2000C HIGH SPEED
TIME SHARED BASIC
INTERNAL MAINTENANCE SPECIFICATIONS

BILL HACCOU
RICH PEARSON

August 10, 1972

Contents

INTRODUCTION

The 2000C (HIGH SPEED) TIME SHARED BASIC SYSTEM consists of three separate programs which are run on two processors. The Communications processor Program is responsible for handling all multiplexed I/O from user terminals. The System contains the BASIC interpreter, executive and library routines and runs on the main processor. The Loader, which also runs on the main processor, is responsible for generating initial systems, backing up the system on mag tape, reloading the entire system and user library, and selectively reloading or backing up users libraries.

HARDWARE CONFIGURATION

1) UP TO 16 TERMINALS

10	PROCESSOR INTERCONNECT
11	PROCESSOR INTERCONNECT
12	TIME BASE GENERATOR
13	1st MULTIPLEXOR
14	1st MULTIPLEXOR
15	DATA SET CONTROL FOR 1st MULTIPLEXOR
16	LINE PRINTER (OPTIONAL)

2) MORE THAN 16 TERMINALS

10	PROCESSOR INTERCONNECT
11	PROCESSOR INTERCONNECT
12	TIME BASE GENERATOR
13	1st MULTIPLEXOR
14	1st MULTIPLEXOR
15	DATA SET CONTROL FOR 1st MULTIPLEXOR
16	2nd MULTIPLEXOR
17	2nd MULTIPLEXOR
20	DATA SET CONTROL FOR 2nd MULTIPLEXOR
21	LINE PRINTER (OPTIONAL)

2000C (HS) TIME SHARED BASIC TABLES

I DIRECTORY

The directory is a table which contains all necessary information about each program or file in the system library. It resides on the drum and may occupy from 1 to 80 drum tracks, depending upon how many discs there are on the system and how many directory tracks per disc are specified by the operator at load time. A core resident table called DIREC contains information on the directory itself.

A directory entry consists of 12 words and has the following format:

WORD	0	user id	
	1	program or	BIT 15 = 1 if protected, 0 if unprotected.
	2	file	BIT 15 = 1 if file, 0 if program
	3	name	BIT 15 = 1 if semi-compiled, 0 if uncompiled
	4	start of program pointer for programs/record size for files	
	5	last reference date (year in bits 9 to 15 day in bits 0 to 8)	
	6	last change date (hour of year)	
	7	drum address (0 if not SANCTIFIED)	
	8	disc	
	9	address	
	10	used only by loader	
	11	length (- words for program + records for file)	

The directory entries are kept sorted on words 0-3. BIT 15 of words 1 and 2 and 3 are not considered in the sorting. Names of fewer than 6 characters are filled out with spaces (40_8).

The last reference date is the most recent date on which the program or file was referred to, while the last change date is the most recent date on which it was altered.

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	-1
1	0	-1
2	0	-1
3	0	-1
4	0	0
5	-1	-1
6	0	0
7	-1	-1
8	0	0
9	0	0
10	0	0
11	0	0

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

II. DIREC

DIREC is a 560 word core resident table which contains information about the directory. It has the following structure:

WORD	0	-length in words of first directory track
	1-4	same as first 4 words of first directory track
	5	unused
	6	drum address of first directory track
	7-13	same as 0-6 but applied to 2nd directory track
	⋮	
	553-559	same as 0-6 but applied to 80th directory track

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

The drum address of a directory is always sector 0 of a track. Each directory track may contain as many as 8184 words = 682 directory entries.

When loading the system from paper tape or mage tape, the operator has the opportunity to specify the number of directory tracks per disc, in the range of 1-10, which is saved in NDIRT. The total number of directory tracks is this number times the number of discs on the system.

III. ID TABLE

The ID table (IDT) is a drum resident table of from 1 to 3 tracks 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	0	user id
	1-3	password (filled with 0's if fewer than 6 characters)
	4	time allowed(in minutes)
	5	time used (in minutes)
	6	disc allowed(in blocks)
	7	disc used (in blocks)

Words 4-7 are 16 bit quantities with values between 0 and 65535.

The 9-word IDEC portion of the EQT has the following format:

WORD	0	first id of first track
	1	drum address of first track
	2	length in -words of first track
	3-5	same as 0-2 but applied to 2nd track
	6-8	same as 0-2 but applied to 3rd track

When loading the system from paper tape or mag tape, the operator has the opportunity to specify the number of id tracks, in the range of 1-3, which is saved in NIDT. Each track may contain as many as 8192 words = 1024 id entries.

IV. AVAILABLE DRUM TABLE

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

WORD	0	drum address
	1	length of area in sectors

Entries are sorted according to word 0. Each entry may refer to as much as one full track, and no two consecutive entries ever refer to two adjacent drum areas (two tracks are not considered to be adjacent).

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

0	177777
1	0

The following two memory locations refer to the ADT:

ADLOC = drum address of ADT
ADLEN = length in-words of ADT

V. DISC ADT

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

WORD	0	disc
	1	address
	2	length of area in blocks

There is one Disc ADT track for each disc on the system and only entries for one particular disc appear on a track. The first 4 blocks of each disc are used by the system and are therefore always unavailable. Thus there are no contiguous areas which overlap discs.

The following words in the EQT refer to the Disc ADT:

DADLC	BSS 8	drum addresses of Disc ADT tracks
DADLN	BSS 8	lengths in -words of Disc ADT tracks

The first word of each BSS refers to logical disc 0, the second to logical disc 1, etc.

VI. LOCKED BLOCKS TABLE

The Locked Blocks Table is a disc-resident table which resides in the 256 words of block 3 of each disc. It contains one two-word entry for each area of the disc that has been MLOCKED. An entry has the following format:

WORD	0	disc address relative to this disc
	1	length of area in blocks

The rest of the table is zero filled.

The disc address is stored as if the disc were logical disc 0, so that packs may be used as any logical disc. The Locked Blocks Table is cleared only when it is determined during the loading procedure that the disc does not have a valid TSB label and the operator requests that one be written. This means that the packs "remember" which blocks are unavailable even if different 2000C systems are loaded.

Vii. FUSS

The FUSS table is a 1024 word table which resides on the drum. Its drum address can be obtained by the instruction.

LDA FUSS,1

FUSS is divided into 32 sections of 32 words each. The 32 words in each section are the 2-word 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 15 of the first word = 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 moving or removing a file in the routines KILL, LOCK, MLOCK, SANCTIFY and DESECRATE 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. Individual entries in a user's FUSS are changed by the execution of ASSIGN statements. It is cleared by BYE, HELLO, KILLID, and sometimes by KILL.

VIII. COMTABLE

The COMTABLE is a list of all user and system commands containing their ASCII codings and drum 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 drum resident programs

COM4 system commands - - all are executed by drum resident programs

COM5 starting addresses for those commands which are listed under COM1 and COM2

COM6 drum 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. Similarly, UCDAB generates a status type for updating the last change date for files after a program is aborted.

IX. LOGGR

LOGGR is a 64-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 15-5 = 60 x hrs + mins
             bits 4-0 = 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 unporcessed 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

X. TELETYPE TABLES

This set of 32 tables, one for each user, contains relevant information about the various terminals. The structure of the tables is as follows:

WORD	0	FLAG
	1	TNUM
	2	DISC
	3	PROG
	4	ID
	5-7	NAME
	8-9	TIME
	10	CLOC
	11	RSTR
	12	STAT
	13	LINK
	14	PLEV
	15	RTIM
	16-20	TEMP

FLAG: bits 0-8 contain information as follows:

BIT	NAME	MEANING IF = 1
0	TERR	errors while reading program in tape mode
1	CFLAG	program is compiled or semi-compiled
2	HFLAG	$HELLO is running
3	TAPEF	user in tape mode
4	UNABT	unable to abort
5	OUTWT	user suspended for output wait
6	COM14	communication from I/O processor
7	ABTRY	abort attempted (while UNABT = 1)
8	CHNFG	program was called from <CHAIN statement>
9	ENDST	error on drum transfer
10	MBUST	error on disc transfer

TNUM: teletype number in bits 12-8; used for sending information to I/O processor.

DISC: drum address of user's swap area

PROG: when user is on the drum 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 drum, 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 routine NAME & GET & CHAIN, and cleared by HELLO. When fewer than 6 characters are in the name, blanks are appended.

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:

-4	port unavilable
-3	enter timeout
-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 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. This includes FILES, CHAIN, and ASSIGN.

1: used for commands RUN,LIST,PUNCH,XPUNCH.

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

4: used for long running programs.

RTIM: the length of time in seconds that it took the user to respond to an <ENTER statement>.

TEMP: used (along with RTIM) to save variables when OPEN, CATALOG, GROUP, LIBRARY, STATUS, DIRECTORY, SDIRECTORY and REPORT are swapped out.

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

```
?FLAG   EQU  0
?TNUM   EQU  1
         :
?TEMP   EQU  16
```

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 $-26 \le N \le 49$ can be referenced by .+N.

XI. EQUIPMENT TABLE

The equipment table is the area of core which describes the resources available to the system. It resides in locations 100-204, as follows:

100-110	IDEC	ID table headers	(see III)
111	NIDT	number of ID Tracks	
112	ADLOC	drum address of ADT	(see IV)
113	ADLEN	length of ADT in -words	
114	NDIRT	number of directory tracks per disc	(see II)
115-124	DADLC	Disc ADT drum addresses	(see V)
125-134	DADLN	Disc ADT lengths in -words	
135-140	?TBL	There is one word in this area for each of the 4 drums. When the word is zero, the particular drum does not exist. Otherwise, bits 7:6 contain the drum prefix and bits 5:0 the high priority select code. The prefix is used by the drum driver as the high order 2 bits of the 8-bit track address.	
141-150	DKTBL	There is one word in this area for each of the 8 discs. When the word is zero, the particular disc does not exist. Otherwise, bits 15:8 contain the high priority select code and 7:0 the unit number	
151-170	TRAX	This is a table of which drum tracks are physically available to the system. Locations 151-154 correspond to drum 0, 155-160 to drum 1, etc. Track 0 of drum 0 is represented by bit 0 of 151, track 1 of drum 0 is represented by bit 1 of 151, etc. A bit is 0 when the track is available, 1 when unavailable.	

The TRAX table is changed only by the following commands:

DRUM - causes all tracks of the specified drum to be made available.

LOCK - all specified tracks are made unavailable.

UNLOCK - all specified tracks are made available.

171-175 SYSID A ten character system identification. It is set in response to the "SYSTEM IDENTIFICATION?" question on paper tape and mag tape loads. It is used in the headers for STATUS, REPORT, DIRECTORY and SDIRECTORY.

176 MAGSC High priority select code for mag tape; if non-existent, MAGSC=0. If bit 15=1, the tape unit is a 7970.

177 NPORT Two's complement of the number of ports on the system. The ports available are numbers Ø thru -NPORT -1.

200 YEAR Year of the century.

201-202 DATIM Time of year. The first word is the hour of the year, and the second is the number of 100 ms units in the hour minus 36000.

203 HDATE Hour of year that the system was last hibernated.

204 SLEPT 0 says that the system has been slept, -1 that it has not. This word is modified only by the sleep and reload procedures and insures that the system may not be reloaded from disc if it has not been slept.

Following the equipment table, in locations 205-216 are another set of words which must correspond with the loader. They are defined as follows:

205	LDBSA	Core address in the loader of the disc bootstrap.
206	LSTDA	Core address of the first loader segment in the System Segment Table (SST).
207	DATLN	Length of the Disc Allocation Table (DAT) in -words.
210	MHAD	Core address of the Moving Head Disc Table (MHTBL)
211	GMQBP	Core address of the routine to get a buffer for disc or drum error messages. Two such routines exist: one for the loader and one for the system.
212	DISCA	Core address of disc driver entry point.
213	DISCB	Core address of disc driver interrupt entry point.
214	MBUSY	Disc driver busy flag.
215	MWORD	Word count for disc driver.
216	DREDP	Core address of disc driver auto restart entry point (used by powerfail/auto restart routine).

XII. SYSTEM SEGMENT TABLE

The System Segment Table (SST) is 53-word table resident in the loader. It is the first portion of the bootstrap and is pointed to by LDBSA. The first word of the table contains - the number of system segments. Each group of 4 words following the first word has the following format:

WORD	1	length of segment in -words
	2	absolute, beginning core address of the segment
	3-4	disc address of the segment

There are 13 segments, ordered as follows:

SEGMENT	1	Interrupt locations (2_8 to 27_8)
	2	System base page (end of EQT to 1777_8)
	3	System linkage area (2002_8 to 2015_8)
	4	System segment 1 (end of Direc to 41777_8)
	5	" " 2 (42000_8 to 51777_8)
	6	" " 3 (52000_8 to 61777_8)
	7	" " 4 (62000_8 to 71777_8)
	8	" " 5 (72000_8 to 77677_8)
	9	Equipment table (100_8 to 216_8)
	10	Direc table (30000_8 to 31057_8)
	11	Loader segment 2 (16000_8 to 25777_8)
	12	Loader Segment 1 (2000_8 to 14677_8)
	13	Disc driver (26000_8 to 27777_8)

Note that this includes all core resident portions of the loader and system except for locations 14700_8 to 15777_8. The first 1000_8 of these words comprise the disc bootstrap and are resident on blocks 1 and 2 of each disc. Locations 15700_8 to 15777_8 may only be used for temporaries.

XIII. DISC ALLOCATION TABLE

The Disc Allocation Table (DAT) is a 276-word table resident in the loader. It is the first portion of Loader Segment 2 and its disc address is pointed to by MEM[LSTDA] + 2 when the SST is in core. The DAT designates the areas allocated on the disc for storage (during a SLEEP or HIBERNATE) of system library programs and system tables normally resident on the drum.

There are 4 sections of the DAT. The first (DATSL) is a 3 word entry consisting of the length in blocks of the system library and the 2-word disc address of the first system library program.

The other 3 sections (DATID, DATDA, and DATDI) contain one 3-word entry for each reserved area on the disc for the Id table, Disc ADT, and Directory tracks respectively. The format of these entries is the length in -words in the first word and the disc address in the second 2 words. For these sections 32 blocks are always reserved for each track, since the tables may grow to this size while the system is running.

XIV. MOVING HEAD DISC TABLE

The Moving Head Disc Table (MHTBL) is a 48 word table resident in the disc driver section of the loader and system. It contains hardware information about the disc on the system as follows:

WORD 0-1 Two-word logical address of the first 128-word hardware sector on logical disc 0.

2 Points to select code/unit number in DKTBL for logical disc 0

3 number of sectors/cylinder

4 number of sectors/track

5 Current cylinder position of heads for logical disc 0 (not used for 2883 discs)

6-11 Same as 0-5 applied to logical disc 1

⋮

42-47 Same as 0-5 applied to logical disc 7

Note that the address in the first two words of each section of the table is a sector address and must be divided by 2 to obtain the block address.

The actual numbers for the 3 kinds of used on the 2000C are as follows:

	2883	2870	7900
WORDS 1-2	Ø	Ø	Ø
	93380_{10}	9744_{10}	19488_{10}
	186760_{10}	19488_{10}	38976_{10}
	280140_{10}	29232_{10}	58464_{10}
	373520_{10}	38976_{10}	77952_{10}
	466900_{10}	48720_{10}	97440_{10}
	560280_{10}	58464_{10}	116928_{10}
	653660_{10}	68208_{10}	136416_{10}
3	460_{10}	48_{10}	96_{10}
4	23_{10}	12_{10}	24_{10}

CORE MAP

	OCTAL LOCATION	
	0	INTERRUPT LINKAGE AND UNINITIALIZED SYSTEM VARIABLES
	100	EQUIPMENT TABLE
	217	CONSTANTS AND SYSTEM VARIABLES
USER	1224	REGISTERS SAVED BY CLOCK
LIBUS	1230	USER SWAP AREA AND SYSTEM LIBRARY WORK AREA (10240 WORDS)
	26000	DISC DRIVER
	30000	DIREC TABLE
	31060	DLOOK ROUTINES
	31701	BASIC
	60000	I/O DRIVERS
	61124	TELETYPE TABLES
	62264	EXECUTIVE
	71624	COMMAND TABLE
	72140	SYSTEM LIBRARY SUBROUTINES
LIBRA	75000	SYSTEM LIBRARY PROGRAMS SWAP AREA (512 WORDS)
	77000	CORE DUMP
	77700	PROTECTED LOADER

DRUM ORGANIZATION

The drum space available to the system consists of from 64 to 256 tracks, depending upon how many drums exist. Each track contains 128 sectors of 64 words each, for a total of 8192 words per track. The loader assigns tracks as follows:

System library routines	(3 tracks)
IDT	(1-3 tracks)
User swap tracks	(1 1/4 - 40 tracks)
Directory	(1-80 tracks)
Disc ADT	(1-8 tracks)
ADT	(1 track)

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

The drum 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 512 words, and so the system reads in exactly 512 words whenever it wants to load such a routine. The loader never assigns a library routine within 7 sectors of the end of a drum track, so that no errors can take place in doing this.

Each directory, IDT, ADT and DISC ADT track is stored beginning at sector 0.

Since the user area is 10240 words long, it cannot fit on a single track. The loader must therefore find adjacent areas on the drum which total 1 1/4 tracks for each user swap area. This will not cause any problems because all drums used on the 2000C have automatic track switching.

During running, each user swap area 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 which are designated as SANCTIFIED by the operator reside on the drum and thus have better access time. They must be less than 8192 words long. The area on the disc where they were originally resident is reserved so they may be copied back at sleep time.

DISC ORGANIZATION

The disc space available to the system is determined by the number and type of discs which exist on the system. The discs are divided into 256 word blocks. There are 46690 such blocks on a 2883 disc, 9744 on a 7900 disc, and 4872 on a 2870 disc.

The first 4 blocks of each disc are reserved for use by the system. Block 0 is a label, which looks like this:

WORD	0	"LB"
	1	"TS"
	2	logical disc number
	3-7	system identification
	8-30	Ø
	31	checksum of words 0-30

Blocks 1 and 2 contain the final disc bootstrap found in the loader, and block 3 contains the locked blocks table for this disc, which is discussed elsewhere.

Disc space for system usage is assigned as follows:

Resident system	130 blocks
System library	126 blocks
IDT	32 blocks/track
Disc ADT	32 blocks/track
Directory	32 blocks/track

All remaining blocks are available for storage of user programs and files. Programs and files are each required to be stored as contiguous blocks of disc. Since the disc is allocated by blocks, each program may cause part of its last block 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. When a program is stored (by the CSAVE routine) it is saved in a semi-compiled form, i.e. the form it is in after the symbol table is built. Both the encoded text and the symbol table are stored, plus 7 words of necessary information.

Files always occupy an integral number of records (1-32767), each file occupying a contiguous area on the disc. BASIC does not treat the individual records in the same logical sequence as the physical sequence, but rather interleaves the records, as follows:

even # of records

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 records

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 records are accessed in a logically ascending order.

DISC AND DRUM ERROR ROUTINES

Disc and drum errors do not cause immediate halts in their respective drivers. Instead, the drivers indicate failures to their calling routines, which take appropriate action. In many cases (particularly most disc errors), the action is merely to inform the affected party and continue normal system operation. Such is the case for all user access to programs and files on the disc or drum.

In certain cases, however, system operation is more significantly affected. The following routines are called after these disc and drum failures:

JETPT

This routine is called when a drum transfer to or from a user's swap area has failed. It proceeds as follows:

1. Remove the user from the queue.
2. Set the port's status to unavailable and clear its ID and flags words.
3. Clear this user's area of the FUSS table.
4. Call TCRIR to inform the user that his port is being zapped.
5. Insert an informative message into the system console message queue and return.

SALVG

This routine is called when a system track with vital information (such as a directory or IDT track) cannot be written back to its assigned drum address, but when recovery might be possible if the information can be saved. It is entered with A containing the address of the word in core which contains the drum address of the track in question. B contains the negative length of the table, which must be in core starting at LIBUS. The operation is as follows:

1. Read the drum ADT, in several portions if necessary, into the upper 2K of user area. Search each piece for an entry large enough to accommodate the table in core. If none is found, go to step 4.

2. Attempt to write the table to the newly found area on the drum. If the write is unsuccessful, go to step 4.

3. If the table is successfully written to the drum, update the drum address in core associated with it, call CLNOT to finish printing any messages in the system message queue as well as a message of success, and halt.

4. Call CLNOT to clean out the message queue and to print a message of SALVG's failure, and then halt.

SICK

This routine is called when the system cannot continue operating (because, for example, it cannot read a library routine or system table), but may be able to be recovered. The routine merely calls CLNOT to finish printing any messages in the queue and a message of hope, and then halts.

DEAD

This routine is called when the system cannot continue operating and has altered its tables in such a way that they contain conflicting information, and recovery is, therefore, impossible. DEAD calls CLNOT to dump the message queue and output a "no recovery" message, and then halts.

MDEAD

This routine is called in situations like those which call DEAD, but which specifically involve possible destruction of a disc's locked blocks table. The procedure is the same as that in DEAD, except that the final message refers to the locked blocks table.

CLNOT

This routine is called when a hardware error has caused a system shut down to be initiated and it is desired to inform the operator and users of what is happening. The procedure is:

1. Call TCRIR to inform all users that the system is going down.
2. If the system teletype driver was outputting, complete the line it was printing.

3. If there are messages in the queue, output them on the system console.
4. If the routine was entered with B nonzero, print the message whose length and text are pointed to by it.
5. Output three X-OFF CR LF's, and return.

TCRIR

This routine is called to inform one or all users of a hardware failure which is fatal to him/them. The procedure is:

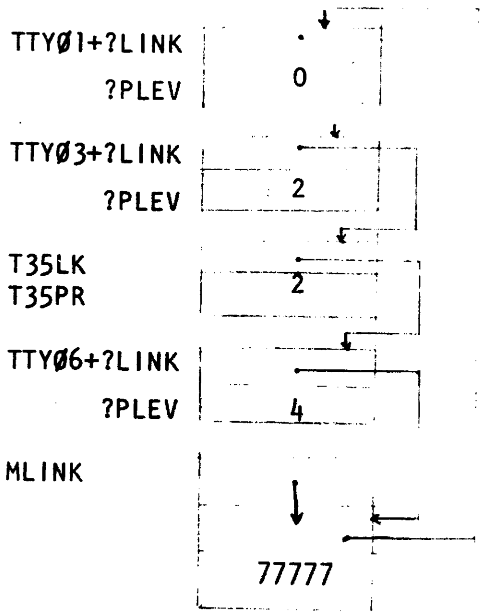
1. Set up port counts and message pointer for the appropriate message.
2. Output the message, one character at a time, to each port to receive it.
3. Output three linefeeds and return.

SYCON

This routine is the non-interrupt output-only teletype driver used by CLNOT to print on the system console. Upon entry, A holds the number of characters to be output. Bit 15 of A = Ø for X-OFF CR LF to be output after the line. B points to the first word of the buffer to be output.

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 34 entries, each entry pointing to the next entry, and the last entry pointing back to the first. The 34 possible entries in the queue are the 32 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, XPUNCH) :	1
Commands for drum-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 reassigned priority 4, and repositioned in the queue according to that priority. Each RUN job is assigned a time slice of two seconds, and if it exhausts that it is assigned another. When executing a <CHAIN statement>, a <FILES statement>, or an <ASSIGN statement>, a RUN job is reassigned a priority of Ø.

The OPEN command is reassigned a priority of 4 when it is suspended after writing file marks in 400 blocks.

After an abort during program execution a user is re-assigned a priority of 0 to run the routine which updates the last change date for files.

LIB points to the location in the COMTABLE of the drum 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 drum resident commands:
MAIN = 0
LIB set to correct drum resident routine

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.

COMMUNICATION BETWEEN SYSTEM MODULES

There are seven system modules that communicate with each other in various ways: the drum driver, the disc driver, I/O Processor driver, system console driver, scheduler, BASIC, and system library routines (HELLO, BYE, KILLID, etc.).

1. Drum Driver.

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

A = drum address	(bits	(15:14)	= drum 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 drum input 0 for drum output)

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

It is the responsibility of the caller to insure that the drum 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 even initiates drum transfers. As a result, the drum will appear to be busy only if the module itself has initiated the transfer.

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

```
JSB DRUM,1
LDA ENDRM
SSA
JMP *-2
SZA
JMP <error location>
<process successful transfer>
```

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

The value of WORD is not modified by the driver.

11. Disc Driver

Any section of the system may call the disc driver to perform a (moving-head) disc transfer. Three parameters are passed to the driver:

A = pointer to disc address (the core address of a two word logical disc block number at which the transfer is to begin)

B = core address (bits 14-0 = core address at which transfer is to begin; bit 15 = 1 for read from disc to core; bit 15 = Ø for write from core to disc)

The variable MWORD = the negative of the number of words to be transferred. If MWORD≥Ø, the driver will cause no transfer, but will position the appropriate disc unit at the specified block.

The disc driver is called by JSB DISCA,I.

The driver determines the logical disc on which the specified block lies, and, if that logical disc is present on the system, processes the requested transfer. While a request is being processed and transfer taking place, the driver busy flag, MBUSY, is set to -1. If the driver is called while MBUSY is so set, it will return without doing anything. If the disc block number passed to the driver does not lie on one of the discs present on the system, the driver will increment the return address by one and return without doing anything. If the driver accepts the request, it will increment the return address by two and return after processing of the request has been initiated.

A moving head disc transfer involves two steps: positioning the heads to the correct area of the disc and performing the actual data transfer. The disc driver returns to its caller while each of these is going on. Command channel interrupts return control to the driver when the operations are complete; the driver checks for successful completion of the operations before proceeding.

The driver for the 287Ø (IOMEC) disc keeps track of the cylinder position of the heads on each of the discs on the system. If a requested transfer is to or from the current cylinder of a disc, the driver does not issue a seek command and suspend pending its completion before starting the read or write. It merely issues an "address record" command to set the disc controller's record address register for the transfer. The 2883-2884 (ISS) disc driver always issues a seek command, since a seek to the current cylinder consumes virtually no time.

A single data transfer on a disc cannot automatically continue from one cylinder to the next. The 287Ø disc has the further restriction that a transfer cannot cross the "mid-cylinder" boundary (between track 1 and track 2). When a data transfer is requested which crosses one or more of these boundaries, the disc driver breaks up the transfer to conform with the restrictions.

When the driver completes handling a request and returns to the caller, MBUSY is set to indicate the outcome of the transfer as follows:

Ø: the requested transfer has been successfully completed.

1: the transfer has failed; the seek (position) operation could not be completed.

2: the transfer has failed; the data transfer was unsuccessful.

3: the transfer has failed; part of the data lies on, or would be written to, a disc which is not present on the system.

A complete disc transfer can be performed by the following sequence:

```
JSB DISCA,I
<return for driver busy>
<return for disc not present>
LDA MBUSY
SSA
JMP *-2
SZA
<process disc error>
<process successful transfer>
```

The disc driver does not modify the contents of MWORD and the A and B registers. The system never suspends a program for a disc transfer.

III. I/O PROCESSOR DRIVERS

There are two drivers used for communication between the main processor and the I/O processor, one for each direction of communication. Each communication consists of one 16 bit word which looks like this:

15 13	12 8	7 4	0
OPCODE	TTY #	DATA OR	OPCODE

The TTY # is the user's port number as found in the ?TNUM word of his teletype table.

The opcode uses bits 15-13, unless they are all 1's, in which case it also uses bits 4-0.

Opcodes which have values of bits 15-13$<4_8$ use bits 7-0 for data, e.g. al ASCII character.

The main processor sends communications on I/O channel 11 and receives them on I/O channel 10. An exception is a communication sent by the main processor which requires a response, which will be received on 11. Communications are initiated by JSB S14SC, I with the communication in the A register.

The following is a list of communications sent by the main processor:

NAME	OPCODE (OCTAL)	FUNCTION
OCR	000000	Output a character
STE	020000	Start timing <ENTER statement>
GTC	040000	Get a character (response required)
PHO	060000	Allowed phones time
SPE	100000	Baud rate information
SBP	120000	Save teletype buffer pointer
RBP	140000	Restore teletype buffer pointer
INI	160000	Initialize system
UIR	160001	User is running
UNR	160002	User no longer running
IWT	160003	User in input wait
HUU	160004	Disconnect user
ULO	160005	User logged on successfully
ECO	160006	Echo-on
ECF	160007	Echo-off

NAME	OPCODE (OCTAL)	FUNCTION
TPO	160010	User in tape mode
ILI	160011	Illegal input? (requires response)
NUC	160012	New user logged on
KAO	160013	Kill all output
ALI	160014	Allow input
OWT	160015	User in output wait
IBF	160016	Is buffer full (requires response)
PSC	160017	Line printer select code
LPR	160020	Line printer request (requires response)
LPD	160021	Line printer disconnect
LPS	160022	Line Printer status (requires response)
BKS	160023	Backspace in teletype buffer
CHS	160024	Character size information (requires response)
STP	160025	Subtype information
WSP	160026	What baud rate (requires response)
WCS	160027	What character size (requires response)
WTP	160030	What terminal type (requires response)
TKO	160031	Telekludge line printer output

Communications initiated by the I/O processor are detailed elsewhere. It should be noted that the main processor ignores communications if they are not consistent, e.g. it will only accept a line of input when the user's status is idle or input wait. The receive driver communicates with the scheduler by setting the COMI4 bit in the ?FLAG word of the user's teletype table and setting the appropriate status.

The I/O processor program is responsible for all multiplexor I/O. Output to the multiplexor 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) ignored.
B = address of WORD 0 of users teletype table
(JSB OUTCH,I).

The OUTCH routine places characters into the user's buffer until it is filled (250 characters), at which point the user is suspended by OUTCH. This is no problem for BASIC, but due to re-entrancy 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. Routines which may fill the buffer, like CATALOG and DIRECTORY, get around the problem by suspending themselves at an appropriate time.

The I/O processor program recognizes aborts and sends them to the main processor. If the user is running a library program (except CATALOG, LIBRARY, GROUP, DIRECTORY, SDIRECTORY, REPORT and STATUS), the abort is ignored, since the routine may be in the process of updating system tables. At other times when aborting could cause trouble, the UNABT bit in the ?FLAG word of the TTY table is set. When the abort is seen, the ABTRY bit is set. Routines which set UNABT have the responsibility of calling ABCHK when aborts will no longer cause harm. ABCHK aborts the user if ABTRY was set.

Input from a user teletype is buffered by lines. When the I/O processor sees a carriage return, it informs the main processor. BASIC, or the command processor, or the library routine, etc. processes the input on a character by character basis.

IV. SYSTEM CONSOLE DRIVER

The system console driver maintains three flags, T35F1, T35F2, and T35F3, which determine its status. The meaning 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.

T35F3: Normally 0, it is set to -1 by the driver at the conclusion of input, and cleared to 0 by the driver after output has been initiated.

The combined values of these flags are more significant:

F1	F2	F3	
0	0	0	Driver is accepting input
0	-1	-1	Input command received and is being processed, but output has not been initiated.
0	-1	0	Output terminated from a system command which is to be reinitiated.
-1	0	0	Outputting
-1	-1	0	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. F2 and the console status (T35ST) are also cleared if a key is struck on the console during output. This will effectively terminate such things as DIRectories, REPorts and STAtuses.

When F3 = -1, log-on and log-off reports as well as the message queue are held off. This guarantees that these messages will not be interfered with by system library program output.

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 TTY35,1
```

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,1
```

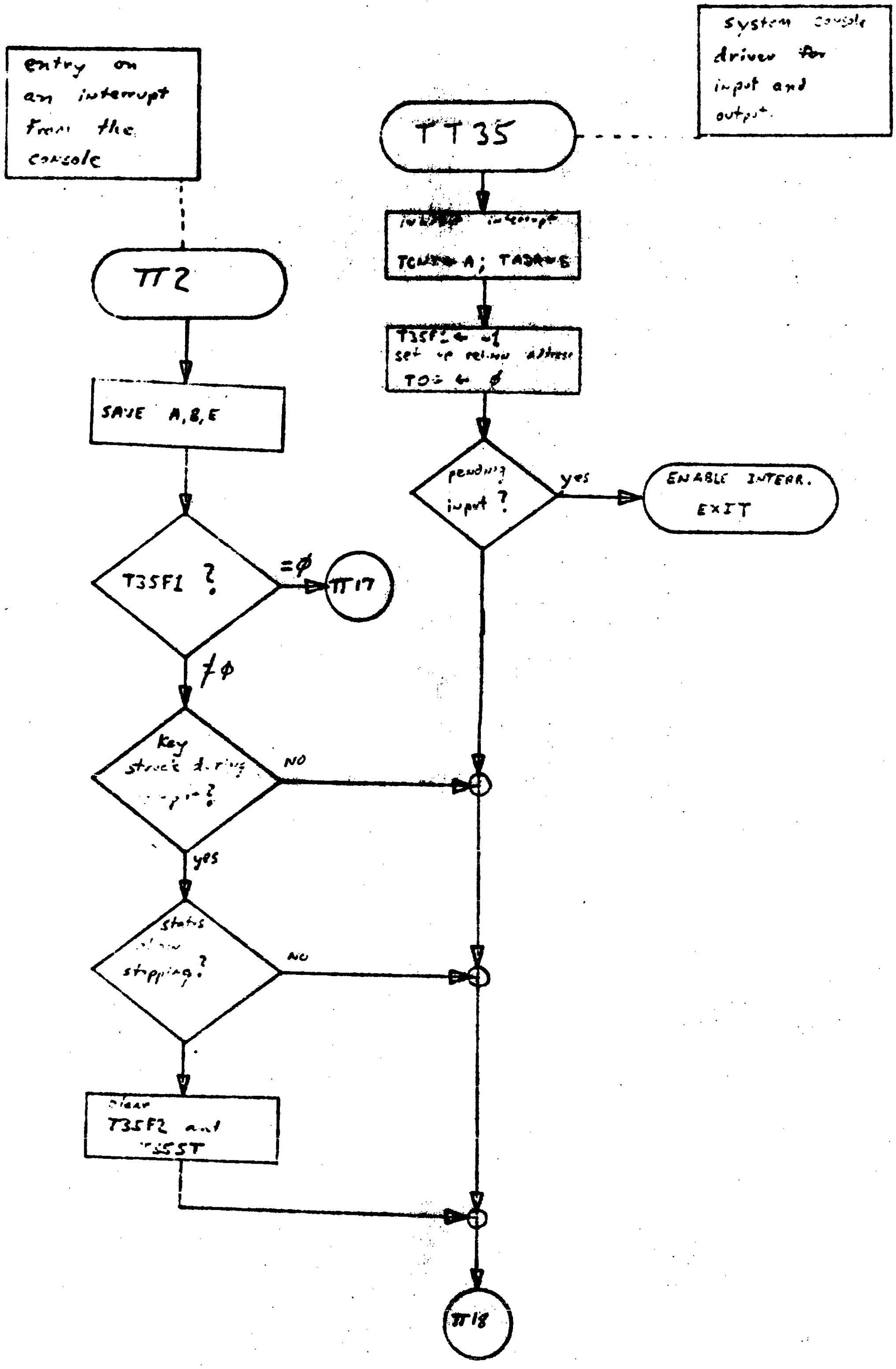
Either BASIC or a system library routine terminates by:

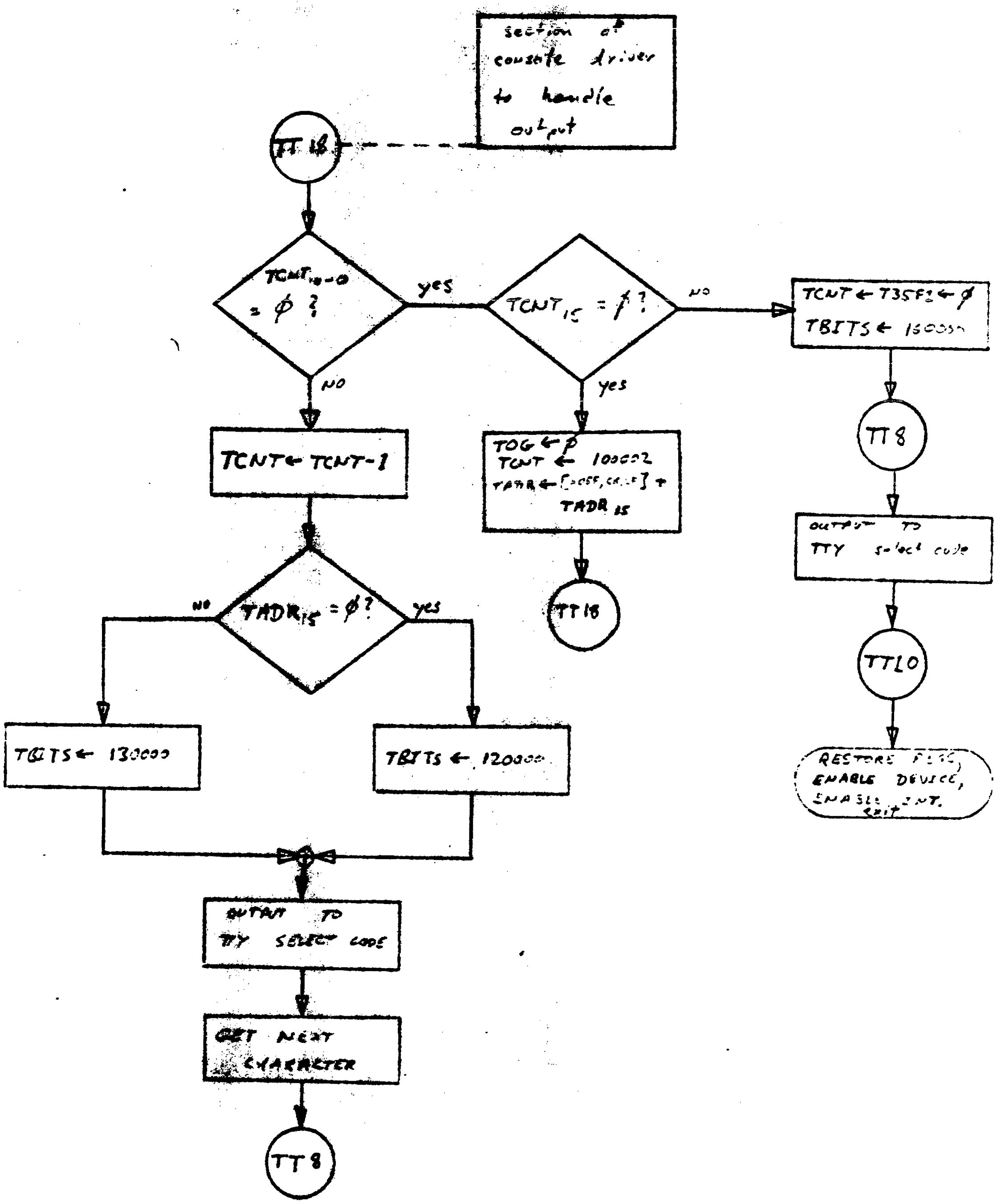
```
JSB SCHEN,1
```

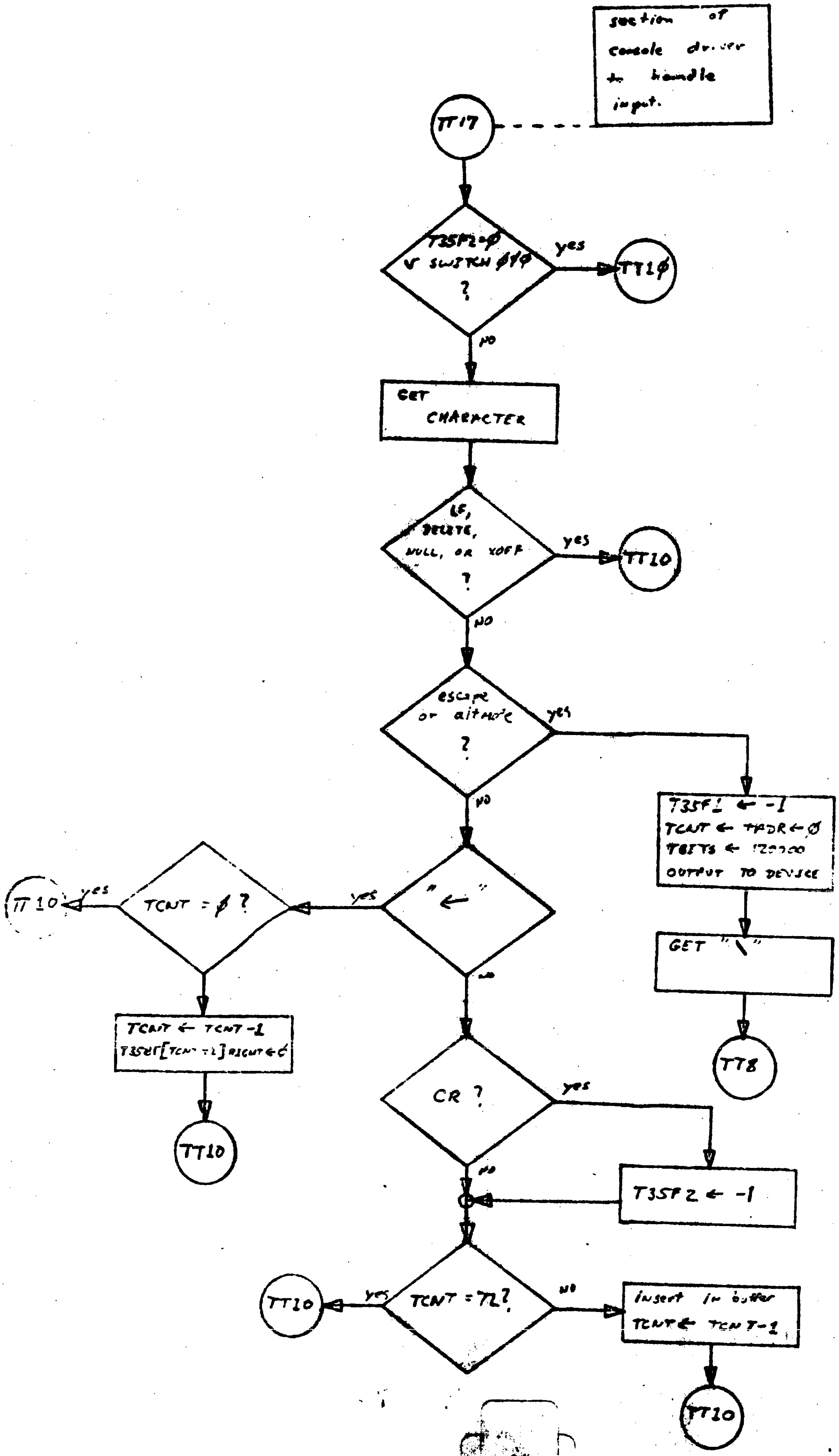
It is possible for BASIC to call a system library routine directly by executing:

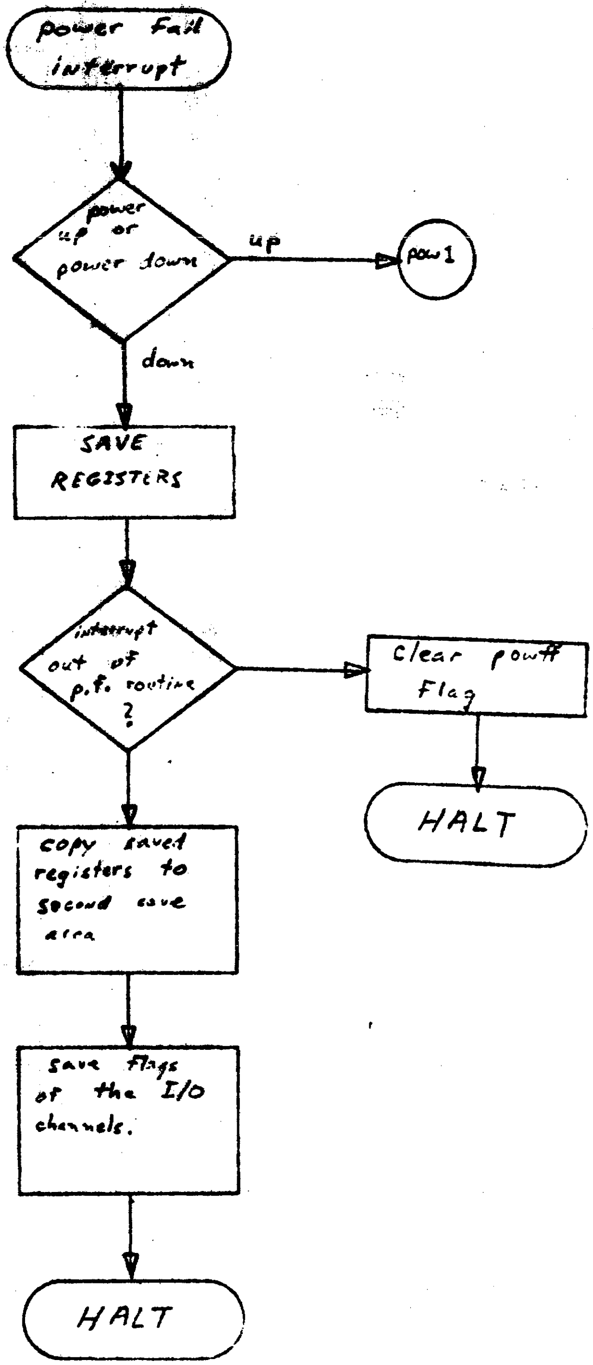
```
JSB SCHLB,1
DEF <location in COMTABLE of drum address of program>
```

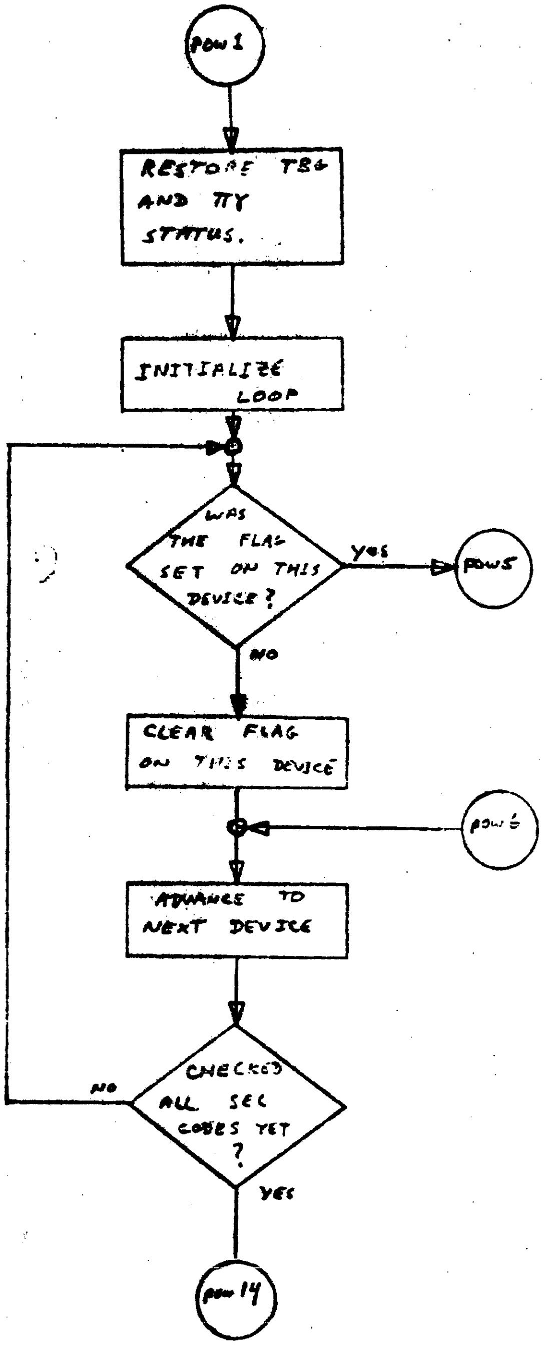
This is done with the FILES, CHAIN and ASSIGN routines. It is necessary that the library routine cooperate with BASIC, i.e., not any program can be so called.

