

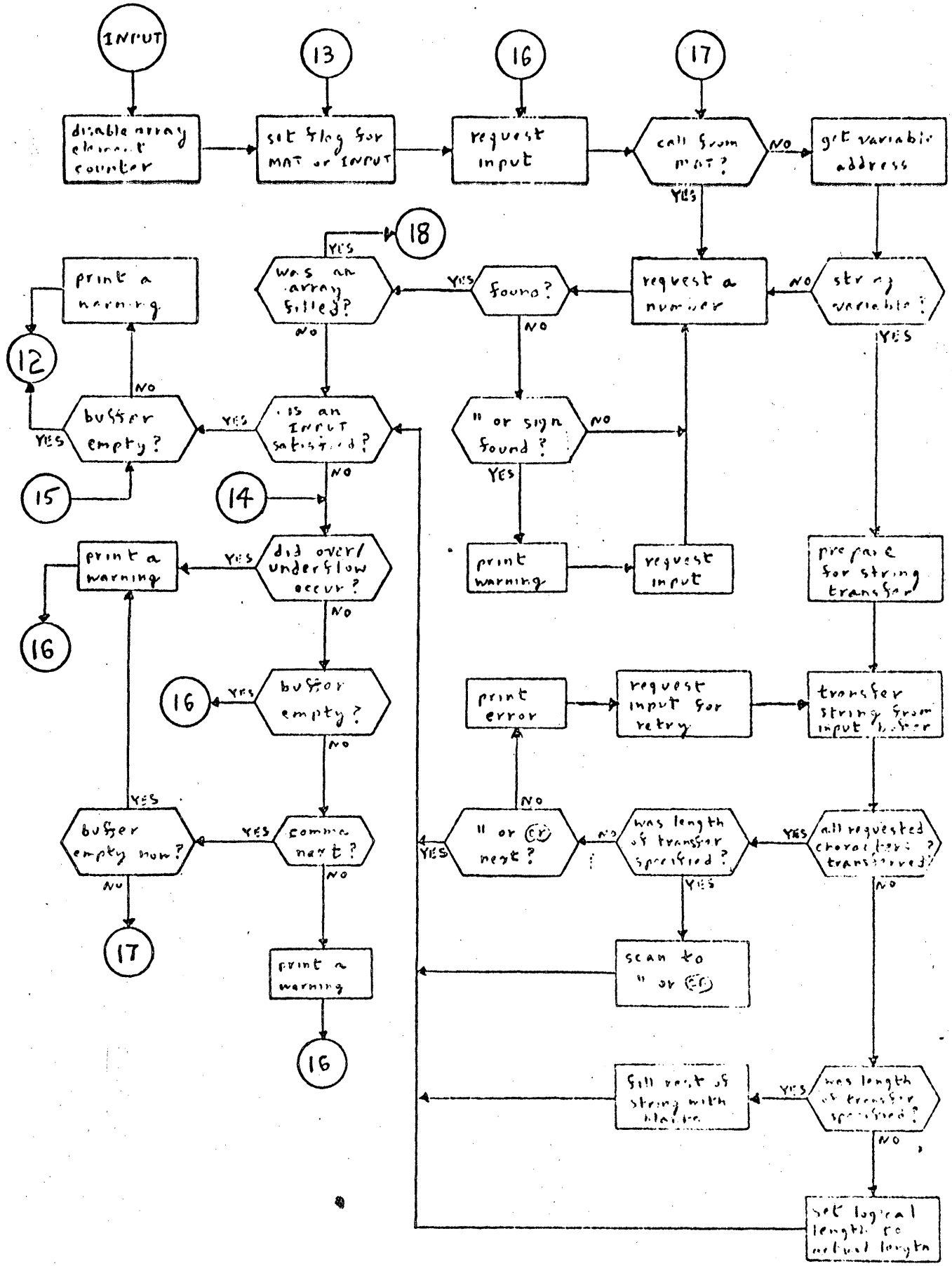
A. Main Loop

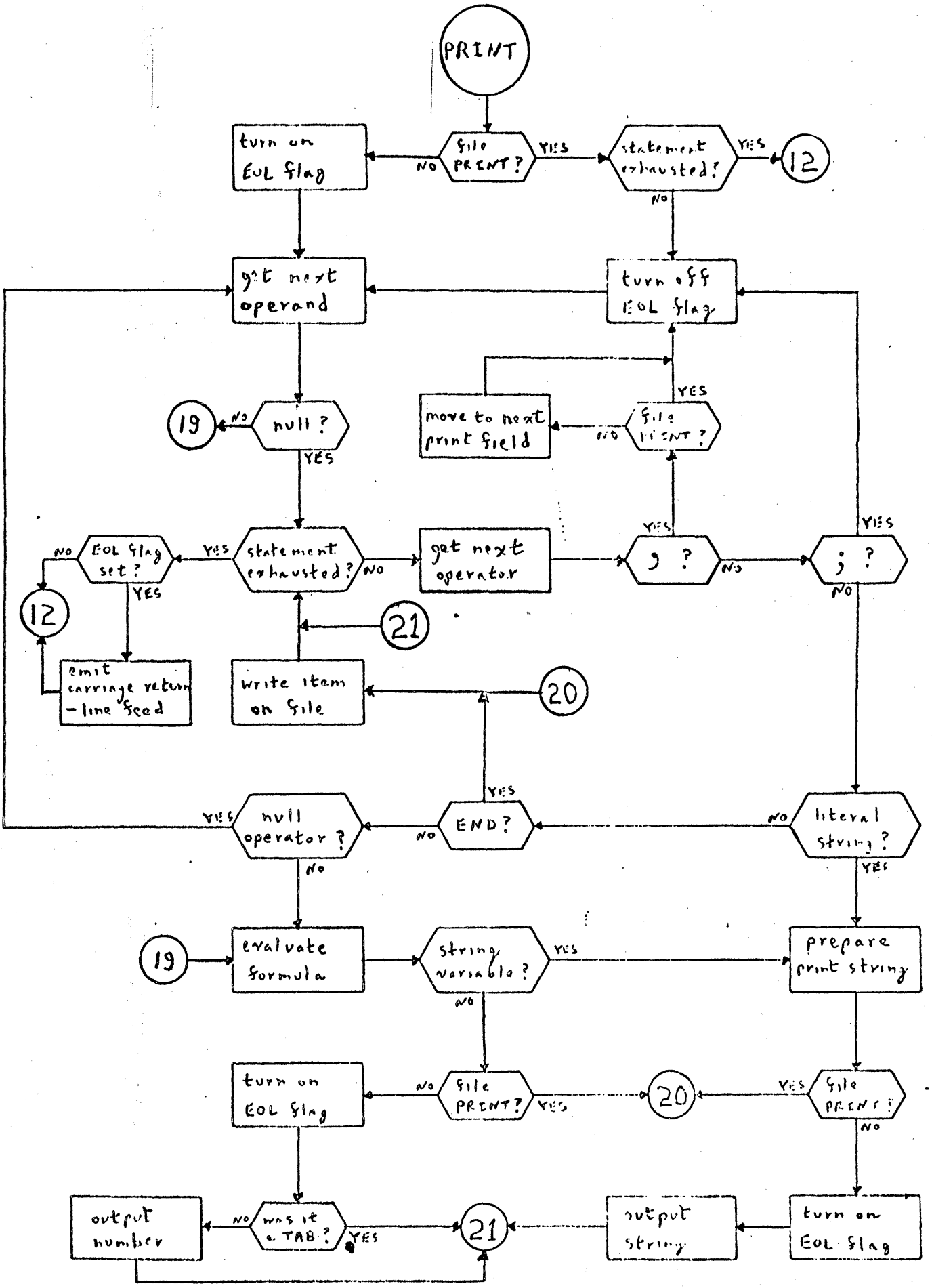


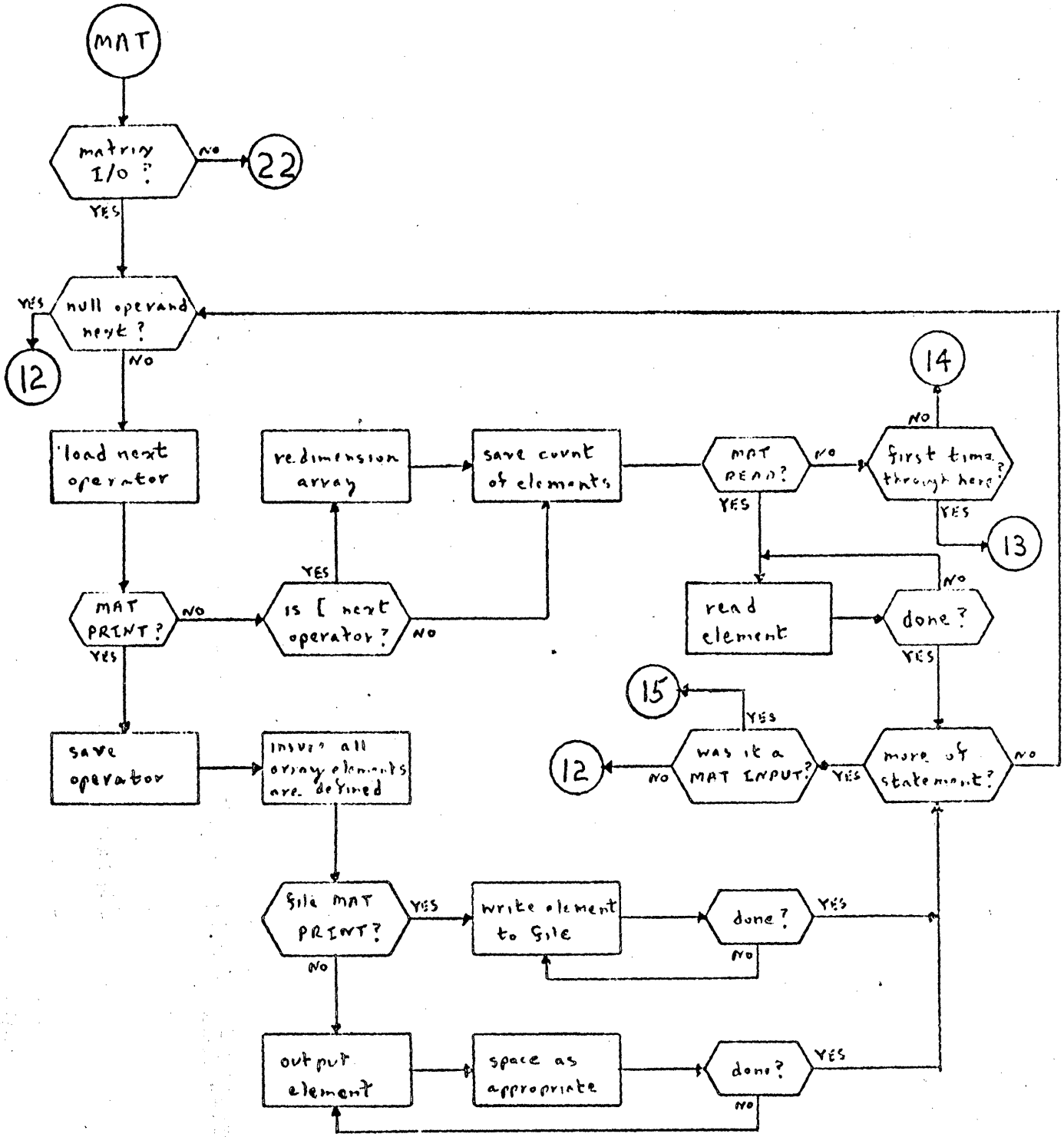
B. Selected Statement Types

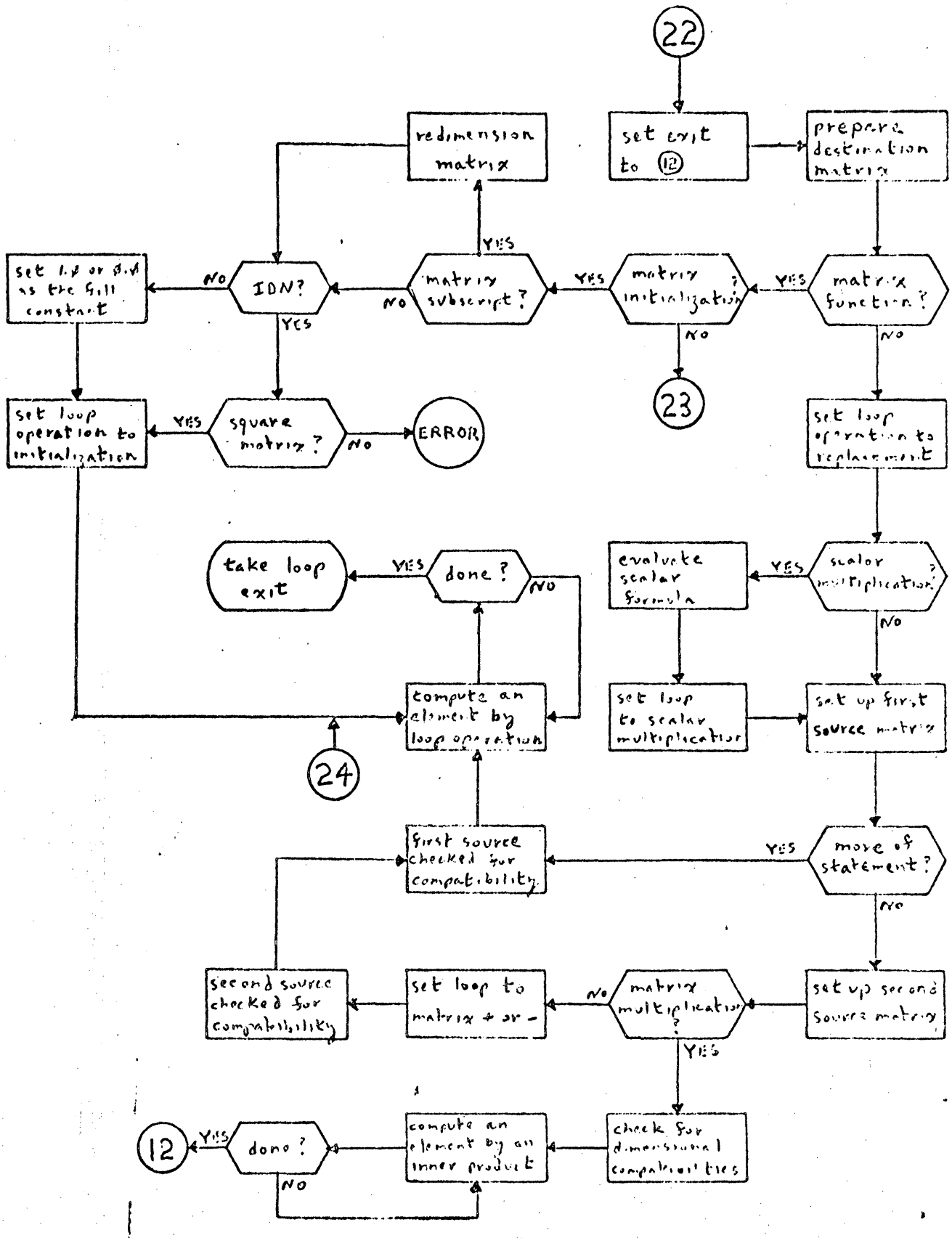


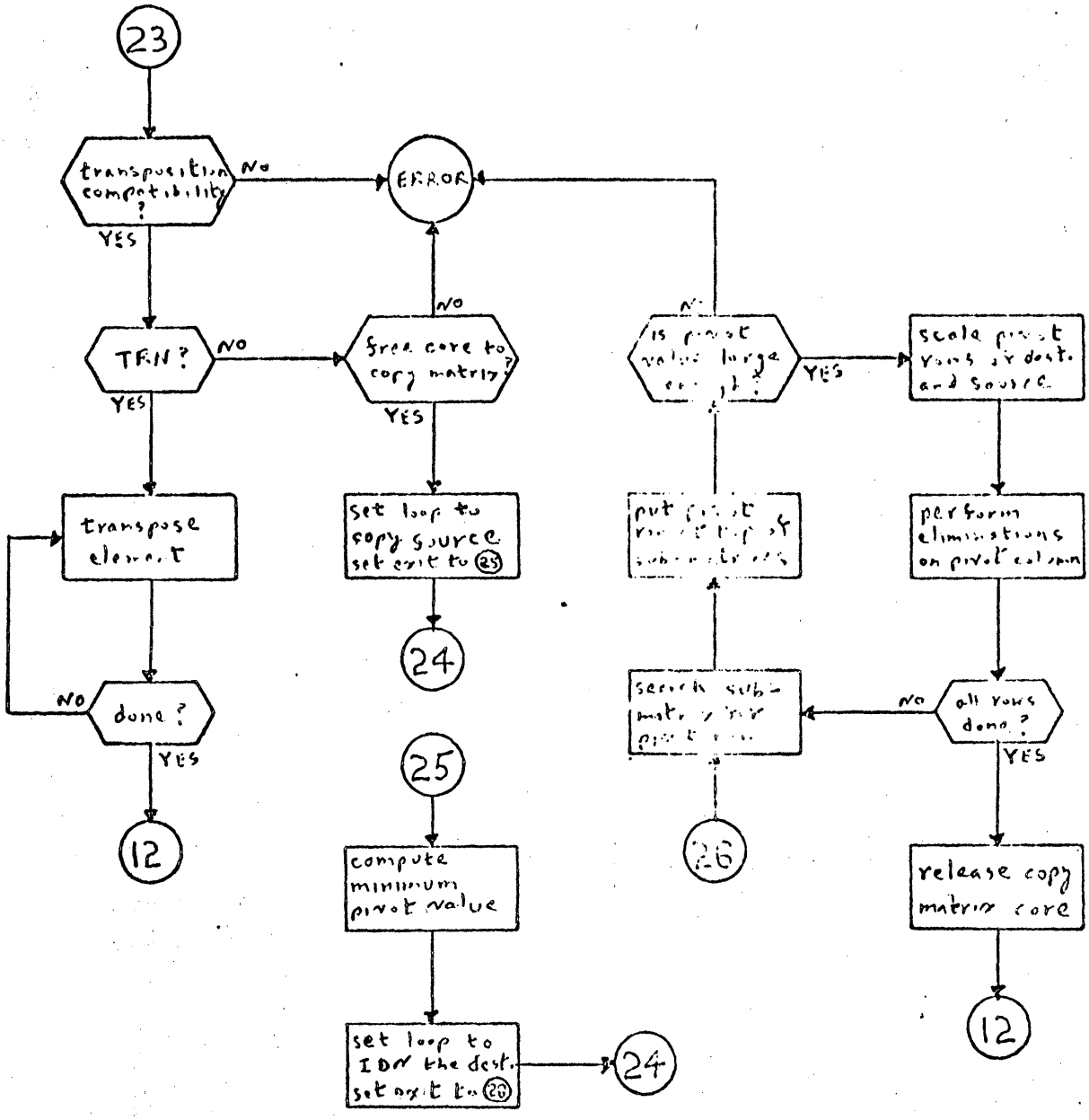

identify for-variable
for-stack entries to examine?
NO
add entry to for-stack
YES
FOR
does next entry match for-variable?
YES
is it the top entry?
YES
put for-variable value address in entry
for-variable set to initial value
NO
NO
move pointer past entry
move old entry to top of stack
enter address of following statement
compute step size
YES
step size given?
save limit value in entry
NO
load 1.0
save step size in entry
set to branch past associated NEXT
eliminate for-stack entry
YES
limit value exceeded?
step size added to for-variable
NO
YES
12
set to branch to statement following FOR
is it the top entry?
NO
eliminate all higher entries
YES
NO
for-stack entries to examine?
YES
does next entry match for-variable?
NO
move pointer past entry
NEXT
identify for-variable


INPUT
13
16
17
disable array element counter
set flag for MAT or INPUT
request input
call from MAT?
NO
get variable address
YES
18
YES
was an array filled?
YES
found?