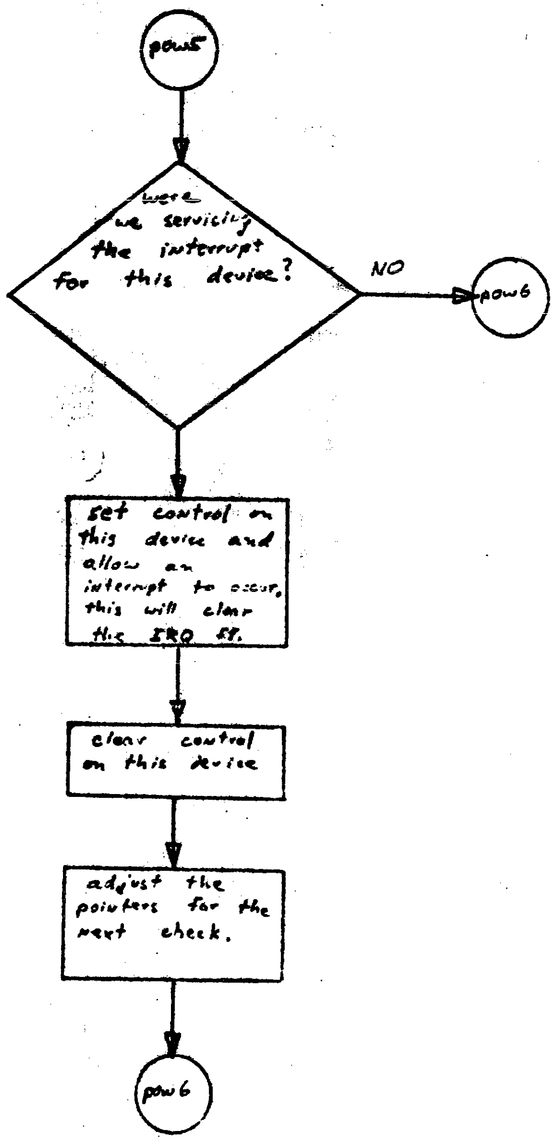

entry on an interrupt from the console
TT2
SAVE A,B,E
T35F1 ?
=ϕ
TT17
≠ϕ
Key struck during output ?
NO
yes
status allow stopping ?
NO
clear T35F2 and T35ST
TT18
System console driver for input and output.
TT35
inhibit interrupt
TCU→A; TADR→B
T35F2 ← 1
set up return address
TO3 ← ϕ
pending input ?
yes
ENABLE INTERR. EXIT

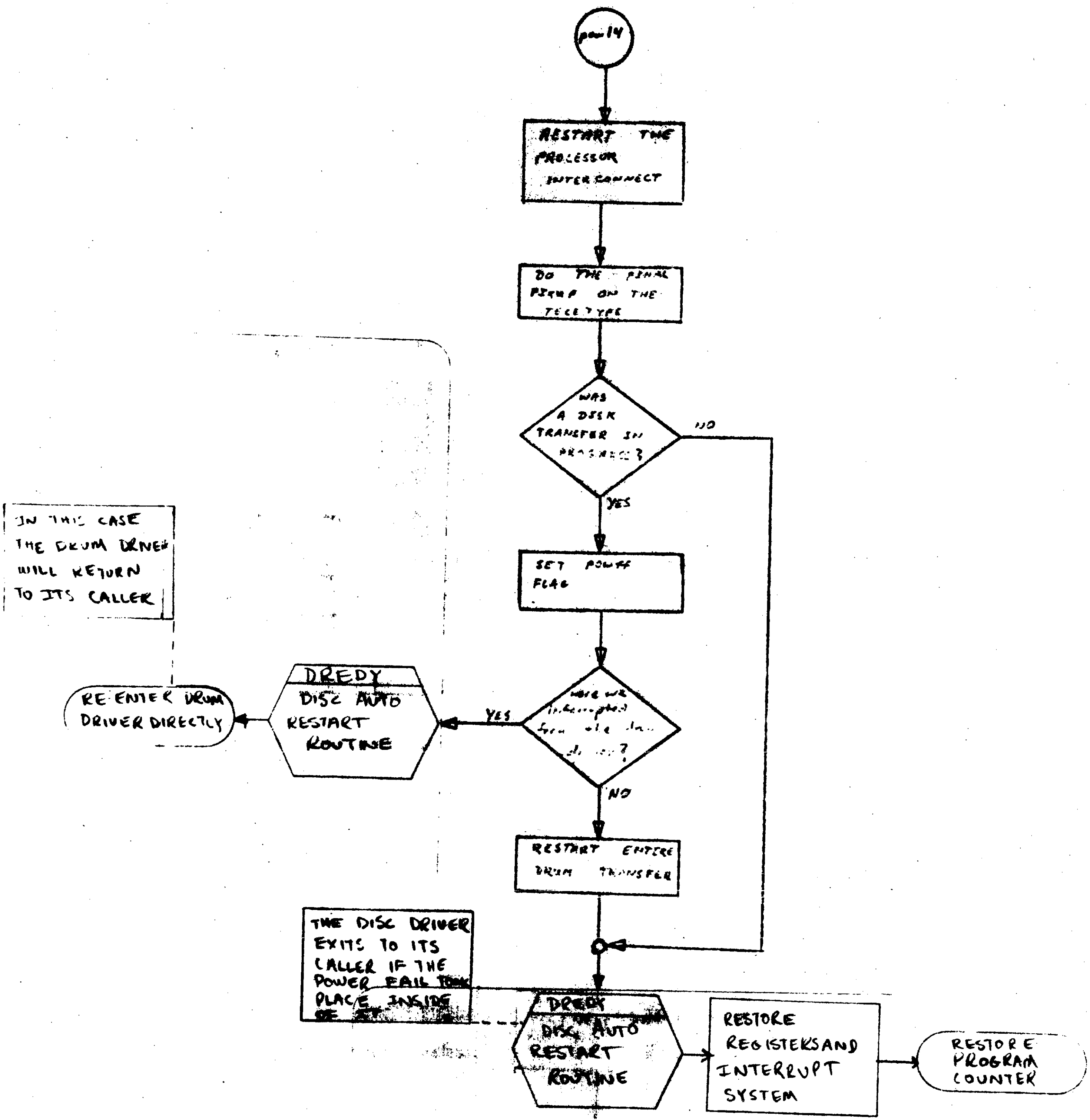

section of
console driver
to handle
output
TT18
TCNT14-0
= φ ?
yes
no
TCNT15 = φ ?
no
yes
TCNT ← T35F2 ← φ
TBITS ← 160030
TT8
OUTPUT TO
TTY select code
TT10
RESTORE REGS,
ENABLE DEVICE,
ENABLE INT.
exit
TOG ← φ
TCNT ← 100002
TADR ← [COFF, CRLF] +
TADR15
TT18
TCNT ← TCNT-1
TADR15 = φ ?
NO
yes
TBITS ← 130000
TBITS ← 120000
OUTPUT TO
TTY SELECT CODE
GET NEXT
CHARACTER
TT8

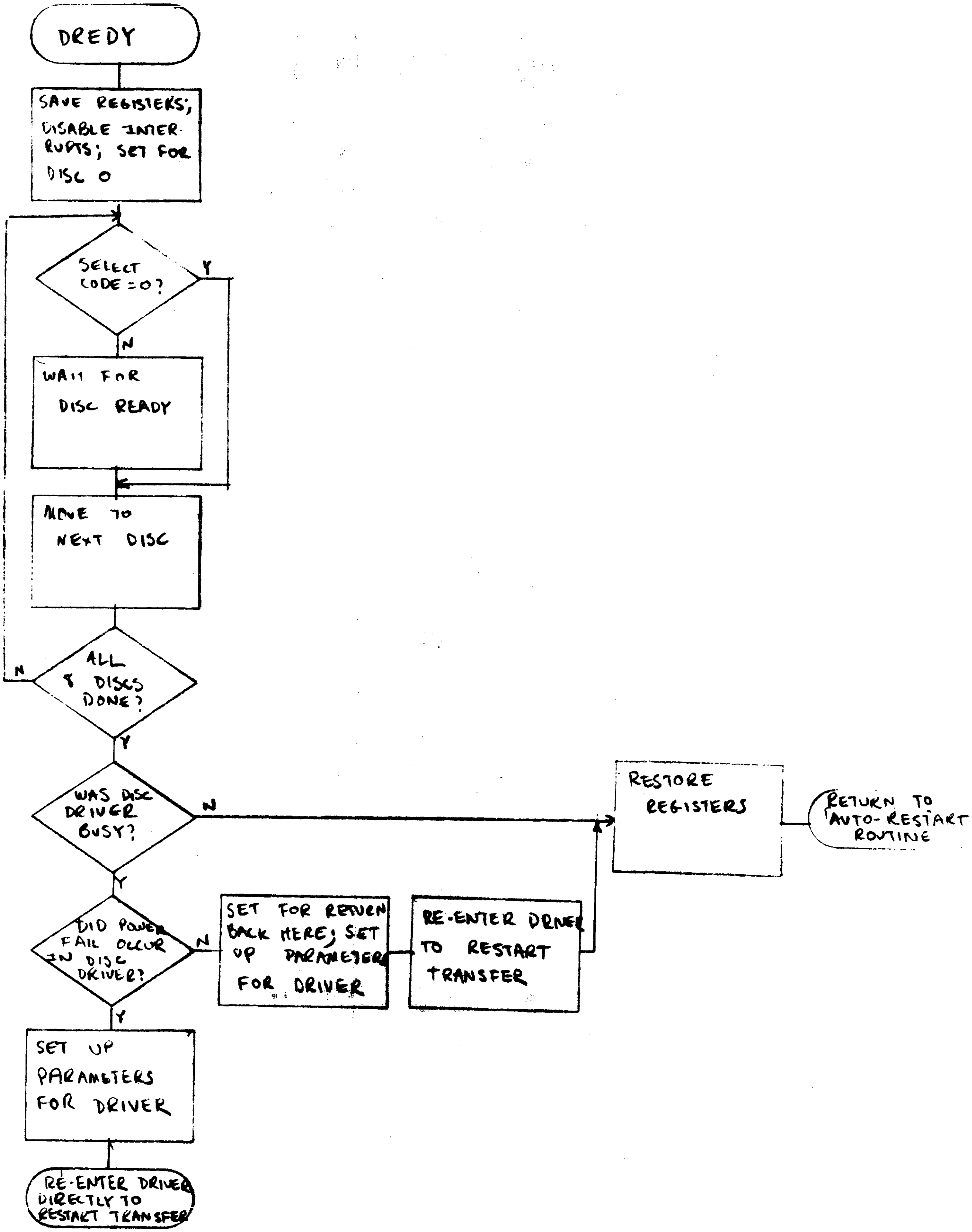

section of console driver to handle input.
TT17
T35F2=Ø V SWITCH Ø1Ø ?
yes
TT1Ø
NO
GET CHARACTER
LF, DELETE, NULL, OR XOFF ?
yes
TT10
NO
escape or altmode ?
yes
T35F1 ← -1
TCNT ← TPDR ← Ø
TBITS ← !20000
OUTPUT TO DEVICE
GET "\"
TT8
NO
"←"
yes
TCNT = Ø ?
yes
TT10
TCNT ← TCNT -1
T35UF[TCNT ÷2] RIGHT ← ¢
TT10
NO
CR ?
yes
T35F2 ← -1
NO
TCNT = 72 ?
yes
TT10
NO
insert in buffer
TCNT ← TCNT-1
TT10


power fail interrupt
power up or power down
up
pow1
down
SAVE REGISTERS
interrupt out of p.f. routine ?
clear powff Flag
HALT
copy saved registers to second save area
save flags of the I/O channels.
HALT

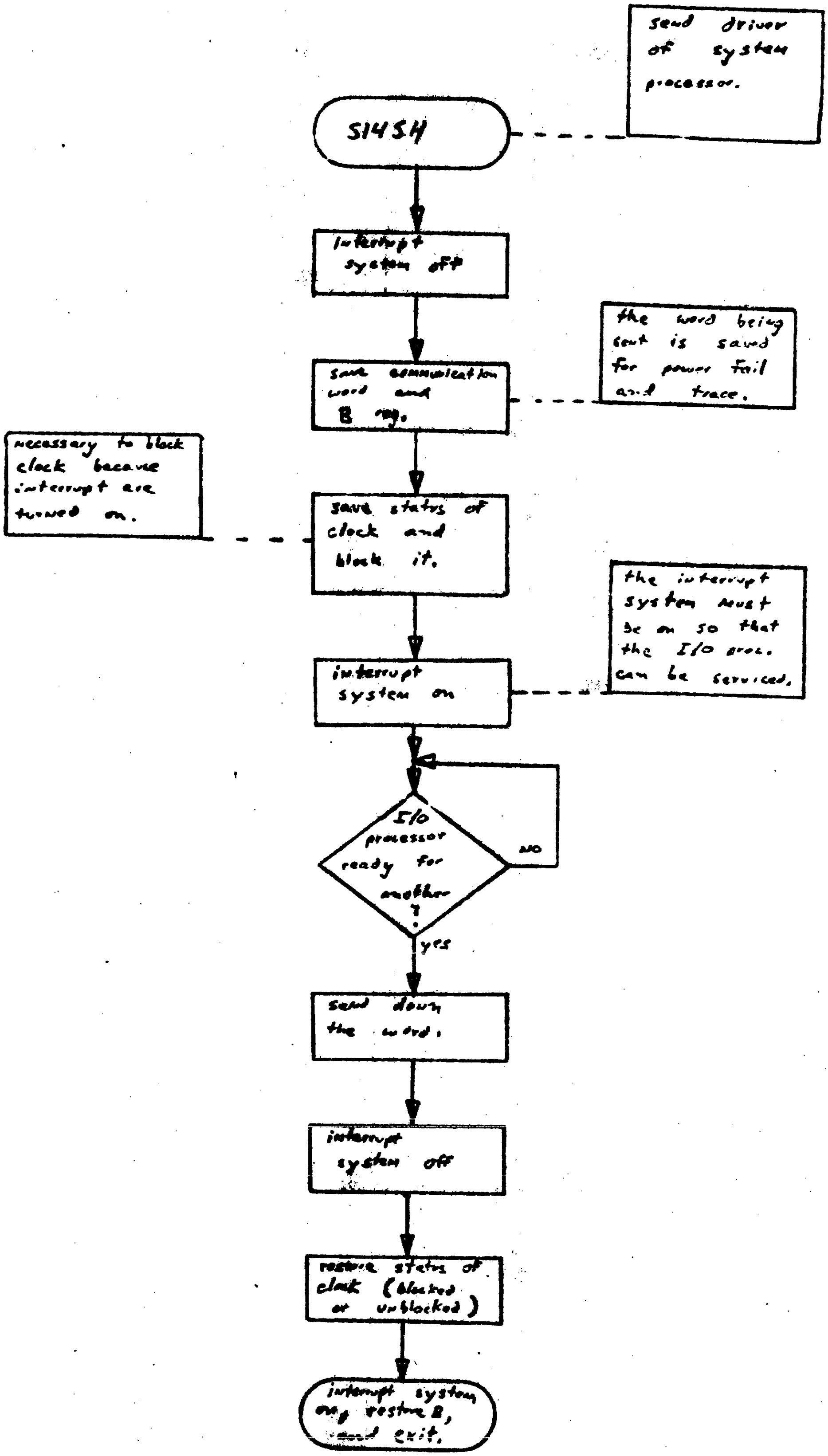

POW 1
RESTORE TBG AND TTY STATUS.
INITIALIZE LOOP
WAS THE FLAG SET ON THIS DEVICE?
YES
POW 5
NO
CLEAR FLAG ON THIS DEVICE
POW 6
ADVANCE TO NEXT DEVICE
CHECKED ALL SEC CODES YET ?
NO
YES
POW 14


pow5
were we servicing the interrupt for this device?
NO
pow6
set control on this device and allow an interrupt to occur, this will clear the IRQ FF.
clear control on this device
adjust the pointers for the next check.
pow6

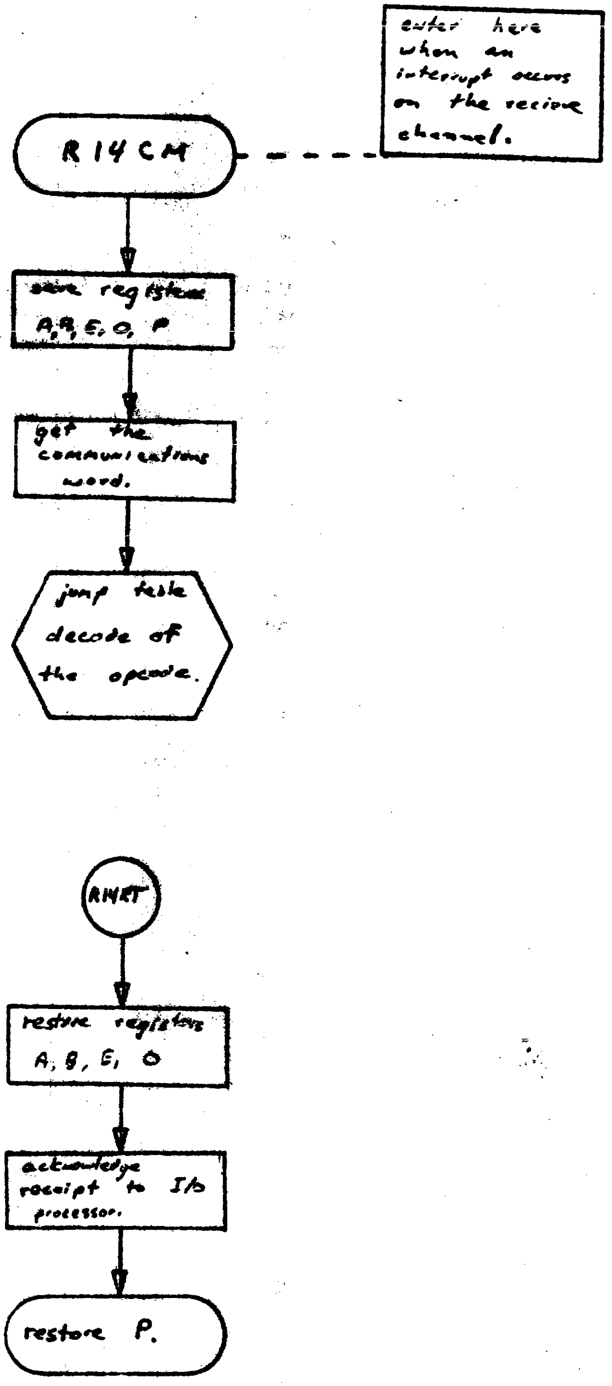

pow14
RESTART THE PROCESSOR INTERCONNECT
DO THE FINAL FIXUP ON THE TELETYPE
WAS A DISK TRANSFER IN PROGRESS?
NO
YES
SET POWFF FLAG
were we interrupted from the drum driver?
YES
NO
RESTART ENTIRE DRUM TRANSFER
DREDY
DISC AUTO RESTART ROUTINE
RE ENTER DRUM DRIVER DIRECTLY
IN THIS CASE THE DRUM DRIVER WILL RETURN TO ITS CALLER
THE DISC DRIVER EXITS TO ITS CALLER IF THE POWER FAIL TOOK PLACE INSIDE OF IT
DREDY
DISC AUTO RESTART ROUTINE
RESTORE REGISTERS AND INTERRUPT SYSTEM
RESTORE PROGRAM COUNTER


DREDY
SAVE REGISTERS; DISABLE INTERRUPTS; SET FOR DISC 0
SELECT CODE = 0?
Y
N
WAIT FOR DISC READY
MOVE TO NEXT DISC
ALL 8 DISCS DONE?
N
Y
WAS DISC DRIVER BUSY?
N
Y
RESTORE REGISTERS
RETURN TO AUTO-RESTART ROUTINE
DID POWER FAIL OCCUR IN DISC DRIVER?
N
Y
SET FOR RETURN BACK HERE; SET UP PARAMETERS FOR DRIVER
RE-ENTER DRIVER TO RESTART TRANSFER
SET UP PARAMETERS FOR DRIVER
RE-ENTER DRIVER DIRECTLY TO RESTART TRANSFER

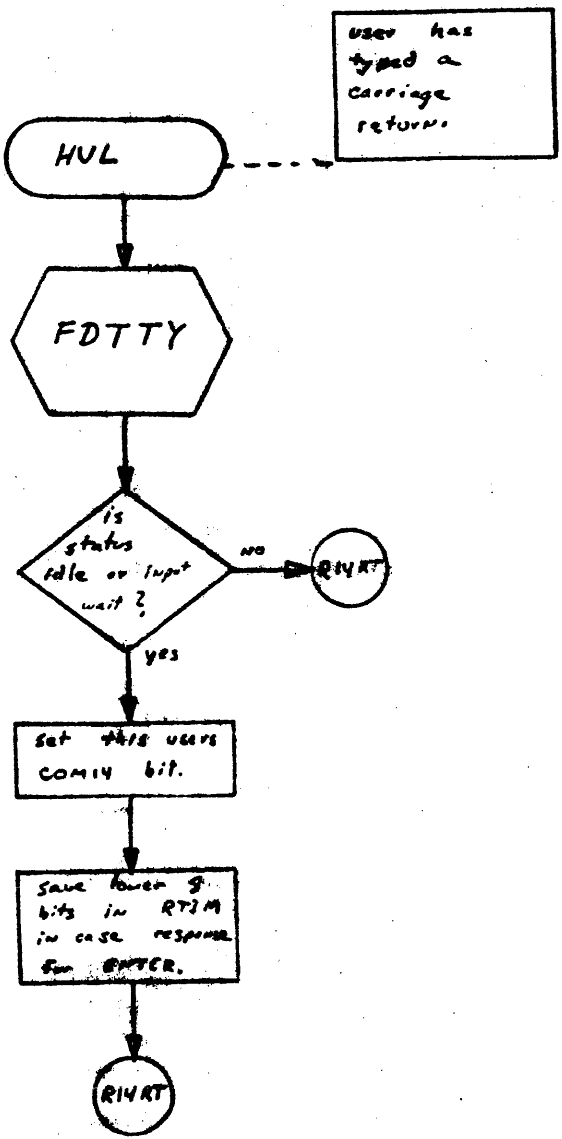
COMMUNICATION DRIVERS -- SEND

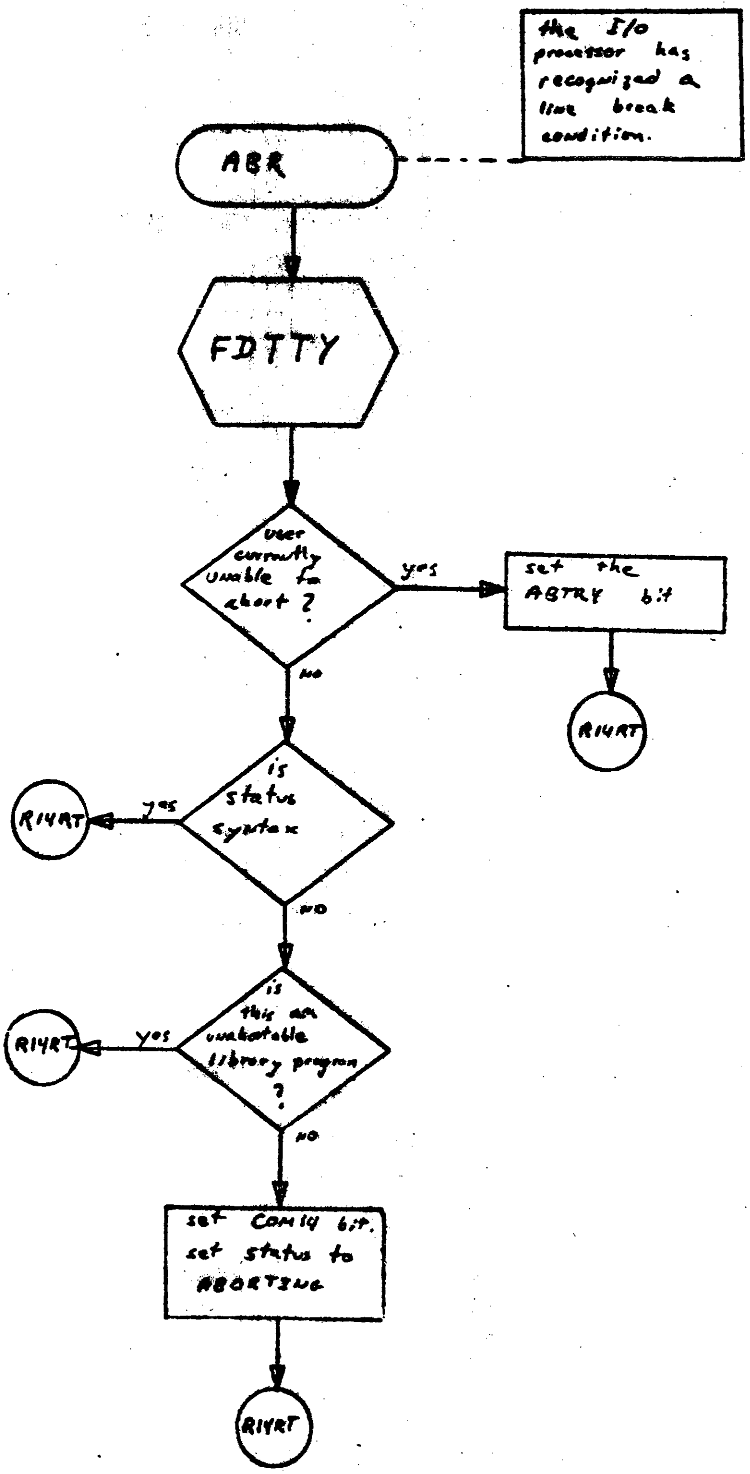

send driver of system processor.
S14SH
interrupt system off
save communication word and B reg.
the word being sent is saved for power fail and trace.
necessary to block clock because interrupt are turned on.
save status of clock and block it.
interrupt system on
the interrupt system must be on so that the I/O proc. can be serviced.
I/O processor ready for another ?
no
yes
send down the word.
interrupt system off
restore status of clock (blocked or unblocked)
interrupt system on, restore B, ~~and~~ exit.

COMMUNICATIONS DRIVERS -- RECEIVE pg 1 of 5

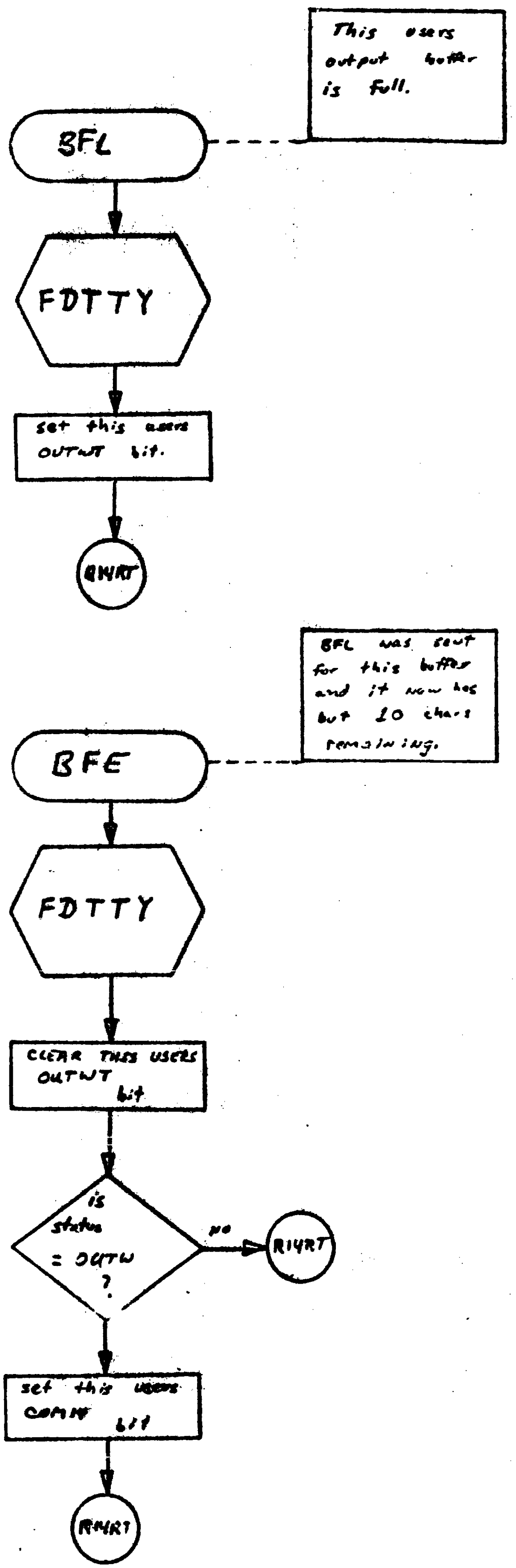

enter here when an interrupt occurs on the receive channel.
R14CM
save registers A,B,E,O,P
get the communications word.
jump table decode of the opcode.
R14RT
restore registers A, B, E, O
acknowledge receipt to I/O processor.
restore P.

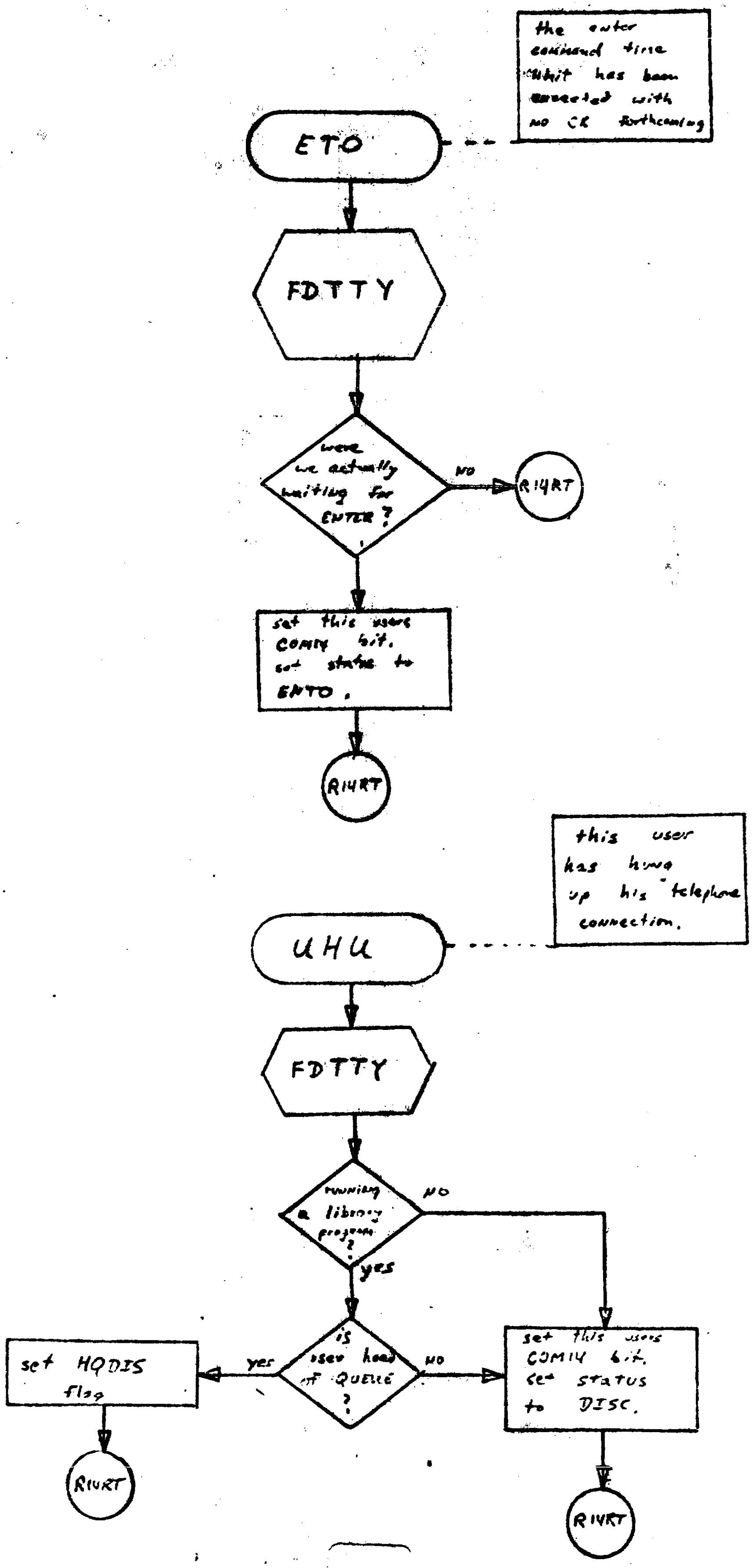
COMMUNICATIONS DRIVERS -- RECEIVE


user has typed a carriage return.
HUL
FDTTY
is status idle or input wait ?
no
R14RT
yes
Set ~~this~~ users COM14 bit.
Save lower 8 bits in RTIM in case response for ENTER.
R14RT

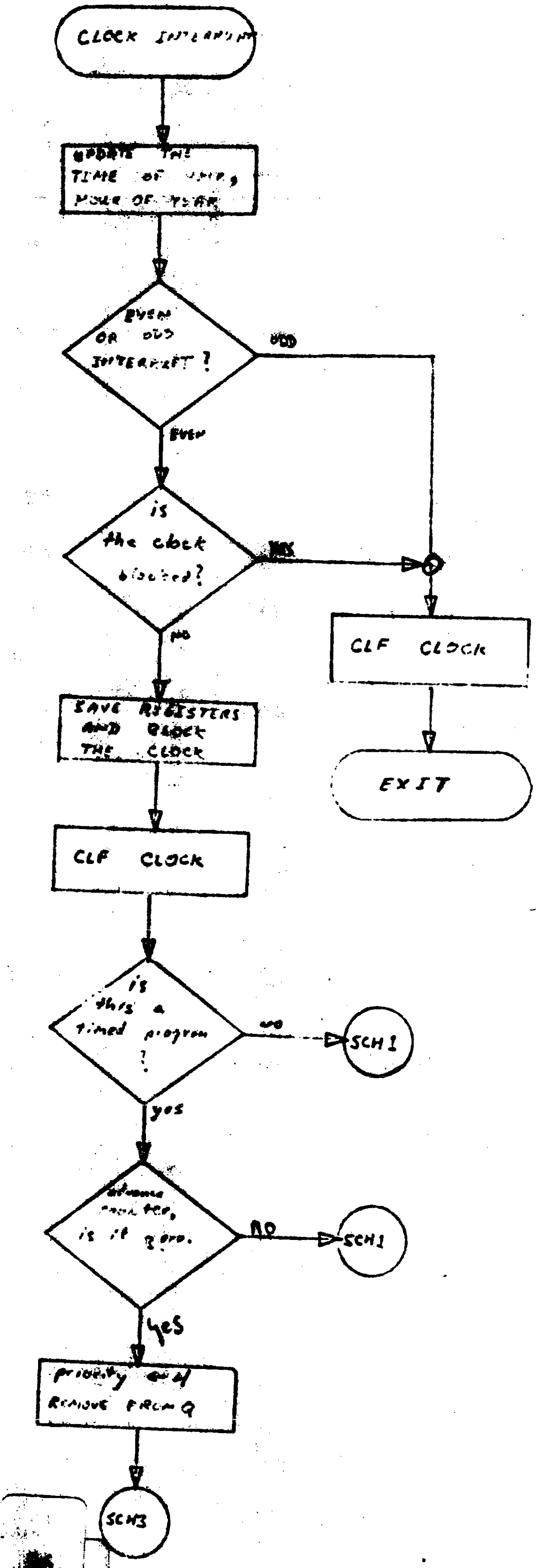

the I/o processor has recognized a line break condition.
ABR
FDTTY
user currently unable to abort ?
yes
no
set the ABTRY bit
R14RT
is status syntax
yes
no
R14RT
is this an unabortable library program ?
yes
no
R14RT
set COM14 bit.
set status to ABORTING
R14RT

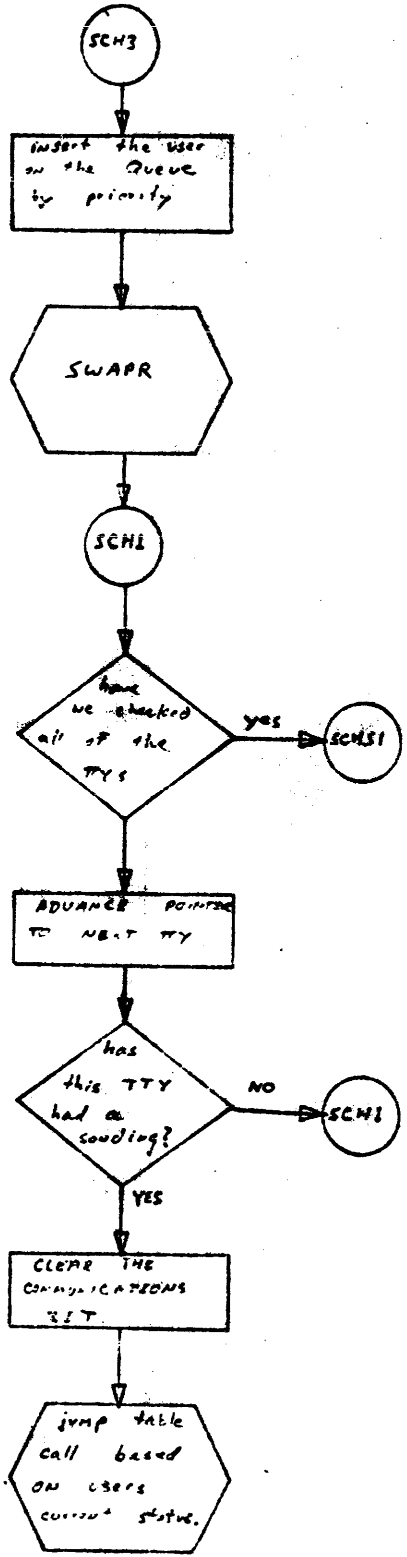
COMMUNICATIONS DRIVERS -- RECEIVE pg 4 of 5

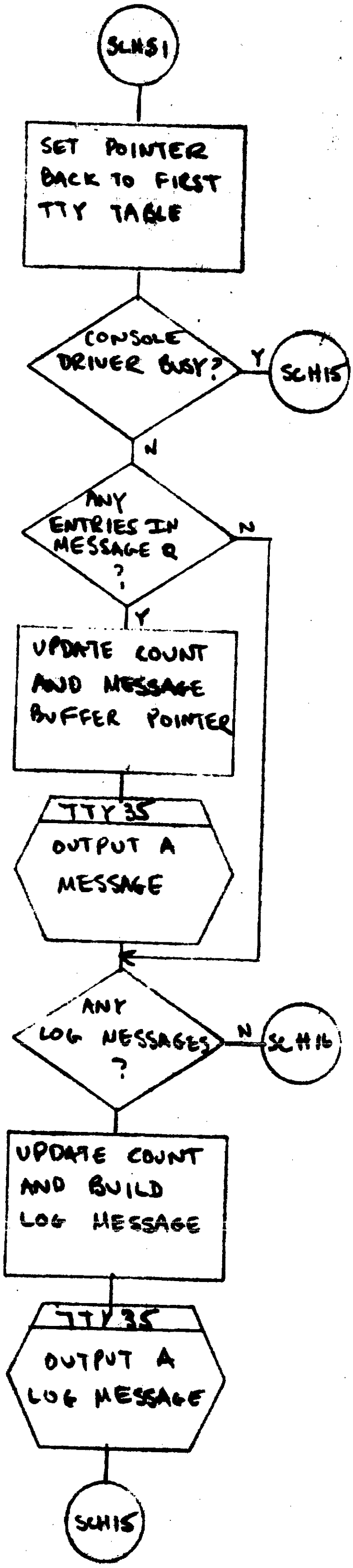

This users output buffer is full.
BFL
FDTTY
set this users OUTWT bit.
RIWRT
BFL was sent for this buffer and it now has but 10 chars remaining.
BFE
FDTTY
CLEAR THIS USERS OUTWT bit
is status = OUTW ?
no
RIWRT
set this users COMIF bit
RIWRT

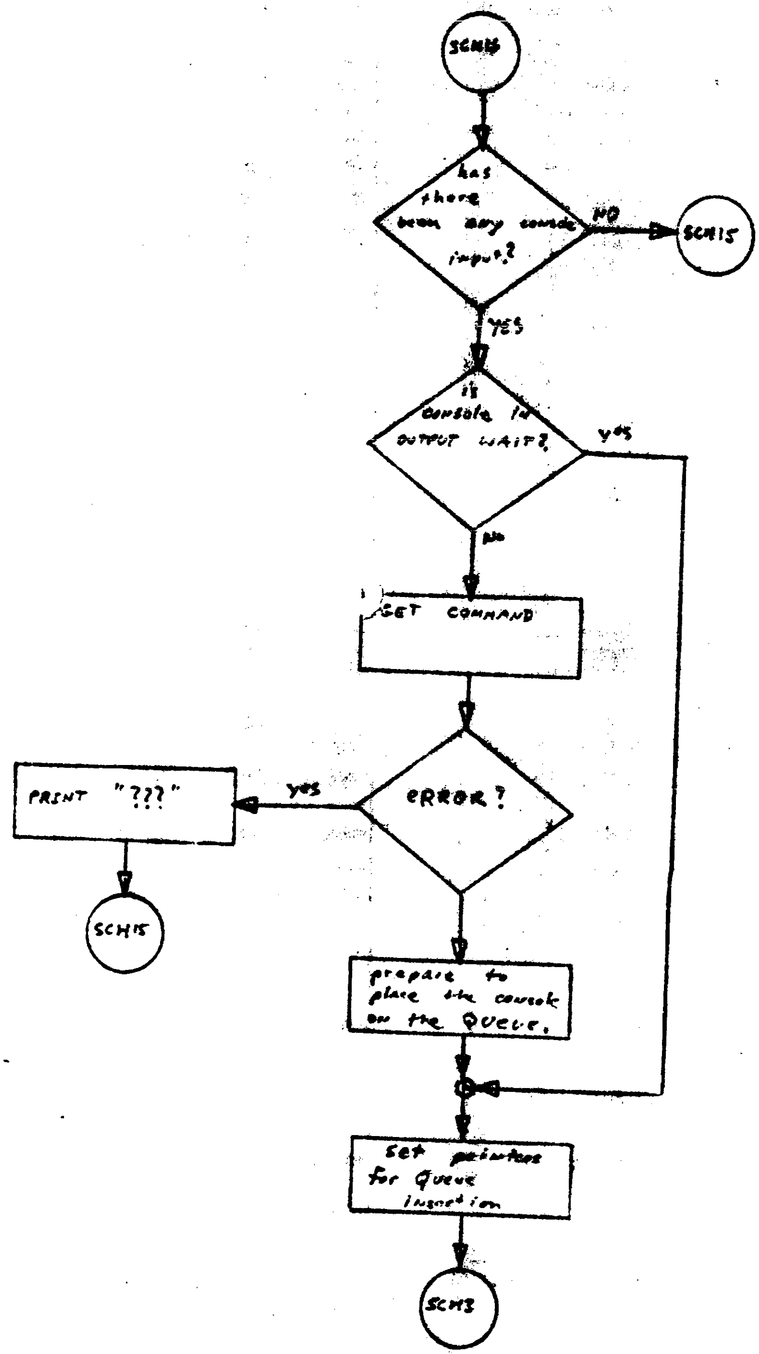

the enter command time limit has been exceeded with no CR forthcoming
ETO
FDTTY
were we actually waiting for ENTER ?
no
R14RT
set this users COM14 bit. set status to ENTO.
R14RT
this user has hung up his telephone connection.
UHU
FDTTY
running a library program ?
no
yes
is user head of QUEUE ?
yes
no
set HQDIS flag
R14RT
set this users COM14 bit. set status to DISC.
R14RT

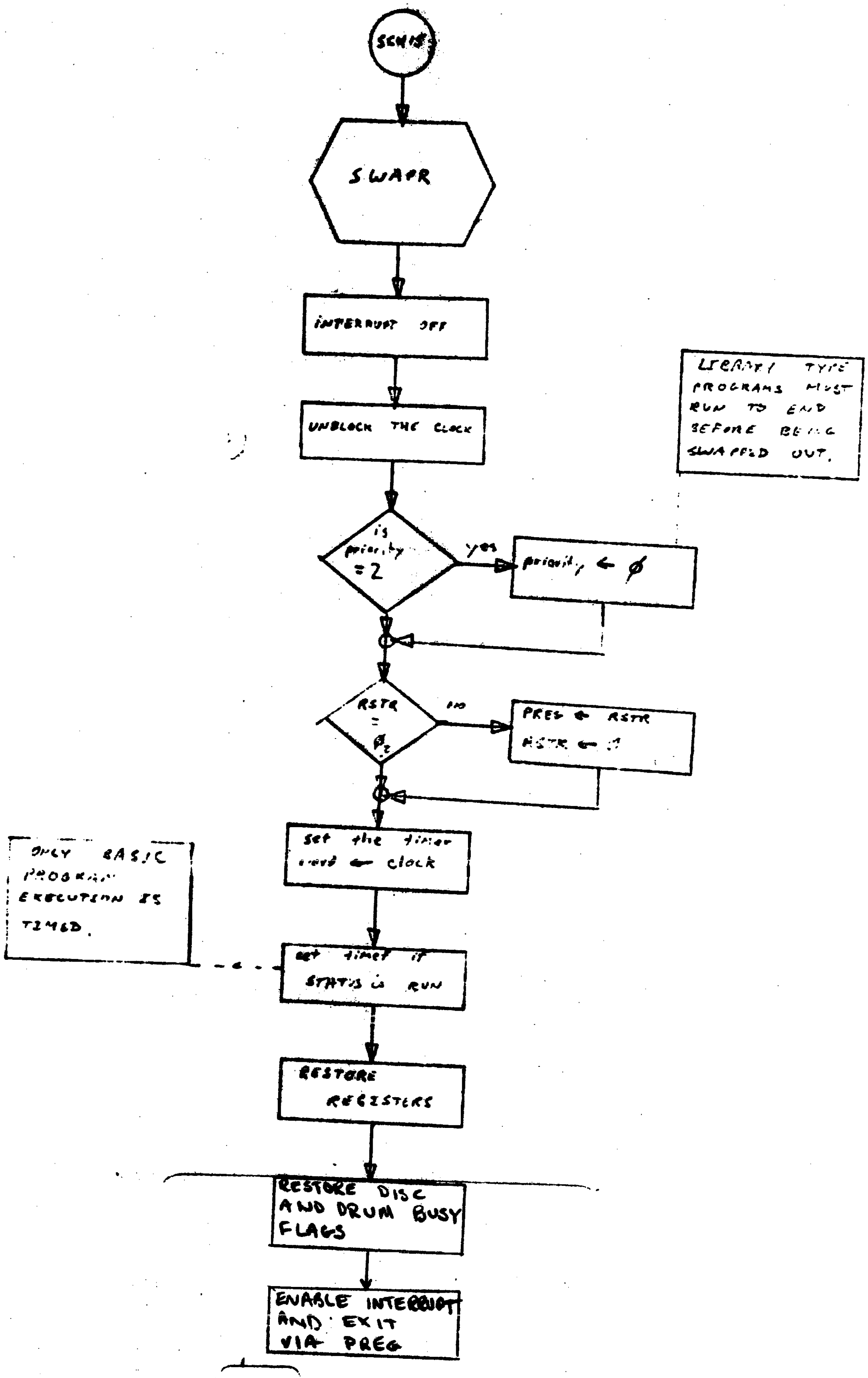
SCHEDULER

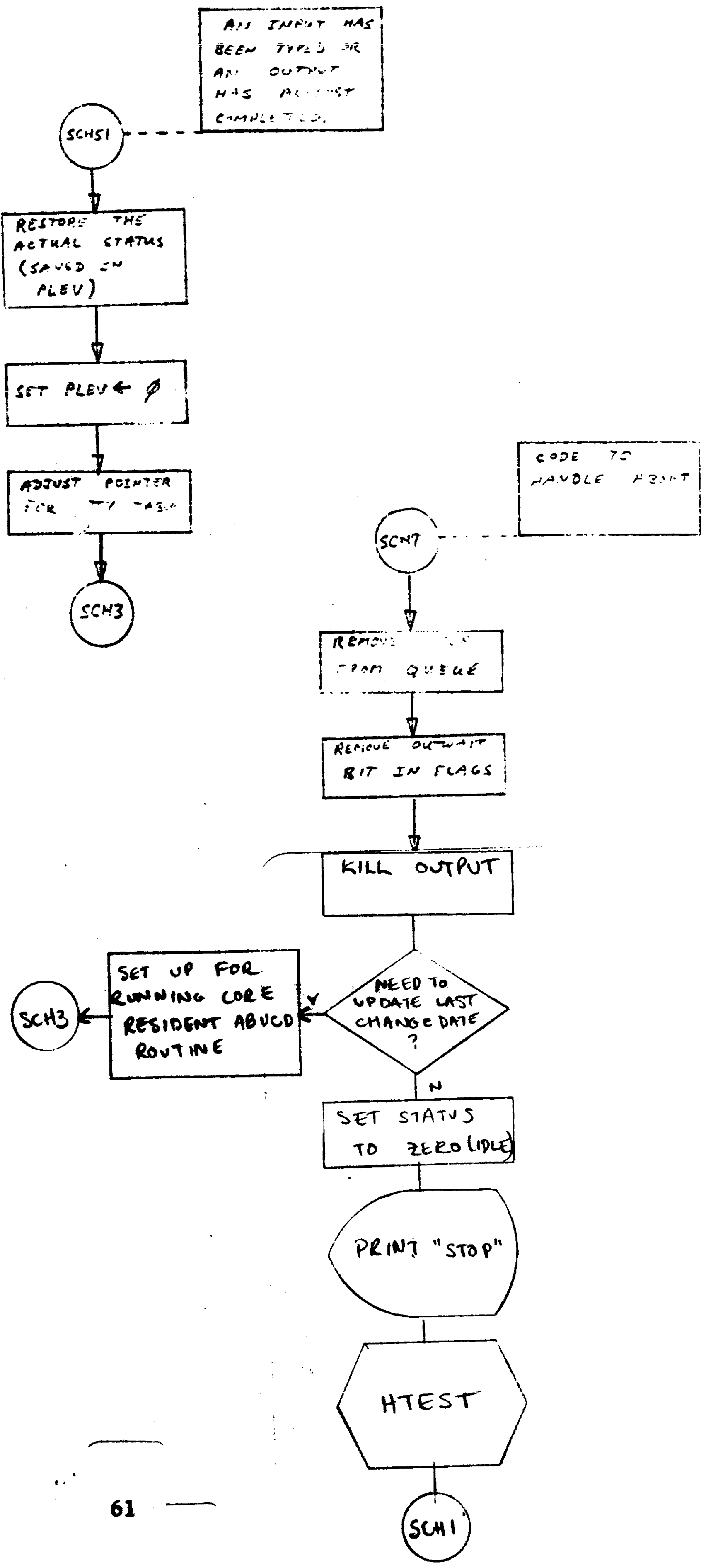

CLOCK INTERRUPT
UPDATE THE TIME OF DAY, HOUR OF YEAR
EVEN OR ODD INTERRUPT ?
ODD
EVEN
is the clock blocked?
YES
NO
CLF CLOCK
EXIT
SAVE REGISTERS AND BLOCK THE CLOCK
CLF CLOCK
is this a timed program ?
no
SCH1
yes
advance counter, is it zero.
NO
SCH1
yes
priority end
REMOVE FROM Q
SCH3

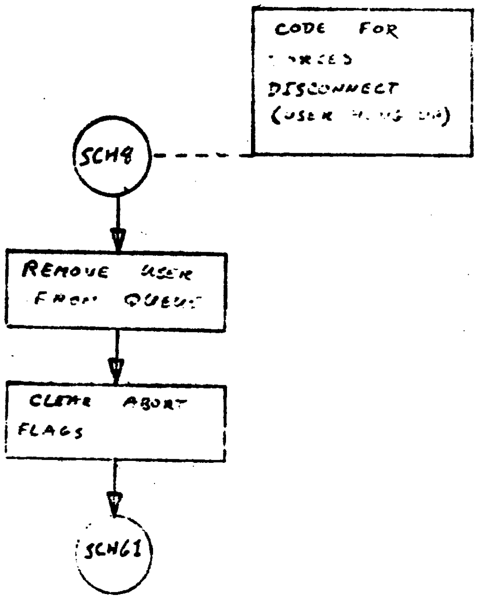

SCH3
insert the user on the Queue by priority
SWAPR
SCH1
have we checked all of the TTYs
yes
SCHS1
ADVANCE POINTER TO NEXT TTY
has this TTY had a souding?
NO
SCH1
YES
CLEAR THE COMMUNICATIONS BIT
jump table call based on users current status.

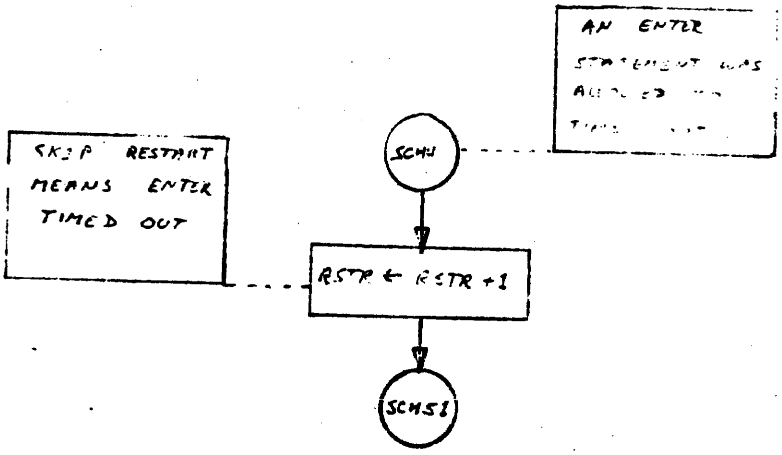

SCHS1
SET POINTER BACK TO FIRST TTY TABLE
CONSOLE DRIVER BUSY?
Y
SCH15
N
ANY ENTRIES IN MESSAGE Q ?
N
Y
UPDATE COUNT AND MESSAGE BUFFER POINTER
TTY35
OUTPUT A MESSAGE
ANY LOG MESSAGES ?
N
SCH16
UPDATE COUNT AND BUILD LOG MESSAGE
TTY35
OUTPUT A LOG MESSAGE
SCH15

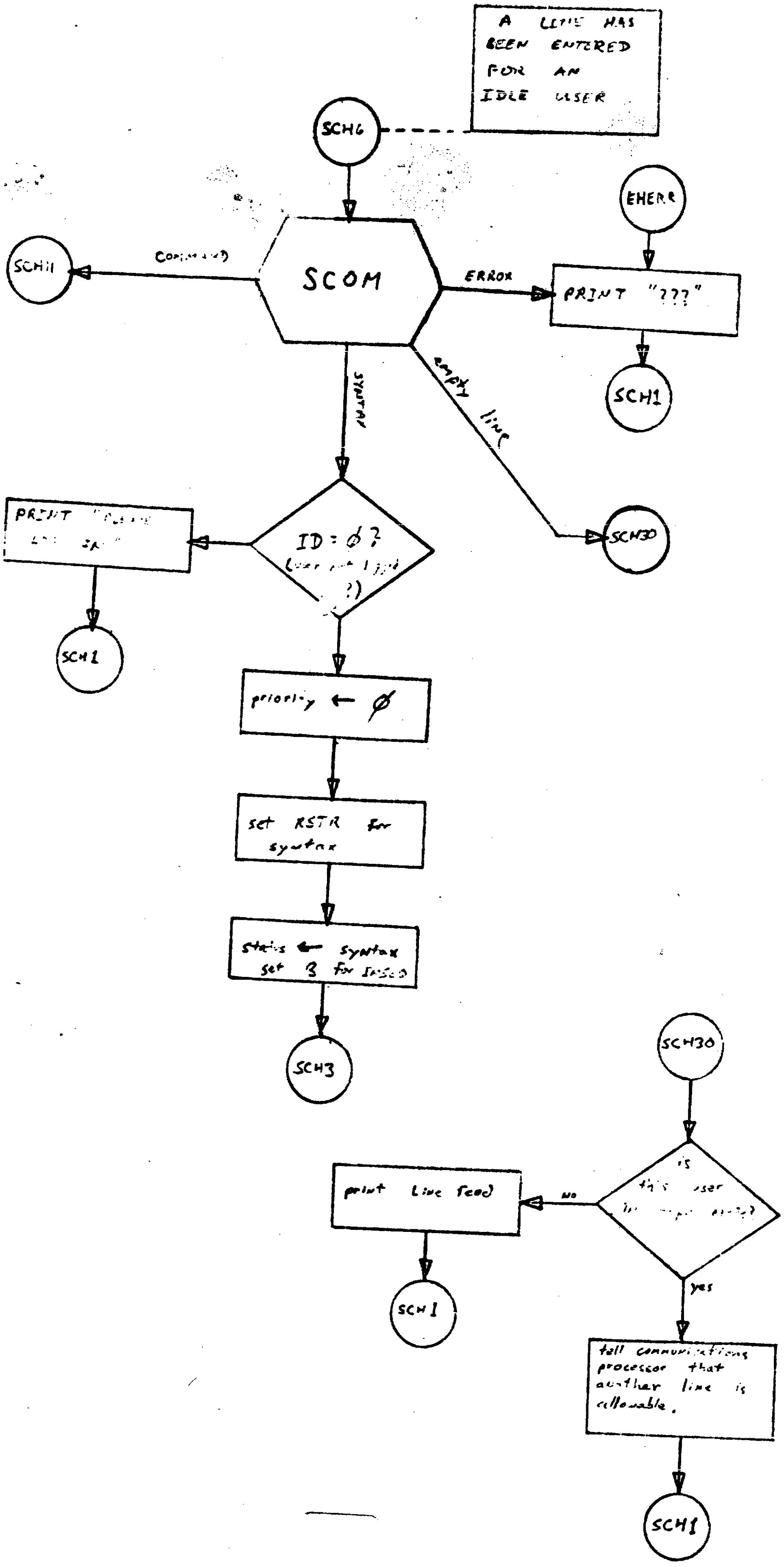

SCH6
has there been any console input?
NO
SCH15
YES
is console in OUTPUT WAIT?
yes
no
SET COMMAND
ERROR?
yes
PRINT "???"
SCH15
prepare to place the console on the Queue.
set pointers for Queue insertion
SCH3

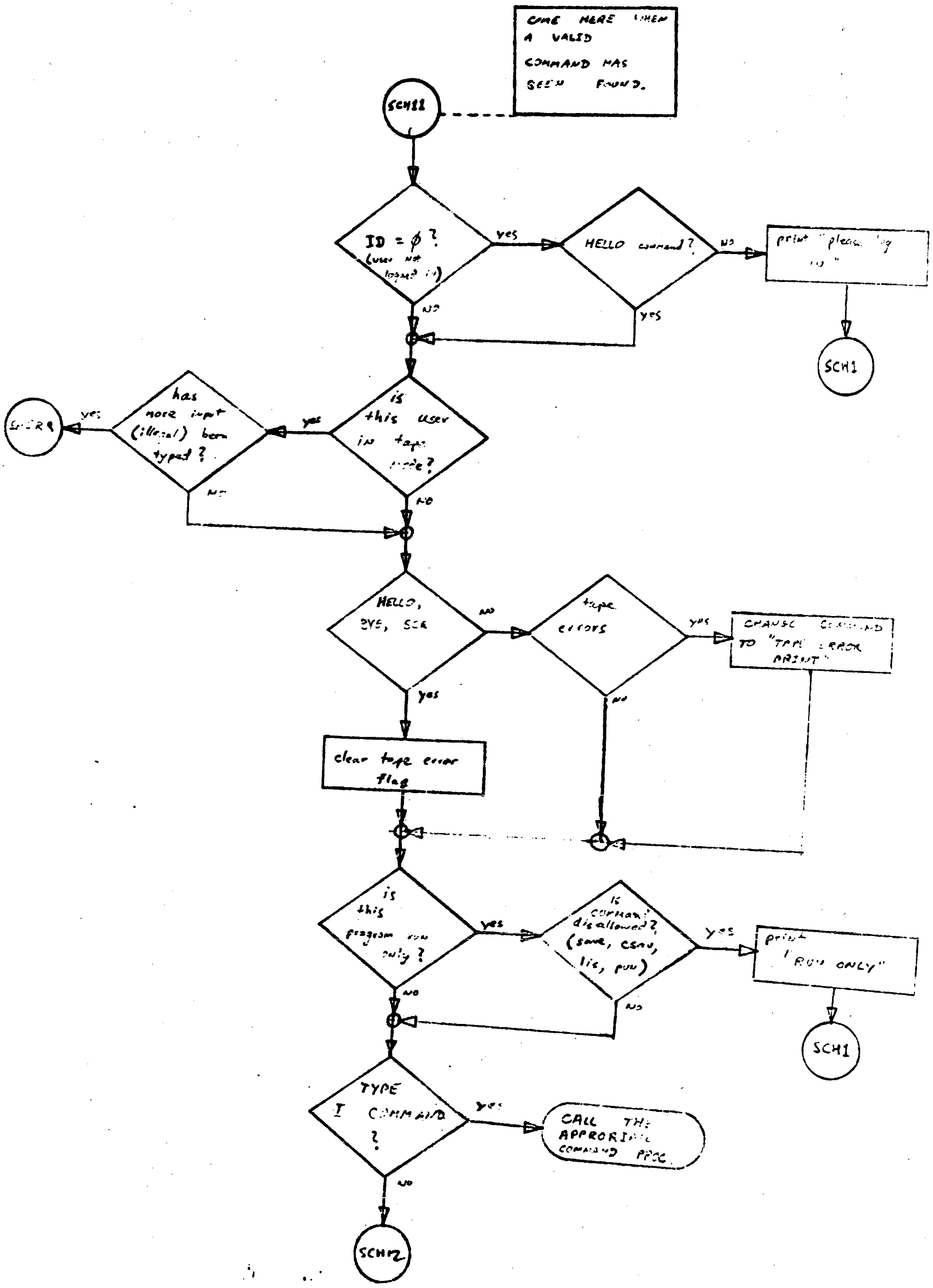

SCHED
SWAPR
INTERRUPT OFF
UNBLOCK THE CLOCK
LIBRARY TYPE PROGRAMS MUST RUN TO END BEFORE BEING SWAPPED OUT.
is priority = 2
yes
priority ← ø
RSTR = ø
no
PREG ← RSTR
RSTR ← ø
set the timer word ← clock
ONLY BASIC PROGRAM EXECUTION IS TIMED.
set timer if STATUS is RUN
RESTORE REGISTERS
RESTORE DISC AND DRUM BUSY FLAGS
ENABLE INTERRUPT AND EXIT VIA PREG

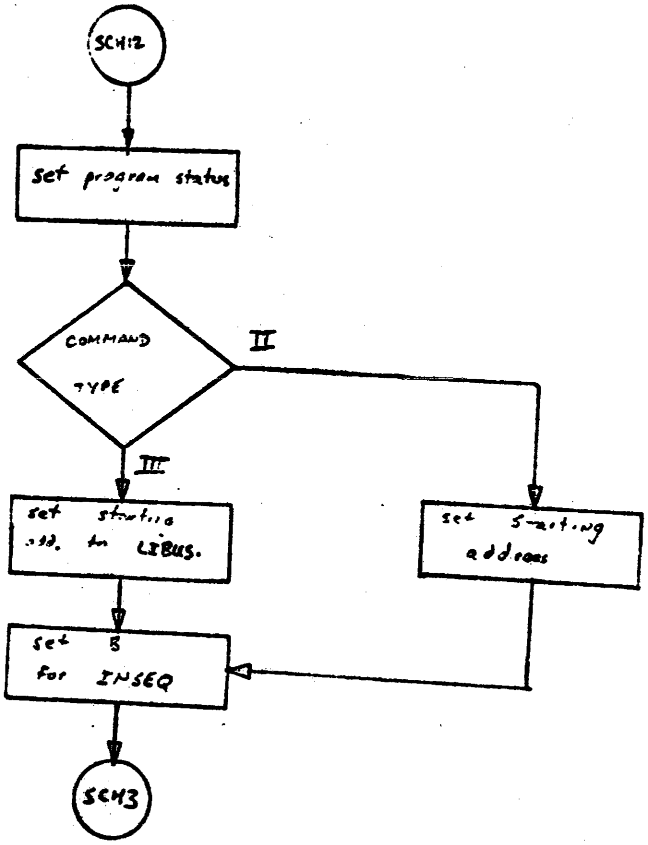

AN INPUT HAS BEEN TYPED OR AN OUTPUT HAS ALMOST COMPLETED.
SCH51
RESTORE THE ACTUAL STATUS (SAVED IN PLEV)
SET PLEV ← ∅
ADJUST POINTER FOR TTY TABLE
SCH3
CODE TO HANDLE [illegible]
SCN7
REMOVE [illegible] FROM QUEUE
REMOVE OUTWAIT BIT IN FLAGS
KILL OUTPUT
NEED TO UPDATE LAST CHANGE DATE ?
Y
SET UP FOR RUNNING CORE RESIDENT ABUCD ROUTINE
SCH3
N
SET STATUS TO ZERO (IDLE)
PRINT "STOP"
HTEST
SCH1

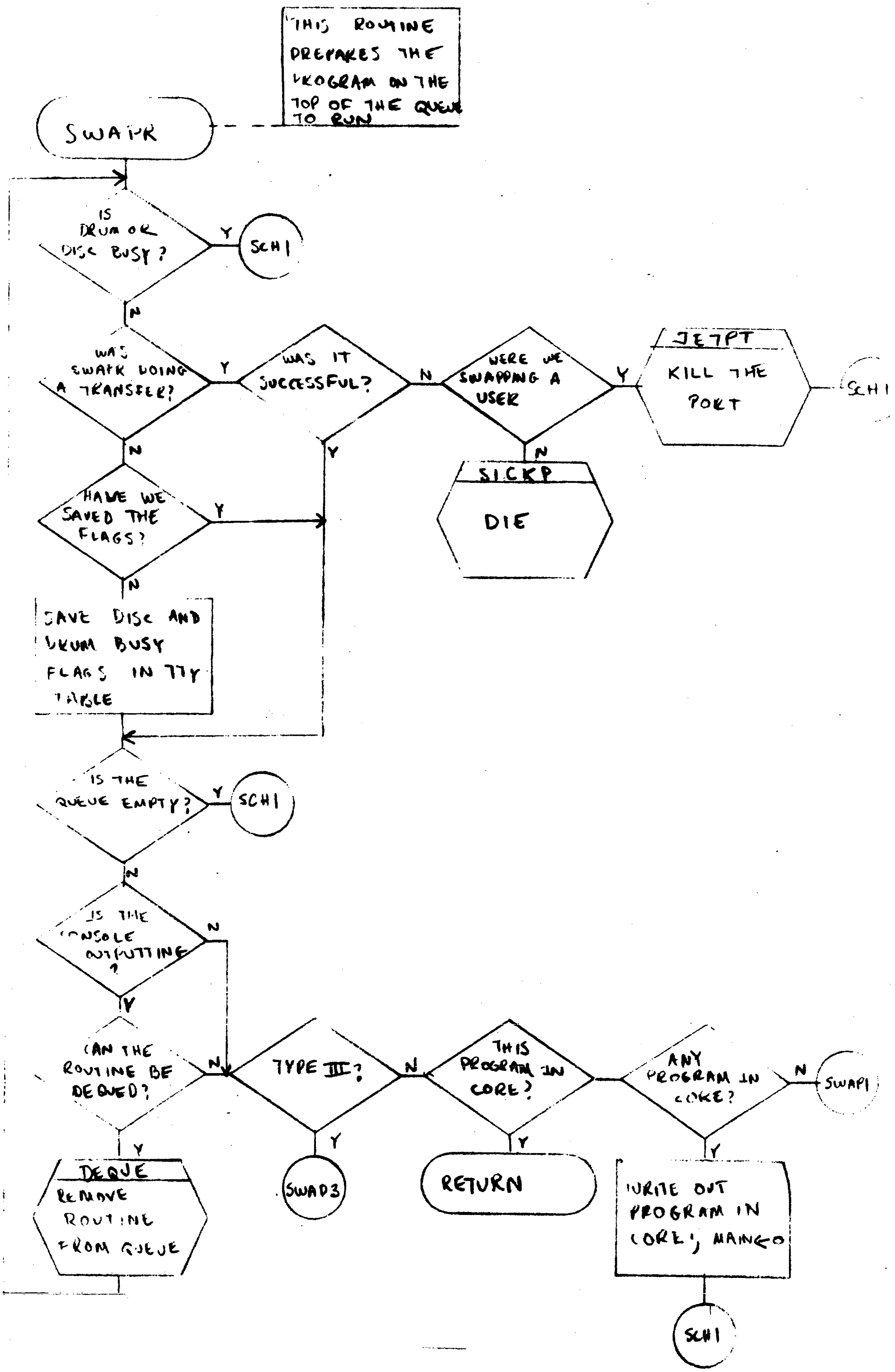

SCH9
CODE FOR FORCED DISCONNECT (USER HUNG UP)
REMOVE USER FROM QUEUE
CLEAR ABORT FLAGS
SCH61


SCH11
AN ENTER STATEMENT WAS ALLOWED ITS TIME [illegible]
SKIP RESTART MEANS ENTER TIMED OUT
RSTR ← RSTR +1
SCH51

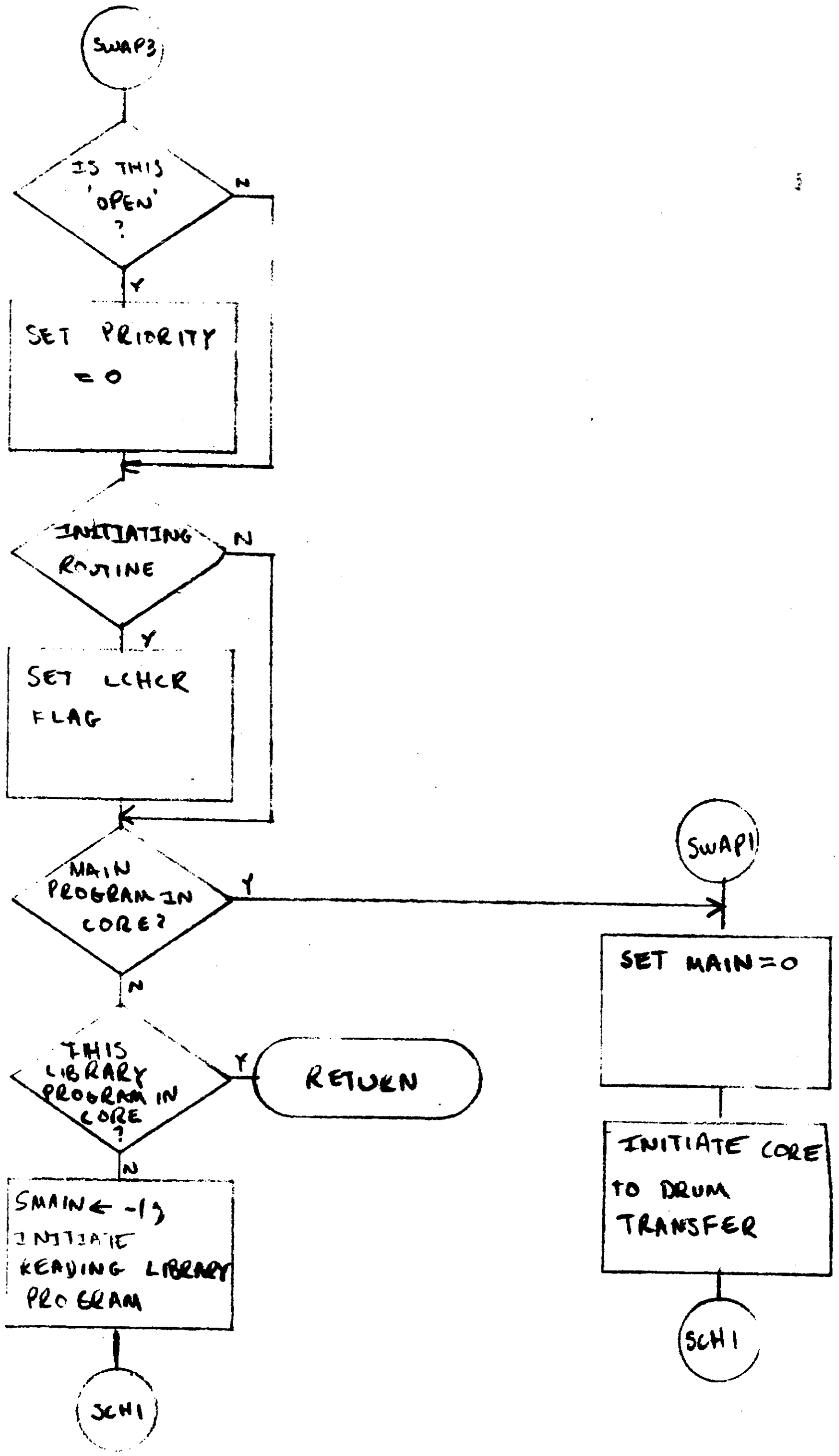

A LINE HAS BEEN ENTERED FOR AN IDLE USER
SCH6
EHERR
SCOM
command
SCH11
ERROR
PRINT "???"
SCH1
SYNTAX
empty line
SCH30
ID = Ø?
(user not logged in?)
PRINT
SCH1
priority ← Ø
set RSTR for syntax
status ← syntax
set 3 for INSCD
SCH3
SCH30
is this user in input mode?
no
print line feed
SCH1
yes
tell communications processor that another line is allowable.
SCH1


COME HERE WHEN A VALID COMMAND HAS BEEN FOUND.
SCH11
ID = ϕ ? (user not logged in)
yes
HELLO command?
NO
print "please log in"
SCH1
yes
NO
is this user in tape mode?
yes
has more input (illegal) been typed?
yes
SCHR9
NO
NO
HELLO, BYE, SCR
NO
tape errors
yes
CHANGE COMMAND TO "TAPE ERROR PRINT"
NO
yes
clear tape error flag
is this program run only?
yes
is command disallowed? (save, csav, lis, pun)
yes
print "RUN ONLY"
SCH1
NO
NO
TYPE I COMMAND ?
yes
CALL THE APPROPRIATE COMMAND PROC.
NO
SCH12


SCH12
set program status
COMMAND TYPE
II
III
set starting add. in LIBUS.
set starting address
set B for INSEQ
SCH3


THIS ROUTINE PREPARES THE PROGRAM ON THE TOP OF THE QUEUE TO RUN
SWAPR
IS DRUM OR DISC BUSY?
Y
SCH1
N
WAS SWAPR DOING A TRANSFER?
Y
WAS IT SUCCESSFUL?
N
WERE WE SWAPPING A USER
Y
JETPT
KILL THE PORT
SCH1
N
SICKP
DIE
Y
N
HAVE WE SAVED THE FLAGS?
Y
N
SAVE DISC AND DRUM BUSY FLAGS IN TTY TABLE
IS THE QUEUE EMPTY?
Y
SCH1
N
IS THE CONSOLE OUTPUTTING ?
N
Y
CAN THE ROUTINE BE DEQUED?
N
Y
DEQUE
REMOVE ROUTINE FROM QUEUE
TYPE III?
Y
SWAD3
N
THIS PROGRAM IN CORE?
Y
RETURN
ANY PROGRAM IN CORE?
N
SWAP1
Y
WRITE OUT PROGRAM IN CORE, MAINGO
SCH1

SCHEDULER


SWAP3
IS THIS 'OPEN' ?
N
Y
SET PRIORITY =0
INITIATING ROUTINE
N
Y
SET LCHCR FLAG
MAIN PROGRAM IN CORE?
Y
SWAP1
SET MAIN=0
INITIATE CORE TO DRUM TRANSFER
SCH1
N
THIS LIBRARY PROGRAM IN CORE ?
Y
RETURN
N
SMAIN← -1; INITIATE READING LIBRARY PROGRAM
SCH1

SYSTEM LIBRARY ROUTINES

FILES

The FILES routine is used by BASIC to process FILES statements 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 15-word entry for each file. Its format is:

1	file length in records (Ø for none)	bit 15=1 if read only
2	logical record size in words	bit 15=1 if "dirty" record bit 14=1 if "dirty" file
3 4	disc or drum address of file's last logical record	drum addresses in words 4, 6, and 8 for sanctified files; for these, words 3, 5, and 7 = 177777_8
5 6	disc or drum address of record currently in core (word 5 = 1ØØØØØ$_8$ if none)	
7 8	disc or drum address of file's first record	
9	pointer to first word beyond core buffer	
10	pointer to current word in buffer	
11	EOF/EOR exit address (Ø for none)	
12 13 14	file name	
15	protect mask	

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

1. Translate characters in FILES statements into the buffer table. FILES statements are pointed to by a four word table in the user swap area which is pointed to by DFILT. FILCT = -5+ # of FILES statements. There may be up to 4 such statements. 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. Those which are specified to be group library files are marked by setting bit 7 of their first word to 1. A "*" alone as a file name is a place holder. The buffer table for the entry is zeroed. Possible errors found in this step are:

   a. File name of 0 or > 6 characters

   b. More than 16 files requested

2. Perform directory search for each specified file. DIRWD is set to the drum address of the directory track in core so that DLOOK doesn't have to read and write the directory for each file. Save the file's drum address (Ø if not sanctified), disc address, file length, and logical record size in its portion of the buffer table. The record size for "*" entries was set to 256 in the previous step. The read-only bit is set if the file is a library file and the user is not the owner. An error occurs if the file is nonexistent or protected. Update the last reference 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. This test is skipped if the user's ID has a letter prefix 'A'. Copy the disc addresses of the requested files into the user's portion of FUSS. Indicate read-only files by setting bit 15 of the upper disc address word.

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.

ASSIGN

The ASSIGN routine is used by BASIC to process an ASSIGN statement in a user's program. The function of the routine is to replace the information currently in the file table referenced by a specified file number with information about a new (usually different) file being assigned to that number. The operation is as follows:

1. If the previous file was written on, update the last changed date word in its directory entry.

2. Search the directory for the new file. If it is not found, is a program, or is protected, exit to non-existent file return. If it has records larger than those of the previous file, exit to that return location. Otherwise save the file's drum address, disc address, length, and record size. Set bit 15 of the length word (read-only) if this is a user accessing a system or group library file not his own. Update the last reference date word in the directory entry for the file.

3. Scan the FUSS table to see if any other user has write capability on the file. If so, set bit 15 of the file's length word unless this is an "A" user. Move the disc address of the file into the appropriate two words of the user's portion of the FUSS table, setting bit 15 of the upper word if the file is read only for this user.

4. Construct the file table entry specified in the description of the FILES routine. Exit to one of three locations, depending on whether the file is: 1) available for reading and writing; 2) read-only (except users Axxx) because of another terminal's read-write access; or 3) read-only because it is a system or group library file.

CHAIN

The CHAIN routine is used by BASIC to process a CHAIN statement in a user's program. The function of the CHAIN routine is to find the program named in the CHAIN statement, retrieve it from the disc, and begin execution. It operates as follows:

1. Dump file buffers.
2. Update the last changed date entry in the directory for each file which was written on.
3. Translate name of program from CHAIN statement. Invalid names exit to error. If preceded by a "$", set up AØØØ search; if preceded by "*", set up group library search; otherwise set for searching on user's ID. Save the line number if any is specified.
4. Perform directory search. Exit to error if not found.
5. Check to make sure that the entry is a program, that it is not ill-stored, and that it will fit. If any of these are not true, exit to the appropriate error.
6. For programs on the disc (not sanctified), initiate a seek by calling the disc driver to perform a zero length transfer.
7. Update date entry in directory and write directory track back to drum.
8. Read in the basic portion of the previous program, including the common area and then append the new program. If the read is unsuccessful, read in the previous program again and exit to error. if successful, move the new program name into the user's table, and if this is a run only program, set the run-only bit, unless the program is in this user's own library. Call SEMIC, which sets up pointers for the language processor, dependent upon whether the program is uncompiled or semi-compiled.
9. Check if an abort was attempted during the previous steps, and if so, abort the user.

<u>CHAIN</u> (cont.)

10. If a line number was specified, search the program for the statement and, if found, put its absolute address into PRGCT. If no line number was specified, set PRGCT equal to SPROG. If the program is null, or if the line number cannot be found, clear the chain bit in the flags word. In any case, exit to SCHBL.

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. Check if the common area has been allocated. If not, call ALCOM, which computes the amount of space required for common. This is used to determine the start-of-program pointer which is saved in word 4 of the directory entry, a device which keeps the common area from being overwritten on GET's and CHAIN's.
4. 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 records) $\leq$ (disc allowed).
5. Search the next disc ADT for the first entry large enough to hold the program. Remember the address of the entry in SAVC.
6. Initiate a seek to the disc address at which program will be written.
7. Perform a directory search on the program to be saved. Fail if such an entry already exists.
8. If the directory track is full, call the SUPERSAVE routine to attempt to reallocate the directory. SUPERSAVE will perform step 9 itself and proceed to step 10.
9. Insert a new directory entry into the directory.
10. Update the IDT and disc ADT.
11. Copy the user's program to its library area. If the write is unsuccessful, set the "ill-stored" bit in the directory and fail.

CSAVE

The CSAVE routine is called by a user to save a program in semi-compiled form. This is the form it has after the symbol table is built. CSAVE operates like SAVE with the following exceptions:

2. and 3. If the user's program is in compiled form, call RSTPT, which restores the symbol table to it's appearance just after it was built, and before variable storage has been allocated. If the program is in uncompiled form, call ALCOM and then jump into the compiler, returning after the symbol table is built.

9. Insert a new entry into the directory, setting bit 15 of word 3 to indicate a semi-compiled program.

11. Read the program and symbol table back from the drum. Prior to writing it to disc, append 7 words which are:

1) SYMTB - symbol table pointer
2) FILCT - -5 + # of FILES statements
3-6) FLSTS - pointers to FILES statements
7) USESN - "USING SEEN" flag

SUPERSAVE

The SUPERSAVE routine is called by the SAVE, CSAVE, COPY, BESTOW 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:LTEMP+7) = disc address of entry

(LTEMP+8) = length of entry

(LTEMP+10) = start of program pointer/record size

(LTEMP+13) = drum address 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 12 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+80) as negative.

3. Now squeeze all the directory entries to the lastmost of the available tracks. This is done by reading the tracks in reverse order and writing 8184 words on each track until we run out of directory entries. The following variables are used in this section:

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

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

K2 = -# of words in core

4. If (# of words on track K1 - K2) > 8184, go to 5. Otherwise update K2 and read this track to the core buffer at location LULEN + K2 (K2 being negative). If K2 ≠ -8184, go to 8. Otherwise set the length into the L1 DIREC entry and write the 8184 word buffer to track L1. Set K2=0 and go to 7.

5. Set ES=(# of words on track K1-K2-8184)/64. That is the number of extra sectors on the track to be read. Set EX = ES * 64. This is the number of extra words. Then read the last (# of words on track K1-EX) words from track K1 to location (LULEN + K2 - # of words on track K1 + EX) and update K2.

6. Write 8184 words to track L1 from location LULEN-8184 and set the length in the L1 DIREC entry. Move the leftover -8184-K2 words to the end of the buffer, resetting K2. Then read the EX words left on track K1 to location LULEN+K2-EX. Set K2=K2-EX.

7. Update L1 to point to the next track to write.

8. Update K1 to point to the next track to read. If we've finished all tracks, go to 9. Otherwise go to 4.

9. If K2 = 0, go to 10. Otherwise write out the -K2 words to track L1 and set the length in the L1 DIREC entry.

10. Zero out the lengths in the DIREC table of all those tracks that no longer have anything written on them.

11. 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 than 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.

12. If L2 = -(PP), we have completely written track L1 so check for L1 = DIREL. If it is, we've written all the tracks, so go to step 18. Otherwise, advance L1 to the next directory track advance PP, set L2 = 0, and repeat this step. If L2 -(PP), go to step 13.

13. If P ≥ 10232, we have read as much as we can, so go to step 15. If K1 = DIREU, there is nothing left to read, so go to step 15. 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 15. 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.

14. If TG = 0, determine if we can insert the new entry. To do this we first determine where the even entry boundary occurs in the core buffer, since we may have read only part of an entry (12 does not divide 64 evenly). If the last entry in the buffer is greater than the entry we are inserting, the entry goes on this track. If this is not the case, go back to step 13. Otherwise, set TG to 1, make a 12-word hole, insert the new entry, set P = P + 12, and go back to step 13.

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

16. 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 finishing 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.

17. 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 12.

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 16.

18. 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; if preceded by a "*", set up for group library 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). Fail if entry is ill-stored (Bit 15 of word 4 of entry is 1).

4. If the program is on the disc, initiate a seek by calling the disc driver to perform a zero length transfer.

5. 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.

6. Set the date into word 5 of the directory entry and write it back.

7. Read in the basic portion of the user area and the common area. Append the library program, reading it in starting with the word specified by the start of program pointer (word 4 of the directory entry). If the read is unsuccessful, read in the previous program again and fail. If successful, move the new program name into the user's table, and if this is a run only program, set the run-only bit unless the program is in this user's own library.

8. Call SEMIC, which sets up pointers as follows: For uncompiled programs, clear CFLAG bit and set SYMTB = 0. For semicompiled programs, set CFLAG bit, move 4 pointers to FILES statements into FLSTS, set FILCT, set SYMTB to point to the first word of the symbol table and set SPTR = 0. For both types of programs set MAIN to point to this user, set SPROG to the start of program pointer and set PBPTR to point past the last word used by the program.

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, and then continues as follows:

4. Check that the program to be appended is not semicompiled and has no common area. Set the date into word 5 of the directory entry and write it back.

5. Load user's current program and call DCMPL. Check that the program to be appended will fit, and if so, read it in at the end of the current program. If the read is unsuccessful, fail.

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 okay, update PBPTR and if the appended program is run-only, set the run-only bit unless the program was retrieved from the user's own library.

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 FLISS, and tell the I/O processor (service routine NUC) that a new user called. 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 4. Check if user has control over the line printer. If not, go to Step 3. Otherwise tell the I/O processor to disconnect the line printer from user (service routine LPD).
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.
4. Translate the new ID code 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.
5. Check if a terminal type was input. If not, assume terminal type #1 and go to Step 6. Otherwise check if terminal type is in the Range 1 through 6. If not, print an error message.
6. Tell I/O processor (service routine STP) which terminal is connected to the port. Check that the time used to date is less than the time allowed.
7. Add a LOGON entry to LOGGR and set the starting time into the user's table. Also insert the ID code, clear the name, clear the program and tell the I/O processor of successful Logon (service routine ULO).
8. Search the directory for a program named HELLO in the library of user Zggg. If not found, or if it is a file, or if it will not fit in core, or if it is ill-stored, or if it can not be read from drum or disc, print READY and terminate.

9. Read in the fixed user area and append HELLO. Call SEMIC, which sets pointers as in SAVE. 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. If user does not have control of the line printer, go to Step 2. Otherwise tell the I/O processor to disconnect the line printer from user. (system calls service routine "LPD" which sets the line printer disconnect flag LPDIS)
2. If user ID is 0, go to Step 3. Otherwise 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.
3. Get ?TYPE from the I/O processor with service routine WTP. If ?TYPE = 0 (ASCII terminal connected to the port), tell I/O processor to restore this port to full duplex. Otherwise tell I/O processor to restore this port to half duplex.
4. Tell I/O processor to disconnect the user (service routine HUU) and 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 (drum) if the entry was sanctified. Restore the space to the appropriate disc 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. He may also specify the first statement to be renumbered and the last statement to be renumbered. If unspecified these are set to the first statement of the program and the last statement of the program respectively.

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, PRINT USING, MAT PRINT USING, 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, except in the case of PRINT USING and MAT PRINT USING, in which case it takes advantage of the fact that there is only one statement number reference. Before calling RENSK for the first time, Q is set to point at the first statement to be renumbered. 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. Translate parameters P and Q. Set RENBA = first statement to be renumbered, RENLA to last statement to be renumbered.

4. Set RENLA to point to the last sequence # $\leq$ RENLA. Also set RENBA to point to the first sequence # $\geq$ RENBA.

5. Insure that there will be no sequence number overlap at either end of the portion of the program to be renumbered and that the new sequence numbers will not exceed 9999.

6. If program is in compiled mode, go to step 9. 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 1,

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

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

why?

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

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

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

these are partitions of statement #'s, and if statements were numbered 1, 2, ..., they would be actual statement numbers

Set Q = SPROG, P ≠ Q-1. (SPROG points to the first statement).

7. Call RENSK to find the next statement reference. If there are none left, go to step 9. Find the largest I for which ERSEC [I] $\leq$ (RENP). If there is none, the statement referenced does not exist, so go to step 8. 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 8.

8. Set (RENP) = (RENP) + 100000_8 and go back to step 7.

9. Change the sequence numbers of all statements to be changed, according to the M, N, BA and LA parameters. If compiled mode, terminate. Otherwise, set Q = SPROG, P = Q-1, and go to step 10.

10. 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.

$N = 48$

$K = 1$

$L = 16$

$ERRSEC[I] = 2 \cdot I + 1 \quad (I < 16) \qquad 1, 3, 5, \ldots, 31$

$ERRSEC[I] = I + 16 + 1 \quad (I \geq 16) \qquad 33, 34, \ldots, 48$

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 "$" or "*", and this is the first character, print an error message and terminate. If a "," 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. Output heading line.
2. Perform directory search on the program with all nulls. Get first directory entry following the one sought.
3. 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 a 'C' if a semi-compiled program or an 'F' if a file (or a blank if neither), then a 'P' if the entry is protected (otherwise, a blank), then a 'S' if the entry is sanctified (otherwise a blank), then the 5-digit program or file length with leading zeroes suppressed, then 3 blanks. Program length is printed in words (stored as negative number) and file length in records.
4. If <4 names have been printed on the line, advance to the next directory entry and return to step 3. Otherwise, copy the name of the last one output into the user's teletype table, output a carriage return and suspend until the buffer is almost empty.
5. Read the name of the last program printed from the teletype table 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 3.

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 AØØØ is used for directory searches instead of the user's id, and ill-stored programs are not listed.

GROUP

The GROUP routine prints a list of all programs and files in the user's group library (the library of the idcode ending in ØØ which has the same letter and first number as the user). Its operation is identical to that of LIBRARY except that the group librarian's idcode is used for directory searches instead of AØØØ.

DIRECTORY - USER CONSOLE

The DIRECTORY routine prints a list of all directory entries on the user console. The entries are printed one per line, and consist of id, name, last reference date, length, disc address and drum address, if any. The operation is as follows:

1. Check that the user's id is A000. If not, fail.

2. Check to see if an idcode was specified. If so, we will start printing the directory with this idcode.

3. Print the heading consisting of system id, date and time and suspend.

4. Print the directory heading and suspend.

5. Set up parameters for directory search for null program name and idcode previously determined (or null idcode if none specified).

6. Perform directory search.

7. Get first directory entry following the one sought. If pseudo-entry, terminate.

8. If id of entry is different from that of the preceding entry, output the ASCII representation of the idcode. Otherwise output four blanks. Save the idcode in the RTIM word of the user's TTY table.

9. Output the six character program name and save it in the TEMP words of the user's TTY table.

10. Output the last reference date.

11. Output a 'C' for semi-compiled programs, an 'F' for files, and/or a 'P' for protected programs or files.

12. Output the length, the disc address, and the drum address, if there is one.

13. Output X-OFF, CR, LF and suspend.

14. Retrieve the parameters for the directory search from the user's TTY table and go to step 6.

SDIRECTORY - USER CONSOLE

The SDIRECTORY routine prints a list of all sanctified programs and files on the user console. The printout is in the same format as a DIRECTORY printout. The routine functions the same as DIRECTORY, except that in step 7 a check is made to see if the entry is sanctified. If it is, the processing continues as in DIRECTORY. Otherwise, the pointer is moved to the next entry and step 7 is repeated.

REPORT - USER CONSOLE

The REPORT command prints IDT information on the user console. For 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 to the IDT until the user logs off. The operation of REPORT is as follows:

1. Check that the user's id is A000. If not, fail.

2. Check to see if an idcode was specified. If so, save it, as we will start printing the report with this idcode. Otherwise save a null idcode.

3. Print the heading consisting of system id, date and time and suspend.

4. Print the report heading and suspend.

5. Retrieve the idcode and find what track its on. Read that track and locate the idcode.

6. Output the id, time and disc of the next three entries. If necessary, read the next id track.

7. If no entries left, print XOFF,CR,LF,LF and terminate. Otherwise save the present idcode + 1, print XOFF,CR,LF and suspend. Go to step 5.

STATUS - USER CONSOLE

The STATUS routine prints a summary of the various system resources and the extent of their utilization on the user console. It operates as follows:

1. Check that the user's id is A000. If not, fail.
2. Print the heading consisting of system id, date and time and suspend.
3. Print MAGSC and a '*' if the mag tape unit is a 7970 (bit 15 of MAGSC=1). Print the select codes of the 4 drums.
4. Print the logical unit, select code, unit number, first block and last block of the discs on the system.
5. Print a list of those tracks which are locked on each drum.
6. Print a list of those disc blocks which have been MLOCKED.
7. Set the ?STAT word in the user's TTY table to the status overlay and suspend so that when we come back the overlay will be read in.
8. Get the line printer select code and type from the I/O processor and print them ('*' if the printer is a 2610A, '**' if the printer is a 2767A). Then print the port number of the current user.
9. Print the drum addresses and lengths of the IDT, ADT, Disc ADT and Directory tracks.
10. Print the drum addresses of the system library tracks and user swap tracks.
11. Print the disc addresses of the 32-block areas reserved for the IDT, Disc ADT and Directory.
12. Print the disc addresses and lengths of the system segments.
13. Terminate.

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 or any group librarian 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. Files are always protected against other users writing on them. The operation is as follows:

1. Check for privileged user.
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 create data files. The user must specify the file name and file length in records. He may also specify a logical record size. The operation of OPEN is as follows:

1. Translate and check the file name and length and record size. File names are subject to the same restrictions as program names. File length must be between 1 and 32767 records, inclusive. Record size, if specified, must be between 64 and 256, inclusive. (The default record size is 256 words.)

2. Check the IDT and disc ADT's to see if a) the user has enough disc space allocated to him to satisfy the command; and b) there is an area on the disc which is large enough to accomodate the file. Save the location of the disc ADT entry and its information, but don't update it until we know 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 disc ADT appropriately.

5. Fill the user area with end-of-file marks (a -1 in the first word of each of 40 256-word blocks). Write this area to the location on the disc reserved for the file. Increment the disc address by 40 blocks and write another 40 records up to 10 times (total) or until the file is full. The last write may be from 1 to 40 blocks in length.

6. If the file has been filled, terminate. If not, save the low word of the disc a-dress immediately beyond the last write in the user's teletype table, along with the file name. Move the user to the bottom of the queue and suspend.

7. Retrieve the file name and partial disc address from the teletype table. Ascertain that the file exists (is in the directory) and that the reconstructed disc address falls within the file. If not, terminate. Otherwise, return to step 5.

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-SPROG. 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 = SYMTB-SPROG.

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 or PLEX1 in the I/0 processor. Bit = 0 implies no echo, 1 implies echo. The user will want echoing if any only if his teletype is full duplex. The command format is:

ECHO-ON for full duplex.
ECHO-OFF for half duplex.

MESSAGE

The MESSAGE command is used to send a message from a user console to the system console. The message is placed in a queue and is ultimately output to the system console by the scheduler. The routine operates as follows:

1. Check if message queue is full. If so, fail.

2. Put a CR-LF and the ASCII representation of the user's port number in the message buffer.

3. Move the message from the user's teletype buffer in the I/O processor to the message buffer.

4. Increment message counter and set pointer to next message buffer.

5. Terminate.

L PRINTER

The LPRINTER command is used to obtain the line printer as the output device. The routine operates as follows:

1. Check to see if the line printer is currently being used. If so print "LP BUSY".

2. Ask the I/O processor to check on the availability of the line printer. The I/O processor will return the following status in the A register:

A≠ 0	Line printer available and on-line. The line printer is assigned to the user and the character string following the command word (if present) will be printed.
A =0	Line printer is not available or not on line. Print "LP NOT AVAILABLE".
A < 0	Line printer available but character string is too long. Print "ILLEGAL FORMAT".

3. Set the LPRINTER command flag, LFLAG, and place the address of the user's port number in the LP user indicator PRIST. Upon completion of the user's next command, PRIST will be cleared, the line printer will be released, and a completion message will be sent to the user. LFLAG is used to indicate that command being terminated is LPRINTER and therefore the line printer is not to be released.

4. Terminate. LFLAG is cleared in the termination routine.

REPORT - SYSTEM CONSOLE

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. Check to see if an idcode was specified. If so, save it, as we will start printing the report with this idcode. Otherwise save a null idcode.
2. Print the heading consisting of system id, date and time and suspend.
3. Print the report heading and suspend.
4. Retrieve the idcode and find what track it's on. Read that track and locate the idcode.
5. Output the id, time and disc of the next three entries to the buffer. If necessary, read the next id track.
6. If no entries left, print the buffer and terminate. Otherwise save the present idcode +1, print the buffer and suspend. Go to step 4.

DIRECTORY - SYSTEM CONSOLE

The DIRECTORY routine prints a list of all directory entries on the system console. The entries are printed one per line, and consist of id, name, last reference date, length, disc address and drum address, if any. The operation is as follows:

1. Check to see if an idcode was specified. If so, we will start printing the directory with this idcode.

2. Print the heading consisting of system id, date and time and suspend.

3. Print the directory heading and suspend.

4. Set up parameters for directory search for null program name and idcode previously determined (or null idcode if none specified).

5. Perform directory search.

6. Get first directory entry following the one sought. If pseudo entry, terminate.

7. 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.

8. Move the 6-character program name to the buffer.

9. Convert the last reference date and put it in the buffer.

10. Convert the drum address and put it in the buffer, unless it is zero.

11. Convert the disc address and length and put them in the buffer.

12. Put a 'C' for semi-compiled programs, an 'F' for files, and/or a 'P' for protected programs or files in the buffer.

13. Print the line and suspend.

14. Set up parameters for directory search. These can be gotten from locations 35, 3, 4, and 5 of the buffer. Go to step 5.

SDIRECTORY - SYSTEM CONSOLE

The SDIRECTORY routine prints a list of all sanctified programs and files on the system console. The printout is in the same format as a DIRECTORY printout. The routine functions the same as DIRECTORY, except that in step 6 a check is made to see if the entry is sanctified. If it is, the processing continues as in DIRECTORY. Otherwise, the pointer is moved to the next entry and step 6 is repeated.

STATUS - SYSTEM CONSOLE

The STATUS routine prints a summary of the various system resources and the extent of their utilization on the system console. It operates as follows:

1. Print the heading consisting of system id, date and time and suspend.

2. Print MAGSC and a '*' if the mag tape unit is a 7970 (bit 15 of MAGSC=1). Print the select codes of the 4 drums.

3. Print the logical unit, select code, unit number, first block and last block of the discs on the system.

4. Print a list of those tracks which are locked on each drum.

5. Print a list of those disc blocks which have been MLOCKED.

6. Get the line printer select code and type from the I/O processor and print them ('*' if the printer is a 2610A, '**' if the printer is a 2767A). Then print the port number of the current user.

7. Set the console status to the status overlay and suspend so that when we come back the overlay will read in.

8. Print the drum addresses and lengths of the IDT, ADT, Disc ADT and Directory tracks.

9. Print the drum addresses of the system library tracks and user swap tracks.

10. Print the disc addresses of the 32-block areas reserved for the IDT, Disc ADT and Directory.

11. Print the disc addresses and lengths of the system segments.

12. Terminate.

ROSTER

The ROSTER routine prints a listing of the id codes of all active users. These are obtained from the ID word in the 32 TTYTABLES. The absence of a user is indicated by the word being zero.

ANNOUNCE

The ANNOUNCE command is used to send a message from the system console to any or all of the user consoles. It operates as follows:

1. Get the port number to send the message to. If 'ALL' specified, set up for sending message to all ports.

2. Output CR-LF-LF, followed by the message, followed by CR-LF-LF to a port. This output is done a character a time, after insuring that the I/O processor can take the character without overflowing the buffer.

3. Move to the next port, and if there are any more ports to do, go to step 2. Otherwise terminate.

NOTE: If a user has the line printer as his output device, the announce message is not sent to him. The flag PRIST indicates the address of the TTY# of the current user.

RESET

The RESET command modifies the time to date of a user's IDT entry. It operates as follows:

1. Set ID = T = 0.

2. If the idcode = "ALL" go to 3, otherwise set ID = the specified ID code.

3. If no time specified, go to 4. Otherwise set T = specified time.

4. Read the IDT track for ID. If ID = 0, go to 5. Otherwise search for the specified idcode. Fail if not found. If found, set its time entry to T, write the IDT track back and terminate.

5. Set the time entry for all the idcodes on this track to T and write it back to drum.

6. Move to the next IDT track. If all are finished, terminate. Otherwise read the IDT track and go to 5.

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 track for this id and locate the specified id. Fail is 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 track back to drum and 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 each user's ?FLAG word in his TTY table.

    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. Tell the I/O processor to disconnect the telephones.

4. Call LCD to update the last change date for files for each port that has a program that is still active.

5. Update the IDT entry for each active user and create a logoff entry in LOGGR.

6. Wait for the console to finish outputting.

7. Read in the loader, turn off all the I/O and the interrupt system, set power fail to halt.

8. Set A = 0 (sleep) and jump to the dump.

HIBERNATE

The HIBERNATE command is identical to the SLEEP command except for the following additions/changes:

0. If MAGSC = 0, fail. Otherwise set the current time into HDATE.
8. Set A = -1 (hibernate) and jump to the dump.

NEWID

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

1. Translate the idcode.

2. Determine what IDT track the idcode is on and read it in.

3. Translate the other parameters.

4. Search the IDT for the specified id. Fail if found. If the track is full go to 5. Otherwise insert the new entry in its appropriate position, update the track length, write the IDT track back to drum and terminate.

5. Scan through IDEC, determine the total length of all IDT tracks, and add 8 for the new entry. If all tracks are full, fail.

6. Divide the total IDT length by the number of IDT tracks to determine their new individual lengths. Insert these in the table at (NNSNN+1:NWSNN+3) as negative.

7. Now redistribute the IDT tracks. The basic idea of the algorithm is to fill the swap area with as much of the IDT 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:

   K1 points to the IDEC entry for track being read

   L1 points to the IDEC entry for track being written

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

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

   S = # of words in core (initially 0)

   SP points to NWSNN entry, telling how many are to be written on L1.

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

8. If L2 = (SP), we have completely written track L1 so check for L1 = NIDC2. If it is, we've written all the tracks, so go to step 14. Otherwise, advance L1 to the next directory track, advance SP, set L2 = 0, and repeat this step. If L2 - (SP), go to step 9.

9. If S ≥ 10232, we have read as much as we can, so go to step 11. If K1 = NIDC3, there is nothing left to read, so go to step 11. 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 11. If it is not zero, read from section K2/64 into core location LIBUS + S. Add the number of words read to S and to K2.

10. If TG = 0, determine if we can insert the new entry. This will be so if K1 = IDLNP and V - LIBD < K2. If this is not the case, go back to step 9. Otherwise, set TG to 1 and insert the new entry in core. Set S to S + 8 and go back to step 9.

11. Write Section. Set SS = 0. This is the number of words written.

12. 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 IDT information. If S-SS<-A, we don't have as much in core as we are capable of writing, so set A = -(((SS-S)÷64) x 64), an exact number of sectors.