request a number
NO
string variable?
print a warning
NO
YES
12
NO
buffer empty?
YES
is an INPUT satisfied?
" or sign found?
NO
YES
15
14
NO
print a warning
YES
did over/ underflow occur?
print warning
request input
prepare for string transfer
16
NO
16
YES
buffer empty?
print error
request input for retry
transfer string from input buffer
NO
YES
buffer empty now?
YES
comma next?
" or ↵ next?
NO
was length of transfer specified?
YES
all requested characters transferred?
YES
NO
NO
NO
17
print a warning
scan to " or ↵
16
fill rest of string with blanks
YES
was length of transfer specified?
NO
set logical length to actual length


PRINT
file PRINT?
NO
YES
turn on EOL flag
statement exhausted?
YES
12
NO
get next operand
turn off EOL flag
19
null ?
NO
YES
move to next print field
file PRINT?
NO
YES
EOL flag set?
NO
YES
statement exhausted?
NO
get next operator
, ?
YES
NO
; ?
YES
NO
12
21
emit carriage return - line feed
write item on file
20
null operator ?
YES
NO
END?
YES
NO
literal string ?
NO
YES
19
evaluate formula
string variable ?
YES
NO
prepare print string
turn on EOL flag
file PRINT?
NO
YES
20
file PRINT?
YES
NO
output number
was it a TAB ?
NO
YES
21
output string
turn on EOL flag