13. If A = 0, we can't write anything, so if SS ≠ D slide the remaining S-SS words in core up to location LIBUS, set SS=0 and S=SS-S. Then go back to step 8. If A≠0, write -A words to sector L2÷64 of track L1. If L2 = 0, set the first word of the L1 IDEC entry to the first word written. Set L2 to L2-A, SS to SS-A, and go back to step 12.

14. Set the new IDT lengths into IDEC 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, or if it ends in '00' and any members of that group are logged on, fail. This is because the files belonging to A000 and group librarians may be accessed by other users, and removing them would be almost impossible.

2. Read the IDT track for the specified id and search it for the id. Fail if not found. Otherwise, delete the entry from the IDT and write it back to the drum.

3. If any user with the specified id is currently on the system, set the id item of this TTY table to 0, set his status to -2, set his COM14 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. Remove all directory entries belonging to this user and build a table which will be used to patch the ADT and Disc ADT. For each directory entry, four words are placed in the table: drum address, length, and disc address.

5. Write the directory back to disc. Read the ADT, call RSFS to return the space released from sanctified programs and files, and write the ADT to drum.

6. Call TBDAD to return released space to the Disc ADT.

7. Terminate.

UNLOCK

The UNLOCK command is used to restore drum 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, (8192 + 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] = 128 unless F = 0, in which case we set MEM [MOVED] = 3, MEM [MOVED + 1] = 125. Also set 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 drum and terminate.

LOCK

The LOCK routine is used to tell the system that certain drum 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 is used for swapping, directory, Disc ADT, IDT, ADT, or system. Fail if they are.

2. Search the directory for sanctified entries on the specified tracks. For each such entry, add a 4-word entry to the patch table consisting of a pointer to the Direc entry, a pointer to the entry in the buffer, and the two-word disc address. There is room for 512 entries in this table. Fail if it overflows.

3. Compare the patch table to the FUSS table. Fail if any of the entries in the patch table are active files.

4. Read the ADT, delete from it all entries for the tracks to be unlocked, and write it back.

5. For each track to be locked, set its TRAX bit to 1.

6. Update the directory using the patch table. For programs, set the drum address in the directory entry to 0. Also set the second word of the patch table entry to 0. For files, remove the entry from the directory. Set the first 2 words of the patch table entry to the id and length of the file.

7. Call TBDAD to return disc space formerly used by removed files to the Disc ADT.

8. Call TBIDT to update the space used for users whose files have been removed due to the locking.

9. Terminate.

MUNLOCK

The MUNLOCK command is used to restore to the system any disc blocks which have previously been MLOCKed. It operates as follows:

1. Interpret the parameters. Fail if the last block is less than the first block, or one of the blocks specified is one of the first 4 blocks of a disc, or the first and last blocks are not on the same disc, or the blocks are on a non-existent disc.

2. Read the Bad Blocks Table and Disc ADT for the disc specified.

3. Search the Bad Blocks Table for the first entry greater than or equal to the first block to be unlocked. If an equal entry is found check to see if the locked block is completely enclosed. If not, update the address and length of the locked block entry, return the unlocked space to the Disc ADT and go to 6. If so, eliminate the completely enclosed entry, return its space to the Disc ADT, and go to 5.

4. If a greater entry was found or if the end of the table was reached, check the preceeding block to see if a portion of it is to be unlocked. If so, modify the entry. If the unlocked portion is in the middle of the block, it will be necessary to make a new entry in the table in addition to modifying the entry. After so doing, return the freed space to the Disc ADT. If the end of the table was reached in step 3, go to 6.

5. Check the next block to see if any portion of it is to be unlocked. If not, go to 6. If it is completely enclosed, eliminate it, return the space to the Disc ADT, and repeat step 5. If only part of it is to be unlocked, return this space to the Disc ADT and modify the address and length.

6. Write the Bad Blocks Table back to the disc and the Disc ADT back to the drum.

7. Terminate.

MLOCK

The MLOCK command is used to make certain disc blocks unavailable to the system because they are faulty or for some other reason. It operates as follows:

1. Interpret the parameters. Fail if the last block is less than the first block, or one of the blocks specified is one of the first 4 blocks of a disc, or the first and last blocks are not on the same disc, or the blocks are on a non-existent disc.

2. Read this System Segment Table. Fail if any of the specified blocks are reserved for system segments.

3. Read the Disc Allocation Table. Fail if any of the specified blocks are reserved for system usage as specified in the DAT.

4. Search the directory for programs or files whose disc address lie in the range of the blocks to be locked. For each such entry found, add a 4-word entry to a table consisting of a pointer to the Direc entry for this track, a pointer to this entry position in the directory buffer, and the two-word disc address. If an entry is found which is partially contained in the area to be locked, put its disc address and length in a special table. There may be 2 such entries. There is room for 512 entries in the main table. If it overflows, fail.

5. Read the FUSS table. Compare the table just built with the FUSS to determine if any of the blocks to be locked contain active files. If so, fail.

6. Read the MLOCK overlay.

7. Read the Bad Blocks Table. Then search it for the first entry greater than or equal to the first block to be locked. If an equal entry is found, and the new block is longer, reset the length and go to 8. If it is not longer, terminate. If an equal entry was not found, check the preceeding entry and if the blocks to be locked will make it longer, update its length. If the table was full and the entries were

not combined, fail. If they were combined, go to 9. If the entry was to go at the end of the table and they were not combined, insert the new entry. Then go to 9. If a greater entry was found and the entries were combined, go to 8. If they were not, insert the new entry (unless the table was full, in which case we fail).

8. Check the next entry to see if it is overlapped by the one we just made. If not, go to 9. If so, eliminate the overlapped entry, and change the length of the new entry if it is longer. Repeat step 8.

9. Write out Bad Blocks Table.

10. If there were no entries in the table of directory entries to be removed, go to 13. Otherwise set TP to point to the first entry in this table. Read the directory and set MOVES=MOVED=LIBD.

11. MEM[TP + 1] = > the disc address of the directory entry. Replace MEM[TP] with the id of the entry, MEM[TP + 1] with the length in blocks, MEM[TP + 2] with the drum address, and, if the drum address is not 0, MEM[TP + 3] with the length in sectors. Call the move routine. Set MOVES=MOVES +12. This eliminates the directory entry we were pointing to. Set TP=TP + 4. Bump CT. If CT=0, go to 12. If MEM[TP]=DI, the next entry is on the same track, so go repeat 11. Otherwise move the end of the directory, write out this directory track, read the new directory track, set MOVES=MOVED=LIBD and repeat 11.

12. Read the ADT, call RSFS for each non-zero drum address entry in the table to return space to the ADT, and write the ADT.

13. Read the Disc ADT for this disc. For each of the two special table entries, if they exist call RADT to return their space. This is done because a locked block may remove only part of a program or file, and it is necessary to put the rest of the space back in the Disc ADT.

14. Remove from the Disc ADT any blocks which lie in the range of the blocks we are locking. Then write the Disc ADT back to drum.

15. If there are no entries in the main table, go to 16. Otherwise call TBIDT, which returns space to the IDT based on the first two words of each 4-word entry in the table we built.

16. Terminate.

COPY

The COPY command is used to copy a program or file from one user's library to the library of another user. It operates as follows:

1. Interpret the parameters.

2. Search the directory for the old entry. Fail if not found.

3. Save the file flag, semi-compiled flag, word 4, drum address, disc address and length.

4. Find out which IDT track the new idcode is on, read it, and search for the new idcode. Fail if not found. Also fail if there is not enough space in the user's library.

5. Search the Disc ADT's for enough disc space to put the new program. Fail if not enough space left.

6. Search the directory for the new entry. Fail if found. If the track is full, call SUPERSAVE and go to 8.

7. Add the new entry to the directory and write the directory track back.

8. Read the IDT, update the space used, and write it back.

9. Read the Disc ADT, update the space used, and write it back.

10. If the old program or file is on the drum, read it from the drum, write it to the new entry's place on the disc and terminate. Otherwise, read it from the disc and write it to the new entry's place on the disc 40 blocks at a time until it is completely copied. Then terminate.

BESTOW

The BESTOW command is used to transfer programs or files from one user's library to another user's library. It operates as follows:

1. Interpret the parameters. If no name is given, set name to null and go to 3.

2. Search the directory for the named entry in the old user's library. Fail if not found. Otherwise go to 4.

3. Search the directory for an entry in the old user's library. If one found, go to 4. If none found and this is the first time thru this step, fail. Otherwise print a message if there were any duplicate entries and terminate.

4. Save pointers to the diretory entry and position in the directory buffers of this entry and also save its length.

5. Search the directory for an entry in the new user's library with this name. If found, bump the duplicate entry counter and go to 11.

6. Read the IDT track for the new user and insure that there is enough space in his library for this entry. If not, fail. Otherwise, update his disc space used and write out the IDT.

7. Read the IDT track for the old user, reduce his disc space used and write out the IDT.

8. Read the directory track for the old entry again. Save pertinent information, eliminate that entry from the directory, and write out the directory track.

9. Search the directory for a place to put the new entry. If the track is full, call SUPERSAVE and go to 11.

10. Insert the new entry in the directory and write the directory track back to drum.

11. Increment the name so we won't find the same entry again. If we were only transferring one entry, terminate. Otherwise, go to 3.

SANCTIFY

The SANCTIFY command is used to copy a program or file from the disc to the drum. The space on the disc is reserved so that it may be copied back at sleep time. The routine operates as follows:

1. Interpret the parameters.

2. Search the directory for the named entry. Fail if not found. Also fail if the entry is longer than 32 blocks or if the entry is already sanctified.

3. If the entry is a program, go to 4. Otherwise read the FUSS table and search it for this entry. Fail if found.

4. Read the ADT. Search it for an area large enough to put this entry. Fail if not found. Otherwise, update the ADT and write it back.

5. Read the directory track for this entry again, update the drum address for this entry, and write the directory track back.

6. Read the program or file from disc and write it to the drum.

7. Terminate.

DESECRATE

The DESECRATE command is used to return a santified program or file to it's area on the disc. Programs are not copied back because there is an identical version already on the disc. The routine operates as follows:

1. Interpret the parameters.

2. Search the directory for the named entry. Fail if not found. Also fail if the entry is not sanctified.

3. If the entry is a program, zero its drum address, write the directory track back, and go to 6. Otherwise read the FUSS table and search it for this entry. Fail if found.

4. Zero the drum address of the file and write the directory track back.

5. Read the file from drum and write it to the disc.

6. Read the ADT, call RSFS to return the drum space to it, and write the ADT back.

7. Terminate.

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 $\leq$ today's date.

3. Make sure that FUSS is empty. This is to avoid killing any active files.

4. Read a directory track. Set P = MOVED = MOVES = LIBD. Set ND to point to the end of the directory.

5. Test the entry pointed to by P to see if it should be deleted. If not, go to 7. Otherwise add a 5-word entry to the patch table consisting of id, length, disc address and drum address. Call the move routine and set MOVES = MOVES + 12.

6. If the patch table is full, write out the interim directory, call PURFX, and read back the directory.

7. Set P = P + 12. If P = ND, we are finished with this directory track, so go to 8. Otherwise to to 5.

8. Call the move routine to move the end of the directory track. Write out the track and move to the next track. If all tracks have been read, call PURFX and terminate. Otherwise go to 4.

The PURFX routine uses the patch table to update the ADT, Disc ADT and IDT as follows:

1. Read the ADT. Set MOVES = MOVED = L8192. Examine each entry for a non-zero drum address, and if one is found call RSFS to return space to the ADT. In any case call the move routine after examining the entry to delete the drum address (i.e., make it into a 4-word entry table). After returning all drum space write the ADT back.

2. Call TBDAD to return disc space to the Disc ADT.

3. Call TBIDT to adjust the disc space used in the IDT for each user who has lost programs or files.

MAGTAPE

The MAGTAPE routine is used to set a select code into the location MAGSC. Typing a '*' after the select code indicates that the tape unit is a 7970 and will force bit 15 of MAGSC to be set.

PHONES

The PHONES command is used to tell the I/O processor how long to allow the user to try to successfully log on before disconnecting him. It is originally assumed to be 120 seconds. It can be reset to from 1 to 255 seconds by the PHONES command.

PRINTER

The PRINTER command is used to tell the I/O processor the select code of the line printer and the line printer type. '*' after the SC indicates a 2610A and '**' indicates a 2767A.

SPEED

The SPEED routine is used to configure the specified port(s) at the specified baud rate and character size. The baud rate to be input is computed with the formula:

$$\frac{14{,}400}{\text{Bit Rate}} - 1$$

Where bit rate = # of Chars. per second X # of bits per character including the start and stop bit(s).

Note: If $\frac{14{,}400}{\text{Bit Rate}}$ is not a whole number, it must be rounded off to the nearest integer.

The baud rate range is from 5 to 191.

The character size to be input is the least significant octal digit of the total number of data bits and stop bit(s) in a character and is either 1 or 2 depending on whether the character contains 1 or 2 stop bits.

For the IBM 2741 Terminal an "*" must be input for character size.

Error Conditions:

The message "ILLEGAL FORMAT" is output if

1. A baud rate <5 or >191 is input.
2. A character size other than 1, 2, or "*" is input.
3. A port number outside the range 0 through 31 is specified.

The message "NO CONF. DONE" is output if the port (configuration of a single port) or at least one port (configuration of more than one port) is logged on.

In each of the above error conditions no configuration will take place.

For each port to be configured the system first initiate service routine "CHS" in the I/O processor. "CHS" sets the character size (0 if an "*" was input!) into the receive and send parometer (?RPRM resp. ?SPRM) kept in the teletype table. If the character size = 0, the echo bit in ?RPRM is set to 0, otherwise it is set to 1. If the character size = 0, ?TYPE is set to 1 (indicating that an IBM 2741 terminal is connected to the port. Otherwise it is set to 0, indicating that an ASCII terminal is connected.

Then the system initiates service routine "SPE". This routine sets the new baud rate in ?SPRM and ?RPRM and sets the parity bit in ?SPRM to 1 ("EVEN" parity will be generated on output) if ?TYPE = 0. Otherwise the parity bit in ?SPRM is set to 0. Finally "SPE" retrieves "?SPRM and ?RPRM and output these parameters to the multiplexer board.

PORT

This routine is used to print out the baud rate and character size for which the port(s) is (are) configured.

The system processor obtains the information from the I/O processor.

If an IBM 2741 is connected to a port, an "*" will be printed for the character size.

Error Conditions:

The message "ILLEGAL FORMAT" will be given, if an illegal port number is specified. No information is given.

This command is available to the system operator and the system master.

If this command is given by a user other than the system master, the error message "priveleged command" is output and no information will be given.

For each port the system uses service routines WSP and WCS to obtain the baud rate resp. character size for which the port is configured.

I/O PROCESSOR PROGRAM

The second computer in the 2000C high speed is used for all of the terminal input and output operations for the system. In addition, it takes care of all of the phones logic (answering and hanging up the telephones) and the timing for the ENTER STATEMENT.

2100A Asynchronous channel Multiplexor

To put the Multiplexor into operation, each port on the interface must be primed with two parameters. The parameters are necessary for transmission and reception of data to/from that port. One parameter is used for the send channel and one parameter for the receive channel of that port. Once primed, those parameters will remain in the channel's memory until the power goes down or a "master clear" is executed.

The parameters consist of 16 bits which have the following functions:

Send Channel Parameter

Bits 0-7 Indicates the rate at which the data bits will be transmitted.

Bits 8-10 Indicates the least significant bits of the number of bits, indlucing stop bits, in a character.

Bit 11 Not used by the system.

Bit 12 If set, ASCII parity will be generated.

Bit 13 If set, interrupt on completion of transmission of data will be enabled.

Bit 14 Must be set.

Bit 15 Must be set.

Receive Channel Parameter

Bits 0-7 Indicates the rate at which the data bits will be received.

Bits 8-10 Indicates the least significant bits of the number of bits, including stop bits, in a character.

Bit 11 Not used by the system.

Bit 12 If set, all received data will be echo'd back to the terminal.

Bit 13 If set, interrupt on reception of data will be enabled.

Bit 14 Must be=0.

Bit 15 Must be set.

The Multiplexor consists of two boards, a "data" board and a "status" board. They must be located in "two consecutive I/O slots; the "data" board in the higher priority slot (lower numbered select code) and the "status" board in the lower priority slot (higher numbered select code). Output of data to a send channel (to be transmitted to the terminal) must be in the format.

Bits 0-10	Data*
Bit 11	Must be set, if a "synchronizing" character must be transmitted.** Otherwise, must be = 0.
Bits 12-13	Immaterial.
Bit 14	Must be set.
Bit 15	Must be = 0.

* The ASCII character must be contained in Bits 0-6. Bit 7 must be = 0 since even ASCII parity must be generated. Bits 8-10 must be = 1. The selectric character (6 bits/char.) must be contained in bits 0-5. Bit 6 must be set or reset according to the odd parity of Bits 0-5. Bit 7 must be = 0 and Bits 8-10 must be = 1.

** A "synchronizing" character is issued by the I/O processor to provide one character time delay. It is used to delay output until the terminal has completed a carriage return or line feed. The number of "synchronizing" characters is dependent on the terminal. The format of the "synchronizing" character is as follows:

Bits 0-6	Must be = 1.
Bit 7	Must be = 1 if terminal is a selectric. Must be = 0 for all other terminals.
Bits 8-10	Must be = 1.
Bit 11	Must be = 1.
Bits 12-13	Immaterial.
Bit 14	Must be = 1.
Bit 15	Must be = 0.

The "synchronizing" character is a non-printable character. Besides, providing a means of time delay, it is used to synchronize the terminal at the beginning of every transmission.

Input of data is done with a "LIA" or "LIB" to the "data" board. Its format is:

Bits 0-6 Data bits.

Bits 7-9 Immaterial.

Bits 10-14 Number of the channel on which the data was received.

Bit 15 Not used by the system.

An "LIA" or "LIB" to the "status" board gives information in the following format:

Bit 0 If=1, the interrupt came from the completion of a character transmission (send channel).
If=0, the interrupt came from the completion of a character reception (receive channel).

Bit 1 Not used by the system.

Bit 2 If=1, a "break" signal was received.

Bit 3 Not used by the system.

Bits 4-9 Immaterial.

Bits 10-14 Number of the channel on which an interrupt occurred.

Bit 15 "Seeking" bit. This bit indicates that a "seek" operation is taking place in the circulating memory of the interface. If=1, no data or parameters should be output to the interface.

Method of outputting the "send" and "receive" parameters to the interface:

1. "LIA upper select code"
2. Check seeking bit. If=1, the previous operation was not yet completed. Go back to 1. If=0, proceed to 3.
3. "OTA lower select code" (the parameter is assumed to be in the "A" register).
4. "OTB upper select code (the channel number is assumed to be in bits 10-14 of the "B" register).
5. "STC lower select code".

Output of data is done in the same way.

MULTIPLEXER DRIVER

The multiplexer driver is used by both multiplexer boards. The driver is divided into five sections:

I. Initialization - One routine for each board
II. Receive channel processing
III. Send channel processing
IV. Abort processing
V. Multiplexer end of processing

I. Initialization

The initialization section has two interrupt entry points, MPXIO for the first board, and MPYIO for the second board. If entry to the driver is made at MPXIO, the registers are saved and then both MUX channels are read and saved. YFLAG is now checked to see if the lower priority board is currently using the driver. If the driver is busy, XFLAG is set, the registers are restored, and the program is exited. If the driver is not busy, no flag is set as the lower priority board cannot interrupt. A check of the multiplexer status determines which processing section (input, output, or abort) is needed to service the interrupt.

If entry is made at MPYIO, the registers are saved; the MUX channels are read and saved; and YFLAG is set. The multiplexer status determines the processing section.

II. Receive Channel Processing

The multiplexer supplies whole characters, each of which are examined on reception and echoed back to the terminal. If the user's terminal is an IBM Selectric (?TYPE ≠ 0), the character is first translated into ASCII. Certain characters (rubouts, feed frams, line feeds, and X-OFF) are ignored. 'Control X' signals that the current line is to be deleted. If the character is a '↩, the buffer pointer is backed up one position. If the user has the line printer as his output device: 'Control Q' causes suspension of output to the line printer; 'Control W' results in resumption of output to the line printer.

II. Receive Channel Processing, Continued

All other characters are appended to the user's buffer.

Upon reception of a carriage return, the system processor is notified that the user has entered a complete line and further character input is blocked. If the line was entered in response to an ENTER statement, the user's response time is also sent to the system.

III. Send Channel Processing

If there are characters left in the user's buffer, a test is made to see if there is line feed or carriage return delay pending. If so, a synchronizing character is output and the delay counter is bumped. If not, the user's next character is plucked from his buffer, translated to IBM code if warranted, and sent to his port. If the character was a line feed or a carriage return, the appropriate delay is set up. If exactly ten characters remain in the user's buffer and if his status is output wait, the system is notified that his buffer is almost empty.

If no characters remain in the user's buffer: He is placed in an idle mode if his program is still running; or he is placed in input mode.

IV. Abort Processing

Unless the aborted occurred on the receive channel with the user in output mode, it is ignored. For a valid abort, the abort request is sent to the system and the user's buffer pointers are reset to the beginning of his buffer.

V. Multiplexer End of Processing

Four interrupt combinations can occur. This logic determines which flags to clear, which multiplexer board to enable, and where to transfer program control.

LINE PRINTER DRIVER

This driver is used for the 2767A, 2778A, and 2610A line printers. Normal entry is from the idle loop and once entered, the driver replaces the idle loop until output is completed. The flag, LPTYP, indicates which line printer is on the system:

2767A = -1	2778A = 0	2610A = 1

Characters are obtained from the user's buffer, and then tested before being sent to the line printer. If the character is a carriage return or a line feed, a print control character is output. If the character is an X-OFF and the next character is an X-OFF, line printer output is temporarily suspended. In addition, control characters and rubouts are ignored and lower case characters are converted to upper case.

If the line printer goes out of READY status, the user's buffer pointers are saved and new pointers are set to an error message buffer. Output to the user's teletype is then initialized and when the transmission has completed, the buffer pointers are reset and the driver waits for READY status.

2100 DATA SET CONTROL INTERFACE

The data set control board is used in the "SCAN" mode so that an interrupt will only occur if a change in either of the two signals ("carrier" and "Data set ready") has been detected. To prime the board for an interrupt is is necessary to output a parameter with the following format:

Bit 0 "Data set ready" bit. If = 0, an interrupt will occur when "Data set ready" comes up. If = 1, an interrupt will occur when "Data set ready" drops.

Bit 1 "Carrier detect" bit. If = 0, an interrupt will occur when "carrier" comes up. If = 1, an interrupt will occur when "carrier" drops.

Bit 2 Enable bit for comparison Logic. If = 1 and if the comparison Logic detects a change in "Data set ready", the flag will be set and scanning is stopped.

Bit 3 Enable bit for comparison Logic. Same as Bit 2, but applies to "carrier detect".

Bit 4 = 1 for "Data terminal ready" on.
= 0 for "Data terminal ready" off.

Bit 5 Must be = 1.

Bit 6 Enable bit for "Data terminal ready". If = 1 and Bit 4 is = 1, "Data terminal ready" will be transferred to the interface.

Bit 7 Must be = 1.

Bits 8-9 Immaterial.

Bits 10-13 Channel Number.

Bit 14 If = 1, Bits 0-3 will be transferred to the interface.

Bit 15 Must be = 1 for operation in "scan" mode.

On an interrupt because of a change of "Data set ready" or "carrier" the obtained status has the format:

Bit 0 If = 0, "Data set ready" has dropped.
If = 1, "Data set ready" has come up.

Bit 1 If = 0, "Carrier" has dropped.
If = 1, "Carrier" has come up.

Bit 2 Has the same value as Bit 2 in the parameter, output to the interface.

Bit 3 Has the same value as bit 3 in the parameter, output to the interface.

Bits 4-7 = 0.

Bits 8-9 Not used by the system.

Bits 10-13 Number of the channel on which the interrupt occurred.

Bits 14-15 =1.

After examining the status and taking the steps necessary to connect, disconnect, set up log-on timing, etc., the interface has to be primed for the next interrupt, based on new conditions of change in "Data set ready" and "carrier". This can be simply accomplished by outputting the obtained status to the board.

DATA SET CONTROL BOARD DRIVER

The driver for this board is used in the "scan" mode so that an interrupt only occurs when a change in the signals "Data set ready" (=CC) and "Carrier detect" (=CF) is detected by the board. As soon as an interrupt occurs, the new status of the channel is compared with the previous status which was saved in ?PPRM teletype table entry. Depending on that comparison one of the following will be executed:

1. The phone is answered, "LTBT" bit in ?STAT is set and log on timing of 120 seconds (subject to change by the system operator with PHOnes command) is stored in ?PHON.
2. "LDBT" bit is set in ?STAT and dropout timing of 2 seconds is stored in ?PHON.
3. "LDBT" bit is reset in ?STAT to signal that connection was restored within the 2 seconds dropout timing.

On exit the new channel status is saved in ?PPRM and output to the board.

<u>NOTE</u>: The "LTBT" or "LDBT" bit will cause the time base generator routine to start timing using the value of ?PHON as a counter.

INITIALIZATION

I. When the I/O processor program is started at 'INI" (initiated from location 2), the following is done:

1. Do a "master clear".
2. Set "CKFLG" to 0 (flag to be used by the power fail routine).
3. Initialize all 32 teletype tables:
   - A. Set ?TYPE = 0 (to terminal type #1).
   - B. Set CR-DELAY and LF-DELAY for terminal type #1 in ?CDLY resp. ?LDLY
   - C. Set ?RPRM to 110 baud, char. size = 2 and echo bit on.
   - D. Set ?SPRM to 110 baud, char. size = 2 and "even" parity.
   - E. Set "Data term. ready" on, "req. to send" on, "carrier detect" off and "data set ready" on.
4. Set phones timing to 120 seconds.
5. Set "NPORT" = 32
6. Initialize processor interconnect board.
7. Initialize power fail board.
8. Go to idle loop.

II. When the I/O processor is updated at the system update entry "INIF", the following is done:

1. Do a "master clear".
2. Get number of available ports and save it in "NNPRT".
3. If "NNPRT" is larger than "NPORT", initialize the ports with port numbers between NPORT and NNPRT (see I, Sub 3). for ports with port numbers above "NNPRT" set the enable bit (Bit #13) in ?RPRM teletype table entry to 0 and set "Data term ready" off in ?PPRM teletype table entry. If "NNPRT" is less than "NPORT", set the enable bit in "Data term. ready" off in ?PPRM for all ports with port numbers above "NNPRT". If "NNPRT" is equal to "NPORT", go to 4.
4. Set in all teletype tables:
   - A. ?CCNT = 0
   - B. ?BPNT = ?BGIN
   - C. ?BSTR = ?BGIN
   - D. ?BHED = ?BGIN
   - E. "IDBT" bit (Bit #3) = 0 in ?STAT
   - F. ?DCNT = 0
   - G. ?SCNT = 0

5. Output ?RPRM & ?SPRM to the multiplexor board(s) and ?PPRM to the data set control board(s) for each port up to the port with port number = "NNPRT".
6. Set "CKFLG" = 1
7. Store contents of "NNPRT" into "NPORT". ("NPORT" indicates now the current number of available ports).
8. Initialize the time base generator board for 100 MS time interval.
9. Initialize the multiplexor board(s) and the data set control baord(s).
10. Go to I, Sub 6.

POWER FAIL AND RECOVERY

If the computer is running when a power failure occurs, the current machine and I/O status is saved. A flag is set to indicate that this status has actually been saved and the program halts.

If the computer is halted when a power failure occurs, the power down interrupt does not occur.

When power is restored, the flag is checked to determine whether or not the power down interrupt was processed. If not, the initialization section (entry point "INI") is called. If so, the program restarts using the saved status and all of the ports buffers and status are retained. The parameters on the multiplexor board(s) and data set control board(s) are re-instated as before power failure. When power is restored, the line printer will be disconnected from a user who had control over it at the time of power failure.

TIME BASE GENERATOR

The time base generator driver is entered every 100 MS. For every available port the driver will scan for "LTBT", "LDBT", "HUBT", "ENBT" and "PDBT" bit in the ?STAT teletype table entry. If one of these bits is set, the following actions will be taken:

A. "LTBT" bit set.

1. Update timing counter in ?PHON for log on timing (See write up of data set control board).
2. If counter becomes zero, reset "LTBT", "LDBT" and "HUBT" bits in ?STAT. Then set "PDBT" bit and output "Data term. ready" off to the appropriate data set control board forcing the phone connection to be broken.
3. Also the "IOBT" bit in ?STAT will be set, ?TYPE will be set to 1 if type #2 terminal is connected; CCNT, ?DCNT and ?SCNT will be set to zero.

B. "LDBT" bit set.

1. Update timing counter in ?PHON for line dropout timing (See write up of data set control board).
2. If counter becomes zero, tell system processor that user has hung up (communication code "UHU").
3. Reset "LDBT", "ENBT" and "ICBT" bits in ?STAT.
4. Execute A, Sub 3.

C. "HUBT" Bit set.

1. Check ?CCNT indicating the number of characters to be output. If ?CCNT not = 0, check if "LTBT" or "LDBT" bits are set. If ?CCNT = 0, execute A, Sub 2 and A, Sub 3.

D. "ENBT" Bit set.

1. Check ?CCNT. If ?CCNT not = 0, check if "LTBT", "DBT" or "HUBT" bit is set. If ?CCNT = 0, update enter timing counter in ?TIMO.
2. If counter becomes zero, remove "ENBT" bit and set "NIBT" bit in ?STAT. If type #2 terminal is connected, set Bits 8 through 12 in ?TYPE. This will force the selectric into receive mode. Fetch ?RPRM from teletype table, set echo bit (Bit #12) to zero and output it to the multiplexor board.

2. (Continued)

   Tell system processor that user was timed out (communication code "ETO").

E. <u>"PDBT" set.</u>

   1. Reset "PDBT Bit in ?STAT
   2. Tell the data set control board to stop scanning.
   3. Take status on the data set control board for the appropriate channel. Output to the data set control board the phones parameter with "Data term. ready" <u>on</u>, "Req. to send" <u>on</u>, "carrier detect" = 0 (if status indicates that it is = 1) and "Data set ready" = 0 (if status indicates that it is = 0) or = 1 (if status indicates that it is = 1).

TELETYPE TABLES

The teletype tables are located in base page and contain information about the system users. Each of the 32 users has one table containing the following entries:

?TNUM Port number in Bits 8-12

?CCNT Used by MPX for counting output characters. It equals -# of characters, including current one.

?BPNT On input - Points to the character location into which the next character will be deposited.

On output- Points to the last character transmitted.

?BSTR On input - Points to the first character of the most recent buffer.

On output- Points to the location into which the next character will be placed by the outcr routine.

?BHED On input - Points to the next character to be fetched.

?BSAV Saved buffer pickup pointer.

?BGIN Points to beginning of physical buffer.

?BEND Points to first character following physical buffer.

?STAT

TPBT	EQU BITØ	User is in tape mode
TPNBT	EQU NBTØ	
STBT	EQU BIT1	User was turned off
STNBT	EQU NBT1	
CXBT	EQU BIT2	'Control X' was hit
IOBT	EQU BIT3	User is in input mode
IONBT	EQU NBT3	
LDBT	EQU BIT4	Line dropout occurred
LDNBT	EQU NBT4	
LTBT	EQU BIT5	Wait for log timing
LTNBT	EQU NBT5	
ENBT	EQU BIT6	Timing for <ENTER>
ENNBT	EQU NBT6	
RNBT	EQU BIT7	User is running
RNNBT	EQU NBT7	

TELETYPE TABLES, Continued

```
PDBT    EQU BIT8        Phone disconnected

NIBT    EQU BIT9        No input allowed
NINBT   EQU NBT9

HUBT    EQU BIT1Ø       Hang user up

XOBT    EQU BIT11       X-OFF was read from terminal

STYP2   EQU BIT12       *

STYP3   EQU BIT13        *

STYP4   EQU B1213        **  Teletype subtypes

STYP5   EQU BIT14        *

STYP6   EQU B1214       *

ICBT    EQU BIT15       Input configuration needed

ICNBT   EQU NBT15

?ATIM                   Contains allowed time for <Enter Statement>
                        execution.

?TIMO                   Timeout value for user executing <Enter
                        Statement>.

?PHON                   Used as time counter for phones logic.

?TYPE                   Terminal Type:  ASCII      =Ø

                                        EBCD       Bit Ø =1
                                                   Bit 15 = Ø
                                        Call/36Ø   Bit Ø = 1


        FOR EBCD & CALL/36Ø TERMINAL:


CDBT    EQU BIT1        Code determined

UCBT    EQU BIT2        Upper case mode
UCNBT   EQU NBT2

CNBT    EQU BIT3        "Cent" character
CNNBT   EQU NBT3

CCBT    EQU BIT4        "CentC" character

CRBT    EQU BIT5        "CR" Bit (Output only!)
CRNBT   EQU NBT5

XBIT    EQU BIT6        "Control X" was input
XNBIT   EQU NBT6
```

```
CBBT    EQU BIT7          "Circle C" was sent
CBNBT   EQU NBT7
            BIT8          Circle D *
            BIT9          SYNC      *   Transmit
            BIT1Ø         Space     **  Interrupt
            BIT11         Space     *   Bits
            BIT12         Space    *
?CDLY                     Carriage return delay (negative).
?LDLY                     Line feed delay (Negative).
?DCNT                     CR and LF delay counter.
?SCNT                     Character counter used for determining
                          carriage return delays.
?RPRM                     Receive channel parameters.
?SPRM                     Send channel parameters.
?PPRM                     Phone parameter.
```

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

```
?TNUM  EQU  Ø
?CCNT  EQU  1
  .
  .
  .
?PPRM  EQU 23
```

These symbols are primarily used for adjusting pointers to the table. For example, if the B register contains a pointer to the STAT entry of some user, the instruction   ADB   .+ ?PHON-?STAT   will point B to his PHON entry.

. is a symbol in base page at the Ø entry of a table of constants from -2Ø to +2Ø. A word containing the value N, where -20 < N < 2Ø can be referenced by .+N.

SELECTRIC CONVERSION ROUTINES

There are two conversion routines in the I/O processor, both of which are entered from the multiplexor driver. The input conversion routine handles Call/360 or EBCD-to-ASCII conversion. The output conversion routine handles ASCII-TO-Call/360 or EBCD conversion.

The conversion routines use a set of bits which are stored in ?TYPE (See teletype tables).

Input Conversion

On entry a check of the "CDBT" bit is made. If it is not set, the code determination section is entered. In this section (if it is the 1st input character in the buffer) the input character is compared with the character "H" (in the "HELLO" command) in EBCD and Call/360 code. If it is the "H" in EBCD code, Bit #Ø and Bit #15 are set to Ø resp. indicating that input came from an EBCD terminal. If it is not the "H" in EBCD code, Bit #Ø and Bit #15 are set to 1 resp. Ø indicating that input came from a Call/360 terminal. Consequently if a user logs on with a 1st character other than an "H" from an EBCD terminal, his input will be treated as coming from a Call/360 terminal. The user's log on command will naturally not be recognized by the system so that the system will output "???" which will appear as "LLL" on the user's terminal. After exit out of the code determination section, the actual conversion routine is entered. If the user logs on correctly, the system will tell the I/O processor that user is logged on and in the "ULO" service routine the "CDBT" will be set.

If, on entry to the input conversion, the "CDBT" is set, the code determination section will be bypassed and the actual conversion routine will be entered.

Output Conversion

On entry a check is made if a "transmit interrupt" (See write up of Selectric terminal) has to be generated. If not, the actual conversion routine will be entered.

Both reoutines use the "CNBT" and "CCBT" bits if a character is input or must be output which requires a two or three character sequence (See manual).

If the selectric character "¢" is being input, the "CNBT" bit will be set. If the "¢" is immediately followed by a character as required by a two character wequence, the "CNBT" bit will be reset and the ASCII-equivalent inserted in the buffer. If not, the "CNBT" will be reset but nothing will be inserted in the buffer. If the "¢" on input is immediately followed by a "C", the "CCBT" bit will be set. (The "CNBT" bit was already set when "¢" was input). If the sequence "¢C" is followed by a character out of the Range A through Z, the "CNBT" & "CCBT" bits are cleared and the equivalent ASCII control character inserted in the buffer.

On output only the two character sequence is involved. If the "¢" is output, the "CNBT" will be set. If the "upper case code" (next character is in upper case and terminal is in lower case mode) or the "lower case code" (next character is in lower case and terminal is in upper case mode) is output, the "CCBT" bit is set. After output of the two character sequence, both bits are cleared.

The "UCBT" bit is set if an "upper case code" is input from the terminal or output to the terminal. If the "UCBT" bit is not set either a "lower case code" was input or output.

The "CRBT" bit is only used on output. The purpose of this bit is to prevent a line feed to be sent to the terminal if preceded by a carriage return. The reason for this being that the selectric is a "new line code" terminal. It will perform a carriage return + line feed on receipt of a carriage return.

The "XBIT" bit is used when the three character sequence "¢CX" (="X^C" in ASCII) is received from the terminal. The purpose of this bit is to let the I/O processor output "¢/" (="/" in ASCII) <u>if</u> the "¢CX" is immediately followed by a carriage return, the "XBIT" is also cleared but no action will be taken by the I.O processor.

The "CBBT" bit is set on output of the "Circle C" code to the terminal.

Bits #8 through #12 are used in the "transmit interrupt" operation.

CONVERSION TABLES FOR THE IBM 2741 TERMINALS TRANSMITTING EBCD & CALL/360 CODES

There are two conversion tables; one table for the EBCD-ASCII (and ASCII-EBCD) conversion and one table for the Call/360 - ASCII (and ASCII-Call/360) conversion. Each table consists of 177 locations, starting at a location pointed to ^{8}by "CTBPI" (for call/360-ASCII Conversion) resp. "CTBP2" (for EBCD-ASCII conversion). The organization of each table is as follows:

The upper part of each location is used for conversion on input, the lower part for conversion on output. The upper part of each location contains the ASCII-equivalent of the input character and the lower part the EBCD or Call/360 equivalent of the output character.

Method of Fetching the Character Equivalent

A. INPUT

Get the pointer to the appropriate conversion table and add to it the octal code of tye input character (if the input character is in upper case, set Bit #6 of the sum just acquired). The sum is the address of the location in the conversion table where the character equivalent (ASCII) is stored. Fetch the contents of this cell, mask off the lower part; the remainder is the ASCII-equivalent of the input character.

B. OUTPUT

Get the pointer to the appropriate conversion table and add to it the octal code of the output character (ASCII). This sum is the address of the location in the conversion table where the character equivalent (EBCD or Call/360) is stored. Fetch the contents of this cell and mask off the upper part. The remainder is the Call/360 of EBCD equivalent of the output character.

If Bit #7 of the Call/360 or EBCD equivalent is set, that character is in upper case. Bit #6 is the parity bit (odd parity).

NOTE: The way to determine which conversion table to use, is to examine the ?TYPE entry of the teletype table. Bit #Ø=1 and Bit #15=Ø for a Call/360 terminal. Bit #Ø=Ø and Bit #15=1 for an EBCD terminal.

HARDWARE CONFIGURATION

I/O PROCESSOR

10 PROCESSOR INTERCONNECT

11 PROCESSOR INTERCONNECT

12 TIME BASE GENERATOR

13 1 ST MULTIPLEXOR

14 1 ST MULTIPLEXOR

15 DATA SET CONTROL FOR 1 ST MULTIPLEXOR

UP TO 16 TERMINALS

16 LINE PRINTER
(optional)

MORE THAN 16 TERMINALS

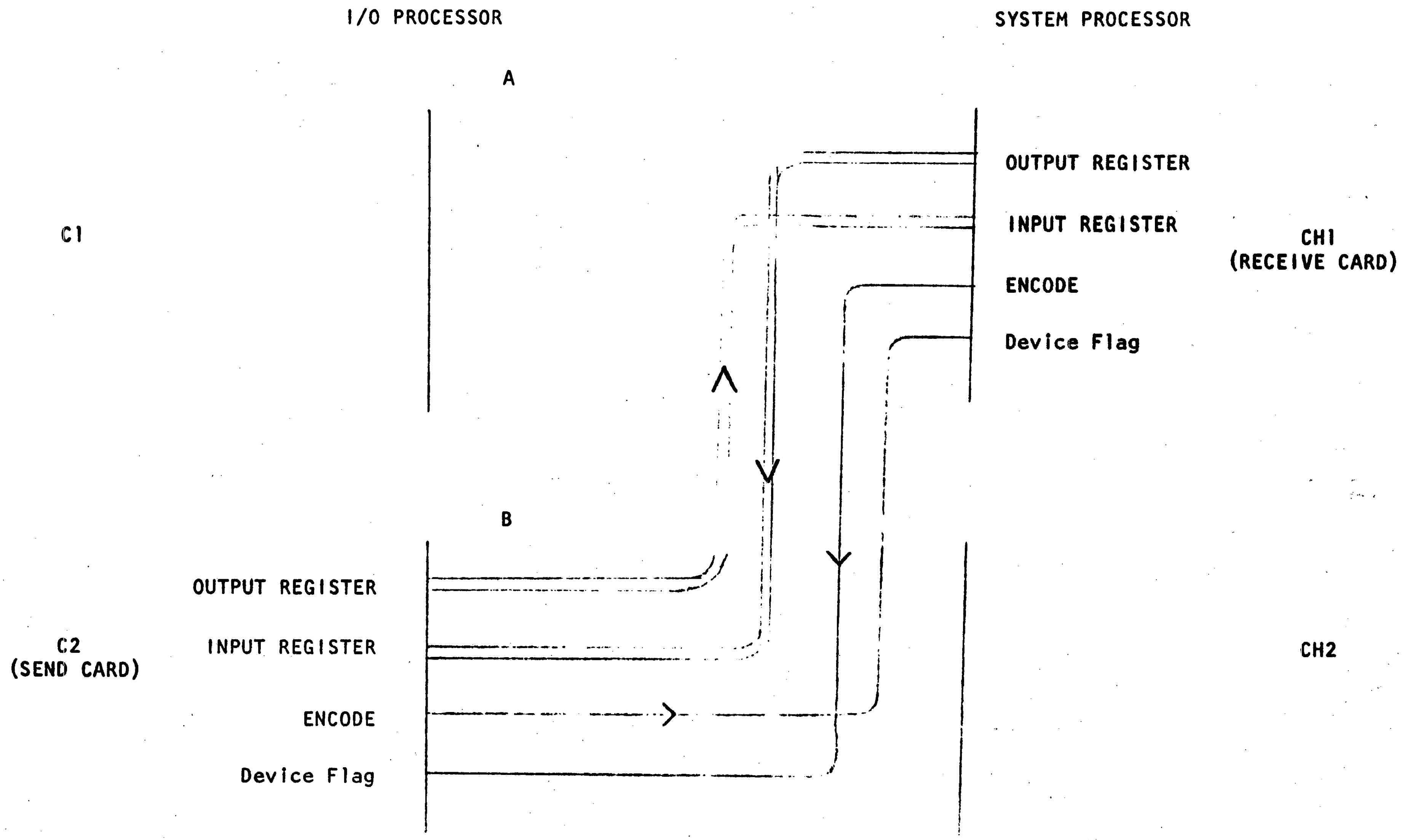
16 2 ND MULTIPLEXOR

17 2 ND MULTIPLEXOR

20 DATA SET CONTROL FOR 2 ND MULTIPLEXOR

21 LINE PRINTER (optional)

PROCESSOR INTERCONNECT



NOTE: CABLE IS NOT SHOWN FOR CH2 - C1 CHANNEL.
IT IS IDENTICAL TO THAT FOR C2 - CH1

In the Idle State, the PI cards are set up as follows:

C1(CH1)	CONTROL & ENCODE:	SET
	FLAG & IRQ :	CLEAR
C2(CH2)	CONTROL & ENCODE:	CLEAR
	FLAG :	SET
	IRQ :	CLEAR

A data transmission operation occurs thusly:

1) Sending machine waits for flag to be set on C2(CH2) indicating that the previous transmission has been processed.

2) Sending machine places data word in output register of C2(CH2) thereby placing it on the input register of CH1(C1). (OTA/B C2(CH2))

3) Sending machine issues STC, CLF to C2(CH2) making the ENCODE LINE go high, setting FLAG ON (CH1 (C1), clearing ENCODE on CH1(C1), and strobing data word into CH1(C1).

4) Sending machine issues CLC to C2(CH2) to prevent an interrupt from that card. The sending machine is now free to return to other tasks.

5) In the receiving machine, T5 will set the IRQ on CH1 (C1). If the interrupt system is enabled and the priority line is high, the IRQ will cause an

interrupt to a service routine.

6) The service routine does an LIA/B from CH1(C1) and decodes the 16 Bit data word.

7) If a response is called for, the receiving machine can load the output register with a data word (OTA/B CH1(C1)).

8) When the receiving machine has completed its processing, it issues an STC,C(CLF) to CH1(C1) which restores the cards to the idle state.

The following is the resultant statuses of the two computers after a command has been sent, received, and acknowledged:

a) The flag is set on C2(CH2) of the sending computer indicating that another transfer is now allowed. This occurred when the receiving computer issued an STC,C to CH1(C1) after it had decoded and executed the command. The STC,C is the acknowledgement to the SEND computer that the RECEIVE computer did receive the transmission.

b) The control on C2(CH2) is cleared by the CLC to C2(CH2). This was done to inhibit the interrupt that normally would occur after the SEND computer outputted the command.

c) The control is set and the flag is cleared on CH1(C1) (from the STC, C acknowledgement)indicating readiness to receive another transmission.

TWO PROCESSOR POWER FAIL CHARACTERISTICS

The two processors in the 2000B system have independent power supplies and consequently, power failure interrupts in either machine may occur at different times.

A problem arises if one computer is powered down, and the other machine attempts to send a transmission. Data will be lost as well as possible subsequent data transmissions. This is apparently caused by stray encode and data levels while power is coming up.

The internal consequences of a lost data transmission are these:

1. A line (Syntax, command, or input) being processed will be garbled.
2. Output characters will be lost. This problem will be hidden by the fact that the current output character is garbled (mux quits sending during character).
3. Under rare circumstances, such communications as echo-on, echo-off, phones-xx will be lost.
4. Terminals on which a carriage return has come in may never have that line processed by the 2116. The terminal will not accept input and the "Break Key" must be used to re-establish communications when power is returned.

5. The 2116 may loose the signal that indicates that the buffer for this user is almost empty. The terminal will stop typing and the program will remain in I/O suspend. The "Break Key" must be used to re-establish communications when power is returned.
6. The 2116 may loose the signal that indicates that the buffer for this user is full. The circular nature of the buffer will cause characters to be typed out of order. The probability of this error is almost zero.
7. If several users are typing on the 2114 and the 2116 is not running, all multiplexor activity may cease (2114 waiting for transmission to be acknowledged). This leads to the classic symptoms, i.e., no response to any struck key (even break), and termination of all output operations, perhaps with a space on the line (teletype chattering).

If the primary power source fails, the two machines will go down within milliseconds of one another and it is not so likely that any transmission will be in progress thereby being lost. If however, an individual processor's power is lost via a depression of the processor's power switch or a malfunction in its power supply one of the above symptoms is sure to occur if there is significant activity on the system.

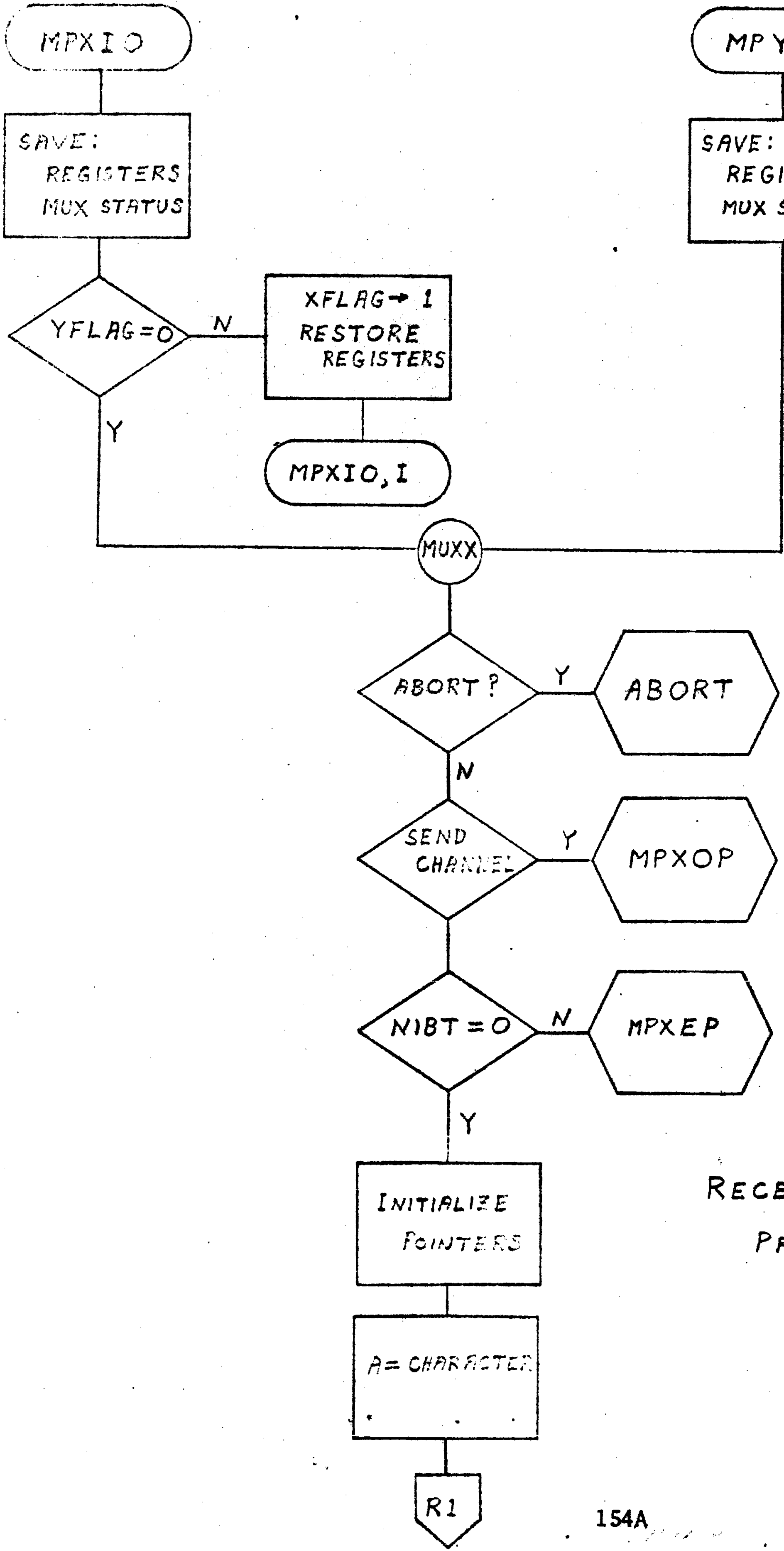
If the 2116 is powered down first, the 4th, 5th, and 7th cases listed are probable. If the 2114 is powered down first, the 2nd case listed is probable. In other words, turn off the 2114 first if the system must be powered down. The two machines should not be turned off together.