MAT
matrix I/O ?
NO
22
YES
null operand next ?
YES
12
NO
load next operator
MAT PRINT ?
NO
YES
save operator
insure all array elements are defined
is [next operator ?
YES
NO
redimension array
save count of elements
file MAT PRINT ?
YES
NO
write element to file
done ?
YES
NO
output element
space as appropriate
done ?
YES
NO
MAT READ ?
NO
YES
first time through here ?
NO
YES
14
13
read element
done ?
NO
YES
15
was it a MAT INPUT ?
YES
NO
12
more of statement ?
YES
NO


22
set exit to (12)
prepare destination matrix
redimension matrix
matrix function?
matrix initialization?
matrix subscript?
IDN?
set I.V or B.V as the fill constant
YES
NO
23
set loop operation to initialization
square matrix?
ERROR
set loop operation to replacement
scalar multiplication?
evaluate scalar formula
take loop exit
done?
compute an element by loop operation
24
set loop to scalar multiplication
set up first source matrix
more of statement?
first source checked for compatibility
set up second source matrix
matrix multiplication?
set loop to matrix + or -
second source checked for compatibility
check for dimensional compatibilities
compute an element by an inner product
done?
12


23
transposition compatibility ?
NO
YES
ERROR
TRIN ?
NO
YES
free core to copy matrix?
NO
YES
is pivot value large enough ?
NO
YES
scale pivot rows in dest. and source
transpose element
set loop to copy source set exit to 25
put pivot row of sub-matrices
perform eliminations on pivot column
NO
done ?
YES
24
search sub-matrix for pivot row
NO
all rows done ?
YES
25
26
compute minimum pivot value
release copy matrix core
12
set loop to IDN the dest. set exit to 26
24
12

SUPPLEMENTARY NOTES ON BASIC

I SYNTAX

The general process of analyzing an input to the language processor is displayed in the section on flow charts. The annotations in the listing explain the actions of the subroutines, while the core map and section on internal representation describe the objects/structures being created or manipulated. The BASIC syntax, in conjunction with the listing, explains the method of identification and recognition of legitimate BASIC statements from the input string.

II Phase 2

A. Compilation

The preliminary section of CMPLE prepares for execution of the program following a successful compilation.. Null programs require no processing. If a sequence number follows the RUN (e.g., RUN - 22Ø) the interpreter's program counter is set to the first statement whose sequence number equals or exceeds the reference, otherwise it is set to the first statement of the user program. If the program is already compiled (as when a program is RUN twice without intervening program modification) PBPTR is set back to the first word following the value table and phase 2 simply reinitializes all of the variables to 'undefined'. Otherwise FILTB is set to Ø so PRNST will not terminate compilation by mistaking it for decompilation.

The symbol table is then built as explained in the listing (Refer to the flow chart for general logic flow and to BASIC Variable Storage Allocation for a visual example.) During compilation SPTR points to the program word being processed and VALTB is either -1 or a pointer to the <FILES statement> if one exists. An error in compilation will cause a call to DCMPL to restore the source form of the program followed by a call to the error routine. If after a successful compilation a <FILES statement> has been found, BASIC calls the system with VALTB pointing to the second word of the statement. The system

analyzes a <FILES statement> and builds the file table, filling in the first, second, and fourth words of each entry.

The symbol routine has two entry points: SSYMT is used for functions and simple variables and ASYMT is used for array and string variables. Because the dimensionality of an array variable may not be known locally (e.g., MAT A = C) some symbols may have two entries. If this is the case, the "don't know" entry will always be farther down in the table (i.e., have a higher core address) than its dimensioned counterpart.

B. Value

VALUE is responsible for detecting deficiencies in the symbol table, allocating storage for the values of symbols (i.e., building the value table), and initializing the values of all variables. Only the last of these functions is performed if a program is already compiled when a RUN command is received. The process of building the value table is described in the listing.

Several errors may be encountered while building the value table. The occurrence of a null symbol (bit pattern of Ø) in the symbol table means that an array symbol is used in the program, but never in such a way that its dimensionality can be determined. If the second word of a function entry is zero, no <DEF statement> for that function appears in the program. Arrays of more than 25ØØ elements are not allowed. For all errors the program is decompiled before the call to the error routine.

C. Decompilation

Programs are decompiled when any error occurs during compilation, building of the file table, building of the value table, or when the program is to be modified or saved in the user library. Since in the first of these only a portion of the program is compiled, the pointer SPTR is used to determine how much to be decompiled (A fully compiled program always has SPTR pointing to the first word following the program). The process is explained in the listing.

D. The routine PRNST

PRNST is used by both CMPLE and DCMPL to scan the program and skip over those portions not affected by compiling. One outstanding pecularity should be noticed. PRNST assumes responsibility for recognizing a ⟨FILES statement⟩ in a program. If a second ⟨FILES statement⟩ is found during compilation the following occurs: 1.) PRNST calls DCMPL 2.) DCMPL calls PRNST 3.) the first ⟨FILES statement⟩ is found and treated as if compilation were taking place 4.) the second ⟨FILES statement⟩ is found and DCMPL is called again, but the first call set CFLAG[1] = Ø so this call returns immediately and PRNST exits to the error routine.

III EXECUTION

A. Main Loop

Upon completion of the value assignment in phase 2, control transfers to XEC. FCORE saves a pointer to the first word following the value table (used in repeated RUNS of a program). After printing the program name XEC proceeds to initialize the file table. A 64-word buffer is allocated for each file and pointers to the word following it are placed in words 5 and 6 of the file table. The disc address of the record in the buffer (word 3) is set to -1 to indicate that no record is present. Word 7 is set to Ø, indicating that no end-of-record/end-of-file exit has been specified. If the file is read-only a message to this effect is printed, following the program name.

Following the preparation of files the initial execution status is set. The initial execution stacks are claimed from free user space and pointers are set to the first constant of the first ⟨DATA statement⟩ , if such exists. The internal print position counter (CHRCT) is set to zero by outputting a carriage return. Phase 2 has already set the BASIC program pointer (PRGCT) to the first statement to be executed.

Execution of a statement simulates the execution of an instruction on a 'BASIC machine'. The sequence number of the statement referenced by PRGCT is saved for possible use by the error routine. PRGCT is advanced to reference

the following statement. The type of the current statement is used to branch to the appropriate routine via a jump table. Individual statement routines return to the top of the loop.

B. Statement execution

<LET statement> execution consists simply of evaluating the formula, which is known to contain at least one assignment operator and to have type compatibility (numeric vs. string) by its acceptance by phase 1.

<IF statement> execution forks on the symbol following the IF. The construction 'IF END' causes the following: the file reference is evaluated and tested for existence as one of the program's requested files; if a legitimate reference, the statement reference following the THEN is placed in the end-of-file word of the file's table entry. If not 'IF END', the decision formula is evaluated and if true the statement reference replaces the value of the interpreter's program counter, PRGCT, via the GOTO mechanism.

<GOTO statement> execution consists of choosing a statement reference to replace the program counter. For simple GOTO's this is done trivially; for multi-branch GOTO's this is done by evaluating the index formula and choosing the statement reference in the corresponding list position. If the index value lies outside the list of statement references, the program counter remains unchanged.

<GOSUB statement> execution follows the pattern for the GOTO except that after choosing the new value for the program counter, the old value is saved on the return stack (stack overflow generating an error condition).

<FOR statement> execution opens an active program loop. The for-stack is searched for an entry with the same for-variable; if found, the entry is eliminated (i.e., the previous <FOR statement> with this for variable is closed). A new entry is set on top of the for-stack (extending the for-stack by six words if no entry was eliminated) and a pointer to the for-variable's value entry is

put into word 1. Since the first formula in the FOR contains an assignment operator, the formula evaluator, FORMX, initializes the for-variable when it determines the initial value. A reference to the statement following the ⟨FOR statement⟩ is put into word 6 of the for-stack entry (the start-of-loop address). Words 2 and 3 save the result of evaluating the limit value formula. If a step size formula appears explicitly it is evaluated, otherwise 1.Ø is taken as the step size. In either case the value of the step size is left in words 4 and 5 of the for-stack entry. The program counter is set to the statement following the associated ⟨NEXT statement⟩ and control transfers to the ⟨NEXT statement⟩ execution code to compare the initial and limit values (see flow chart).

⟨NEXT statement⟩ execution decides whether to iterate a loop or close it. The for-stack is searched for an entry with the same for-variable. If none is found the statement is ignored and control passes to the following statement. If the entry is found, any entries above it (more recent entries) are eliminated; i.e., they are assumed to belong to nested loops which were not closed by exceeding their limit value but exited otherwise. The value of the for-variable is then incremented by the step size and the new value tested by subtracting the limit value and using the sign of the step size to determine whether a non-negative or non-positive result indicates 'success'. If the result is 'success', the program counter is loaded from word 6 of the for-stack entry (the reference to the statement following the ⟨FOR STATEMENT⟩). If the result is not 'success', the for-stack entry is eliminated. At this point the program counter already points to the statement following the ⟨NEXT statement⟩ so exit is simply to the main execution loop.

⟨RETURN statement⟩ execution merely loads the program counter from the top entry of the return stack. An error condition is generated if the return stack is empty.

⟨INPUT statement⟩ execution assigns values to the input list for both INPUT and MAT INPUT. INITF = Ø and MCNT is meaningless when executing an ⟨INPUT statement⟩ ; For MAT INPUT, INITF = -1 and MCNT holds the number (in 2's complement) of elements of the current array as yet unassigned values.

IFCNT holds the ordinal number of the current item in the current record (Note that IFCNT is not cumulative over the entire execution of a statement requesting input unless the request is met entirely by one line from the teletype.).

The general approach in execution is to determine the address and type of a variable in the input list and then attempt to satisfy it from the input record. When an error occurs in the above process, it is explained along with any necessary corrective action and the value assignment is attempted again, so that errors in the input record will not terminate program execution. For simple input if the next variable in the list is of numeric type its value table address is placed into SBPTR; for array input the base address of the array is put into SBPTR. After filling a simple variable the next variable from the list is taken and a new address generated; after filling an array element SBPTR has been advanced to the next element by the numeric input routine so no new address need be calculated. When MCNT rolls over to zero (an array has been filled) control exits to the MAT INPUT code, which may return with another array's base address in SBPTR and MCNT reset appropriately. If the input record is empty but the variable list is not yet exhausted a request for additional input is made (signified by '??' rather than the initial '?'). SERR is needed as a flag to indicate if under/overflow occurred while converting the latest numeric input, since the error message will have destroyed any additional information in the input record. When looking for a number, the input record is scanned for the first sign (+ or -), digit, or decimal point, which begins the number. Any other characters will be ignored except the ", which will generate a recoverable error.

String input requires fairly complicated analysis of the data transfer. If the string variable does not specify the transfer length (does not have a double subscript), then the next string in the input record is transferred in its entirety and the logical length of the variable set appropriately. If the next string does not fit, a message is printed and a new string value requested. If the string variable specifies the transfer length then exactly that much of the next string in the input record will be transferred, either truncated or extended by blanks as necessary to achieve the specified length. The 'next strin

in the input record begins with the next non-blank character or, if it is a ", the following character, blanks included. The string ends with the first " (which is not part of the string) encountered or with the carriage return (also not part of the string) if no " appears.

Every data item in the input record must be followed by a comma or carriage return and a comma must be followed by another data item. Failure to observe the above will generate recoverable errors. INTMP holds the type of data being sought, INTMP = Ø for a number or INTMP # 0 for a string, and is used by the error recovery code to prepare for the retry.

⟨READ statement⟩ execution assigns values to variables in the list. FDATA is primed to obtain values from either a file or the ⟨DATA statement⟩ s, depending on the presence or lack of a file reference following the READ. A mismatch in type between the variable and the next data item, or a string too long to fit into its designated destination, will generate an error and terminate execution.

⟨PRINT statement⟩ execution consists of identifying items in the print list and sending the appropriate media equivalent to the teletype or disc file. An initial file reference identifies the statement as a file write and turns off the end-of-line mode; its absence identifies a teletype write and turns on the end-of-line mode. A comma or semicolon turns off the end-of-line mode and generates enough blanks to advance to the next field of 15 characters, if a teletype write. A literal string is written as a string of characters, less quotes, and turns on the end-of-line mode if a teletype write. An END writes an end-of-file mark on the file; it cannot occur in a teletype write. Formulas in the print string are evaluated and the results examined. Formulas which are string variables evaluate to their contents, which is then treated as a literal string. If not a string variable but within a file write statement, the floating point value of the formula is written on the file in its two-word binary representation. If a teletype write, floating point values are converted to an ASCII character string of the decimal equivalent. TAB can only occur in a teletype write; the evaluation of the TAB itself produces the desired action, so the value returned is thrown away, along with a following comma if one exists. For a teletype write all formulas

turn on the end-of-line mode. If the end-of-line mode is on after processing the last print item, a carriage return-line feed is printed (This can only occur in a teletype write.).

Before writing a quantity BASIC insures that sufficient space is available to accommodate it. CHRCT keeps track of the current print position on the teletype line (Ø-71). If the character string sent to the teletype would require non-blank characters to be printed past position 71, a carriage return-line feed is output first and CHRCT set to Ø. If an item sent to a file requires more words than remain in the current record, BASIC automatically advances to the next record if in serial mode or exits to the end-of-record code if in record mode.

⟨RESTORE statement⟩ execution resets the pointers to the DATA block. Beginning at the statement specified, or at the first statement in the program if none is specified, the pointers are set to the first ⟨DATA statement⟩ found, or to the out-of-data condition if none is found.

⟨END statement⟩ and ⟨STOP statement⟩ execution terminates the program run. Since each requested file has a 64-word buffer in core, the last record written on a file does not exist on the disc in its updated form. Thus END and STOP must force the buffer of each read/write file onto its proper disc sector. Following this, the word DONE is sent to the teletype and control exits to the scheduler.

⟨MAT statement⟩ execution involves many disparate tasks. The forms of the ⟨MAT statement⟩ may be classified as array I/O, array assignment, array initialization, and the array functions TRN and INV. For conciseness in coding, all forms other than array I/O use some common program segments.

Array I/O prepares each array in the list in the same fashion. SBPTR is set to the dynamic dimensions of the array (base address -2) and the operator following the array identifier is picked up for examination. At this point

MAT PRINT follows a separate path than MAT READ and MAT INPUT. The following operator is noted as spacing the elements (comma or end-of-statement) or packing them (semicolon). VCHK examines the array and generates an error if any of its elements have value 'undefined'. The dynamic row and column lengths are saved in 2's complement. If the MAT PRINT references a file, the array elements are written one by one in rows, each element in its two-word binary form. If the MAT PRINT references the teletype, rows are double spaced and the elements within a row are spaced or packed as noted above, each element in its ASCII decimal form. Both MAT READ and MAT INPUT redimension the array if the following operator is a left bracket (i.e., begins a matrix subscript). MCNT is set to the number of elements in the array, in 2's complement. MAT READ calls FDATA for element values while MAT INPUT transfers to the ⟨INPUT STATEMENT⟩ execution to obtain element values. MTØ acts as a flag for MAT INPUT, differentiating the first call for input from subsequent calls and saving the input character following the last element value used from the input record. After completing I/O on an array, a common section of code prepares the next array in the list or, if no more remain, terminates the statement execution. MAT INPUT returns to the input code to clean up there, MAT PRINT and MAT READ return directly to the main execution loop.

Array assignment consists of preparing the destination and source arrays and executing a loop which assigns the destination array elements one by one. The general procedure is to assign a jump to the element computation code to MOP, an exit address to MEXIT to use after completing the destination array, and a count of the elements to MCNT, in 2's complement. The code to compute an element returns to MLOP1, MLOP2, or MLOP3 depending on the number of arrays involved which require updating of the element address. Each operation checks the dimensions of the arrays involved to insure that the operation is well-defined; and all elements of the source matrices are checked to make sure none have value 'undefined'. Matrix multiplication does not use the element computation loop, instead it uses row and column counters to tell when it is done and computes destination array elements by inner products of the rows and columns of its source matrices.

Array initialization also uses the element computation loop. The initialization program first redimensions the destination array (if a matrix subscript is given) and then chooses the appropriate constant for the element values. IDN acts like ZER except that it insists that the destination array be 'square' and sets a special counter to choose 1.Ø for the value of main diagonal elements.

TRN and INV are handled apart from the other matrix functions. For both of these, the elements of the source matrix are checked against the 'undefined value'. The source and destination matrices are then checked for transpositional compatibility. If TRN, then proceed to transfer the columns of the source matrix to the rows of the destination matrix.

INV uses the Gauss-Jordan algorithm with row pivoting. This procedure converts a copy of the cource matrix into the identity matrix and converts an identity matrix into the inverse by applying the same set of operations to both. Since the source matrix is destroyed in the process, it is first copied into free user space and the copy treated thereafter as the source. A side effect of the copying produces the element of largest absolute value, which is used to compute a lower bound on the allowable magnitude of pivot elements. INV then calls IDN to set the destination matrix to an identity matrix, having the side effect of checking that the matrix is square.

Diagonalization of the source matrix and production of the inverse now proceeds on a row-by-row basis. The next unreduced column of the source is searched for the pivot element (the largest in magnitude). If necessary, rows are swapped to put the pivot element on the main diagonal (the corresponding rows of the destination matrix must also be swapped). If the pivot element is smaller in magnitude than the previously computed lower bound, the matrix is too nearly singular to invert and execution is terminated. Otherwise, the pivot rows of both matrices are divided through by the pivot element. Now all other elements in the pivot column are eliminated by subtracting the appropriate multiple of the pivot row from each of the other rows. Advantage

is taken of those pivot column elements which are already zero and of the fact that elements of the pivot row to the left of the pivot column have been set to zero by previous steps. After diagonalization of the source matrix and consequent creation of the inverse, the user space occupied by the source copy is released.

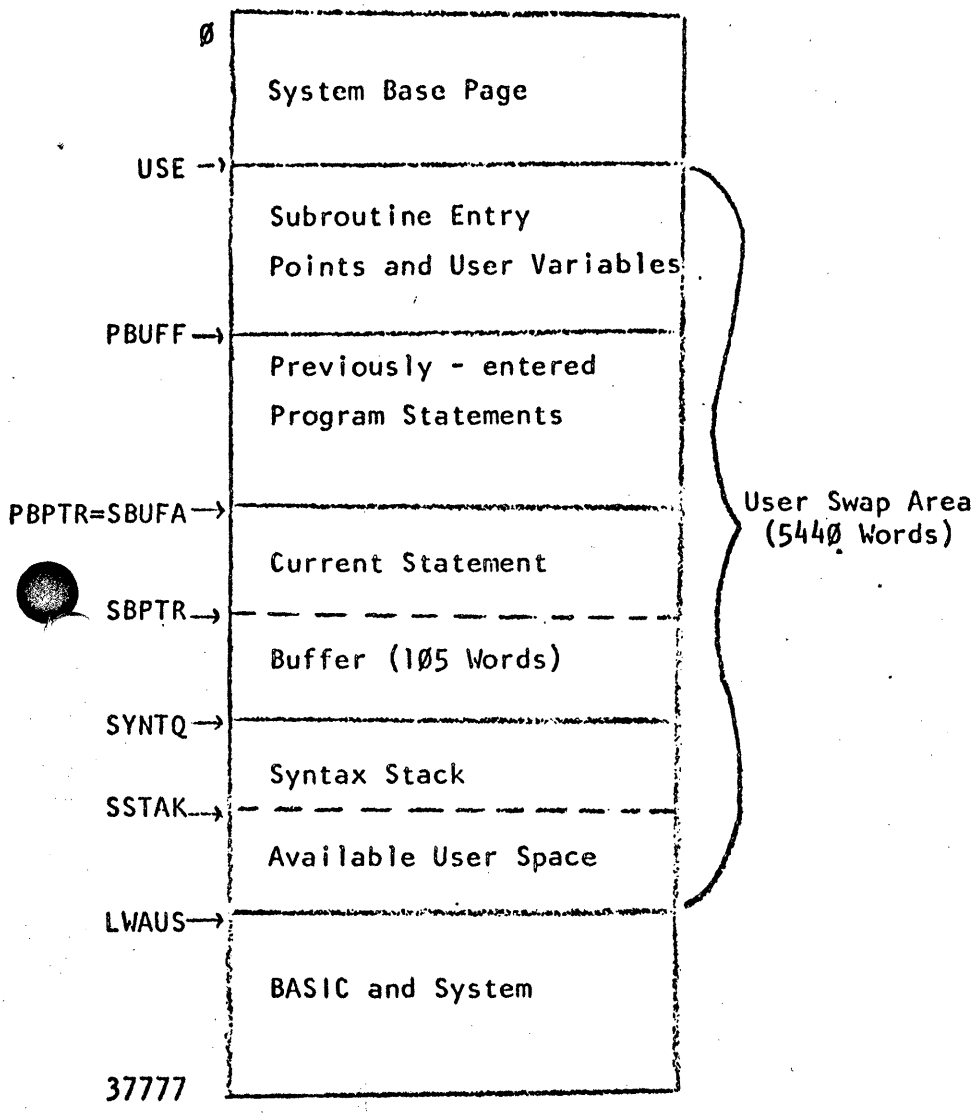
The other statement types are declarative in nature. Execution of them consists solely of skipping over to the statement following.