If it is necessary to power down the system, and a common power switch does not exist, it is necessary to power down the 2114 prior to the 2116. Restart procedure dictates that the 2114 is powered up last.

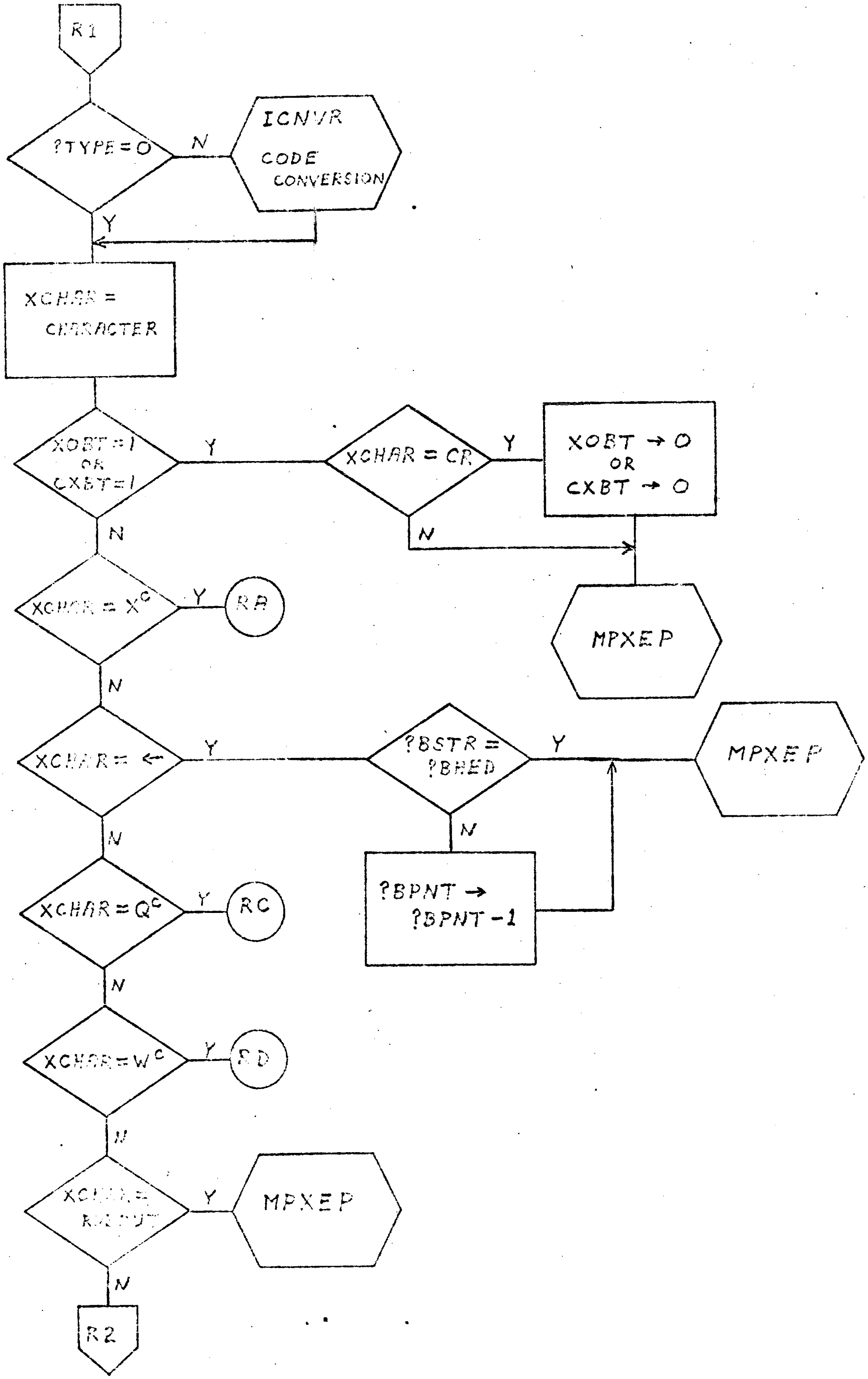
CORE ALLOCATION IN I/O PROCESSOR PROGRAM

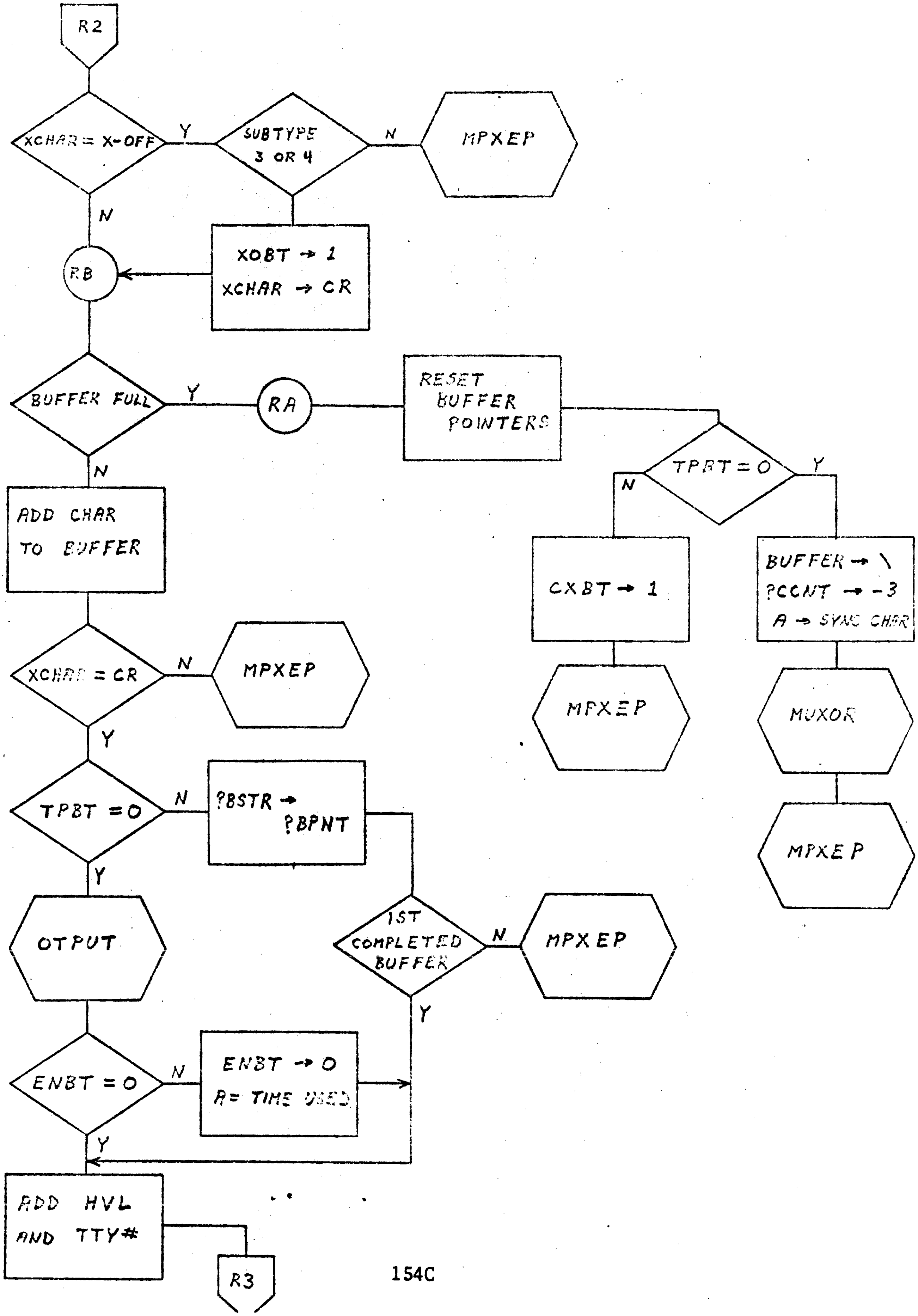
0000	interrupt locations, variables, & constants
0350	TELETYPE TABLES
1550	SYSTEM PROCESSOR DRIVERS
1700	multiplexor driver routines
3000	LINE PRINTER DRIVER ROUTINES
3350	SYSTEM PROCESSOR SERVICE ROUTINES
4600	PHONES LOGIC
5000	SELECTRIC CONVERSION ROUTINES
6200	TIME BASE GENERATOR ROUTINES
6500	INITIALIZATION
7000	POWER FAIL
7700	TELETYPE BUFFERS 250 CHAR. EACH
17700	BBL
17777	

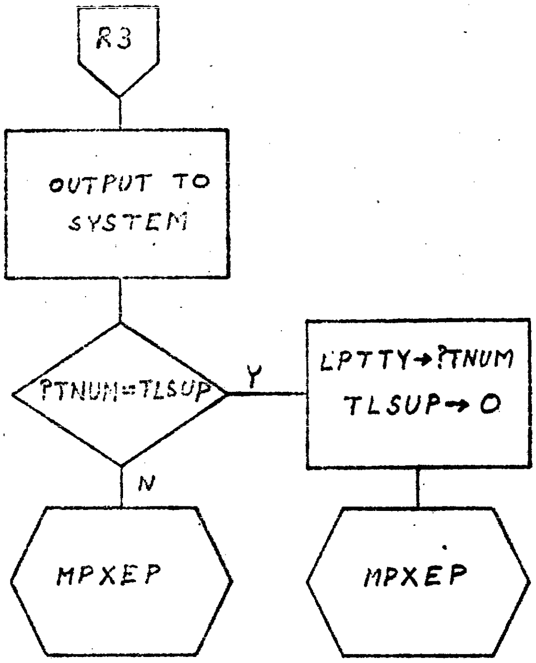
INITIALIZATION SECTION

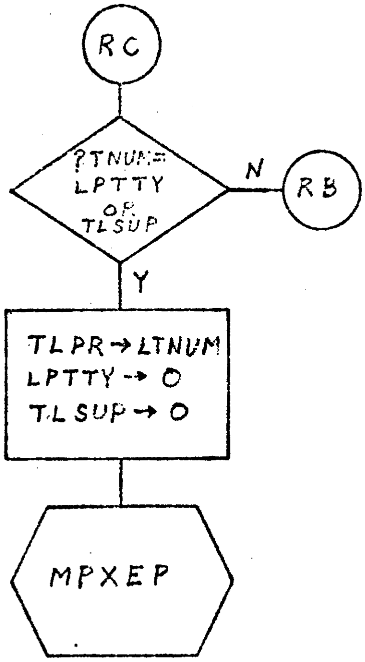


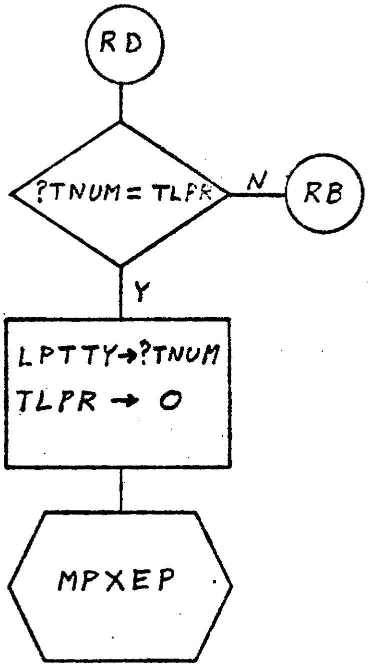
RECEIVE CHANNEL
PROCESSING

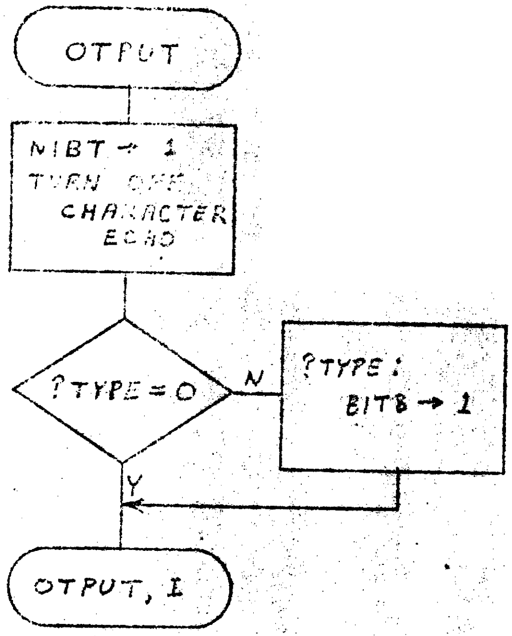

R1
?TYPE = O
N
ICNVR
CODE
CONVERSION
Y
XCHAR =
CHARACTER
XOBT = 1
OR
CXBT = 1
Y
XCHAR = CR
Y
XOBT → O
OR
CXBT → O
N
N
XCHAR = X^C
Y
RA
MPXEP
N
XCHAR = ←
Y
?BSTR =
?BHED
Y
MPXEP
N
?BPNT →
?BPNT - 1
N
XCHAR = Q^C
Y
RC
N
XCHAR = W^C
Y
RD
N
XCHAR =
RUBOUT
Y
MPXEP
N
R2

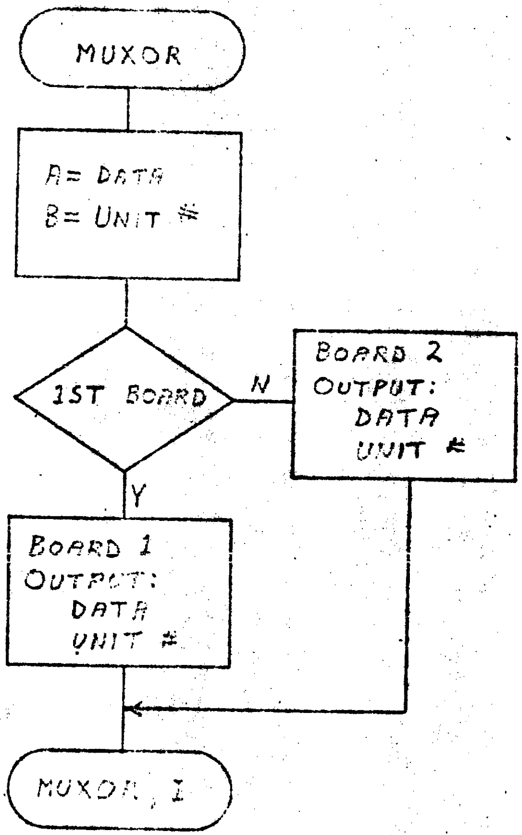

R2
XCHAR = X-OFF
Y
SUBTYPE 3 OR 4
N
MPXEP
N
XOBT → 1
XCHAR → CR
RB
BUFFER FULL
Y
RA
RESET BUFFER POINTERS
TPBT = O
N
Y
CXBT → 1
MPXEP
BUFFER → \
?CCNT → -3
A → SYNC CHAR
MUXOR
MPXEP
N
ADD CHAR TO BUFFER
XCHAR = CR
N
MPXEP
Y
TPBT = O
N
?BSTR → ?BPNT
Y
OTPUT
1ST COMPLETED BUFFER
N
MPXEP
Y
ENBT = O
N
ENBT → O
A= TIME USED
Y
ADD HVL AND TTY#
R3


R3
OUTPUT TO SYSTEM
?TNUM=TLSUP
Y
N
LPTTY→?TNUM
TLSUP→0
MPXEP
MPXEP


RC
?TNUM= LPTTY OR TLSUP
N
RB
Y
TLPR→LTNUM
LPTTY→0
TLSUP→0
MPXEP

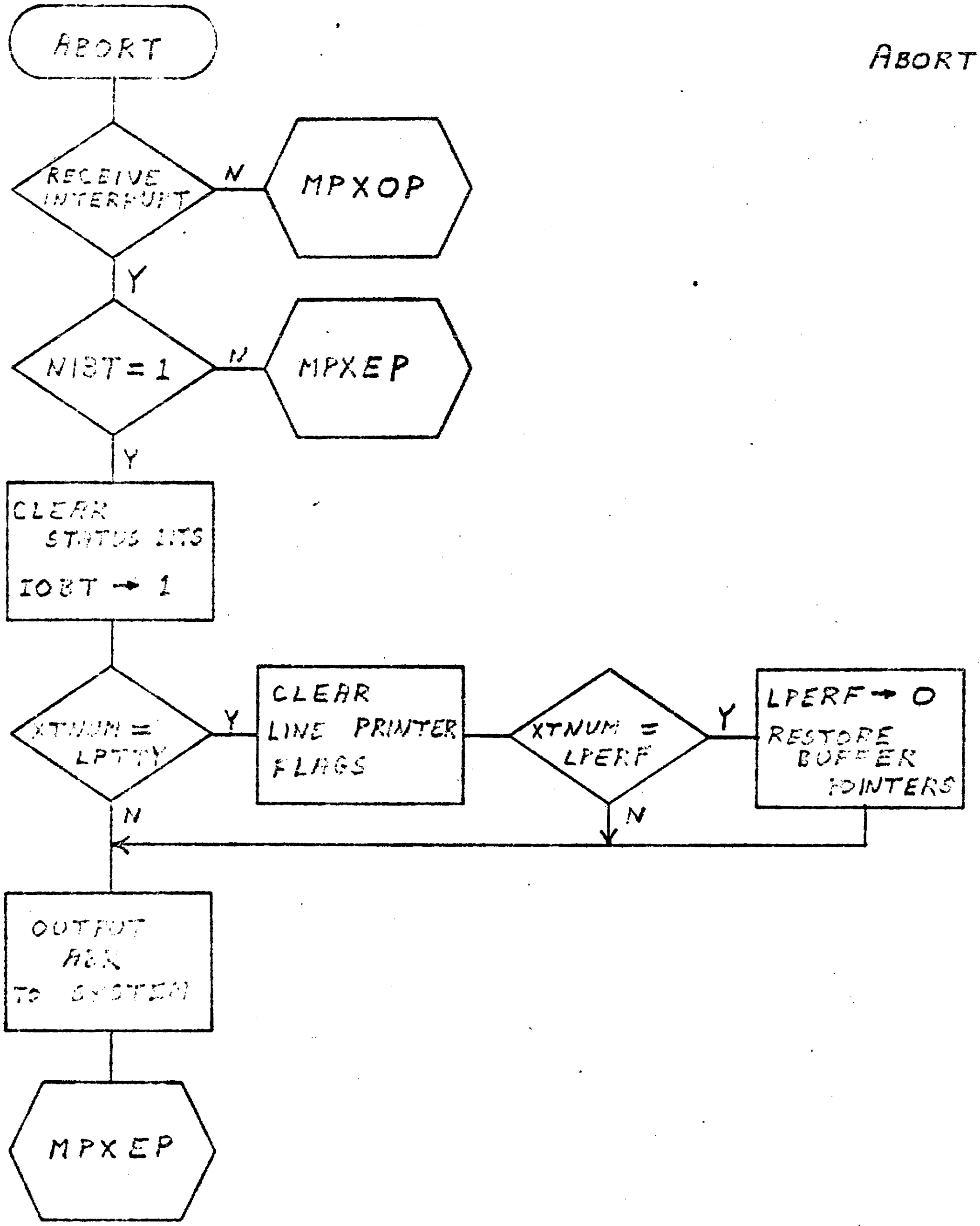

RD
?TNUM=TLPR
N
RB
Y
LPTTY→?TNUM
TLPR→0
MPXEP



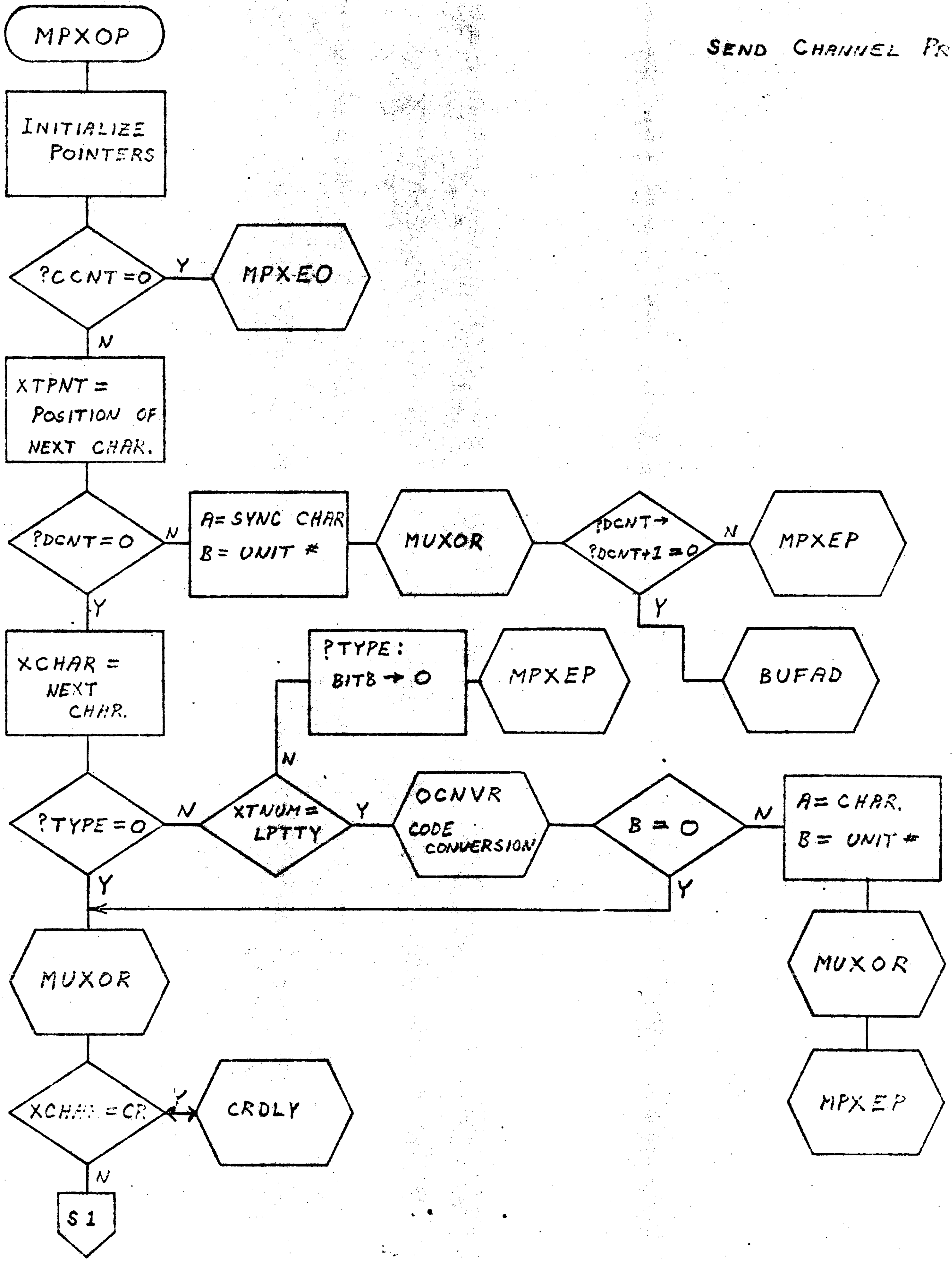


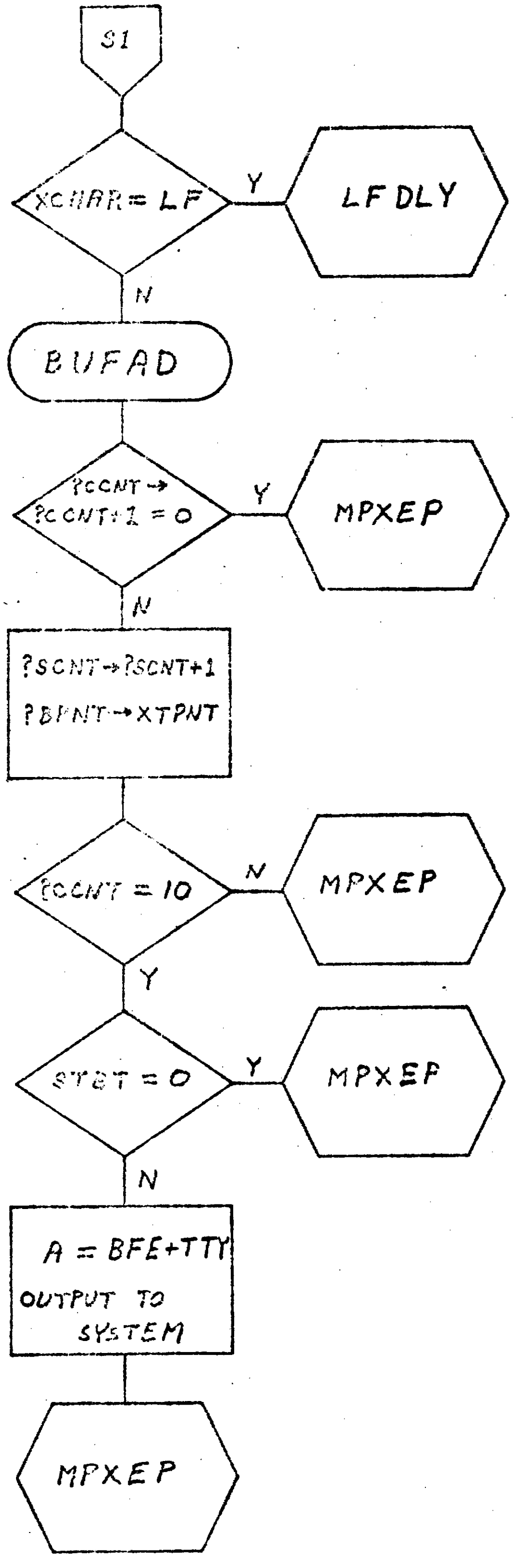
MULTIPLEXER OUTPUT ROUTINE

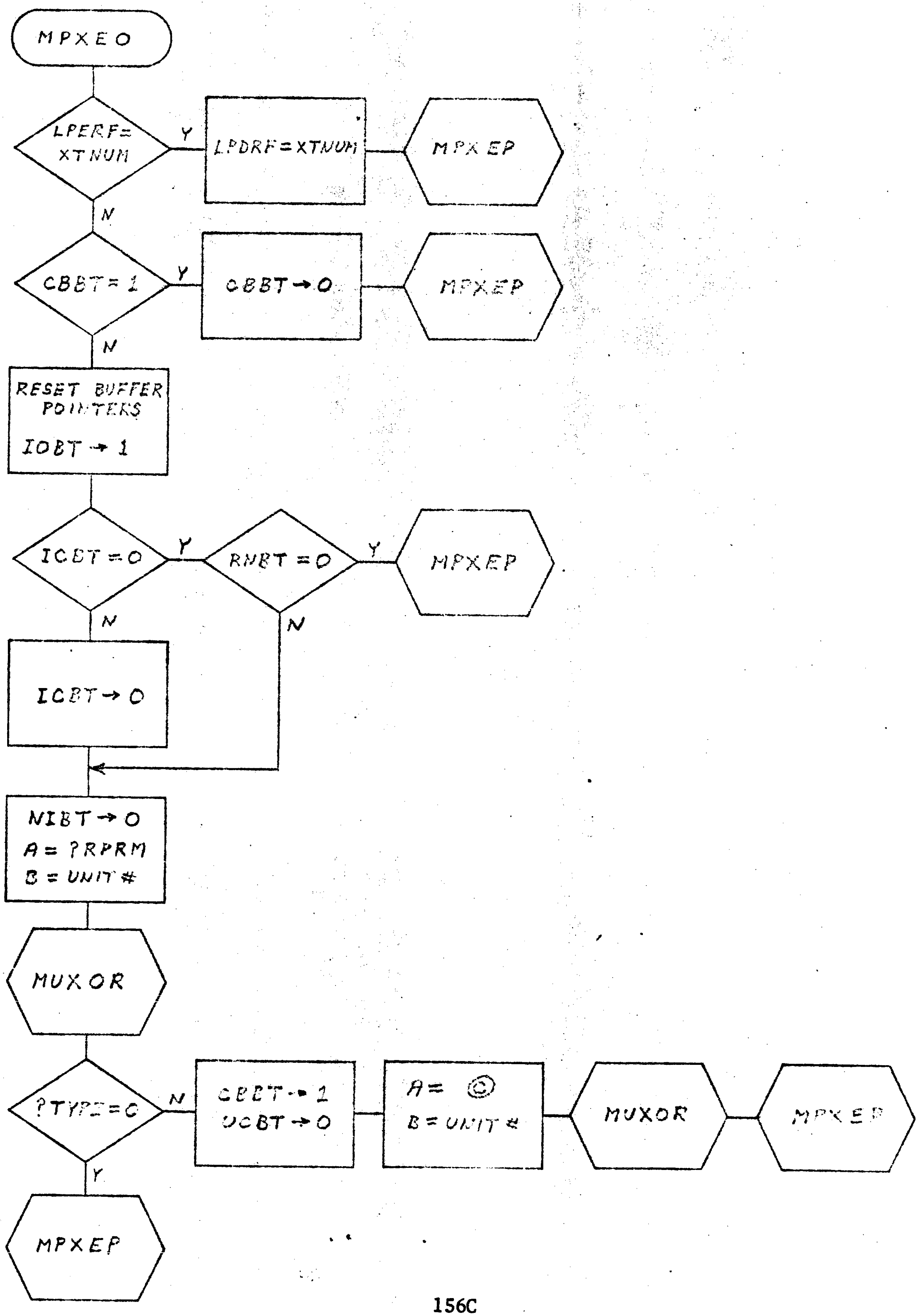
ABORT PROCESSING

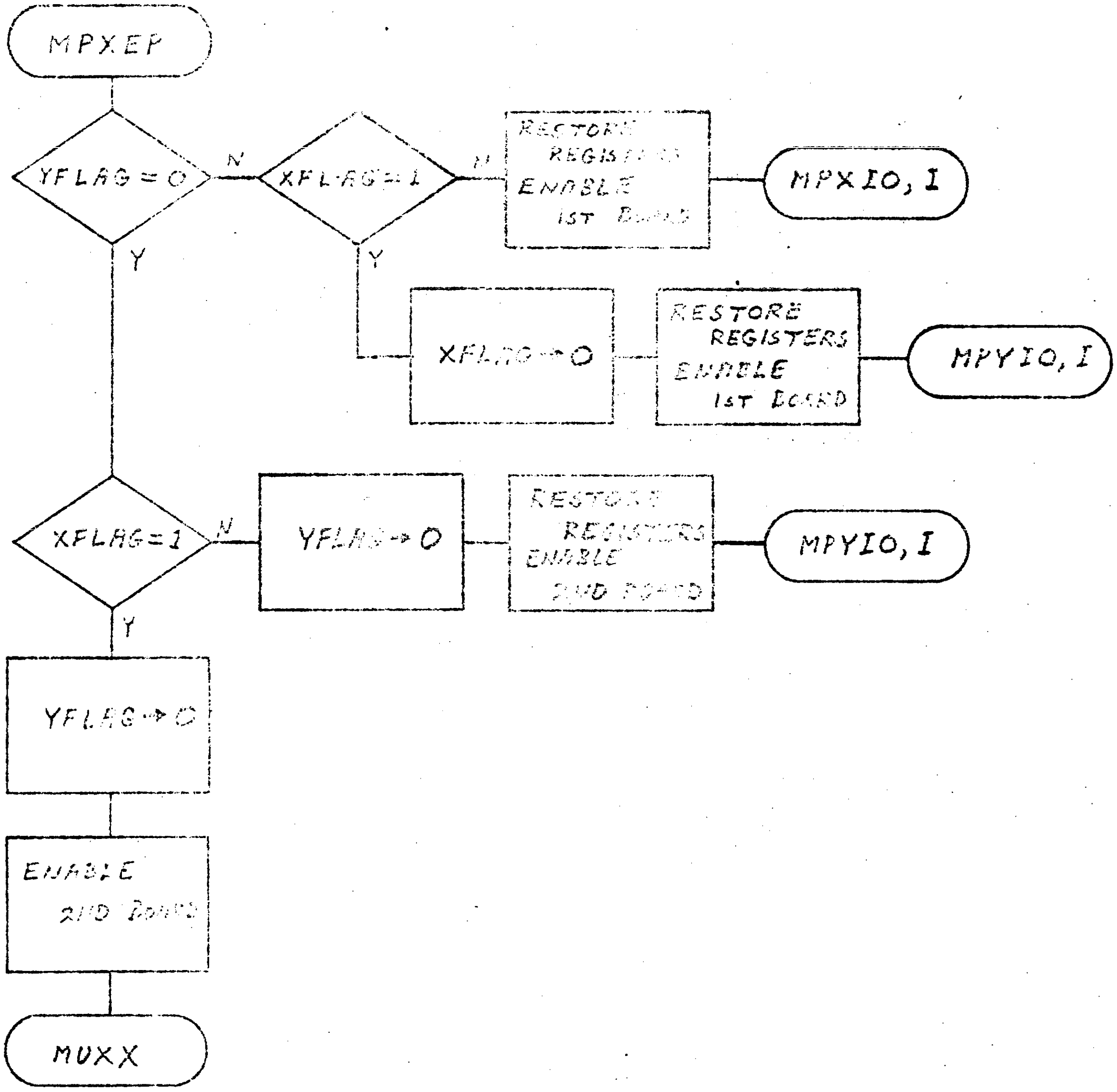


SEND CHANNEL PROCESS



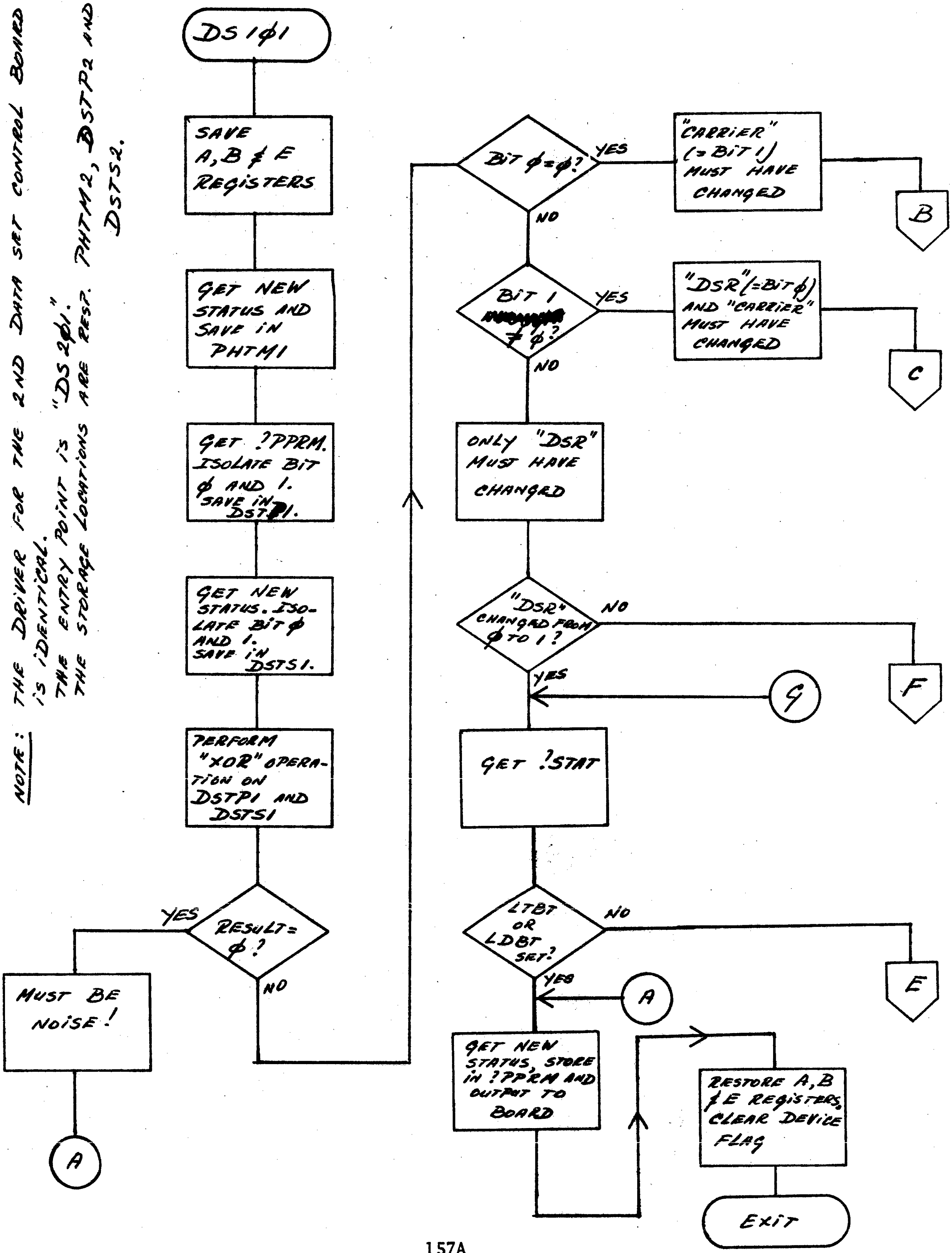

S1
XCHAR = LF
Y
LFDLY
N
BUFAD
?CCNT →
?CCNT+1 = 0
Y
MPXEP
N
?SCNT→?SCNT+1
?BPNT→XTPNT
?CCNT = 10
N
MPXEP
Y
STAT = 0
Y
MPXEP
N
A = BFE+TTY
OUTPUT TO
SYSTEM
MPXEP


MPXEO
LPERF= XTNUM
Y
LPDRF=XTNUM
MPXEP
N
CBBT=1
Y
CBBT→0
MPXEP
N
RESET BUFFER POINTERS
IOBT→1
ICBT=0
Y
RNBT=0
Y
MPXEP
N
N
ICBT→0
NIBT→0
A=?RPRM
B=UNIT#
MUXOR
?TYPE=0
N
CBBT→1
UCBT→0
A= ©
B=UNIT#
MUXOR
MPXEP
Y
MPXEP

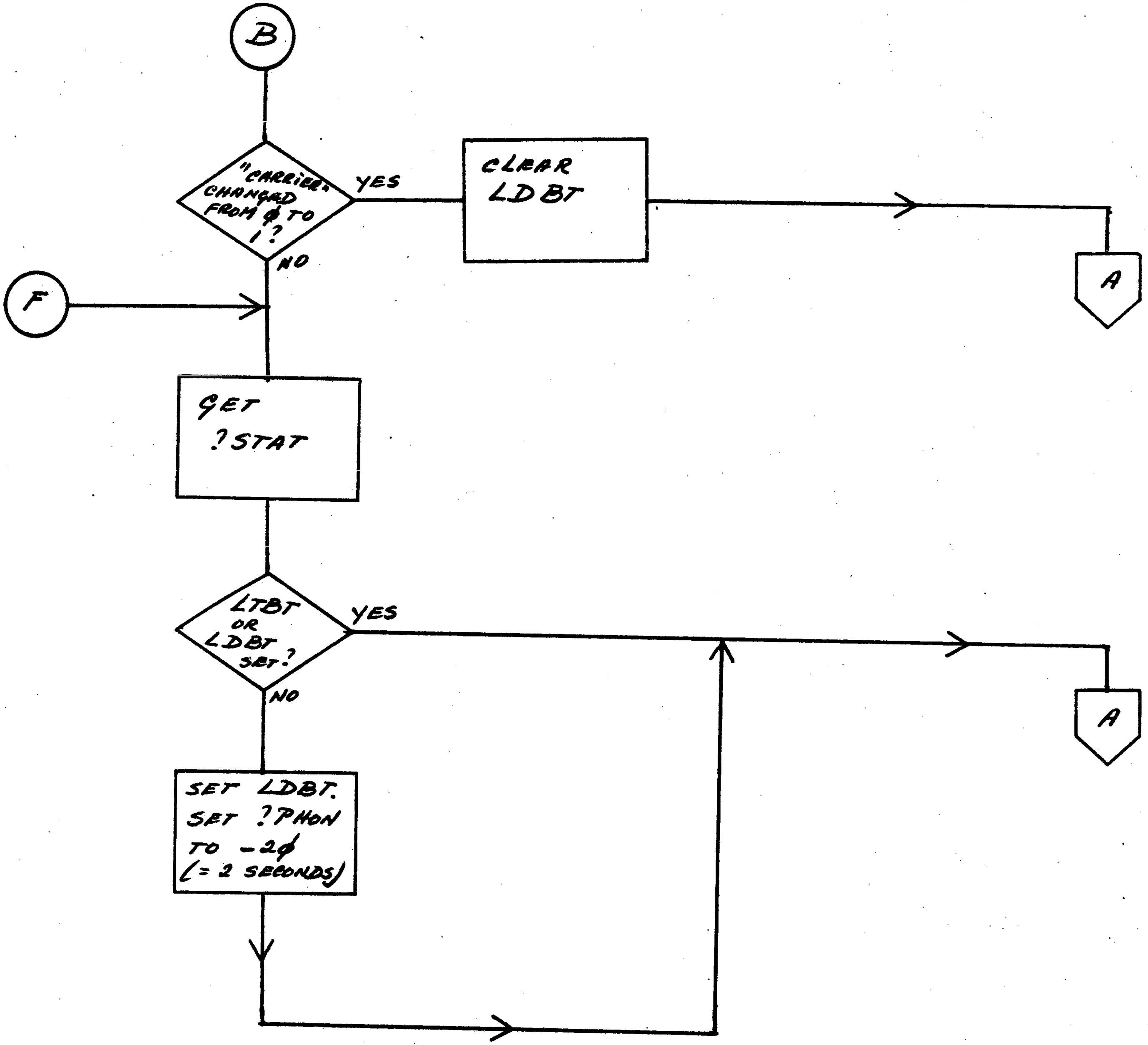

MPXEP
YFLAG = 0
N
XFLAG = 1
N
RESTORE REGISTERS
ENABLE 1ST BOARD
MPXIO, I
Y
Y
XFLAG → 0
RESTORE REGISTERS
ENABLE 1ST BOARD
MPYIO, I
XFLAG = 1
N
YFLAG → 0
RESTORE REGISTERS
ENABLE 2ND BOARD
MPYIO, I
Y
YFLAG → 0
ENABLE 2ND BOARD
MUXX

DATA SET CONTROL BOARD (1 OF 3)

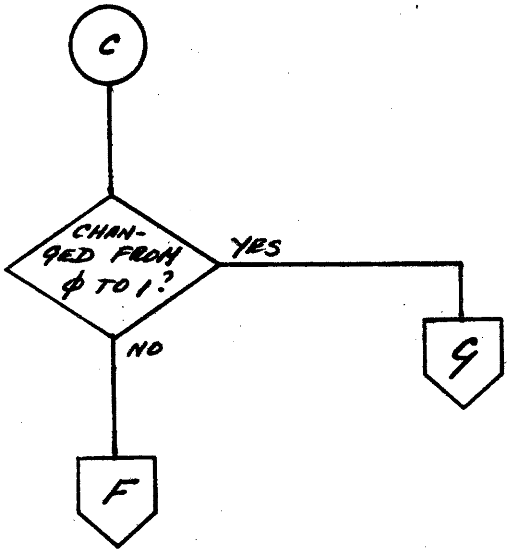
NOTE: THE DRIVER FOR THE 2ND DATA SET CONTROL BOARD IS IDENTICAL.
THE ENTRY POINT IS "DS2Ø1".
THE STORAGE LOCATIONS ARE RESP. PHTM2, DSTP2 AND DSTS2.

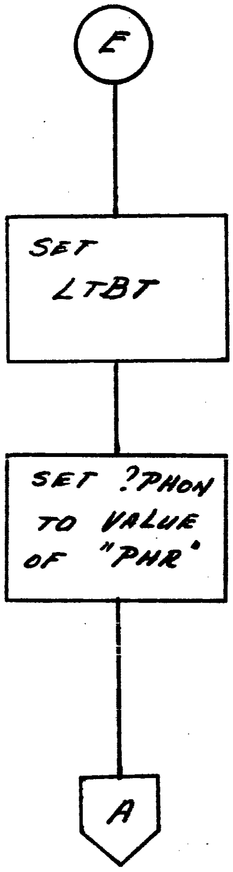


DS1Ø1
SAVE A,B & E REGISTERS
GET NEW STATUS AND SAVE IN PHTM1
GET ?PPRM. ISOLATE BIT Ø AND 1. SAVE IN DSTP1.
GET NEW STATUS. ISOLATE BIT Ø AND 1. SAVE IN DSTS1.
PERFORM "XOR" OPERATION ON DSTP1 AND DSTS1
RESULT = Ø?
YES
NO
MUST BE NOISE!
A
BIT Ø = Ø?
YES
NO
"CARRIER" (= BIT 1) MUST HAVE CHANGED
B
BIT 1 ≠ Ø?
YES
NO
"DSR" (= BIT Ø) AND "CARRIER" MUST HAVE CHANGED
C
ONLY "DSR" MUST HAVE CHANGED
"DSR" CHANGED FROM Ø TO 1?
NO
YES
F
G
GET ?STAT
LTBT OR LDBT SET?
NO
YES
E
A
GET NEW STATUS, STORE IN ?PPRM AND OUTPUT TO BOARD
RESTORE A,B & E REGISTERS, CLEAR DEVICE FLAG
EXIT

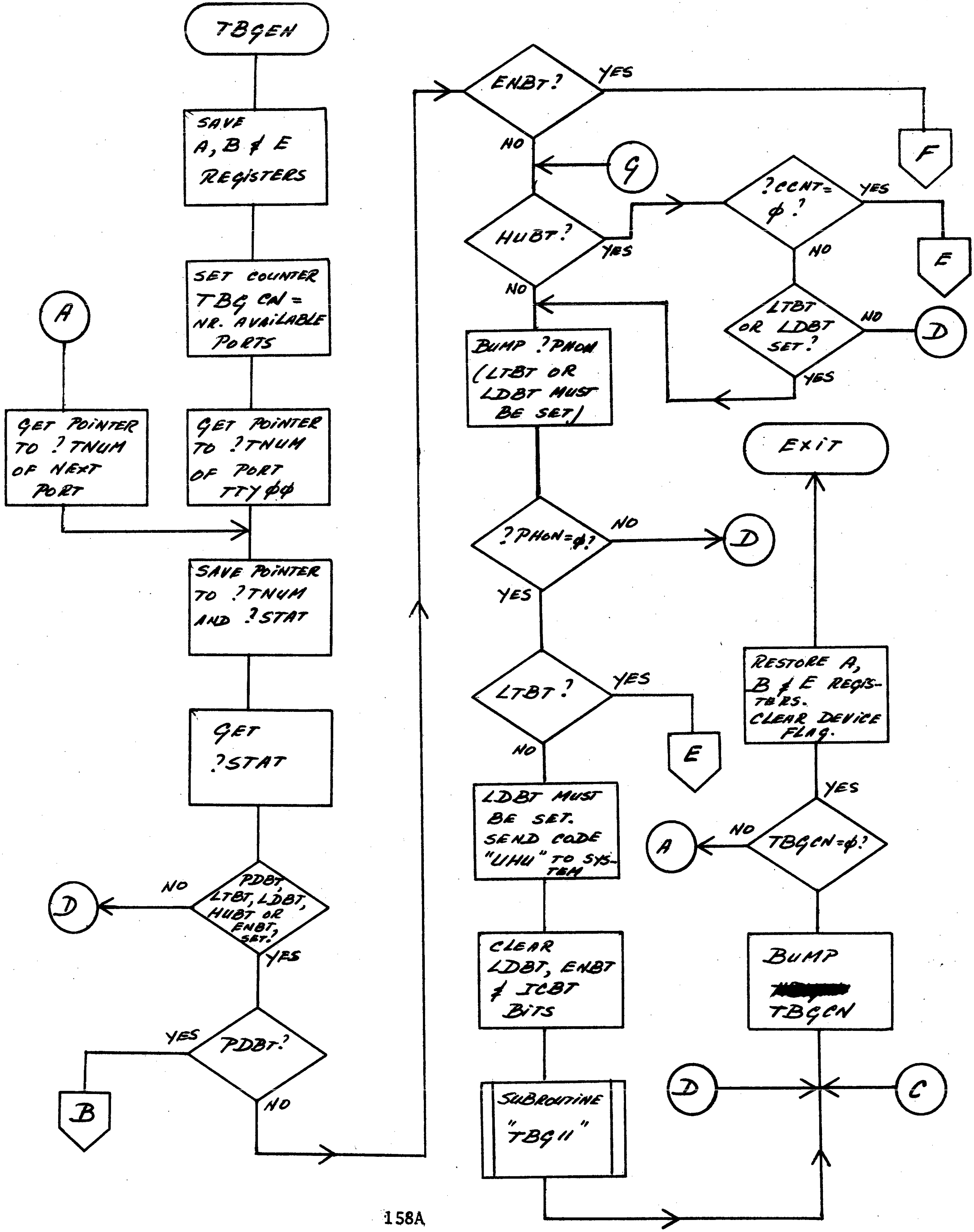

B
"CARRIER" CHANGED FROM ∅ TO 1?
YES
CLEAR LDBT
A
NO
F
GET ?STAT
LTBT OR LDBT SET?
YES
A
NO
SET LDBT. SET ?PHON TO -2∅ (= 2 SECONDS)

DATA SET CONTROL BOARD (3 OF 3)



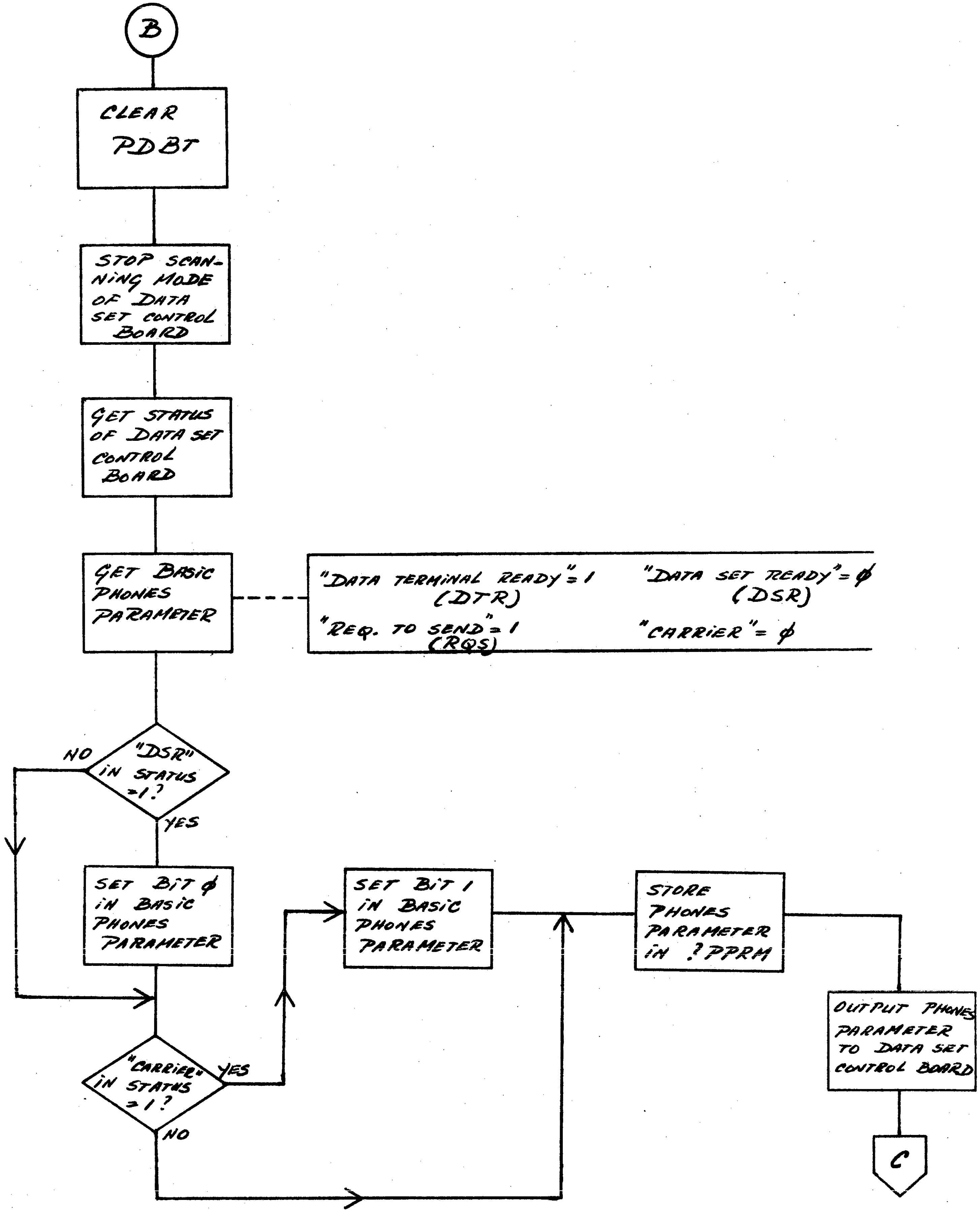


TIME BASE GENERATOR (1 OF 6)

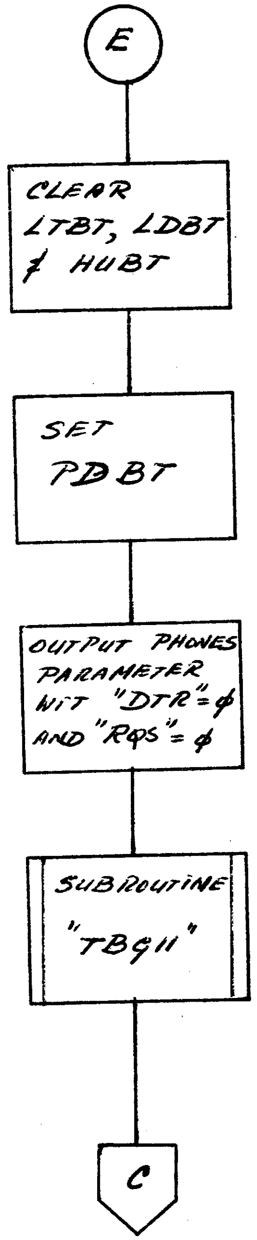


TBGEN
SAVE A, B ¢ E REGISTERS
SET COUNTER TBG CN = NR. AVAILABLE PORTS
A
GET POINTER TO ?TNUM OF NEXT PORT
GET POINTER TO ?TNUM OF PORT TTY ØØ
SAVE POINTER TO ?TNUM AND ?STAT
GET ?STAT
PDBT, LTBT, LDBT, HUBT OR ENBT, SET?
NO
D
YES
PDBT?
YES
B
NO
ENBT?
YES
F
NO
G
HUBT?
YES
?CCNT= Ø?
YES
E
NO
LTBT OR LDBT SET?
NO
D
YES
NO
BUMP ?PHON (LTBT OR LDBT MUST BE SET)
?PHON=Ø?
NO
D
YES
LTBT?
YES
E
NO
LDBT MUST BE SET. SEND CODE "UHU" TO SYSTEM
CLEAR LDBT, ENBT ¢ ICBT BITS
SUBROUTINE "TBG11"
EXIT
RESTORE A, B ¢ E REGISTERS. CLEAR DEVICE FLAG.
YES
TBGCN=Ø?
NO
A
BUMP [redacted] TBGCN
D
C

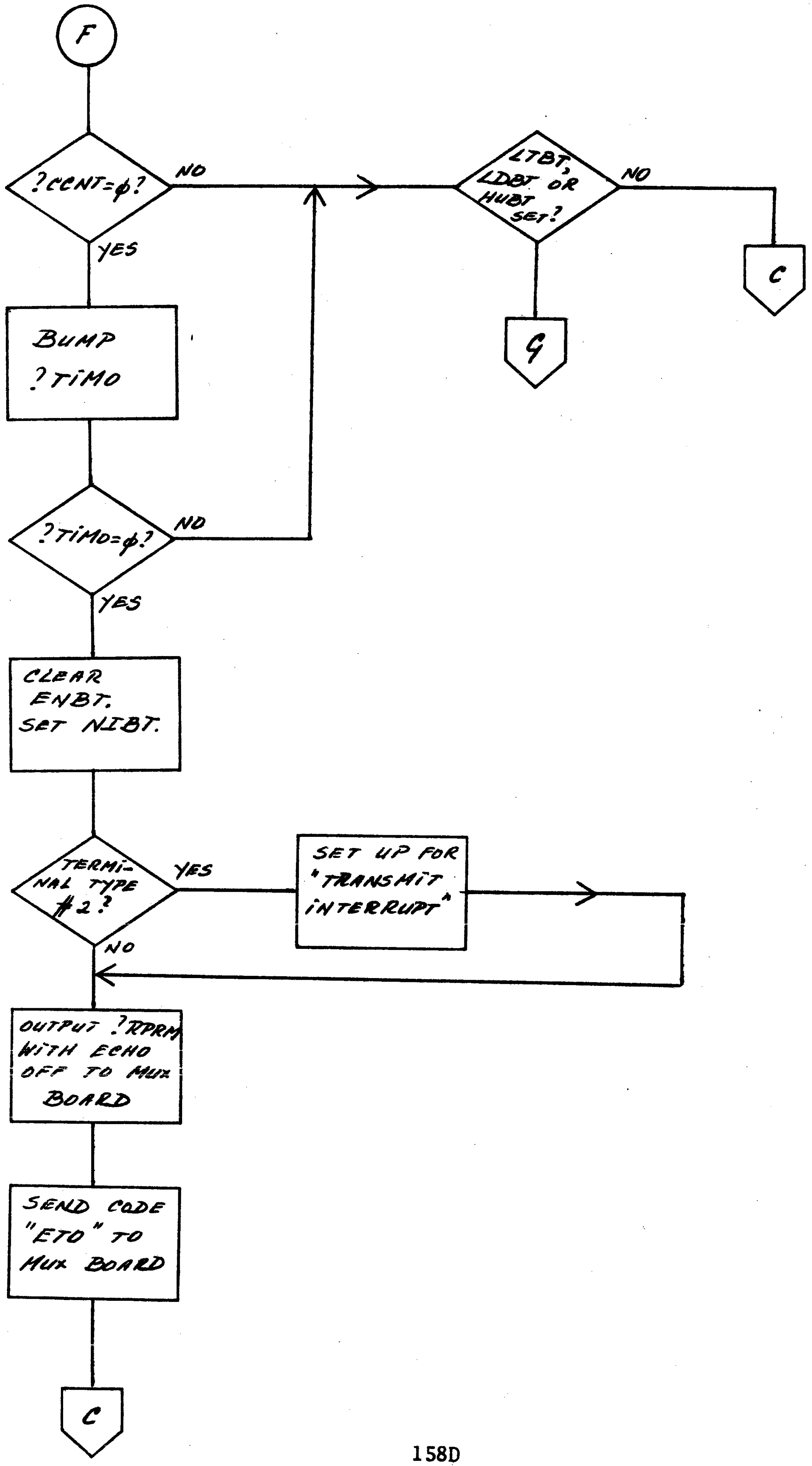
TIME BASE GENERATOR (2 OF 5)


B
CLEAR PDBT
STOP SCANNING MODE OF DATA SET CONTROL BOARD
GET STATUS OF DATA SET CONTROL BOARD
GET BASIC PHONES PARAMETER
"DATA TERMINAL READY" = 1 (DTR)
"DATA SET READY" = ϕ (DSR)
"REQ. TO SEND" = 1 (RQS)
"CARRIER" = ϕ
NO
"DSR" IN STATUS =1?
YES
SET BIT ϕ IN BASIC PHONES PARAMETER
SET BIT 1 IN BASIC PHONES PARAMETER
STORE PHONES PARAMETER IN ?PPRM
OUTPUT PHONES PARAMETER TO DATA SET CONTROL BOARD
"CARRIER" IN STATUS =1?
YES
NO
C

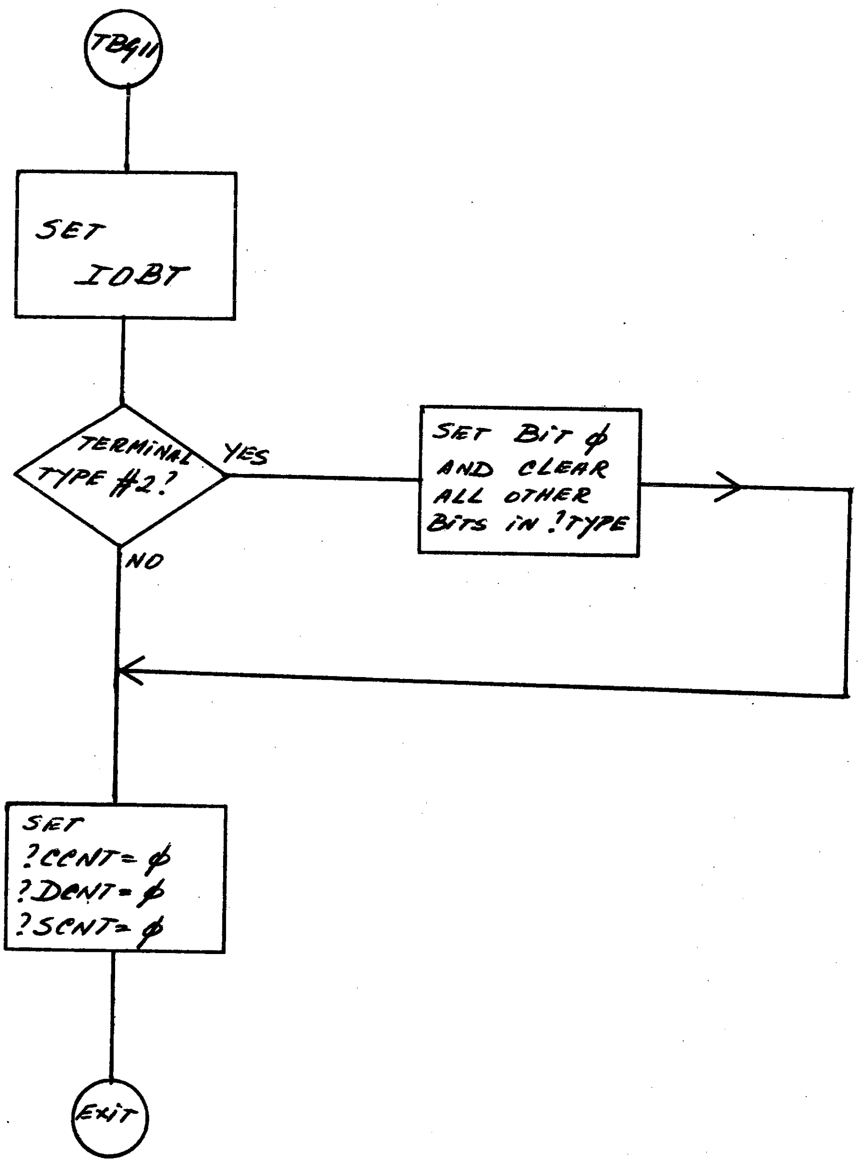
TIME BASE GENERATOR (3 OF 5)



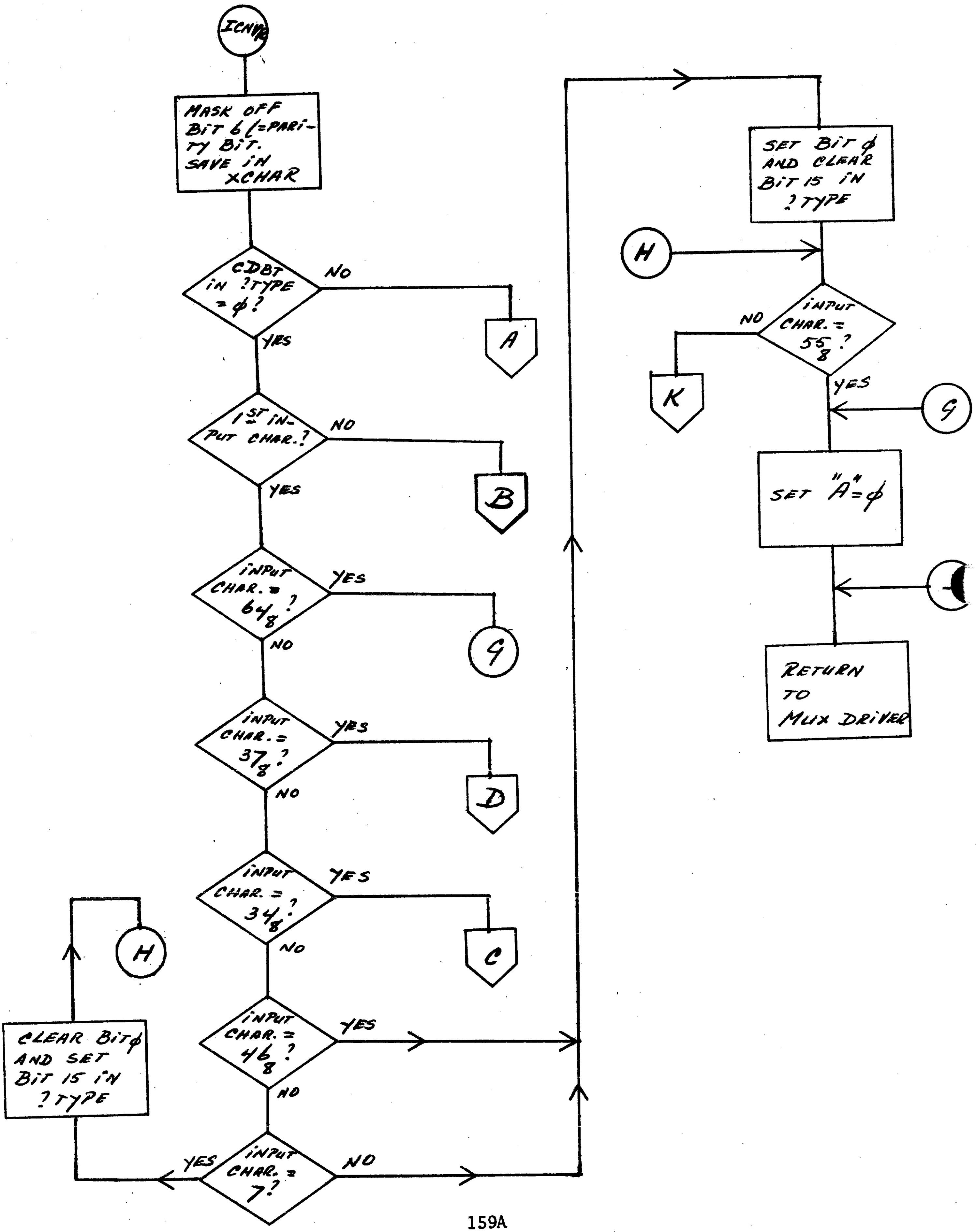
TIME BASE GENERATOR (4 OF 5)



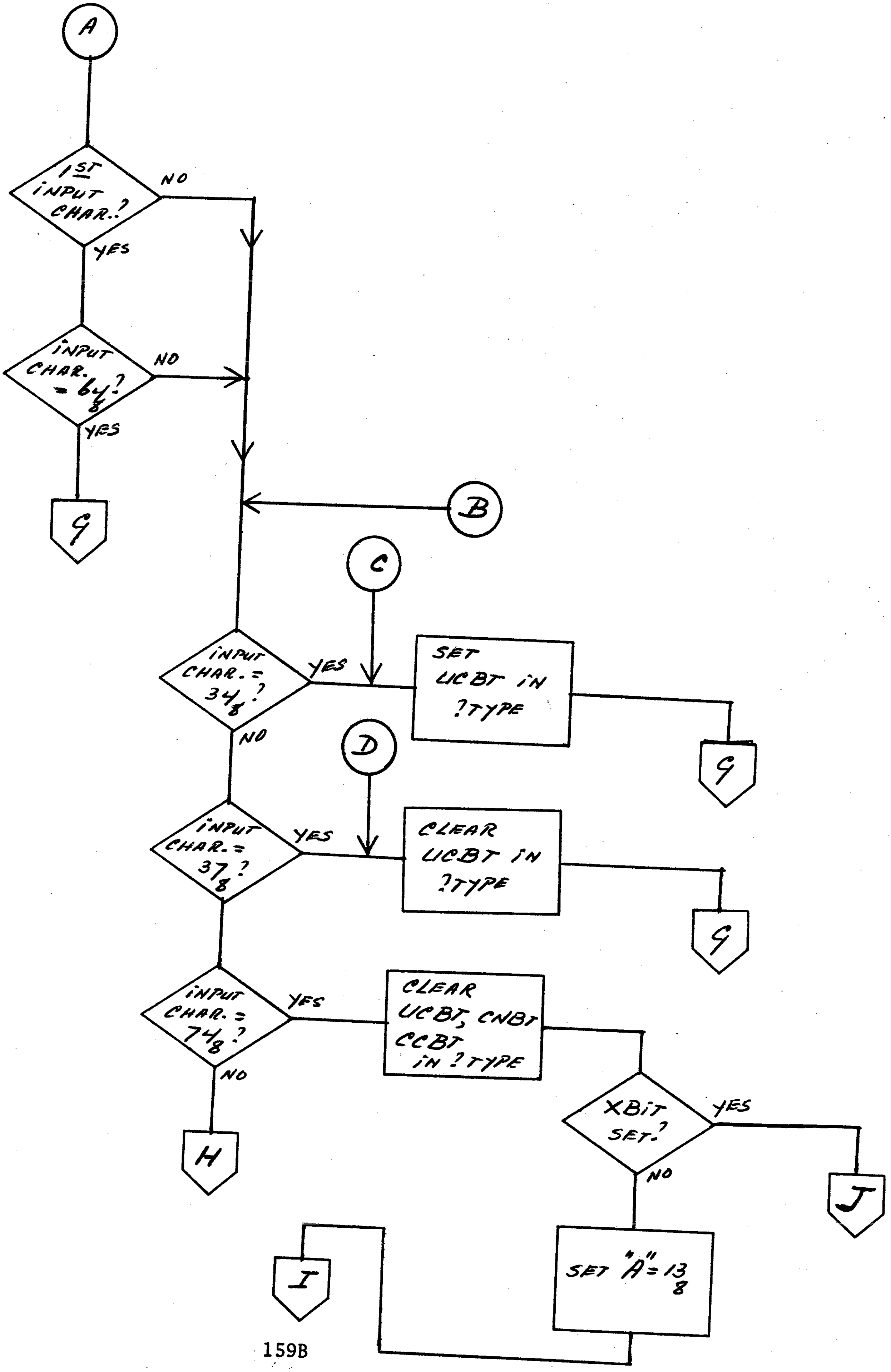
SUBROUTINE "TBG11"


TBG11
SET IOBT
TERMINAL TYPE #2?
YES
NO
SET BIT φ AND CLEAR ALL OTHER BITS IN ?TYPE
SET ?CCNT=φ ?DCNT=φ ?SCNT=φ
EXIT

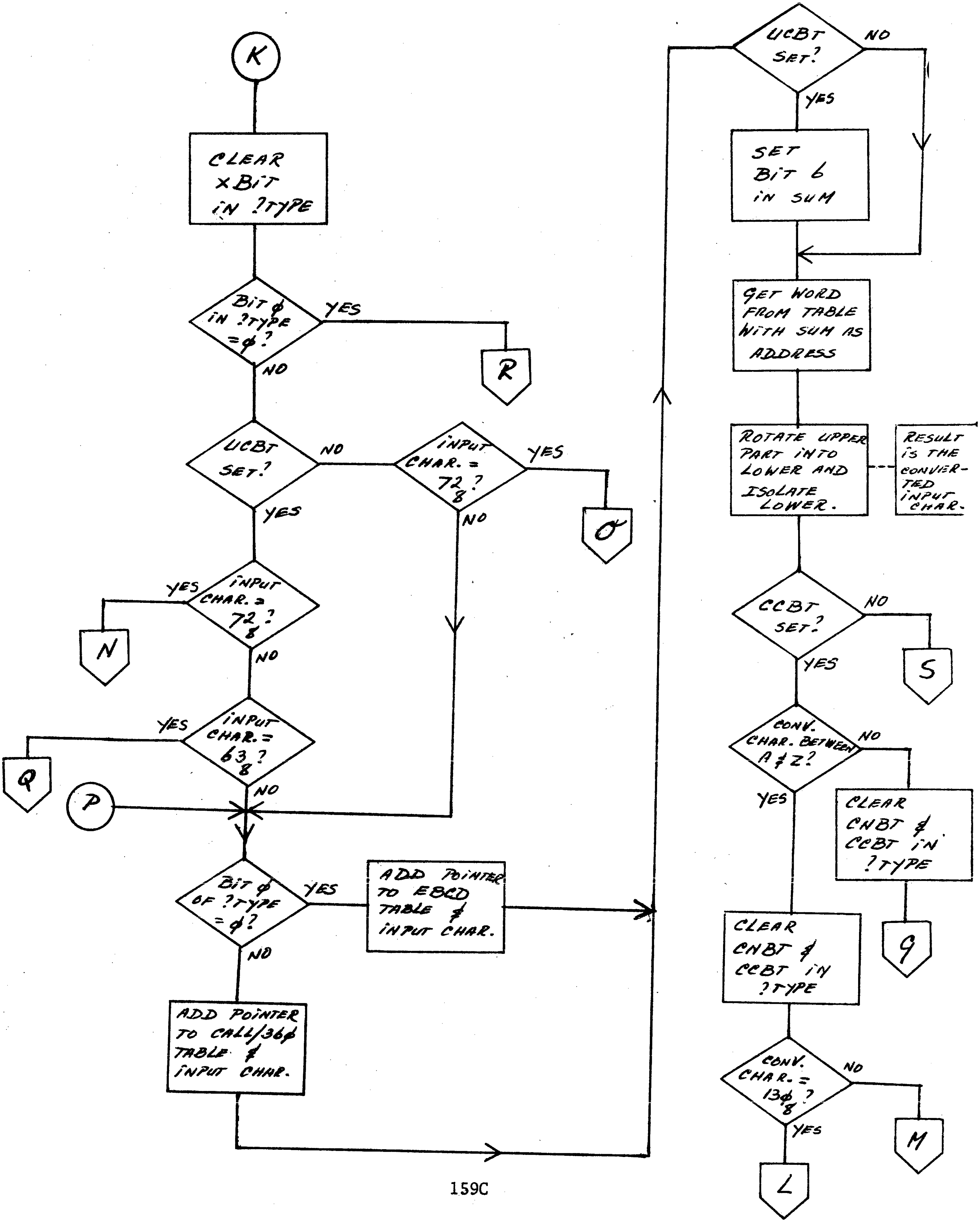
SELECTRIC INPUT CONVERSION (1 OF 9)



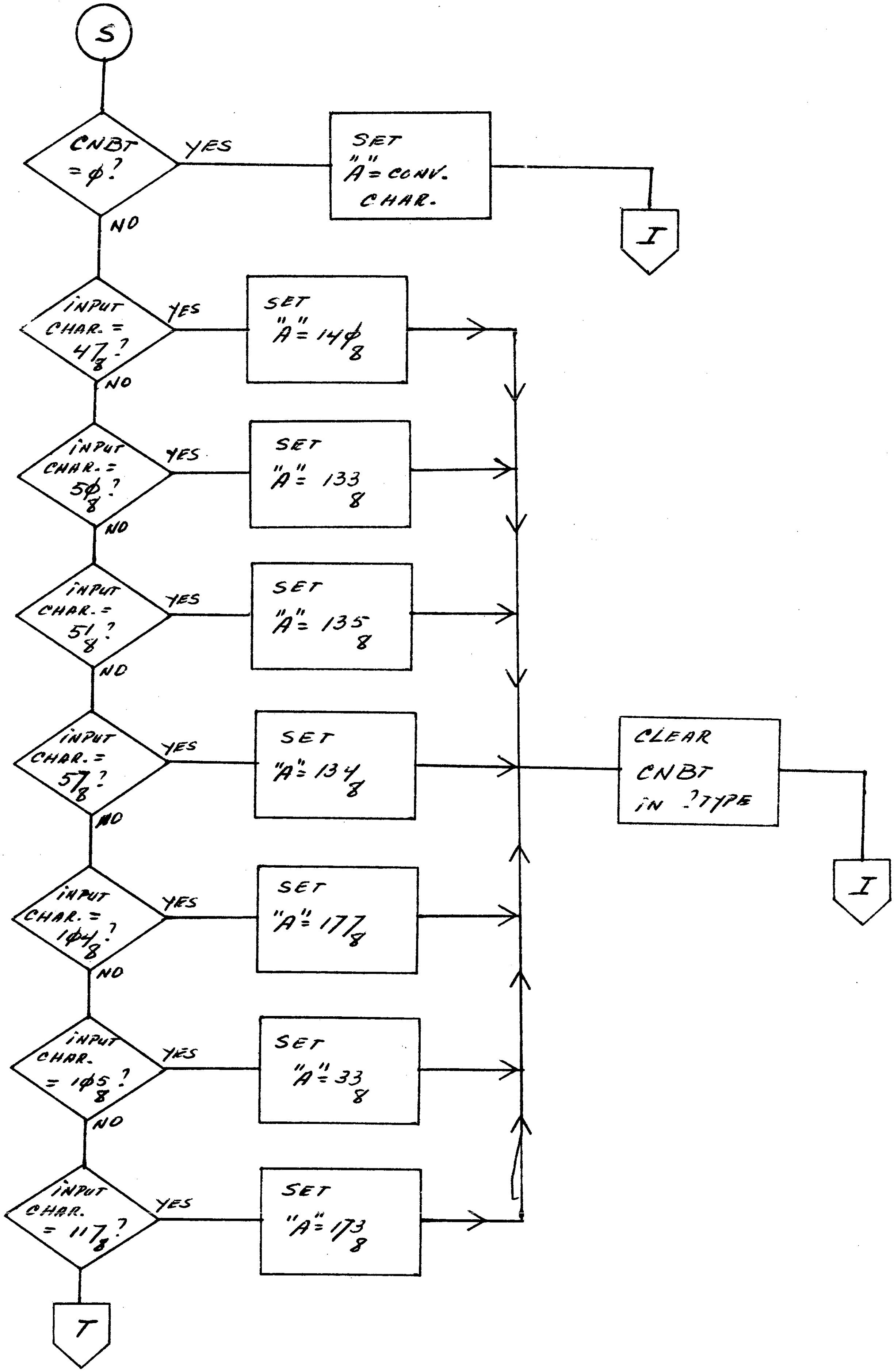
SELECTRIC INPUT CONVERSION (2 OF 9)



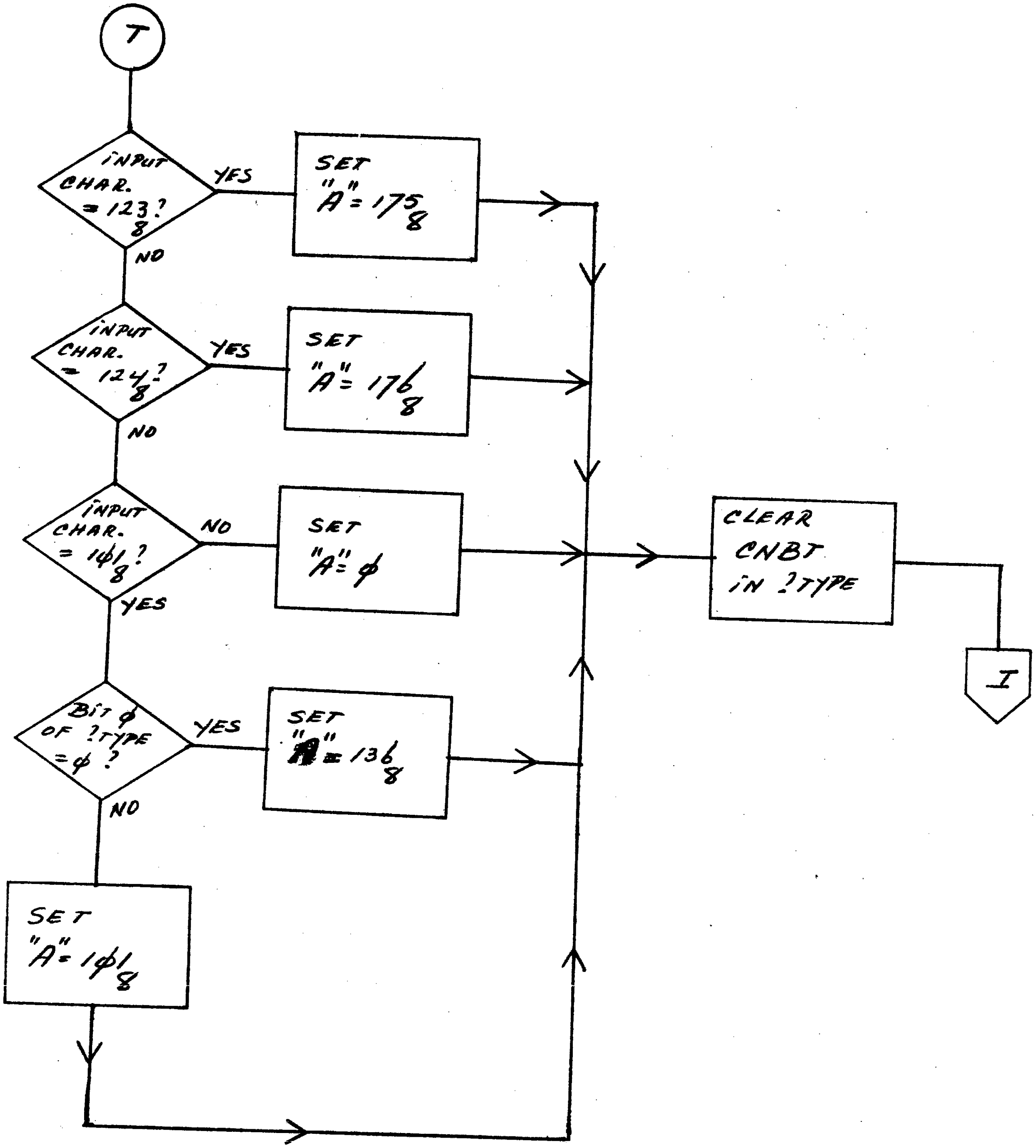
SELECTRIC INPUT CONVERSION (3 OF 9)


K
CLEAR XBIT IN ?TYPE
BIT φ IN ?TYPE =φ?
YES
R
NO
UCBT SET?
NO
INPUT CHAR. = 72₈?
YES
O
NO
YES
INPUT CHAR. = 72₈?
YES
N
NO
INPUT CHAR. = 63₈?
YES
Q
NO
P
BIT φ OF ?TYPE =φ?
YES
ADD POINTER TO EBCD TABLE & INPUT CHAR.
NO
ADD POINTER TO CALL/36φ TABLE & INPUT CHAR.
UCBT SET?
NO
YES
SET BIT 6 IN SUM
GET WORD FROM TABLE WITH SUM AS ADDRESS
ROTATE UPPER PART INTO LOWER AND ISOLATE LOWER.
RESULT IS THE CONVERTED INPUT CHAR.
CCBT SET?
NO
S
YES
CONV. CHAR. BETWEEN A & Z?
NO
CLEAR CNBT & CCBT IN ?TYPE
9
YES
CLEAR CNBT & CCBT IN ?TYPE
CONV. CHAR. = 13φ₈?
NO
M
YES
L

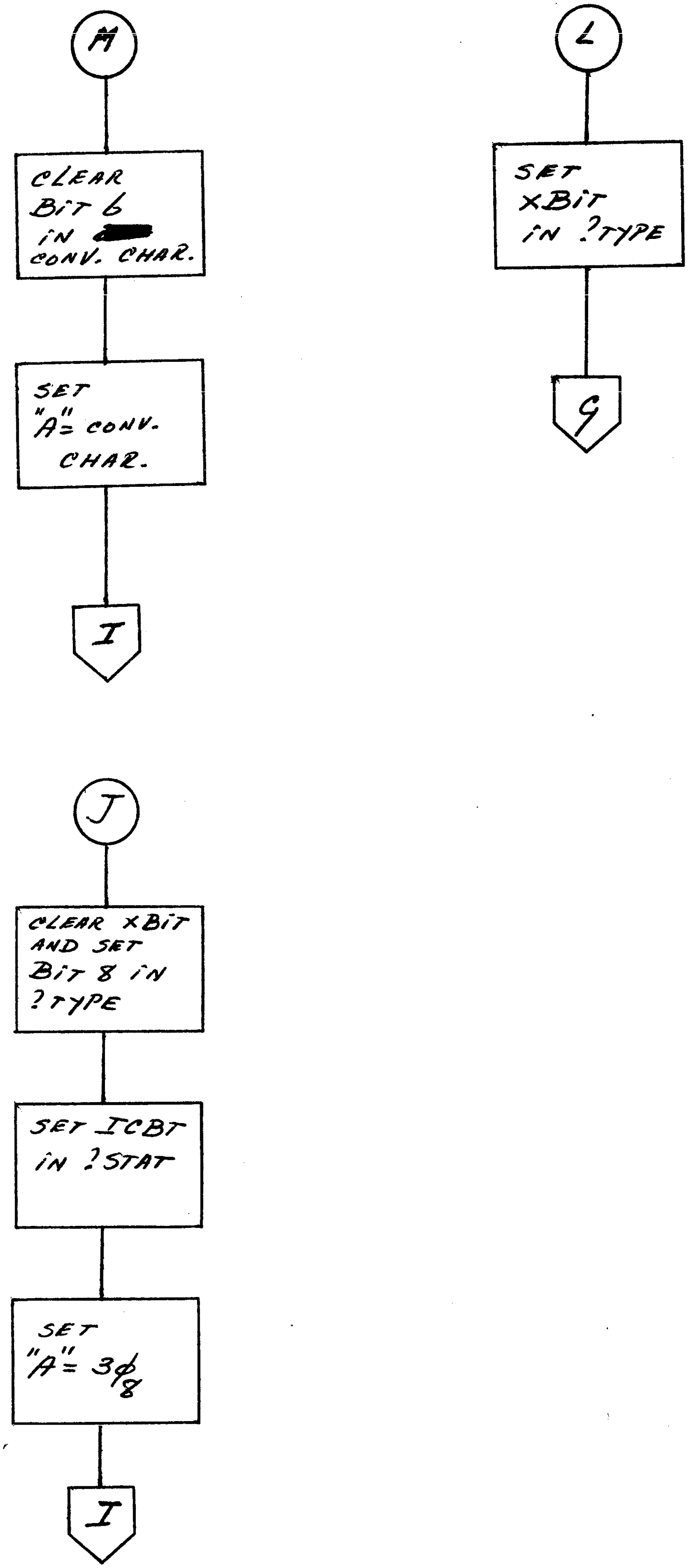
SELECTRIC INPUT CONVERSION (4 OF 9)



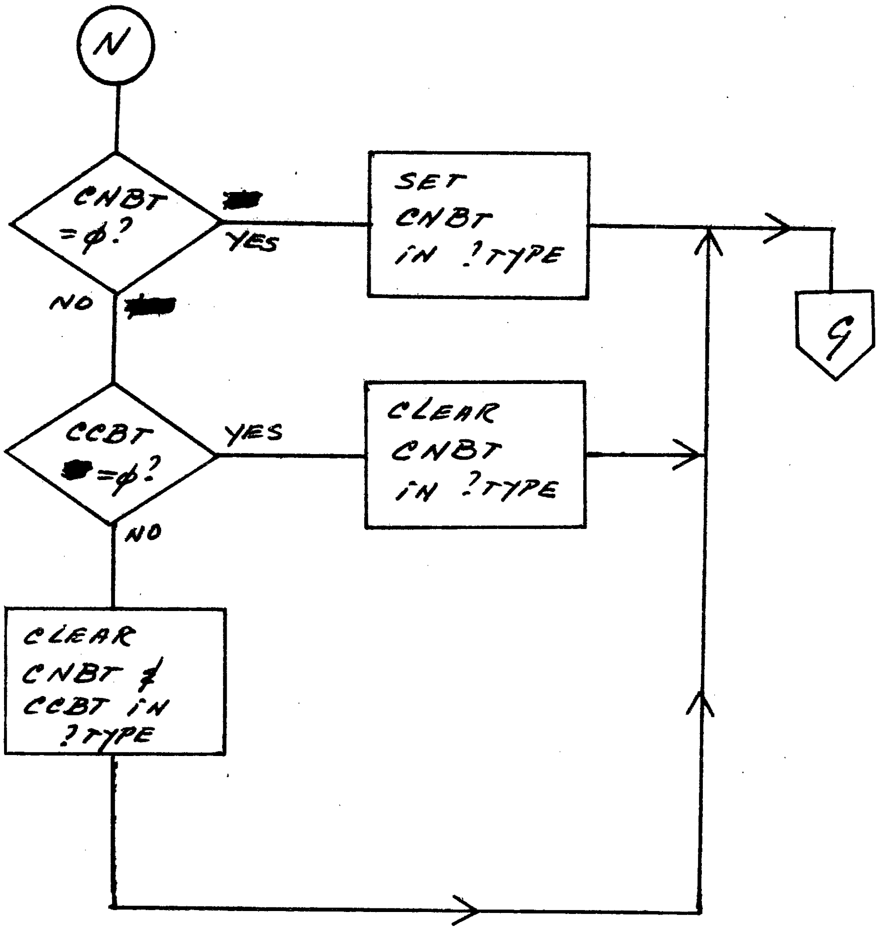
SELECTRIC INPUT CONVERSION (5 OF 9)



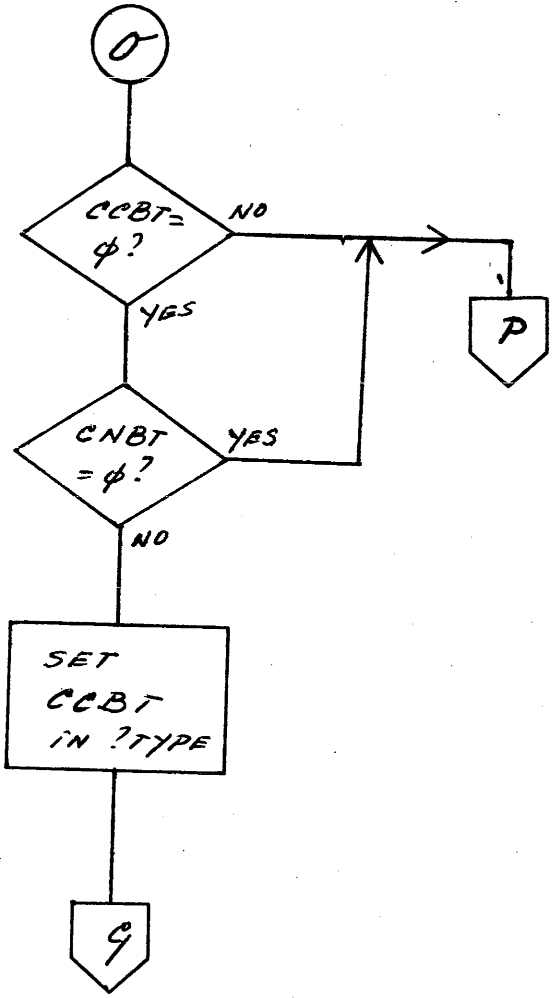
SELECTRIC INPUT CONVERSION (6 OF 9)

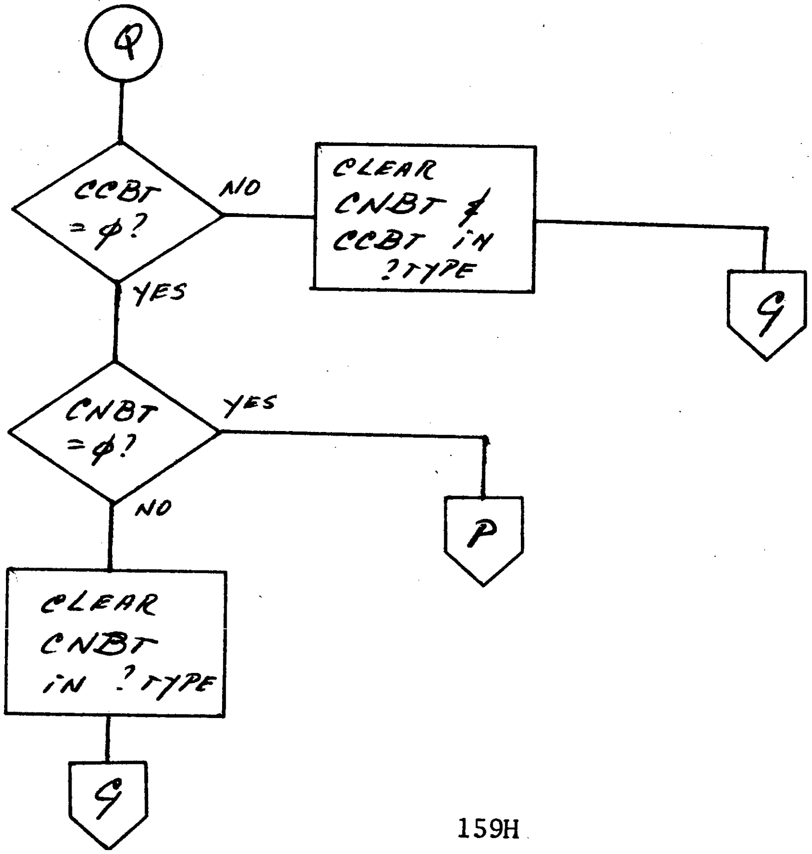


SELECTRIC INPUT CONVERSION (7 OF 9)

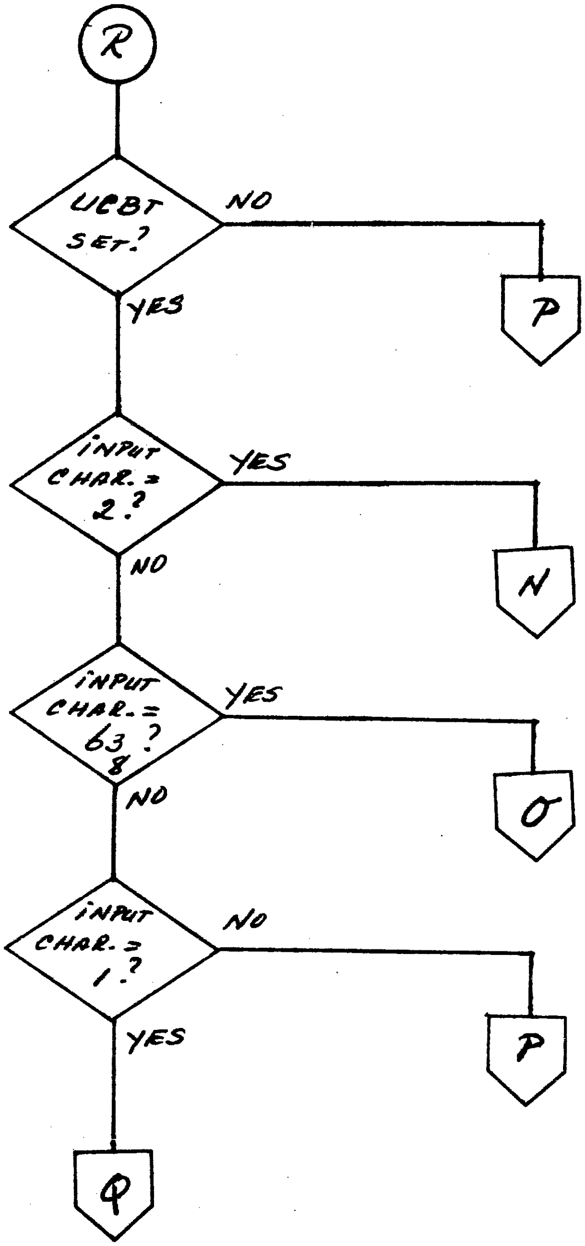


SELECTRIC INPUT CONVERSION (8 OF 9)

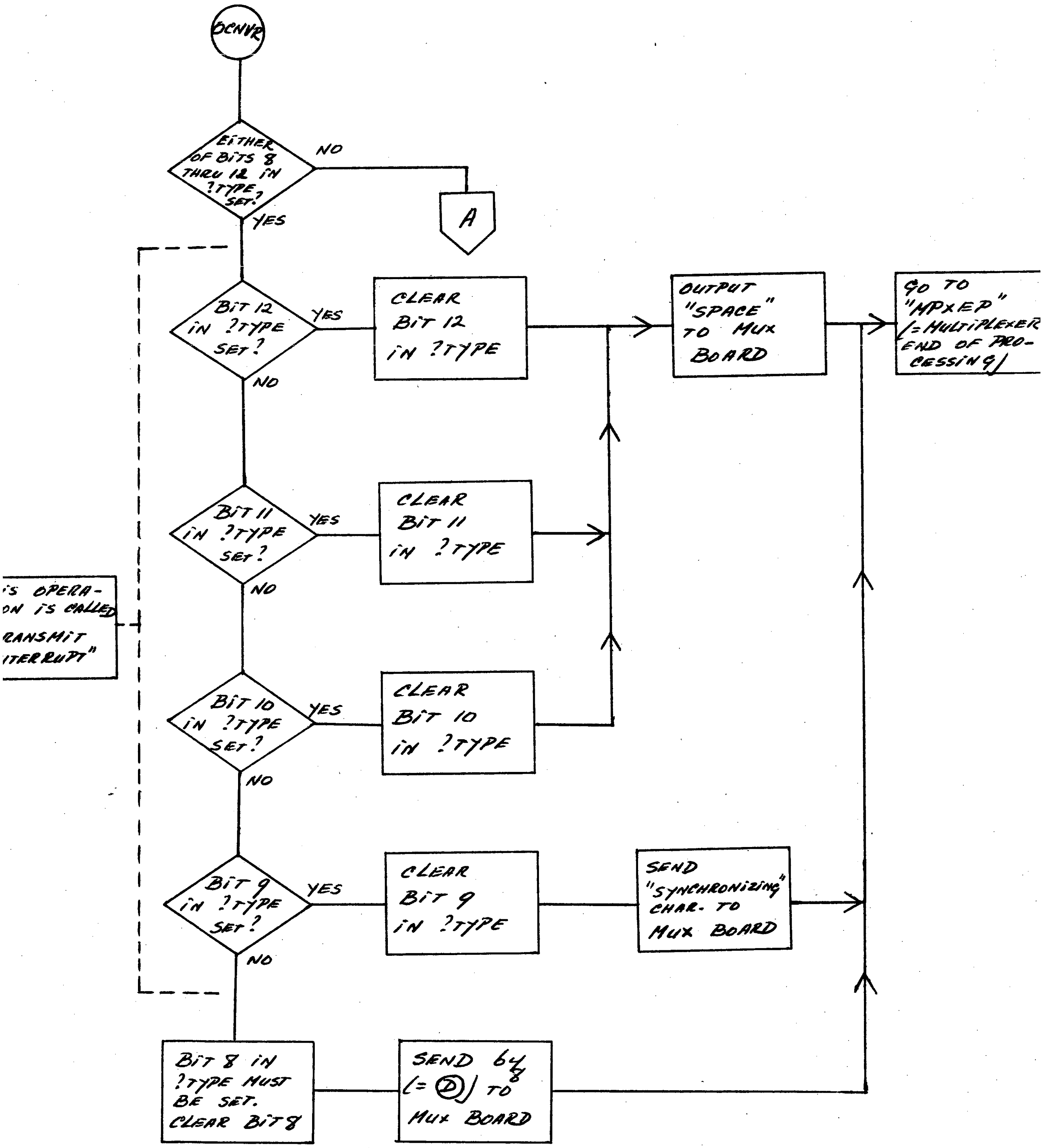




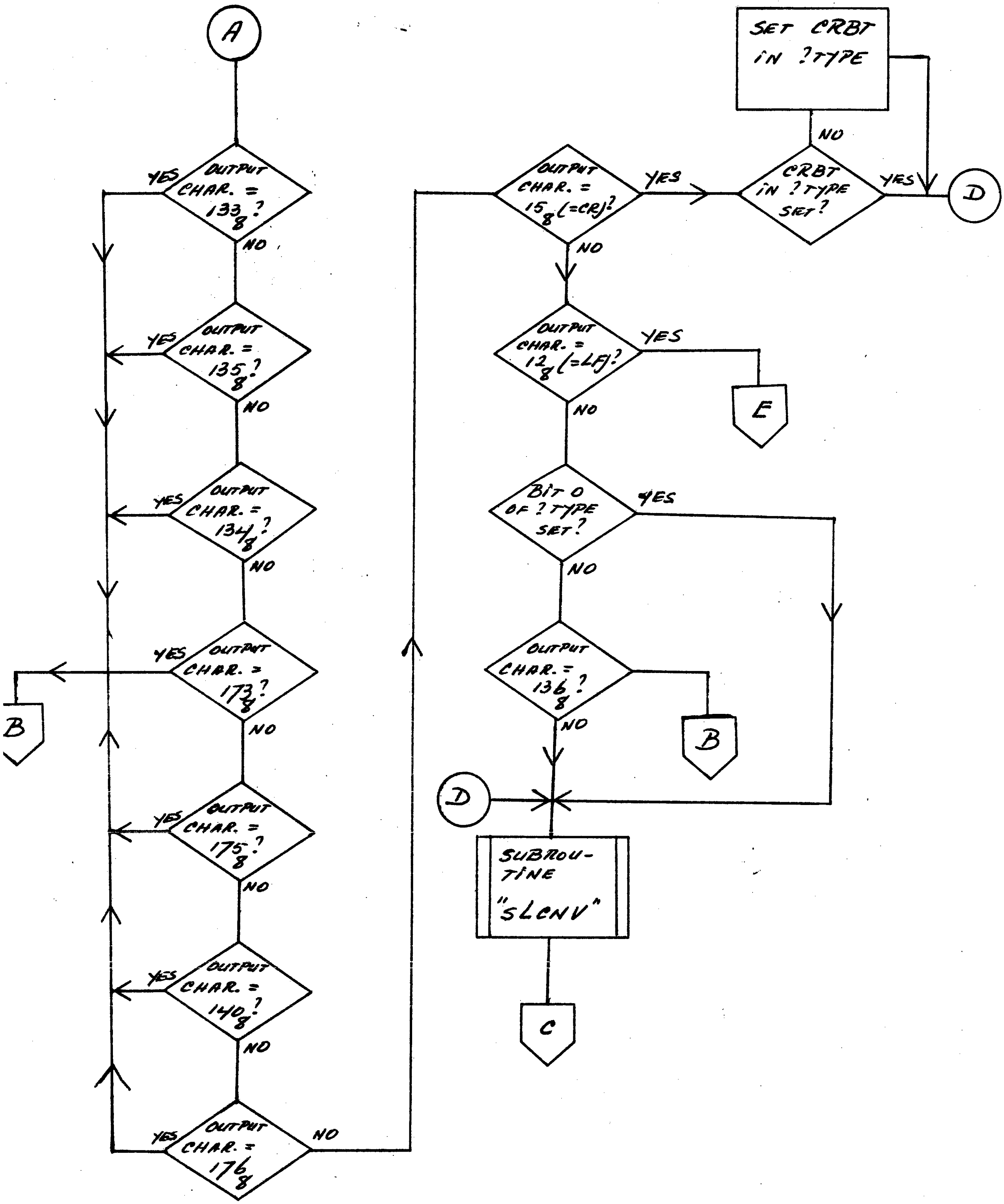
SELECTRIC INPUT CONVERSION (9 OF 9)