NOTES ON THE ERROR ROUTINES

Errors are handled by routine SERR, reached by a jump through the base page table beginning at SERRS. A JSB SERRS + i,I signifies detection of error i. The alternative bases RERRS and WERRS are conveniences to denote subsections of the table used for run-time errors and warning-only errors. The actions taken by SERR are explained in the listing; but notice that the 'BAD INPUT' error is singled out, its processing is completed by the input execution routine upon return from SERR.

Syntax errors detected while in tape mode are handled by accepting error psuedo-statements in place of the erroneous statements. Since these psuedo-statements will be replaced by any subsequently received statements with the same line number, provision is made in FNDPS, which returns the location of a statement when given its sequence number, to decrement the error counter (ERRCT) whenever the statement found is an error psuedo-statement (an error psuedo-statement will only be found by FNDPS when another statement with the same sequence number is ready to replace it). Over/underflows detected during number conversions in syntax mode cause warning messages to be issued only after accepting the statement, if it is otherwise correct. Since no printing can be done while in tape mode, the routine CHOUF suppresses setting of the flag and these potential errors are not reported when in tape mode.

SYNTAX (Phase 1)

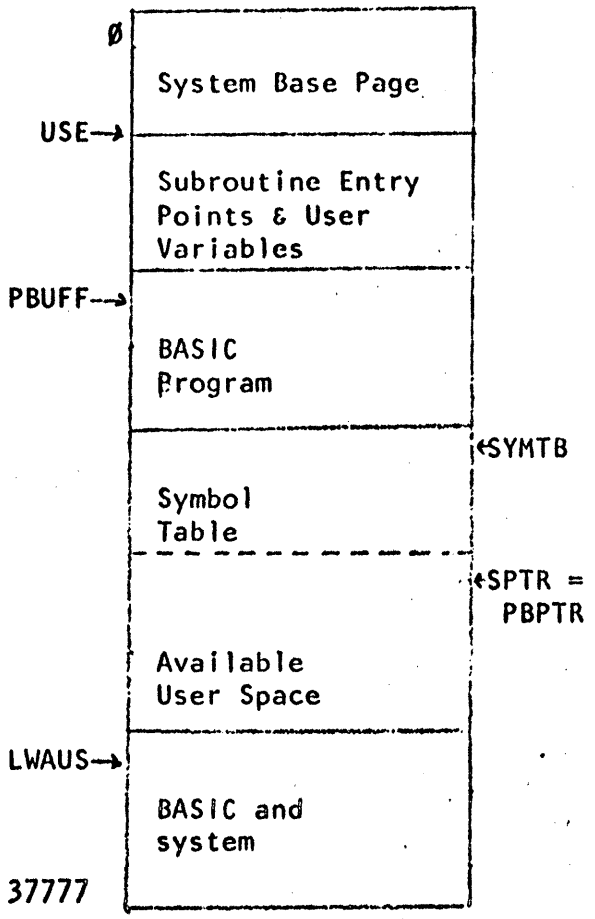


Pointers

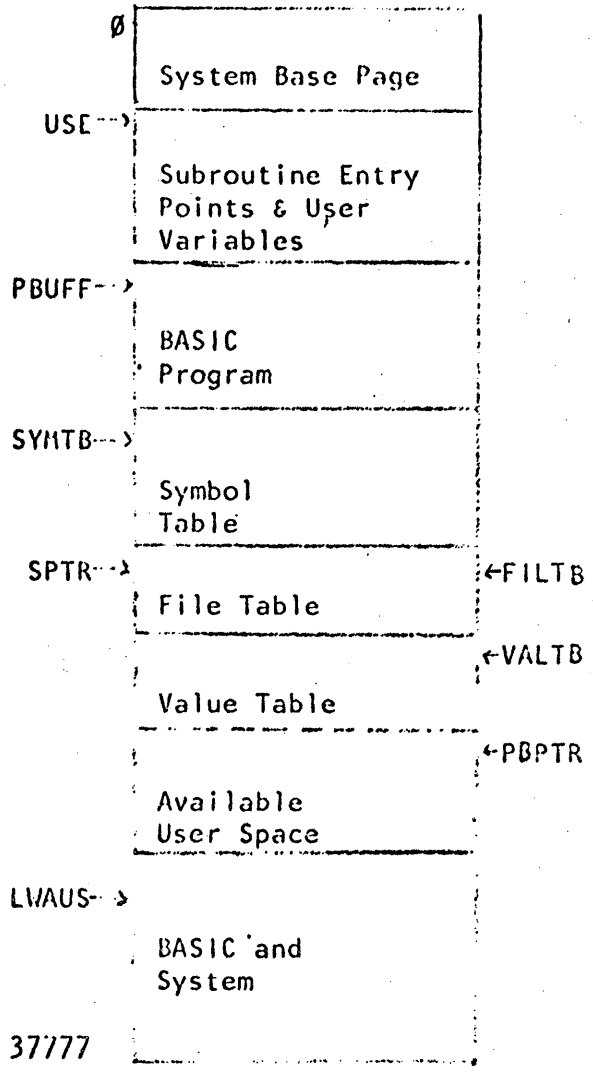
USE	Fixed, first word of user swap area.
PBUFF	Fixed, first word of program space.
SBUFA	Variable, first word of statement being syntaxed.
PBPTR	Variable, first word of program space not used by previously accepted program statements.
SBPTR	Variable, first word not used by statement being syntaxed.
SYNTQ	Variable, first word of syntax stack.
SSTAK	Variable, last word of syntax stack.
LWAUS	Fixed, first word not in user swap area.

COMPILATION (Phase II)

Compilation



Value Storage Allocation



SYMTB - Variable, first word of symbol table.

SPTR - Variable, first word not used by symbol table .

FILTB - Variable, first word of file table.

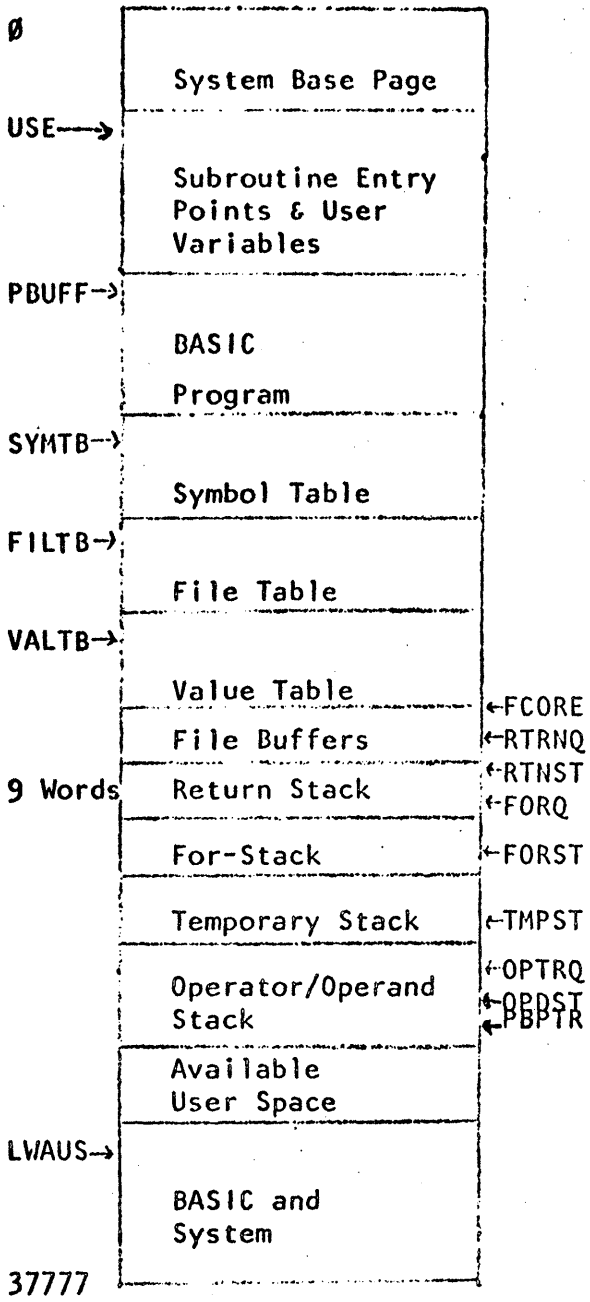
VALTB - Variable, first word of symbol value table
(FILTB = VALTB if no ‹FILES statement› is in program)

PBPTR - Variable, first word available of user space.

SYMTB and SPTR are not changed after compilation.

FILTB and VALTB are not changed after allocating value storage.

EXECUTION (PHASE III)



FCORE - Variable, first word not used by Phase II

RTRNQ - Variable, bottom of return stack (first word preceding return stack)

RTNST - Variable, top of return stack

FORQ - Variable, bottom of for-stack (sixth word preceding for-stack)

FORST - Variable, top of for-stack (points to latest 6-word entry)

TMPST - Variable, top of temporary stack (points to latest 2-word entry)

OPTRQ - Variable, bottom of operator stack

OPDST - Variable, top of operand stack.

PBPTR - Variable, top of operator stack.

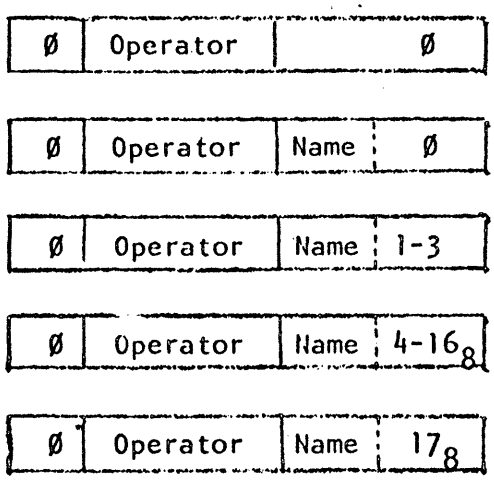
FCORE, RTRNQ, and FORQ are not changed after initiating execution.

Entries on the operator and operand stack are one word each and interleave (i.e., alternate words belong to one stack). All stacks beyond the return stack grow and shrink as needed so long as user space is available.

BASIC Internal Representation

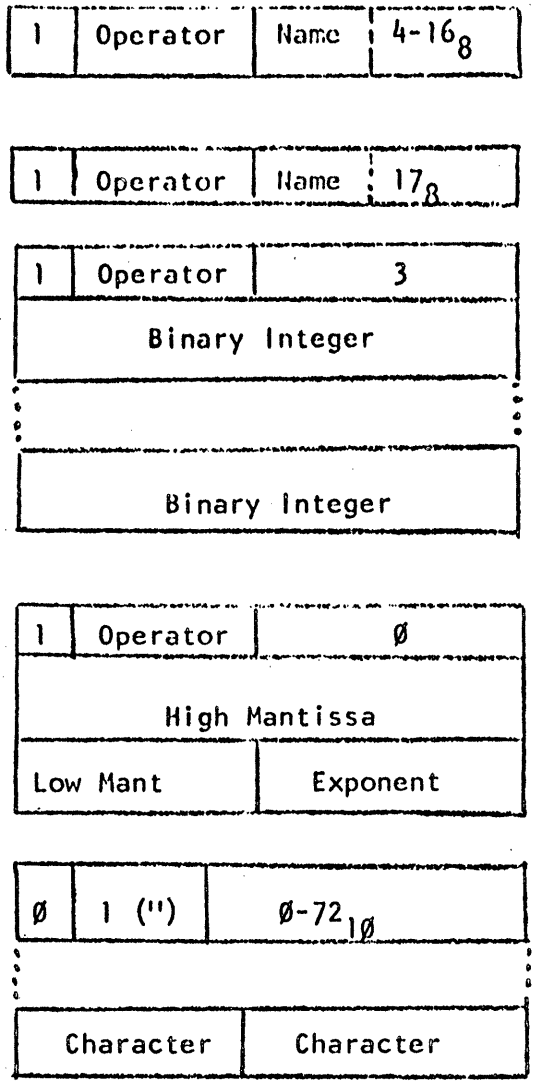
BASIC statements are represented internally by the sequence number followed by the length in words (including the sequence number and length words) followed by the statement body. The statement body is composed almost entirely of operator-operand pairs which occupy from one to three words each. Null operands and operators are used when necessary to maintain the operator-operand correspondence. The operator resides in bits 14-9 of a word; the operand uses bit 15, bits 8-Ø, and sometimes whole additional words immediately following.

'Variable' Operands



Bits 8-Ø are generally divided into two fields as follows: a name field (bits 8-4) and a type field (bits 3-Ø). The name field holds a value between 1 and 32_8 corresponding to A-Z (for functions, corresponding to FNA through FNZ). A type of Ø identifies a string variable (e.g. 3,Ø represents C$). Types 1 and 2 identify array variables of dimensionality one and two respectively (e.g. 4,2 represents D[*,*]) while type 3 identifies an array variable whose dimensionality cannot be determined by its immediate context. Type 4 identifies a simple variable with no digit (e.g. 1,4 represents A) while types 5 - 16_8 identify simple variables whose names include the digit Ø - 9_{10} respectively (e.g. 6,7 represents F2). Type 17_8 identifies a programmer-defined function (e.g. 32_8, 17_8 represents FNZ).

'Constant' Operands



A parameter (which can only appear inside a <DEF statement>) differs from a simple variable only in that bit 15 is set. The name of a pre-defined function may range, in the standard system, from 1 to 16_8 or 24_8 to 30_8 (TAB to TYP or ZER to TRN). A flagged (bit 15 set) operand of 3 identifies either a formal dimension in a <DIM statement> (value in following word) or a branch address list (one or more statement sequence numbers in the following words). A flagged operand of Ø indicates that the following two words hold a floating-point constant (all numerical constants within a program are so represented). The operator with internal code 1 is ", which signals the start of a string constant. The operand portion of the word has a value from Ø to $72_{1Ø}$, indicating the number of characters in the constant. The string follows, two characters per word, and the closing " is not explicitly represented internally.

The table below gives the internal representation of the BASIC operators. Those operators which manipulate the formula evaluation stack during execution have associated priorities. All numbers are in octal notation.

BASIC Operators

CODE	PRIORITY	ASCII	CODE	PRIORITY	ASCII	CODE	ASCII
Ø	Ø	(end-of-formula)	26	5	<	54	FOR
1		"	27	5	#	55	NEXT
2		,	3Ø	5	=(equal)	56	GOSUB
3		;	31		(unused)	57	RETURN
4		# (file)	32		(unused	6Ø	END
5		(unused)	33		(unused)	61	STOP
6		(unused)	34		(unused)	62	DATA
7		(unused)	35		(unused)	63	INPUT
1Ø	1	)	36	4	AND	64	READ
11	1	]	37	3	OR	65	PRINT
12	13(1)	[	4Ø	6	MIN	66	RESTORE
13	13(1)	(	41	6	MAX	67	MAT
14	11	+(unary)	42	5	<>	7Ø	FILES
15	11	-(unary)	43	5	>=	71	'IMPLIED' LET
16	2	,(subscript)	44	5	<=	72	(unused)
17	2	=(assignment)	45	11	NOT	73	(unused)
2Ø	7	+	46		LET	74	OF
21	7	-	47		DIM	75	THEN
22	1Ø	*	5Ø		DEF	76	TO
23	1Ø	/	51		REM	77	STEP
24	12	↑	52		GOTO		
25	5	>	53		IF		

Some examples of BASIC statements in their internal form are given below. Note that actual function parameter formulas, ⟨DEF statements⟩ formulas, and subscript formulas appearing in ⟨MAT statements⟩ require end-of-formula operators to signal their end whereas most formulas end either with the first operator which does not manipulate the formula evaluation stack or with the end of the statement. Note also that constants are considered signed only within a ⟨DATA statement⟩. ASCII numbers are decimal, internal numbers are octal in the presentation below.

```
1Ø      LET  W1 = Y = (B = C) ↑ 3*A[1,J+K]

   12                    sequence number        2Ø DIM A[5], C[6,12]
   21                     length

Ø | 46 | 27 | 6          LET W1                  24
Ø | 17 | 31 | 4          =    Y                  14
Ø | 17 |    | Ø          =                     Ø | 47 |  1 | 1
Ø | 13 |  2 | 4          (    B                1 | 12 |    | 3
Ø | 3Ø |  3 | 4          =    C                  |      5
Ø | 1Ø |    | Ø          )                     Ø | 11 |    | Ø
1 | 24 |    | Ø          ↑                     Ø |  2 |  3 | 2
Ø3ØØØØ                   3.Ø                   1 | 12 |    | 3
ØØØØØ4                                           |      6
Ø | 22 |  1 | 2          * A                   1 | 16 |    | 3
Ø | 12 | 11 | 4          [ 1                     |     14 |
Ø | 16 | 12 | 4          , J                   Ø | 11 |    | Ø
Ø | 2Ø | 13 | 4          + K
Ø |  Ø |    | Ø          (end-of-formula)
Ø | 11 |    | Ø          ]
```

```
30 DEF FNC (X) = X + A0
```

```
      36
       7
0 | 50 |  3 | 17
1 | 13 | 30 |  4
0 | 10 |    |  0
1 | 17 | 30 |  4
  | 20 |  1 |  5
0 |  0 |    |  0
```

```
40 REM ARK
```

```
      50
       5
0 | 51 | 40
 040522
 045400
```

```
50 GOTO A OF 10, 20, 30
```

```
      62
       7
0 | 52 |  1 | 4
1 | 74 |    | 3
      12
      24
      36
```

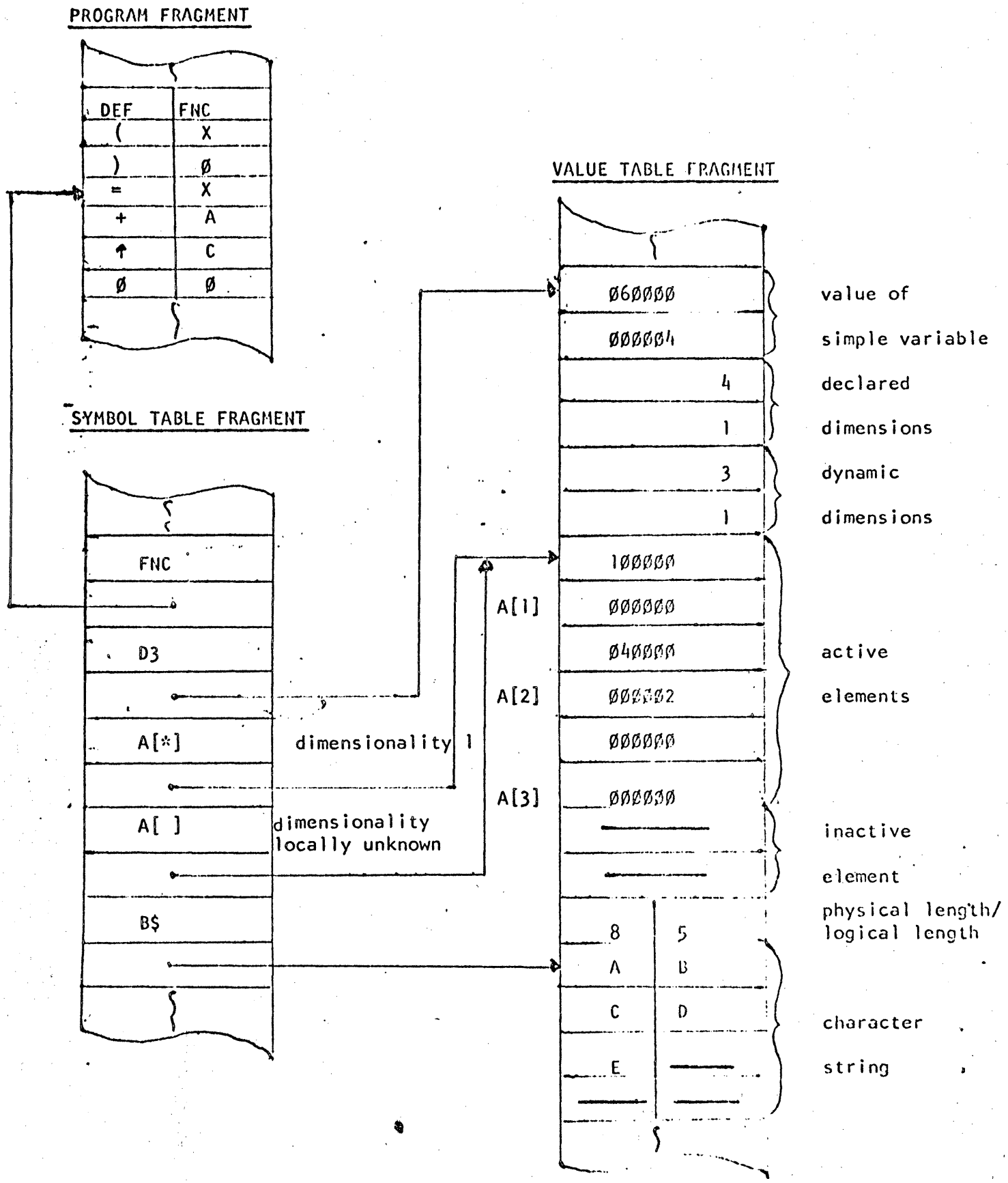
```
60 DATA -1, "ABC"
```

```
      74
      11
1 | 62 |    | 0
 100000
 000000
0 |  2 |    | 0
0 |  1 |    | 3
 040502
 041400
```

```
70 MAT READ #K;A[I]
```

```
    106
     11
0 | 67 |    | 0
0 | 64 |    | 0
0 |  4 | 13 | 4
0 |  3 |  1 | 1
0 | 12 | 11 | 4
0 |    |    | 0
0 | 11 |    | 0
```