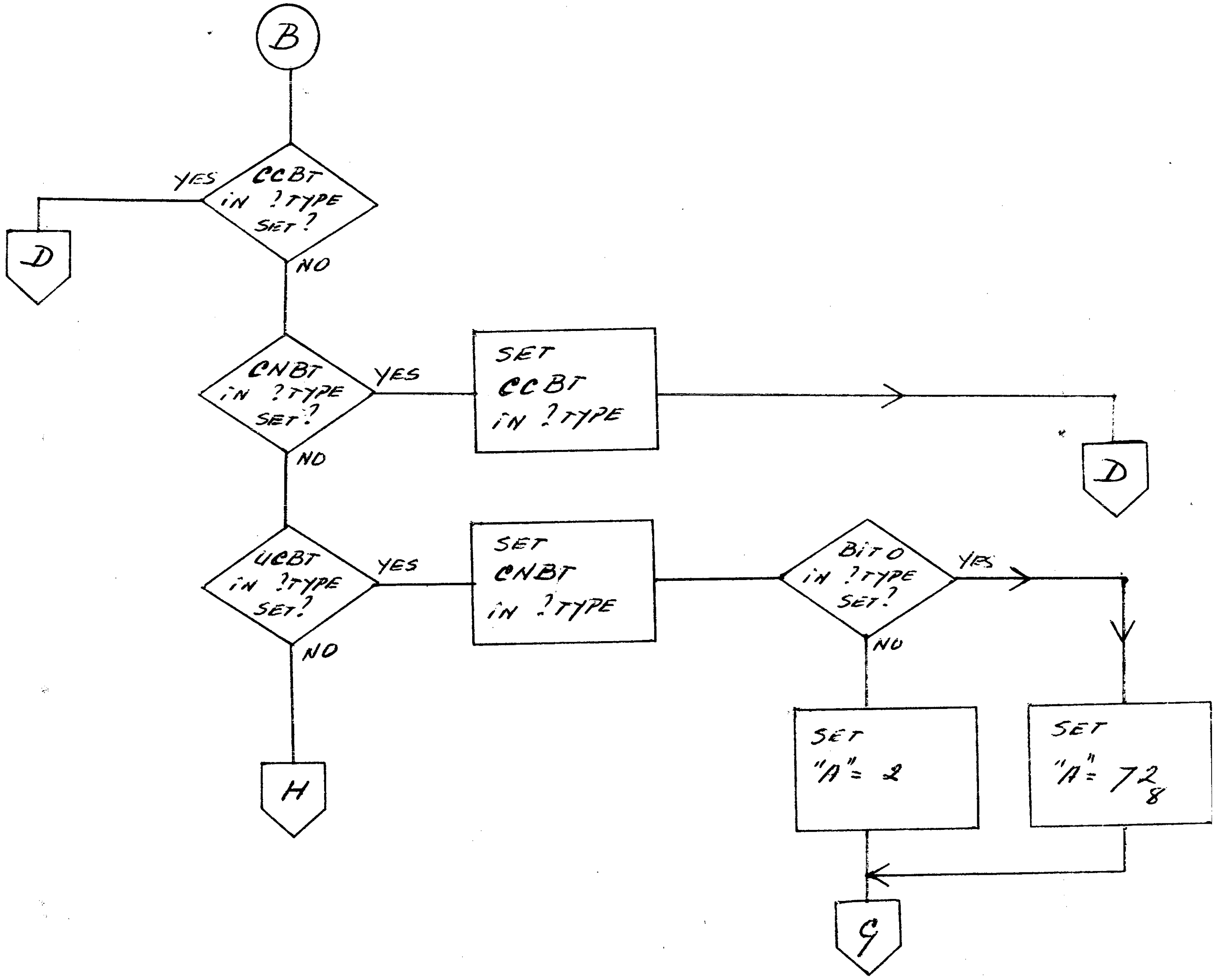
SELECTRIC OUTPUT CONVERSION (1 OF 6)



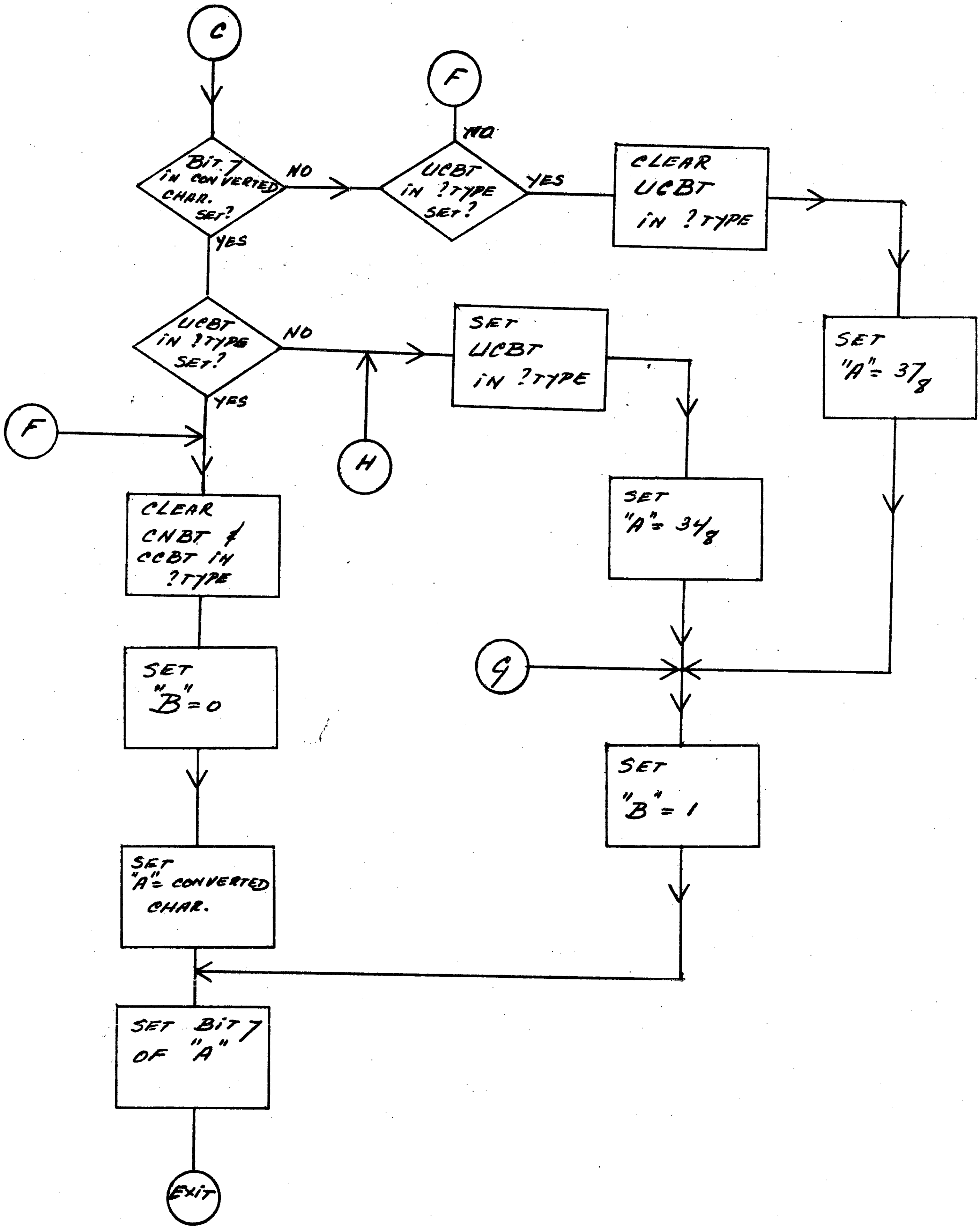
SELECTRIC OUTPUT CONVERSION (2 OF 6)



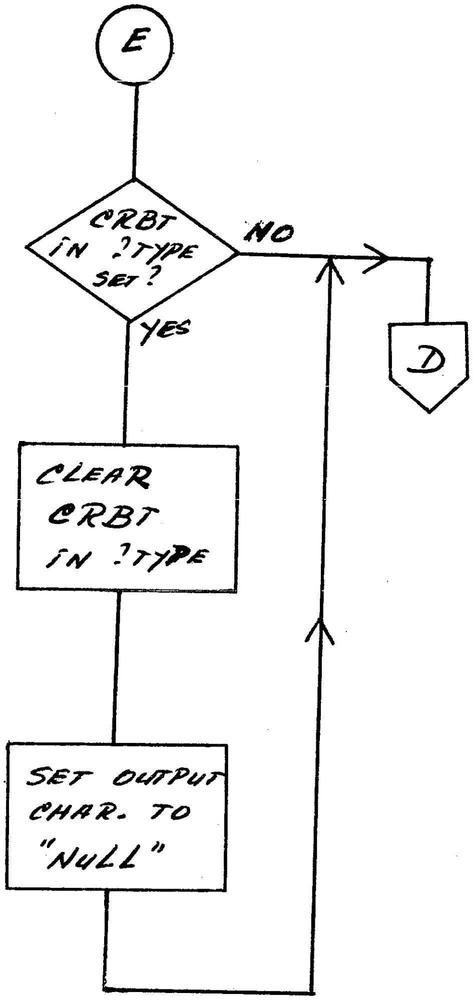
SELECTRIC OUTPUT CONVERSION (3 OF 6)



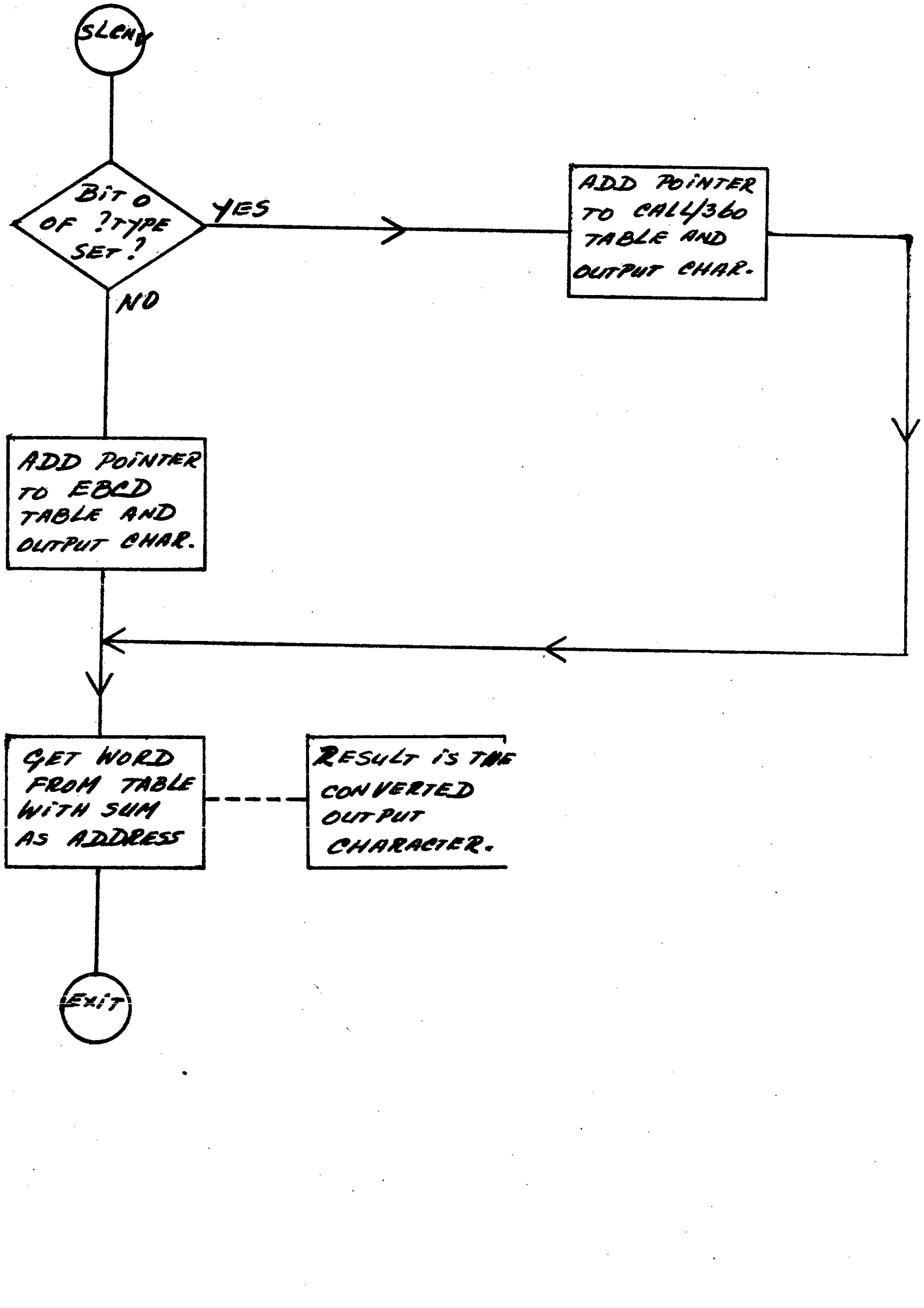
SELECTRIC OUTPUT CONVERSION (4 OF 6)



SELECTRIC OUTPUT CONVERSION (5 OF 6)

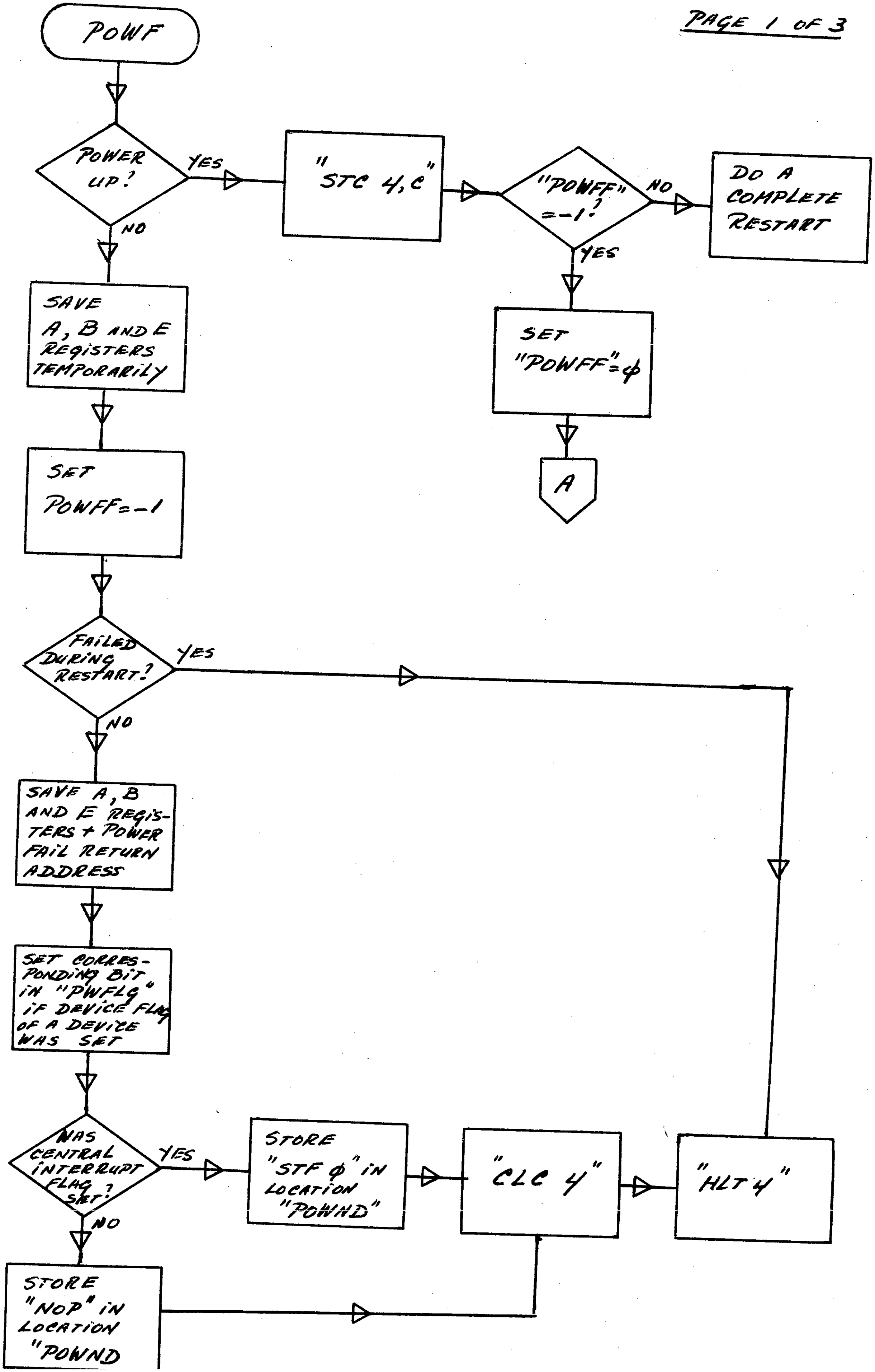


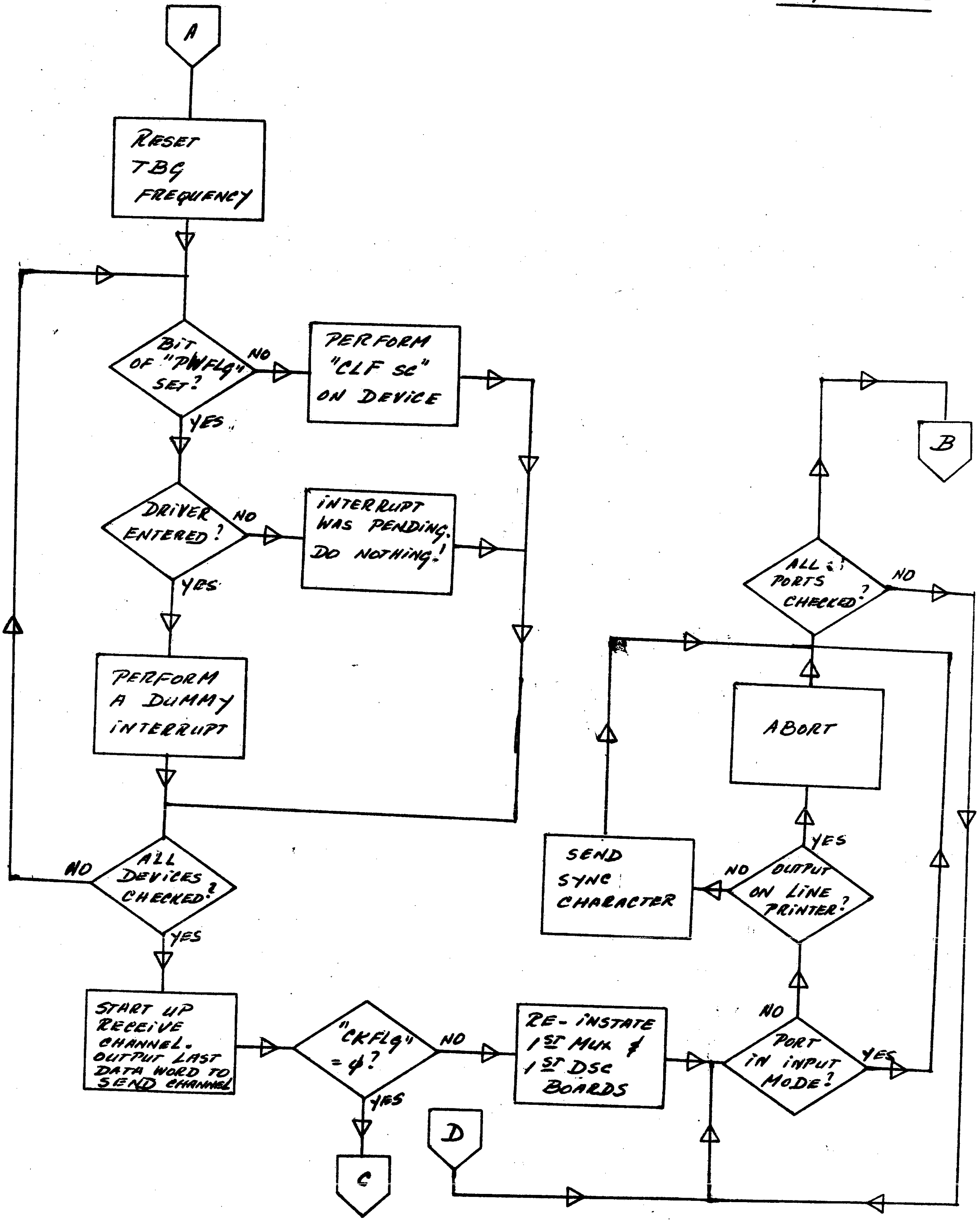
SUBROUTINE "SLCNV"

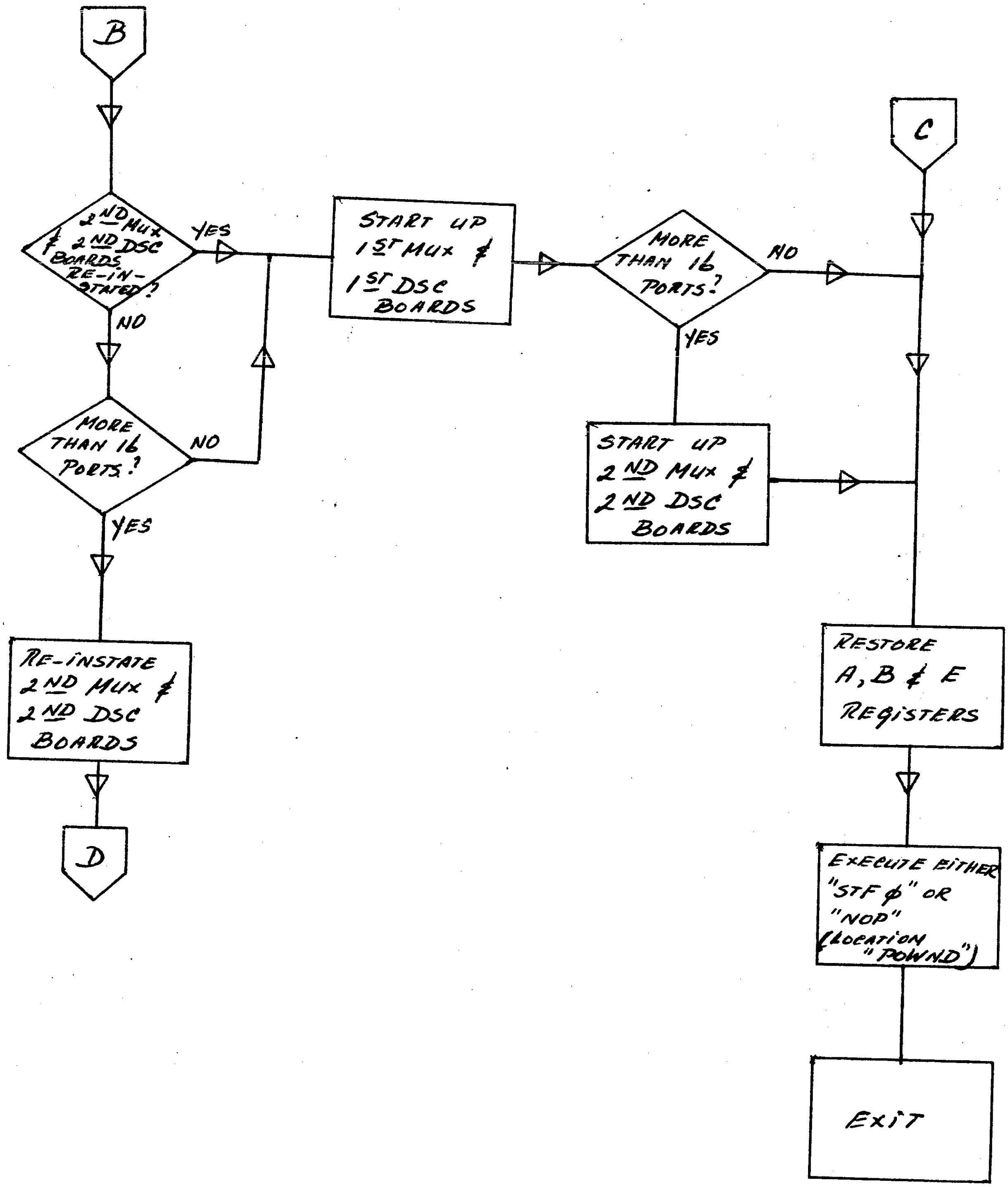

SLCNV
BIT 0 OF ?TYPE SET?
YES
NO
ADD POINTER TO CALL/360 TABLE AND OUTPUT CHAR.
ADD POINTER TO EBCD TABLE AND OUTPUT CHAR.
GET WORD FROM TABLE WITH SUM AS ADDRESS
RESULT IS THE CONVERTED OUTPUT CHARACTER.
EXIT

POWER FAIL/AUTO RESTART

PAGE 1 OF 3


POWF
POWER UP?
YES
NO
"STC 4,C"
"POWFF" =-1?
NO
YES
DO A COMPLETE RESTART
SET "POWFF"=φ
A
SAVE A, B AND E REGISTERS TEMPORARILY
SET POWFF=-1
FAILED DURING RESTART?
YES
NO
SAVE A, B AND E REGISTERS + POWER FAIL RETURN ADDRESS
SET CORRESPONDING BIT IN "PWFLG" IF DEVICE FLAG OF A DEVICE WAS SET
WAS CENTRAL INTERRUPT FLAG SET?
YES
NO
STORE "STF φ" IN LOCATION "POWND"
"CLC 4"
"HLT 4"
STORE "NOP" IN LOCATION "POWND


A
RESET TBG FREQUENCY
BIT OF "PWFLG" SET?
NO
PERFORM "CLF SC" ON DEVICE
YES
DRIVER ENTERED?
NO
INTERRUPT WAS PENDING. DO NOTHING!
YES
PERFORM A DUMMY INTERRUPT
ALL DEVICES CHECKED?
NO
YES
START UP RECEIVE CHANNEL. OUTPUT LAST DATA WORD TO SEND CHANNEL
"CKFLG" = ϕ?
NO
YES
C
D
RE-INSTATE 1ST MUX & 1ST DSC BOARDS
PORT IN INPUT MODE?
NO
YES
OUTPUT ON LINE PRINTER?
YES
NO
SEND SYNC CHARACTER
ABORT
ALL PORTS CHECKED?
NO
B


B
2ND MUX & 2ND DSC BOARDS RE-INSTATED?
YES
NO
START UP 1ST MUX & 1ST DSC BOARDS
MORE THAN 16 PORTS?
NO
YES
MORE THAN 16 PORTS?
NO
YES
RE-INSTATE 2ND MUX & 2ND DSC BOARDS
D
C
START UP 2ND MUX & 2ND DSC BOARDS
RESTORE A, B & E REGISTERS
EXECUTE EITHER "STF ϕ" OR "NOP" (LOCATION "POWND")
EXIT

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 common area has not been allocated, ALCOM is called to compute the space needed and move the program accordingly. If the program is already compiled (SYMTB=SPTR≠0) PBPTR is set back to the first word following the format stack (FCORE) and phase 2 simply reinitializes all of the variables to undefined. If the program is semi-compiled (SPTR=0, SYMTB≠0) we may skip building the symbol table. 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). Also, at this time statement number references are replaced by absolute addresses. This is facilitated by dividing the program into 32 parts and building an 64 word table in ERSEC containing the first statement number and address of each part. During compilation SPTR points to the program word being processed. Pointers to <FILES statements> are stored in

FLSTS and a count of them is kept in FILCT. 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 at least one <FILES statement> has been found, BASIC calls the system, which analyzes the <FILES statements> and builds the file table, filling all but the fifth, sixth, ninth, tenth, eleventh and fifteenth 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=B) some symbols may have two entries. If this is the case, the "don't know" entry will always be father 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 common area), and initializing the values of all variables except those in common. 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. Note that for arrays in common, the declared dimensions in the <COM statement> are checked against those in the common area. If they match and the dynamic dimensions are consistent (i.e., less than or equal to the declared ones) then the values are left alone. Otherwise they are set to undefined and both sets of dimensions are set equal to those in the <COM statement>. For strings, the physical length is checked against the declared length and the logical length tested to be less than or equal to the physical length. If these tests fail the physical length is set to the declared length and the logical length is set to zero. Simple variables in common are left untouched.

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 5000 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 program is moved so that SPROG=PBUFF (no common area). 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. PRNST assumes responsibility for recognizing extra <FILES statements> and <COM statements> that are out of order. If such an error condition is encountered, SPTR is set to point before the statement which caused the error (it hasn't been compiled). Then PRNST calls DCMPL, which calls PRNST. The statement causing the error is not seen this time, so PRNST and DCMPL can exit correctly.

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 format stack (used in repeated RUNS of a program). After printing the program name (unless the program was CHAINED to) XEC proceeds to initialize the file table. A buffer the size of a logical record is allocated for each file and pointers to the word following it are placed in words 9 and 10 of the file table. The first word of the disc address of the record in the buffer (word 5) is set to 100000_8 to indicate that no record is present. Word 11 is set to Ø, indicating that no end-of-record/end-of-file exit has been specified. Word 15 is set to 0 as a null protectmask. If the file is read-only a message to this effect is printed, following the program name, unless the program was CHAINED to.

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 or 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 string' 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 entry.

<ENTER statement> execution assigns a value to a string variable or a simple variable. If a '#' follows the ENTER, the user's port number (0-31) is assigned to the first variable. The <ENTER statement> is timed and the length of time it took to respond (in seconds) is assigned to another variable. The input analysis proceeds much like an input statement with one variable, with the notable exception that no error messages are printed. Instead, the response time variable is negated if an error occurs. If the user does not respond within the alloted time, the response time variable is set to -256. This is non-ambiguous since response times are between 1 and 255 seconds inclusive. Also, for string input leading blanks are non stripped off and quote marks are allowed as characters.

<READ statement> execution assigns values to variables in the list. FDATA is primed to obtain values from either a file of 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 and 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 exits. 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.

<PRINT USING statement> execution sends formatted output to the teletype. TEMP1 is set to point to the first operand, NCH is set to the number of characters in the format string for a partial string or 0 for a full string, B = > the first word of the format string and A = character of the string to start with. Then the formatter is called, and it takes care of

retrieving the operands, formatting their values, and outputting them. See description of formatter elsewhere.

<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 one record 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 block. Also, the last change date for each file must be updated if any records have been changed. Following this, the word DONE is sent to the teletype and control exits to the scheduler.

<CHAIN statement> execution consists of calling the CHAIN library routine to get the named program from the disc and start execution of it.

<ASSIGN statement> execution changes the file referred to by a specified file number. After interpreting the file name and file number and dumping the last record of the previous file, it calls the ASSIGN library routine to update the last change date for the previous file and put information for the new file in the file table. Control is transferred back to BASIC to set the return code in the variable specified for it and set the protectmask in the file table if one is specified.

<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 USING calls the formatter just as PRINT USING does. The EVEXP routine in the formatter takes care of picking up the elements of the array one by one, in rows. 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 innter 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 source 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.

FORMATTER

Upon entry, B contains the address of the format string and A contains an index describing which character of the string the formatter should begin with. (This is for substring expressions.) The variable NCH will be zero if the entire string from character (A) on is to be considered. Otherwise it will contain a character count.

The routine grabs off the carriage control character, if any, and saves it. It then searches for a delimiting character ('/', ',', ')' or end of string) and processes the specification up to that delimiter. The characters of the specification are examined and stored 1 character/word on a stack. Replicators are converted to binary and negated. Flags are set to indicate string, integer, fixed or floating point specifications. Literal strings are outputted directly from the stack and absence of any flag being set indicates a blank specification. The stack is then processed from top to bottom and each character or binary replicator and character causes appropriate output.

Strings are handled in a straight forward manner and may be output only if the string flag is set. Numbers are converted from binary to decimal and are stored 1 decimal digit/word in a number holding buffer. For integer or fixed specifications, the numbers are stored with decimal exponent of Ø and output directly according to the specification. For a floating specification, the number is stored with a maximum of 7 digits to the left of the decimal point and the decimal exponent is set accordingly. The number is then output in a straight forward manner.

When the stack has been exhausted, the delimiter(s) are processed. If the end of the string has been reached, and there remain expressions to output, the string is reprocessed from the beginning. If the end has not been reached, the next delimiter is found and the specification is processed as above. If there were no more expressions to output the carriage control character is processed and execution terminates.

Grouped specifications are handled by saving pointers to the beginning of the group and upon notice of the end, returning and reprocessing the entire group.

Formatter Utility Routines:

MTL1 expects an unpacked floating point number in MANT1, MANT2 and EXP and returns a number there which has been made greater than 1. EXPON holds the count of multiplications necessary to make the number greater than 1.

DTL1 expects an unpacked floating point number in MANT1, MANT2 and EXP and returns a number there on return. The A register contains a count of the number of divides necessary to make the number less than 1.

ROUND expects an unpacked floating point number in MANT1, MANT2 or EXP and rounds the number in the number holding buffer.

OUTBL and OUTCL are self explanatory.

DSRCH searches the format string starting at character pointed to by DP for a delimiter. If one is found, it is returned in the A register, and DP points to its location in the string. If the end of the string is reached and no delimiter is found, DP points one character past the end of the string.

MCHAR expects the address of a character in the A register. It returns that character in the A register. If the 0-Bit is set, blanks are ignored and the first non-blank character after that address is returned. In this case, if a delimiter is reached, the address of this delimiter (i.e., DP) is returned in the A register.

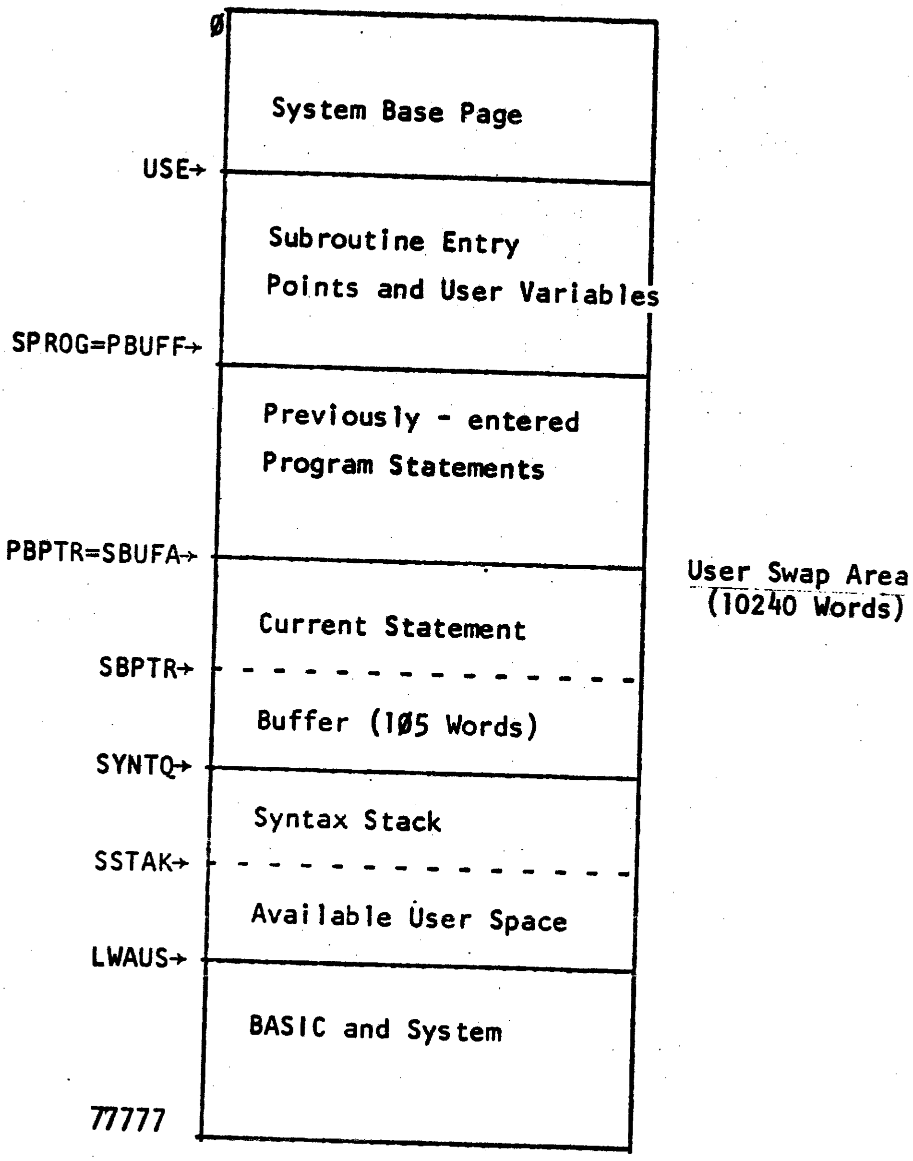
EVEXP is responsible for extracting the next variable to be output by the formatter. FFLG determines whether this is a MAT PRINT USING or a PRINT USING statement. For matrices, the first time EVEXP is called it verifies that all array elements are valid and returns the first element. Subsequent calls to EVEXP return array elements one at a time on a row by row basis. Numerical values are returned in the A and B registers and strings are returned with a pointer in the A register and the number of characters in the B register. EVEXP also evaluates the functions TAB, LIN and SPA and then goes to the next operand.

NOTES ON THE ERROR ROUTINES

Errors are handled 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, FERRS and WERRS are conveniences to denote subsections of the table used for run-time errors, format errors and warning-only errors. After printing a format error message, the offending format string is also printed. 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.

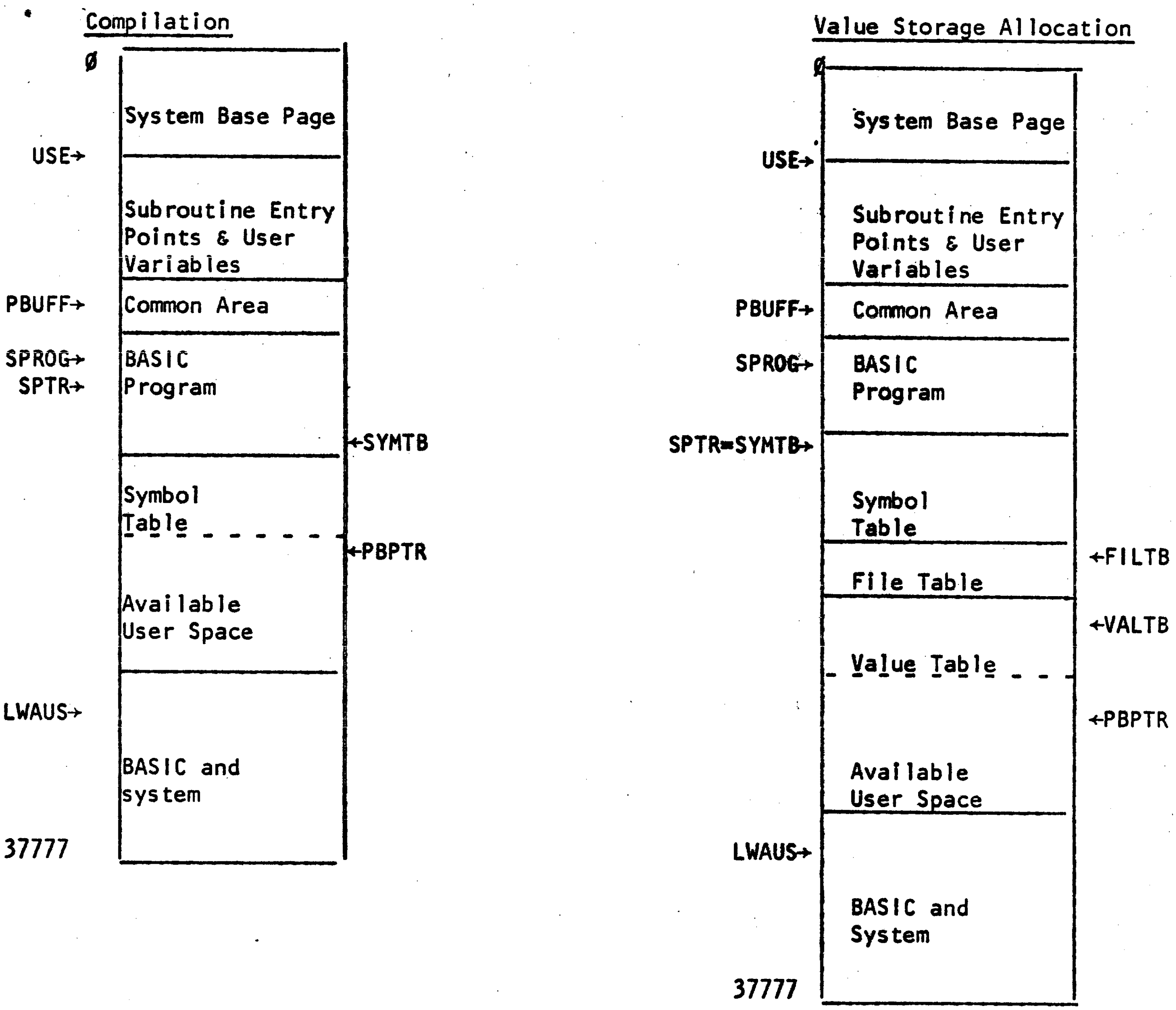
SYNTAX (Phase 1)



Pointers

USE	Fixed, first word of user swap area.
PBUFF	Fixed, first word of program space.
SPROG	Fixed, first word of program.
SBUFA	Variable, first word o statement being syntaxe
PBPTR	Variable, first word o program space not used 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)



SPROG - Variable, first word of program

SYMTB - Variable, first word of symbol table.

SPTR - Variable, word of program being processed.

FILTB - Variable, first word of file table.

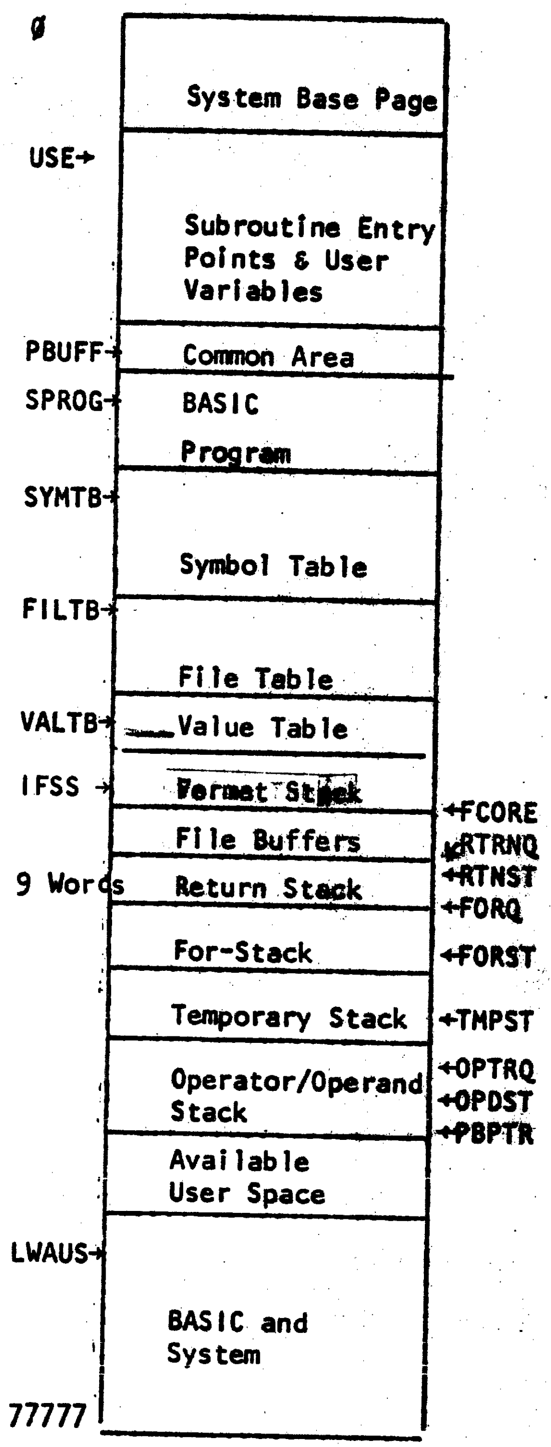
VALTB - Variable, first word of symbol value table
(FILTB = VALTB if no <FILES statement> is in program)

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)



IFSS - Variable, first word of format stack.

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

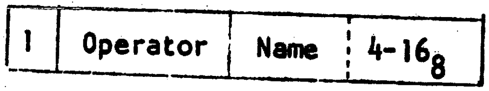
BASIC statements are represented internally by the sequence number followed by the length in words (including the sequence number and length words) followed by the statement body. The statement body is composed almost entirely of operator-operand pairs which occupy from one to three words each. Null 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

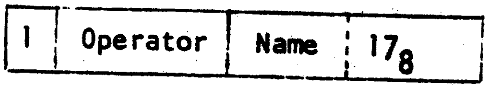
Ø	Operator	Ø		Null Operand
Ø	Operator	Name	Ø	String Variable
Ø	Operator	Name	1-3	Array Variable
Ø	Operator	Name	$4\text{-}16_8$	Simple Variable
Ø	Operator	Name	17_8	Function Variable

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 $5\text{-}16_8$ identify simple variables whose names include the digit $Ø\text{-}9_{10}$ respectively (e.g. 6,7 represents F2). Type 17_8 identifies a programmer-defined function (e.g. 32_8, 17_8 represents FNZ).