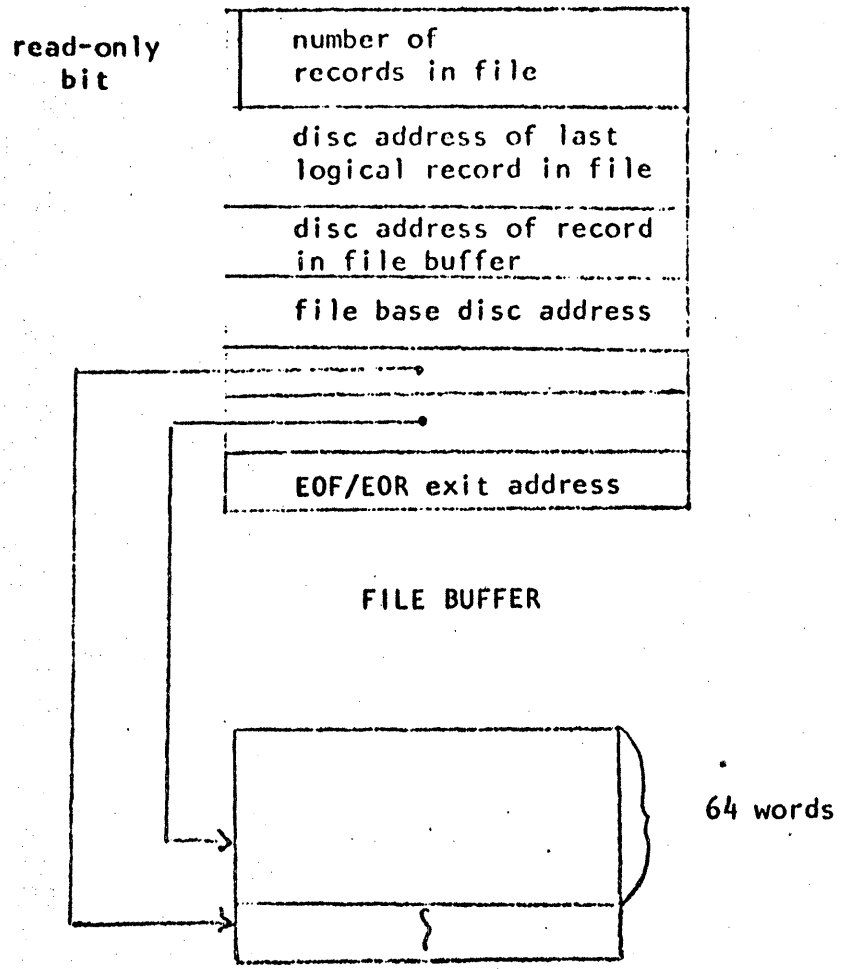
BASIC Variable Storage Allocation



The symbol table consists of two-word entries, one for each unique symbol occurring in the user's program. The first word of an entry is the internal representation of the symbol as previously described. The second word of the entry is a pointer to the value of the symbol. For a programmer-defined function the value is the defining formula in the ‹DEF statement›. The value of a simple variable is a two-word floating point number. The value pointer of an array is its base address (i.e. the address of its first element); when an array is dynamically redimensioned to occupy less than its physically allocated storage, it occupies a contiguous block justified to the low core portion of its element space. Since array symbols may not have dimensionality locally defined (e.g. MAT A=B), array symbols may have a "don't know" entry in the symbol table in addition to the dimensioned entry. Both entries have the same value pointer. The declared and dynamic dimensions occupy the four words preceding the element space in the value table. The value of a string is also its base address. A string is a character array (packed two elements per word in contrast to the two words per element for numerical arrays). Its physical (declared) length and logical (dynamic) length occupy the word immediately preceding its value space.

The value table is simply the concatenation of the values for the symbols in the program, excepting programmer-defined functions.

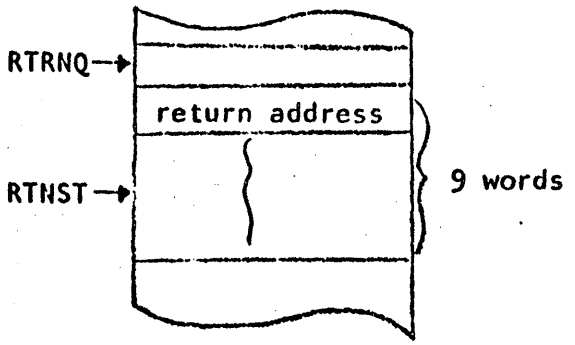
FILE TABLE ENTRY



The file table consists of one seven-word entry for each file in the ⟨FILES statement⟩. Bit 15 of the first word is set if the file was busy when requested or is a public file (available on a read-only basis). A 64-word buffer is associated with each file entry and is accessed through pointers in its file entry. An intra-record pointer designates the next portion of the record to be written or read. A fixed pointer to the first word in the buffer acts as a bound on the intra-buffer pointer.

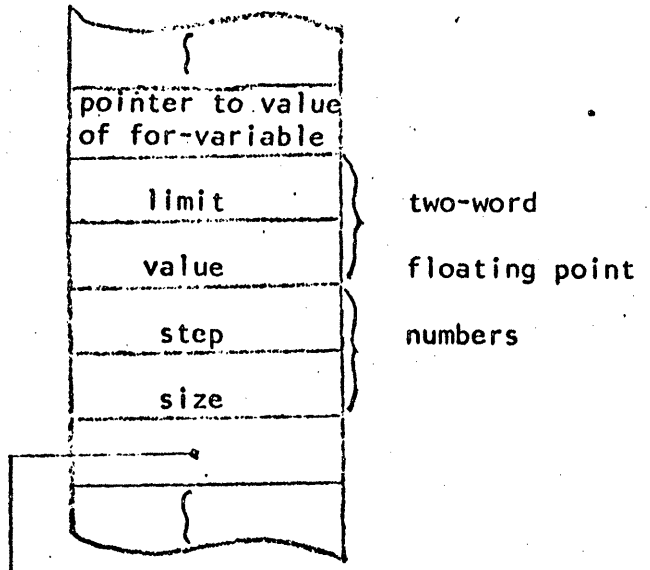
BASIC Run-Time Stacks

Return Stack



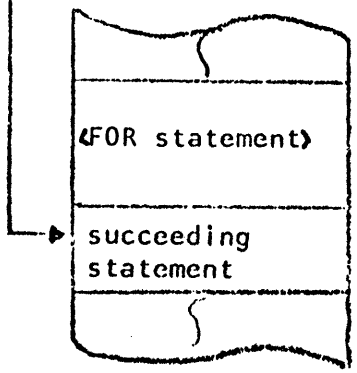
The return stack is of fixed length, holding from Ø to 9 one-word entries at any time. An entry is the absolute address of the statement following the GOSUB which placed the entry on the stack.

For-Stack Entry

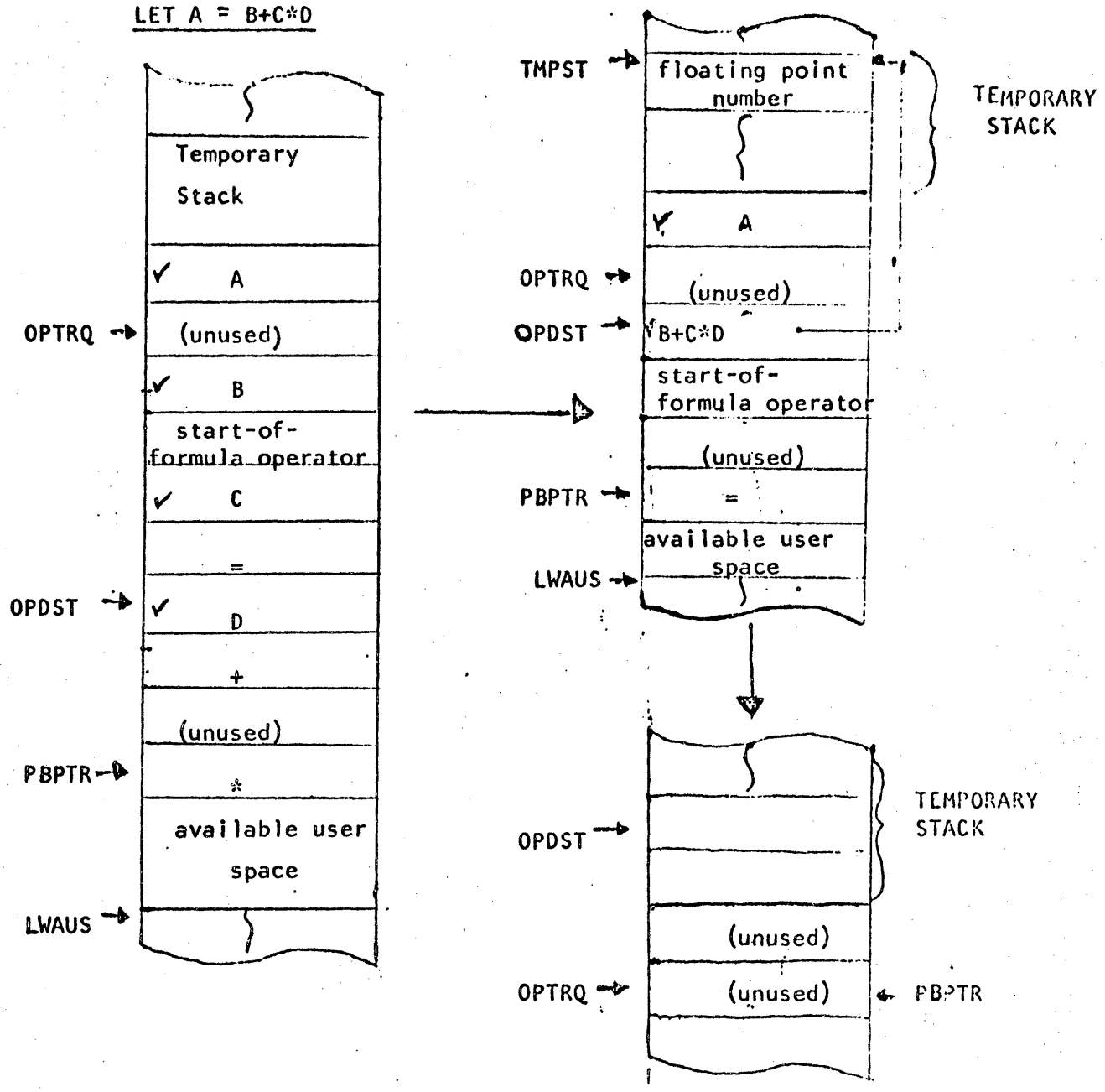


The for-stack is of variable length, containing one six-word entry for each for-loop which is currently active. Since the limit value and stop size are kept in the entry, they may not be changed within the for-loop. The value of the for-variable is the one kept in the value table, so this may be altered by statements within the for-loop.

Program Fragment



OPERATOR/OPERAND STACK FRAGMENTS



All operands (checked words) are addresses (i.e., C represents a pointer to the value of the simple variable C). Bits 7 - Ø of an operator entry contain the operators identifying code (See 'Basic Operations' Table) while bits 15-8 contain the operator's priority. Note the alternate-word structure of the stacks. The temporary stack holds intermediate values during the formula evaluation.

BASIC Language Processor Tables

The two areas of core labelled SBJTB and USER contain the mechanism allowing different users to exercise different portions of the language processor without interference. The language processor makes its subroutine calls to the labels in the area beginning with USER. The word following a subroutine entry point is an indirect jump through the appropriate address in the area following SBJTB. When a user is displaced by the system, his registers are saved at USER and the area of core from USER to PBPTR,I inclusive is dumped onto his track of the disc. Thus, a complete record of the language processor's status with respect to him is preserved. [The only things particular to a user which remain when he is swapped out are his own teletype table, teletype buffer, and the bit flags CFLAG, TERR, and TAPEF.] Since the bit flags are modified only in the bit belonging to the active user, information belonging to quiescent users is never modified.

The tables headed by PDFTB (which must be in base page), SYNTB, XECTB, and FOJT are jump tables. The method in the last three cases is to compute a decision number, add the base address of the table, and transfer through the entry thus designated. The pre-defined function table is used by the formula evaluator to enter the code for evaluating pre-defined functions.

The tables headed by QUOTE and MCBOS have several uses. Their entries are explained in the listing and their use will be explained in those routines which access them. The Error Jump Table (at SERRS) is explained along with the error routines.

• SYNTAX REQUIREMENTS OF TSB

LEGEND

::= "is defined as..."
| "or"
< > enclose an element of Time Shared BASIC

LANGUAGE RULES

1. Exponents have 1 or 2 digit integers only.
2. A <parameter> primary appears only in the defining formula of a <DEF statement>.
3. A <sequence number> must lie between 1 and 9999 inclusive.
4. An array bound must lie between 1 and 9999 inclusive; a string variable bound must lie between 1 and 72 inclusive.
5. The character string for a <REM statement> may include the character ".
6. An array may not be transposed into itself, nor may it be both an operand and the result of a matrix multiplication.

Note: Parentheses, (), and square brackets, [], are accepted interchangeably by the syntax analyzer.

Continued on the next page.

SYNTAX REQUIREMENTS OF TSB

<constant>	::=	<number>|+<number>|-<number>|<literal string>
<number>	::=	<decimal number>|<decimal number><exponent part>
<decimal number>	::=	<integer>|<integer>.|<integer>.<integer>|.<integer>
<integer>	::=	<digit>|<integer><digit>
<digit>	::=	Ø|1|2|3|4|5|6|7|8|9
<exponent part>	::=	E<integer>|E+<integer>|E-integer *(see rule 1)*
<literal string>	::=	"<character string>"
<character string>	::=	<character>|<character string><character>
<character>	::=	any ASCII character except null, line feed, return, x-off, alt-mode, escape, ←, " , and rubout
<variable>	::=	<simple variable>|<subscripted variable>
<simple variable>	::=	<letter>|<letter><digit>
<letter>	::=	A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<subscripted variable>	::=	<letter>(<sublist>)
<sublist>	::=	<expression>|<expression>,<expression>
<string variable>	::=	<string simple variable>|(<sublist>)|<string simple variable
<string simple variable>	::=	<letter>$
<expression>	::=	<conjunction>|<expression>OR<conjunction>
<conjunction>	::=	<relation>|<conjunction>AND<relation>
<relation>	::=	<minmax>|<minmax><relational operator><minmax>
<minmax>	::=	<sum>|<minmax>MIN<sum>|<minmax>MAX<sum>
<sum>	::=	<term>|<sum>+<term>|<sum>-<term>
<term>	::=	<factor>|<subterm>*<factor>|<subterm>/<factor>
<subterm>	::=	<denial>|<signed factor>

SYNTAX REQUIREMENTS OF TSB, CONTINUED

<denial>	::=	<factor>|NOT<factor>
<signed factor>	::=	+<factor>|-<factor>
<factor>	::=	<primary>|<factor>↑<primary>
<primary>	::=	<variable>|<number>|<functional>|<parameter> *(rule 2)*| (<expression>)
<relational operator>	::=	<|<=|=|#|<>|>=|>
<parameter>	::=	<letter>|<letter><digit>
<functional>	::=	<function identifier>(<expression>)| <pre-defined function>(<expression>)| LEN (<string simple variable>)
<function identifier>	::=	FN <letter>
<pre-defined function>	::=	SIN|COS|TAN|ATN|EXP|LOG|ABS|SQR|INT|RND|SGN|TYP
<source string>	::=	<string variable>|<literal string>
<destination string>	::=	<string variable>
<file reference>	::=	#<file formula>|#<file formula>,<record formula>
<file formula>	::=	<expression>
<record formula>	::=	<expression>
<array identifier>	::=	<letter>
<sequence number>	::=	<integer> *(see rule 3)*
<program statement>	::=	<sequence number><BASIC statement>carriage return
<BASIC statement>	::=	<LET statement>|<IF statement>|<GOTO statement>| <GOSUB statement>|<RETURN statement>|<FOR statement>| <NEXT statement>|<STOP statement>|<END statement>| <DATA statement>|<READ statement>|<INPUT statement>| <PRINT statement>|<RESTORE statement>|<DIM statement>| <DEF statement>|<FILES statement>|<REM statement>| <MAT statement>
<LET statement>	::=	LET <leftpart><expression>| LET <destination string>=<source string>| <leftpart><expression>| <destination string>=<source string>
<leftpart>	::=	<variable>=|<leftpart><variable>=
<IF statement>	::=	IF<decision expression>THEN<sequence number>| IF END #<file formula>THEN<sequence number>
<decision expression>	::=	<expression>| <comparison string 1><relational operator> <comparison string 2>
<comparison string 1>	::=	<string variable>
<comparison string 2>	::=	<string variable>|<literal string>

SYNTAX REQUIREMENTS OF TSB, CONTINUED

```
<GOTO statement>          ::=  GOTO <sequence number>|
                               GOTO <expression>OF<sequence list>

<sequence list>           ::=  <sequence number>|<sequence list>,<sequence number>

<GOSUB statement>         ::=  GOSUB <sequence number>|
                               GOSUB <expression>OF <sequence list>

<RETURN statement>        ::=  RETURN

<FOR statement>           ::=  FOR <for variable>=<initial value>TO<final value>|
                               <FOR <for variable>=<initial value>TO<final value>
                                                          STEP<step size>

<for variable>            ::=  <simple variable>

<initial value>           ::=  <expression>

<final value>             ::=  <expression>

<step size>               ::=  <expression>

<NEXT statement>          ::=  NEXT<for variable>

<STOP statement>          ::=  STOP

<END statement>           ::=  END

<DATA statement>          ::=  DATA<constant>|<DATA statement>,<constant>

<READ statement>          ::=  READ<variable list>|READ<file reference>|
                               READ<file reference>;<variable list>

<variable list>           ::=  <read variable>|<variable list>,<read variable>

<read variable>           ::=  <variable>|<destination string>

<INPUT statement>         ::=  INPUT<variable list>

<PRINT statement>         ::=  <type statement>|<file write statement>|
                               PRINT<file reference>

<type statement>          ::=  <print 1>|<print 2>

<print 1>                 ::=  PRINT|<print 2>,|<print 2>;|<print 3>

<print 2>                 ::=  <print 1><print expression>|<print 3>

<print 3>                 ::=  <type statement><literal string>

<print expression>        ::=  <expression>|TAB(<expression>)|<source string>

<file write statement>    ::=  PRINT<file reference>;<write expression>|
                               <file write statement>,<write expression>|
                               <file write statement>;<write expression>|
                               <file write statement><literal string>|
                               <file write statement><literal string>
                                                    <write expression>

<write expression>        ::=  <expression>|END|<source string>

<RESTORE statement>       ::=  RESTORE|RESTORE<sequence number>
```

SYNTAX REQUIREMENTS OF TSB, CONTINUED

<DIM statement> ::= DIM<dimspec>|<DIM statement>,<dimspec>

<dimspec> ::= <array identifier>(<bound>)|
<array identifier>(<bound>,<bound>)|
<string simple variable>(<bound>)

<bound> ::= <integer> *(see rule 4)*

<DEF statement> ::= DEF<function identifier>(<parameter>)=<expression>

<FILES statement> ::= FILES<file name>|<FILES statement>,<file name>

<file name> ::= a string of 1 to 6 printing characters

<REM statement> ::= REM<character string> *(see rule 5)*

<MAT statement> ::= <MAT READ statement>|<MAT INPUT statement>|
<MAT PRINT statement>|<MAT initialization statement>|
<MAT assignment statement>

<MAT READ statement> ::= MAT READ<actual array>|
MAT READ<file reference>;<actual array>|
<MAT READ statement>,<actual array>

<actual array> ::= <array identifier>|<array identifier>(<dimensions>)

<dimensions> ::= <expression>|<expression>,<expression>

<MAT INPUT statement> ::= MAT INPUT<actual array>|
<MAT INPUT statement>,<actual array>

<MAT PRINT statement> ::= <MAT PRINT 1>|<MAT PRINT 2>

<MAT PRINT 1> ::= MAT PRINT<array identifier>|
MAT PRINT<file reference>;<array identifier>|
<MAT PRINT 2><array identifier>

<MAT PRINT2> ::= <MAT PRINT 1>,|<MAT PRINT 1>;

<MAT initialization statement>::= MAT<array identifier>=<initialization function>|
MAT<array identifier>=<initialization function>
(<dimensions>)

<initialization function>::= ZER|CON|IDN

<MAT assignment statement> *(rule 6)* ::= MAT<array identifier>=<array identifier>|
MAT<array identifier>=<array identifier><mat operator>
<array identifier>|
MAT<array identifier>=INV(<array identifier>)
MAT<array identifier>=TRN(<array identifier>)|
MAT<array identifier>=(<expression>)*<array identifier>

<mat operator> ::= +|-|*