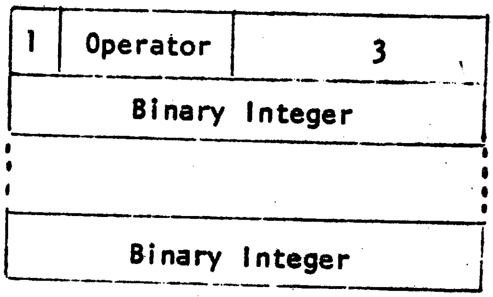
'Constant' Operands



Parameter

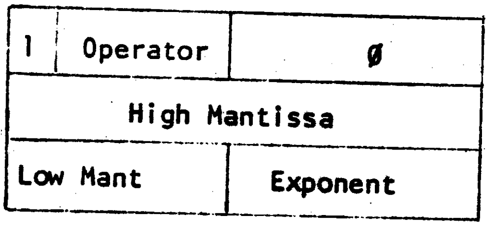


Pre-defined Function

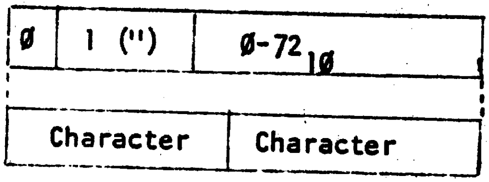


Formal Dimension/

Branch Address List



Numerical Constant



String Constant

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 21$_8$ or 24$_8$ to 30$_8$ (TAB to COS or ZER to TRN). A flagged (bit 15 set) operand of 3 identifies either a formal dimension in a <DIM statement> or <COM 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	4	AND	6Ø	END
5		(unused)	33	3	OR	61	STOP
6		(unused)	34	6	MIN	62	DATA
7		(unused)	35	6	MAX	63	INPUT
1Ø	1	)	36	5	<>	64	READ
11	1	]	37	5	>=	65	PRINT
12	13(1)	[	4Ø	5	<=	66	RESTORE
13	13(1)	(	41	11	NOT	67	MAT
14	11	+(unary)	42	11	ASSIGN	7Ø	FILES
15	11	-(unary)	43		USING	71	CHAIN
16	2	,(subscript)	44		IMAGE	72	ENTER
17	2	=(assignment)	45		COM	73	'IMPLIED' LET
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 WI = Y = (B = C) + 3*A[1,J+K]

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

```
2Ø DIM A[5], C[6,12]

    24
    14
Ø | 47 |  1 | 1
1 | 12 |    | 3
          5
Ø | 11 |    | Ø
Ø |  2 |  3 | 2
1 | 12 |    | 3
          6
1 | 16 |    | 3
         14
Ø | 11 |    | Ø
```

20
12
DIM A[
[d [illegible]
5
] , [illegible]
C [illegible]

```
3Ø  DEF  FNC (X) = X + AØ

        36            30
         7             7
Ø | 50 | 3 | 17      DEF ,
1 | 13 |3Ø |  4       ( ,  X
Ø | 1Ø |   |  Ø       ) ,  —
1 | 17 |3Ø |  4       = ,  X
  | 2Ø | 1 |  5       + ,  AO
Ø |  Ø |   |  Ø      eof , —
```

```
4Ø  REM  ARK

     5Ø             40
      5              5
Ø|51|4Ø          REM | 0
 Ø4Ø522            A R
 Ø454ØØ            K  O
```

```
5Ø  GOTO  A  OF  1Ø, 2Ø, 3Ø

        62            50
         7             7
Ø | 52 | 1 | 4     GOTO , A
1 | 74 |   | 3      OF ,  FBL
        12             10
        24             20
        36             30
```

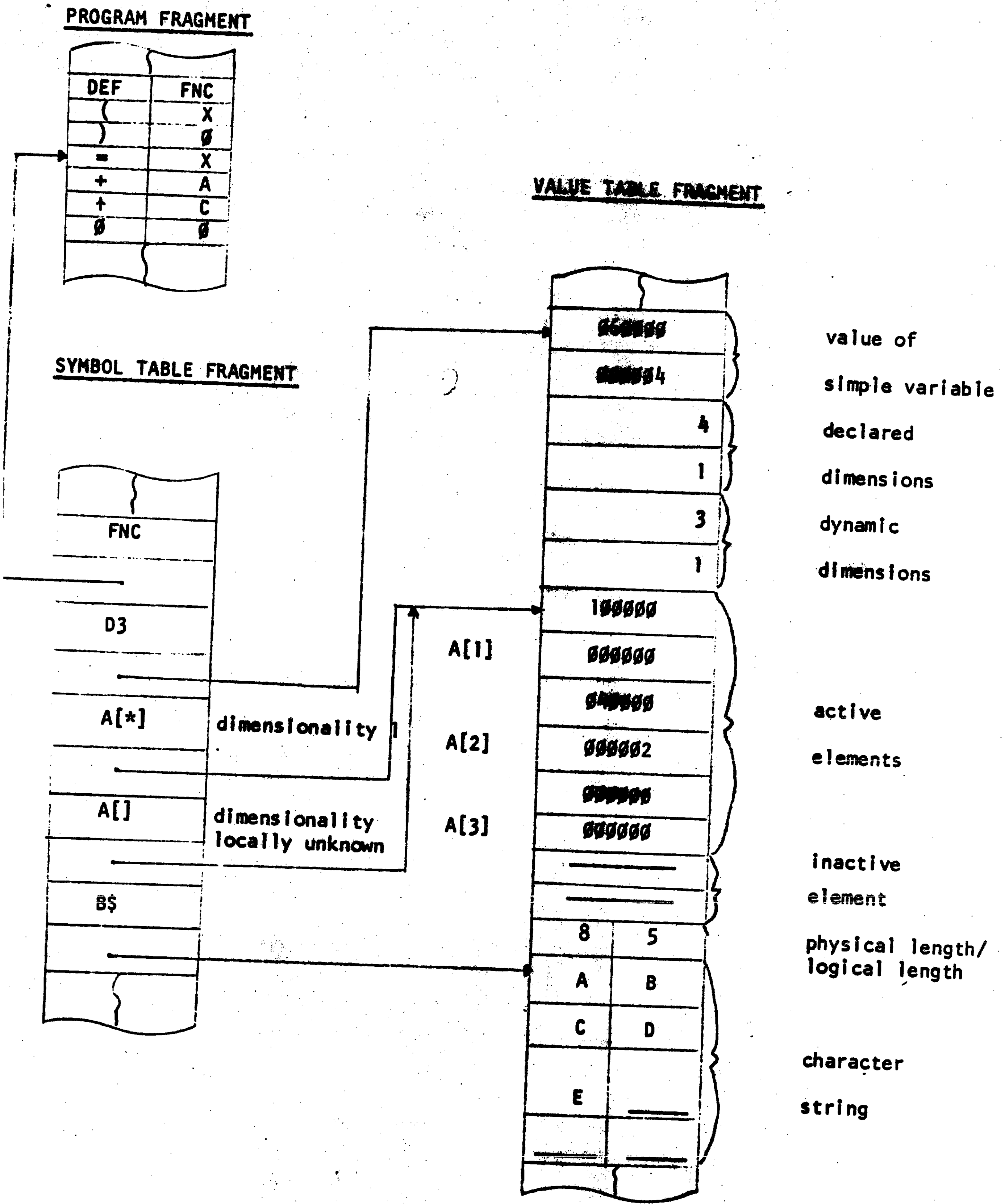
```
6Ø  DATA  -1, "ABC"

      74              60
      11               7
  1|62|   |Ø         DATA
   1ØØØØØ             -1
   ØØØØØØ
 Ø |2|    |Ø           ,
 Ø |1|    |3
   Ø4Ø5Ø2          A    B
   Ø414ØØ          C
```

```
7Ø MAT READ #K;A[I]

     1Ø6            70
      11             9
Ø | 67 |    | Ø     MAT
Ø | 64 |    | Ø     READ
Ø |  4 | 13 | 4     # ,  K
Ø |  3 |  1 | 1     ; ,  A[]
Ø | 12 | 11 | 4     [ ,  I
Ø |    |    | Ø    eof   —
Ø | 11 |    | Ø     ] ,  —
```

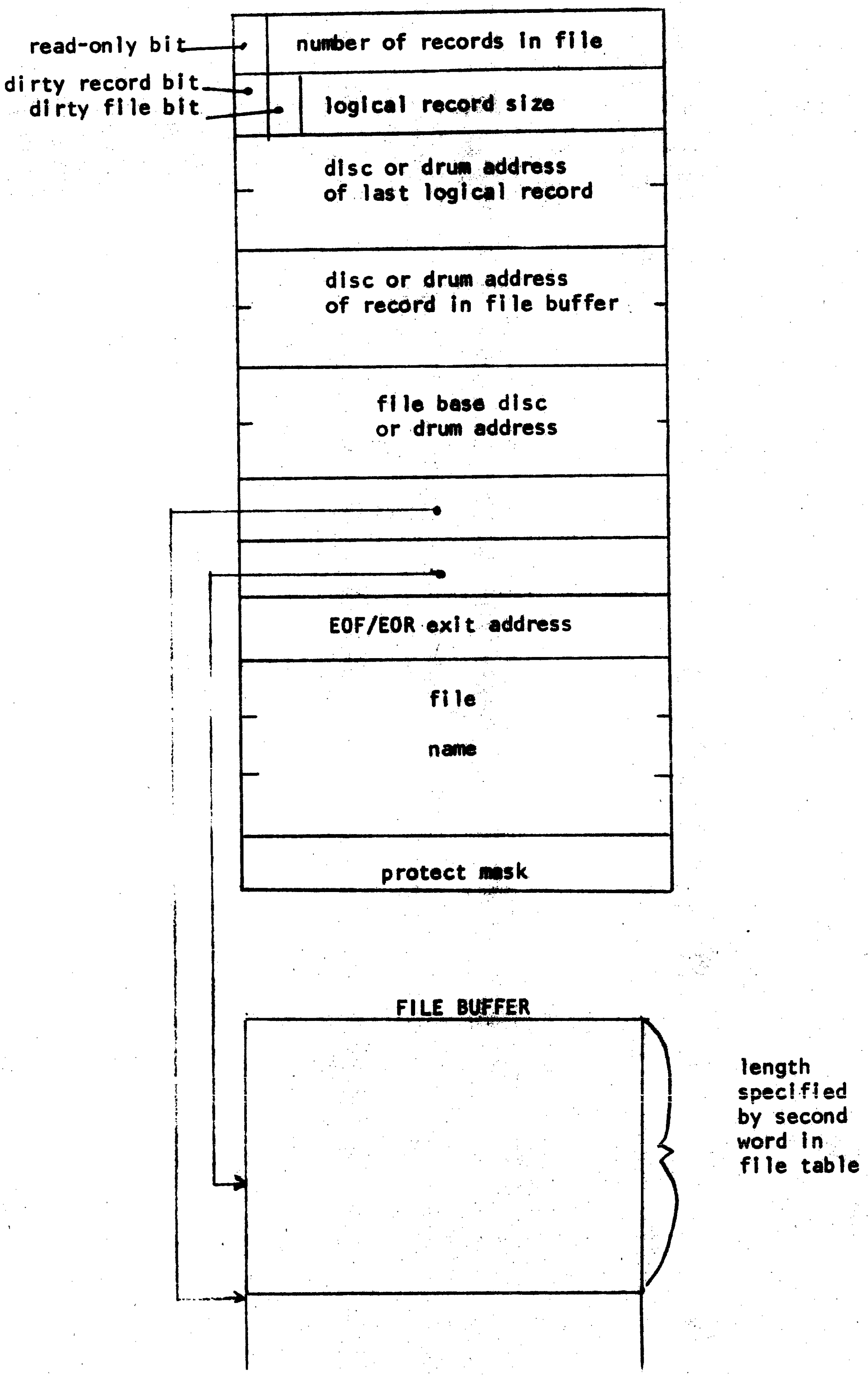
BASIC Variable Storage Allocation



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 and common area are simply the concatenation of the values for the symbols in the program, excepting programmer-defined functions.

FILE TABLE ENTRY


read-only bit
dirty record bit
dirty file bit
number of records in file
logical record size
disc or drum address of last logical record
disc or drum address of record in file buffer
file base disc or drum address
EOF/EOR exit address
file name
protect mask
FILE BUFFER
length specified by second word in file table

FILE TABLE

The file table consists of one fifteen-word entry for each file or place-holder ("*") in the FILES statements. Bit 15 of the first word is set if another user had read/write access to the file when it was requested (except for Axxx users) or if the file is a library file not being accessed by its owner. Bit 15 of the second word is set when an item is stored in the buffer, so that only records which are changed will be written back to the disc or drum. Bit 14 of the second word is set when a record is written to disc or drum and is used during program termination as a basis for updating the last changed date word in the file's directory entry.

A logical record-sized buffer is associated with each file table entry, and is accessed through pointers in the entry. An intra-buffer pointer designates the next portion of the record to be written or read. A fixed pointer to the first word not in the buffer acts as a bound on the intra-buffer pointer.

FILE CONTENTS

There are 4 data types possible in a file. A string has bit 9=1 and the length in characters in the lowest 7 bits of the first word, followed by the string packed 2 characters per word. A two-word floating point number has the upper two bits of the first word different, except for a zero, which has both words zero. An end-of-file is a -1, and an end-of-record is a -2, in the first word.

Data written to or read from a file is first exclusively ORed with the fifteenth word of the file table entry. This has no effect, of course, unless that word is nonzero. It will be nonzero only if an ASSIGN statement has been used to specify the file, and the statement included a protect mask parameter. End-of-file marks, end-of-record marks, and the first word of strings are not masked. Floating point numbers are masked when they are written to or read from the file buffer. Strings are masked when the buffer is read from or written to the disc or drum.

Update Last Change Date Routine

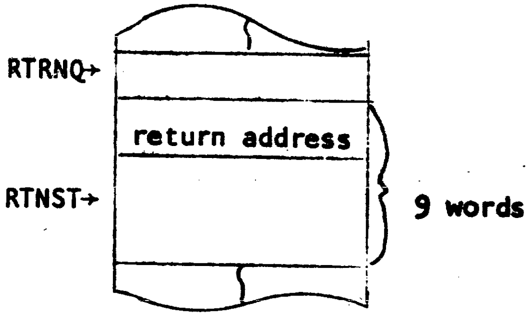
Each file and program entry in the directory has a word containing the hour of the year when the entry was last changed. It is necessary to update this word for files when a program is terminated for any of the following reasons: normal termination, CHAINing to a new program, error termination, abort and when a SLEEP or HIBERNATE command is issued.

The DFCHK bit in the user's ?FLAG word in his TTY table is set to 1 if there were any files statements in the program. This determines whether the LCD routine will be called. When it is, each file table entry is examined. If bit 14 of word 2 = 1, the file has been written on, so the last change date must be updated. This bit is set by the WRBUF routine.

The only abnormality in calling LCD occurs following an abort. The user is taken off the queue and re-inserted with priority 0 to run a core resident routine called ABUCD which writes the user to the swap track, calls LCD, and returns to the scheduler to finish aborting the user.

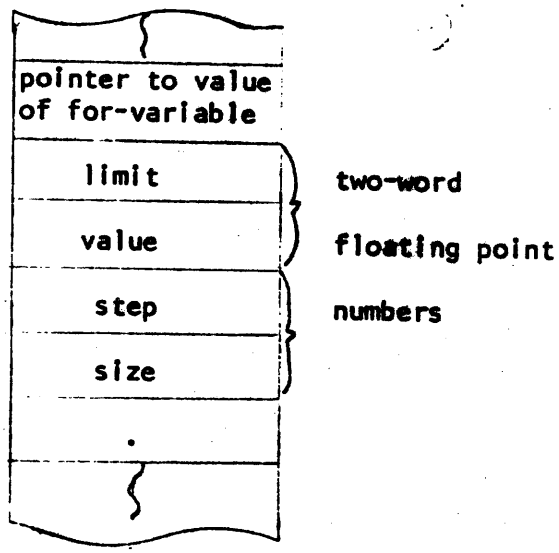
BASIC Run-Time Stacks

Return Stack



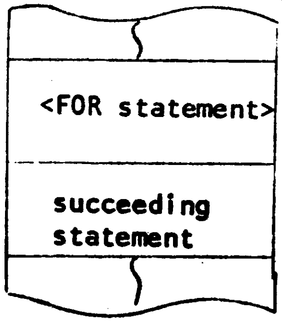
The return stack is of fixed length, holding from Ø to 9 one-word entries at any time. An entry is the absolute address of the statement following the GOSUB which placed the entry on the stack.

For-Stack Entry

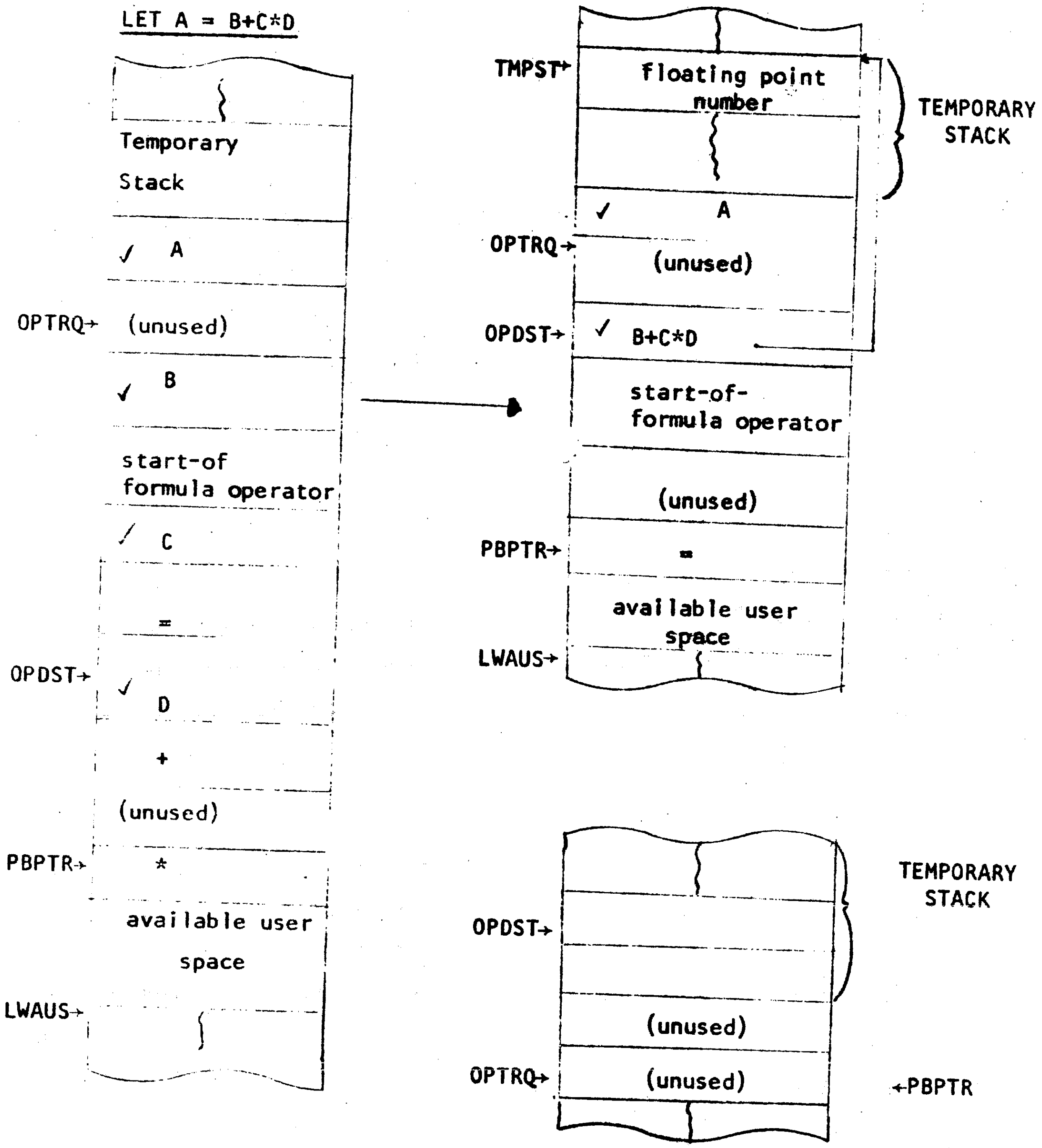


The for-stack is of variable length, containing one six-word entry for each for-loop which is currently active. Since the limit value and step size are kept in the entry, they may not be changed within the for-loop. The value of the for-variable is the one kept in the value table, so this may be altered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

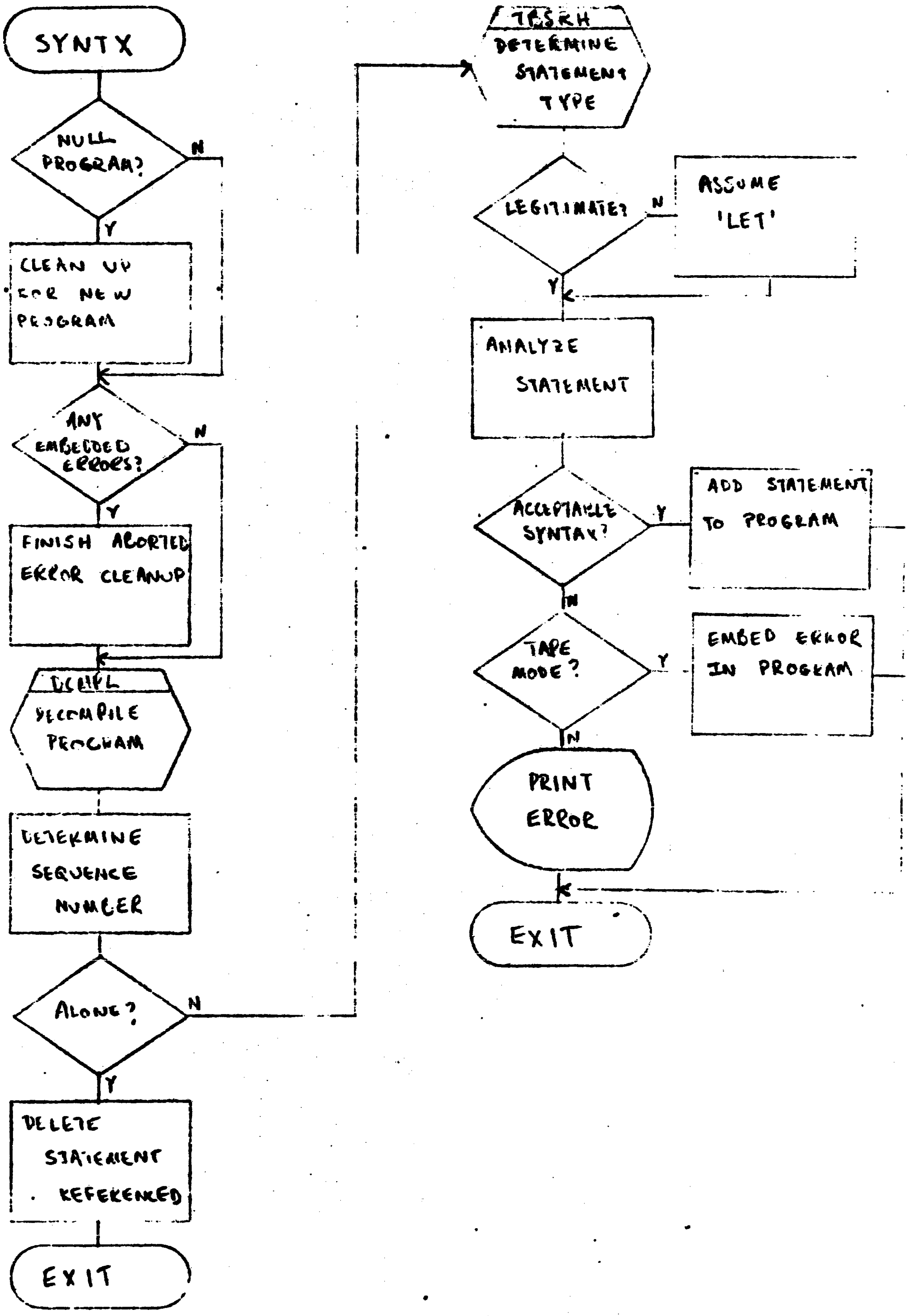
BASIC Language Processor Tables

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 thing particular to a user which remains when he is swapped out is his own teletype table.

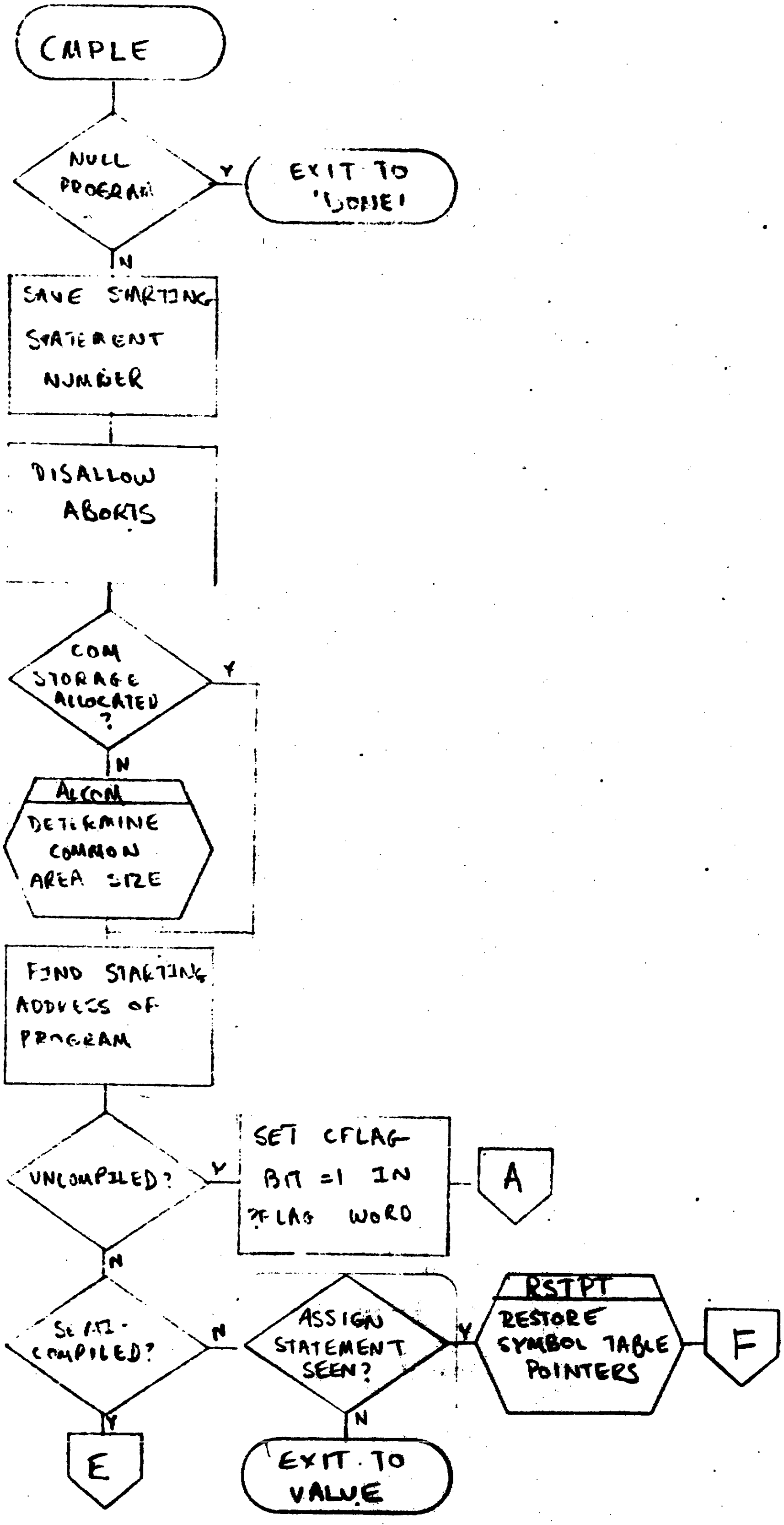
The tables headed by PDFTB (which must be in base page), SYNTB, XECTB, and FOJT are jump tables. The method in the last three cases is to compute a decision number, add the base address of the table, and transfer through the entry thus designated. The pre-defined function table is used by the formula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

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

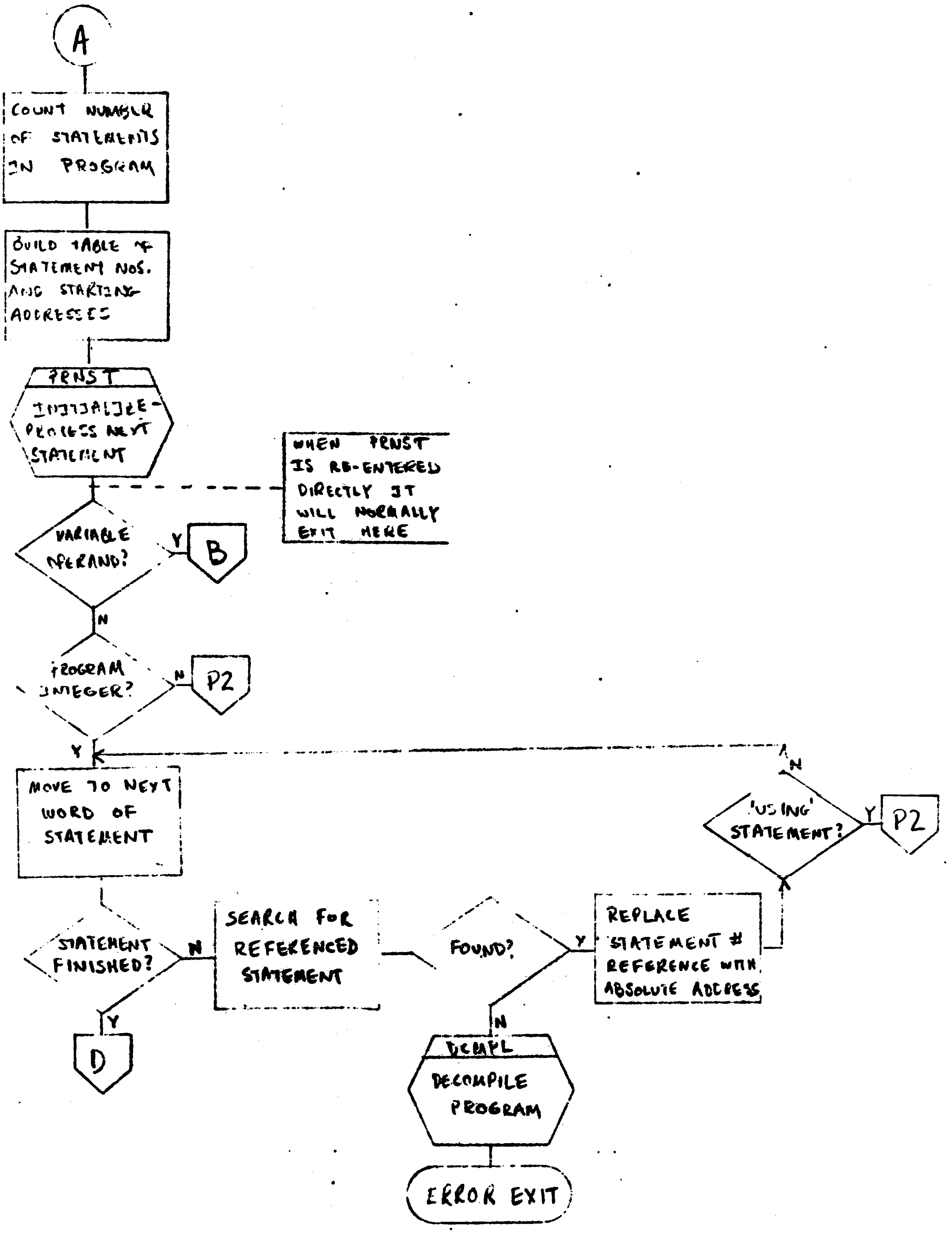
SYNTAX


SYNTX
NULL PROGRAM?
N
Y
CLEAN UP FOR NEW PROGRAM
ANY EMBEDDED ERRORS?
N
Y
FINISH ABORTED ERROR CLEANUP
DCMPL
DECOMPILE PROGRAM
DETERMINE SEQUENCE NUMBER
ALONE?
N
Y
DELETE STATEMENT REFERENCED
EXIT
TESRH
DETERMINE STATEMENT TYPE
LEGITIMATE?
N
ASSUME 'LET'
Y
ANALYZE STATEMENT
ACCEPTABLE SYNTAX?
Y
ADD STATEMENT TO PROGRAM
N
TAPE MODE?
Y
EMBED ERROR IN PROGRAM
N
PRINT ERROR
EXIT

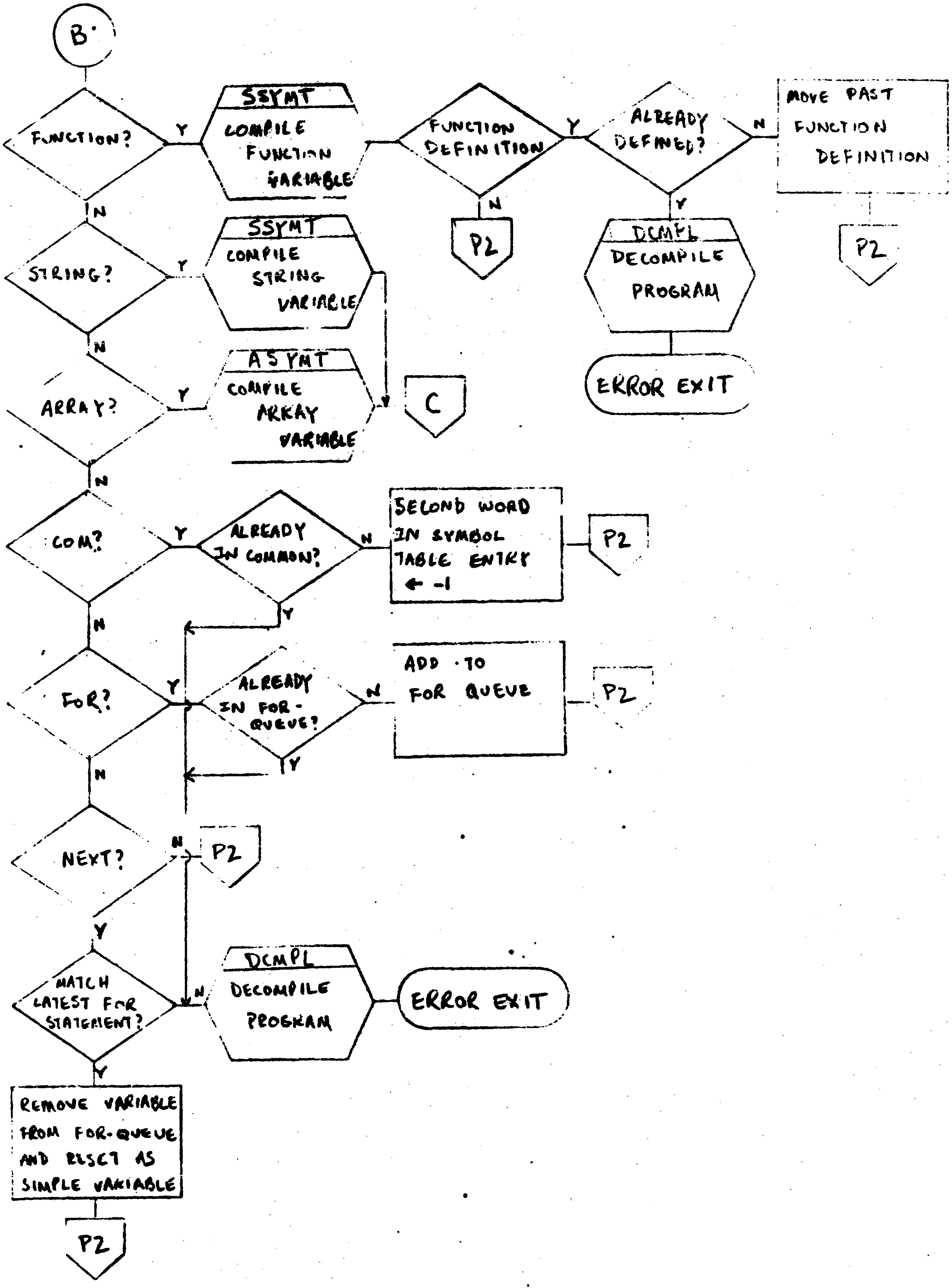
COMPILATION (1 of 4)



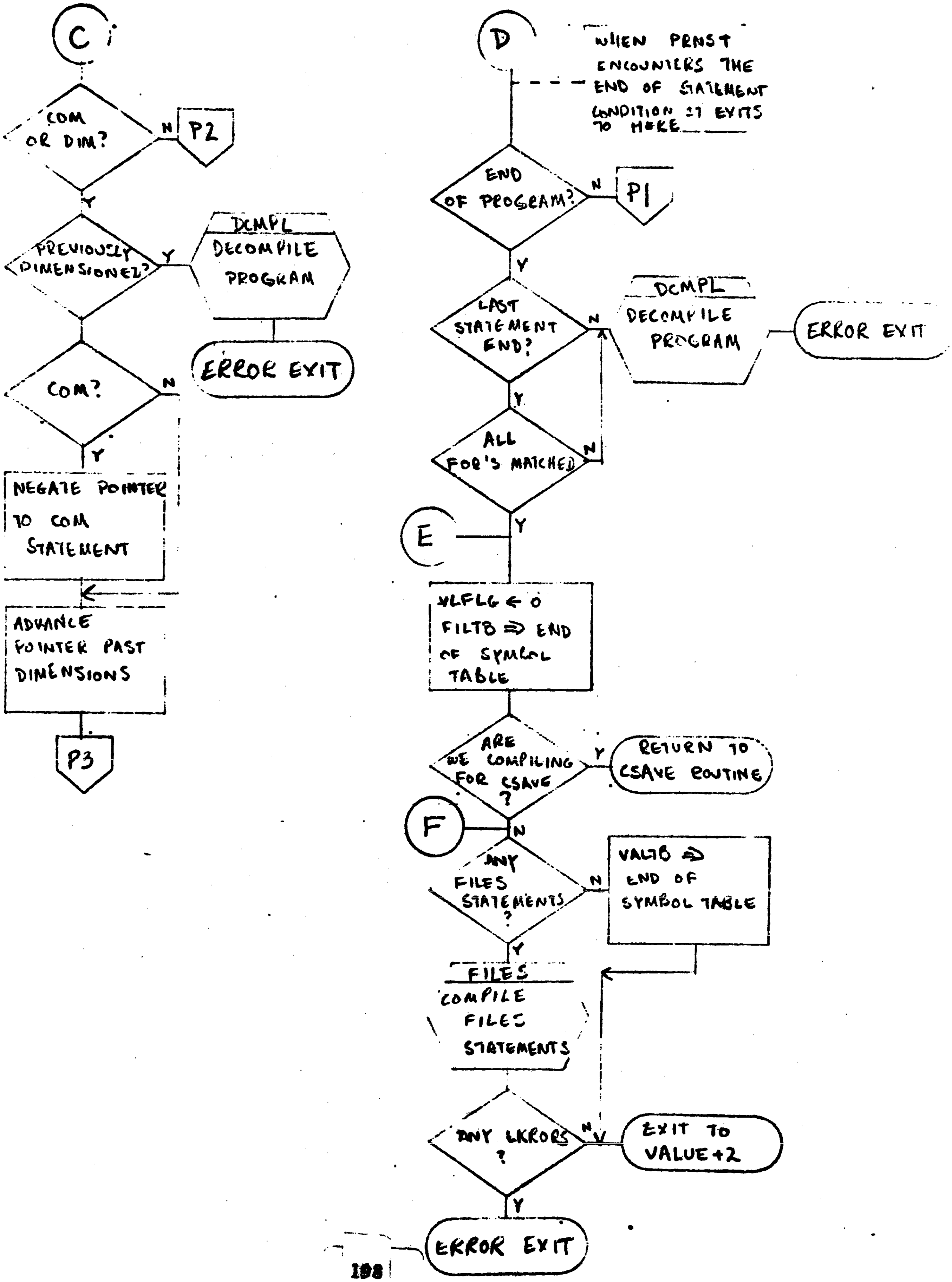
COMPILATION (2 of 4)


A
COUNT NUMBER OF STATEMENTS IN PROGRAM
BUILD TABLE OF STATEMENT NOS. AND STARTING ADDRESSES
PRNST
INITIALIZE - PROCESS NEXT STATEMENT
WHEN PRNST IS RE-ENTERED DIRECTLY IT WILL NORMALLY EXIT HERE
VARIABLE OPERAND?
Y
B
N
PROGRAM INTEGER?
N
P2
Y
MOVE TO NEXT WORD OF STATEMENT
STATEMENT FINISHED?
Y
D
N
SEARCH FOR REFERENCED STATEMENT
FOUND?
Y
REPLACE STATEMENT # REFERENCE WITH ABSOLUTE ADDRESS
'USING' STATEMENT?
Y
P2
N
N
DCMPL
DECOMPILE PROGRAM
ERROR EXIT

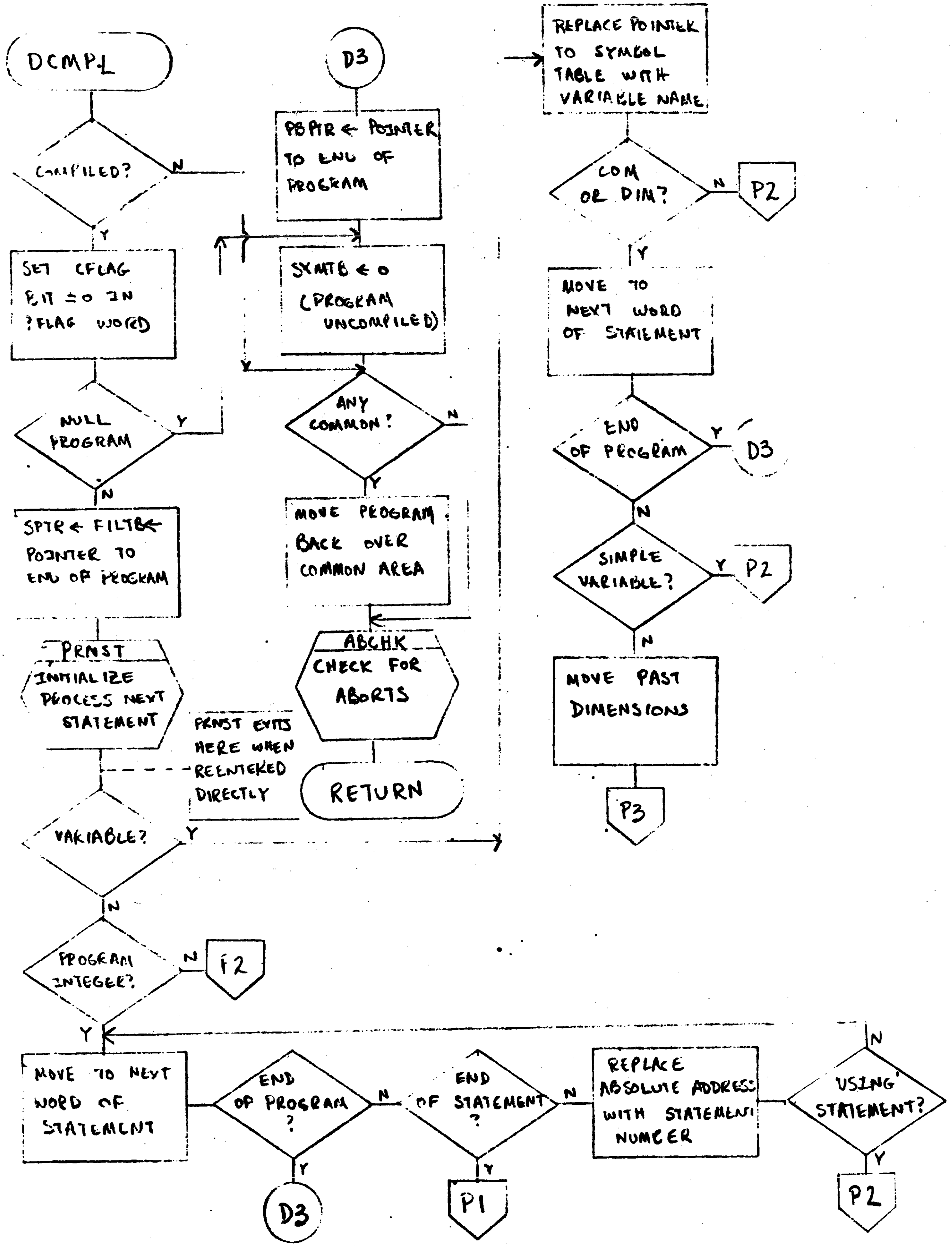
COMPILATION (3 of 4)



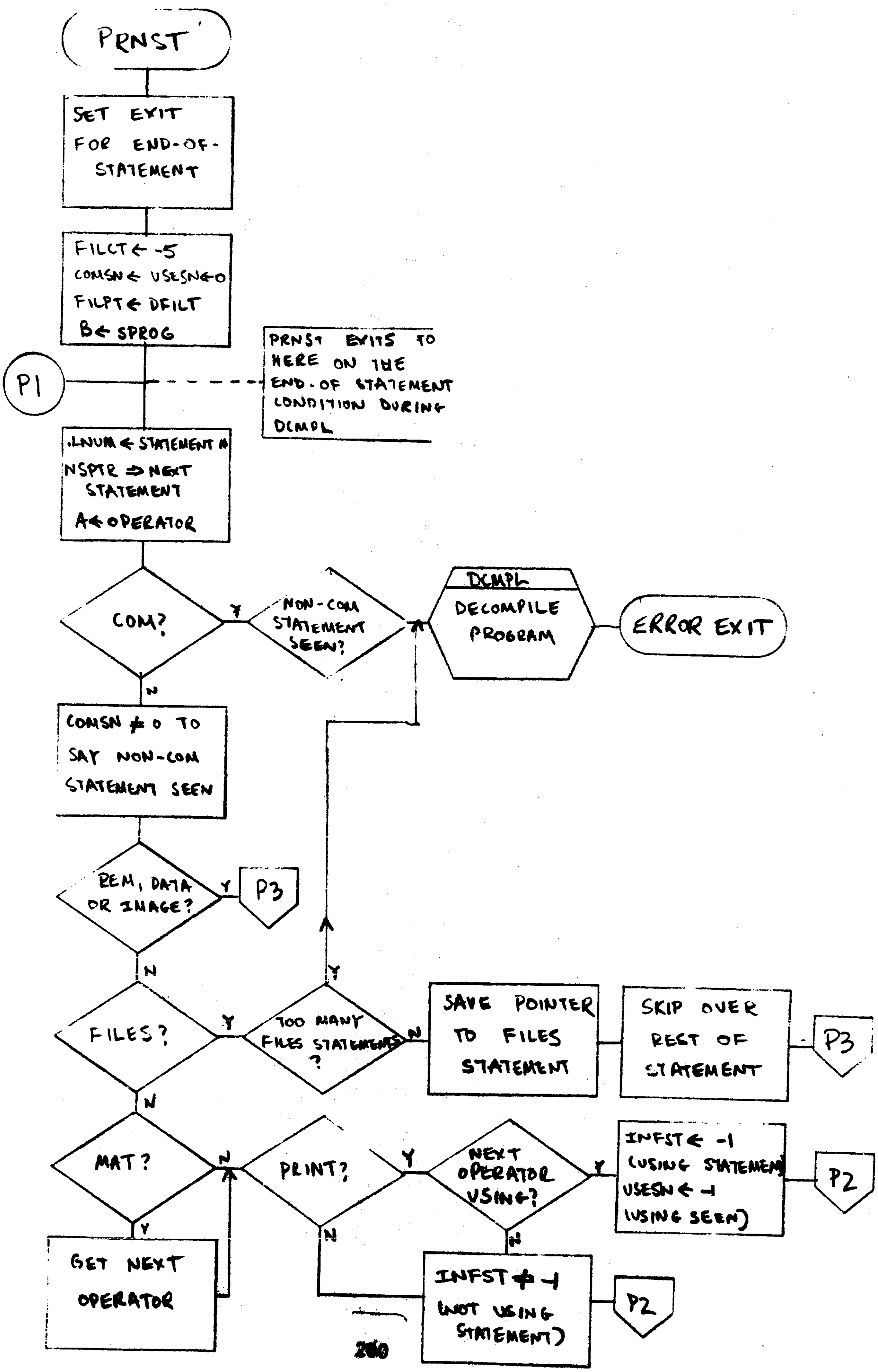
COMPILATION (4 of 4)



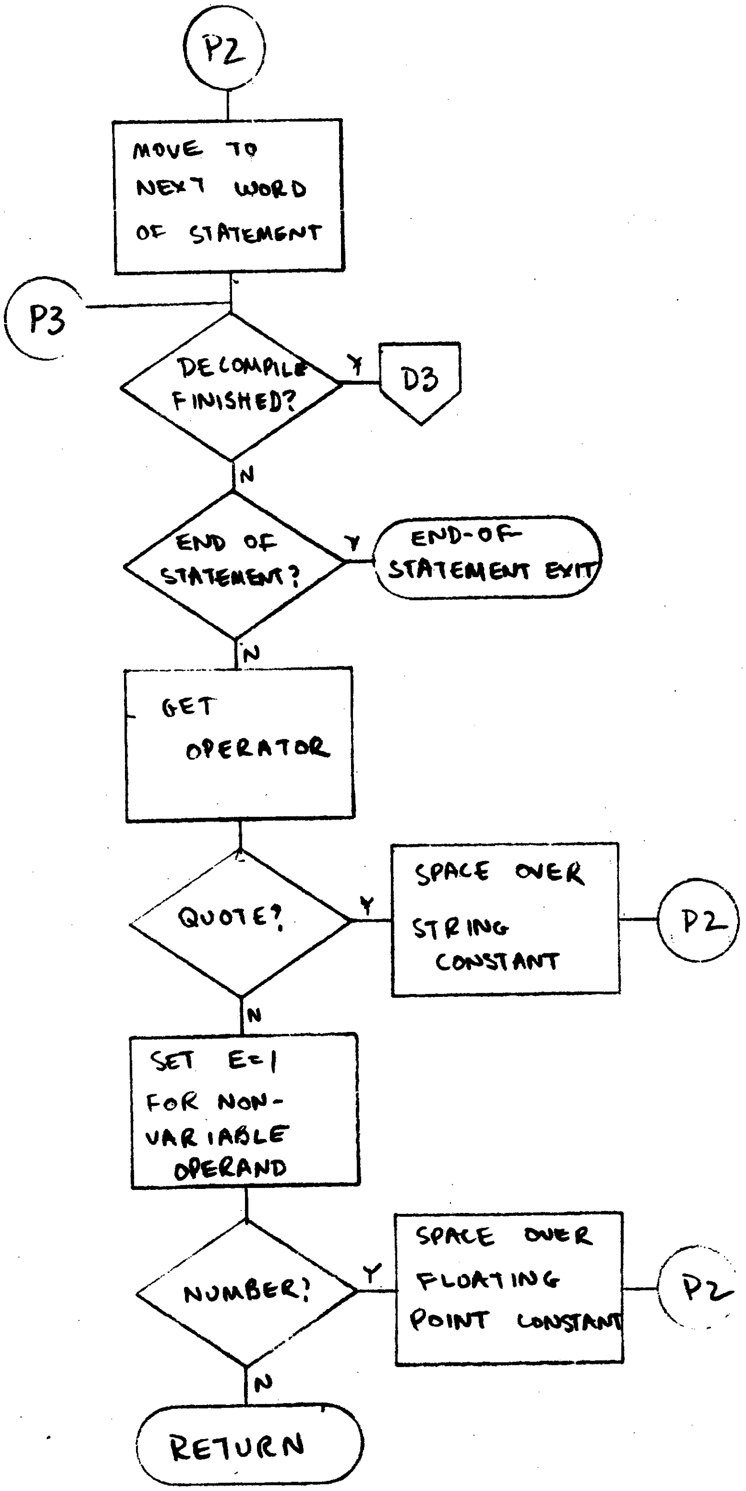
DECOMPILATION


DCMPL
COMPILED?
N
Y
SET CFLAG BIT ±0 IN ?FLAG WORD
NULL PROGRAM
Y
N
SPTR ← FILTB← POINTER TO END OF PROGRAM
PRNST
INITIALIZE PROCESS NEXT STATEMENT
PRNST EXITS HERE WHEN REENTERED DIRECTLY
VARIABLE?
Y
N
PROGRAM INTEGER?
N
F2
Y
MOVE TO NEXT WORD OF STATEMENT
END OF PROGRAM ?
Y
D3
N
END OF STATEMENT ?
Y
P1
N
REPLACE ABSOLUTE ADDRESS WITH STATEMENT NUMBER
USING STATEMENT?
N
Y
P2
D3
PBPTR ← POINTER TO END OF PROGRAM
SYMTB ← 0 (PROGRAM UNCOMPILED)
Y
ANY COMMON?
N
Y
MOVE PROGRAM BACK OVER COMMON AREA
ABCHK
CHECK FOR ABORTS
RETURN
REPLACE POINTER TO SYMBOL TABLE WITH VARIABLE NAME
COM OR DIM?
N
P2
Y
MOVE TO NEXT WORD OF STATEMENT
END OF PROGRAM
Y
D3
N
SIMPLE VARIABLE?
Y
P2
N
MOVE PAST DIMENSIONS
P3

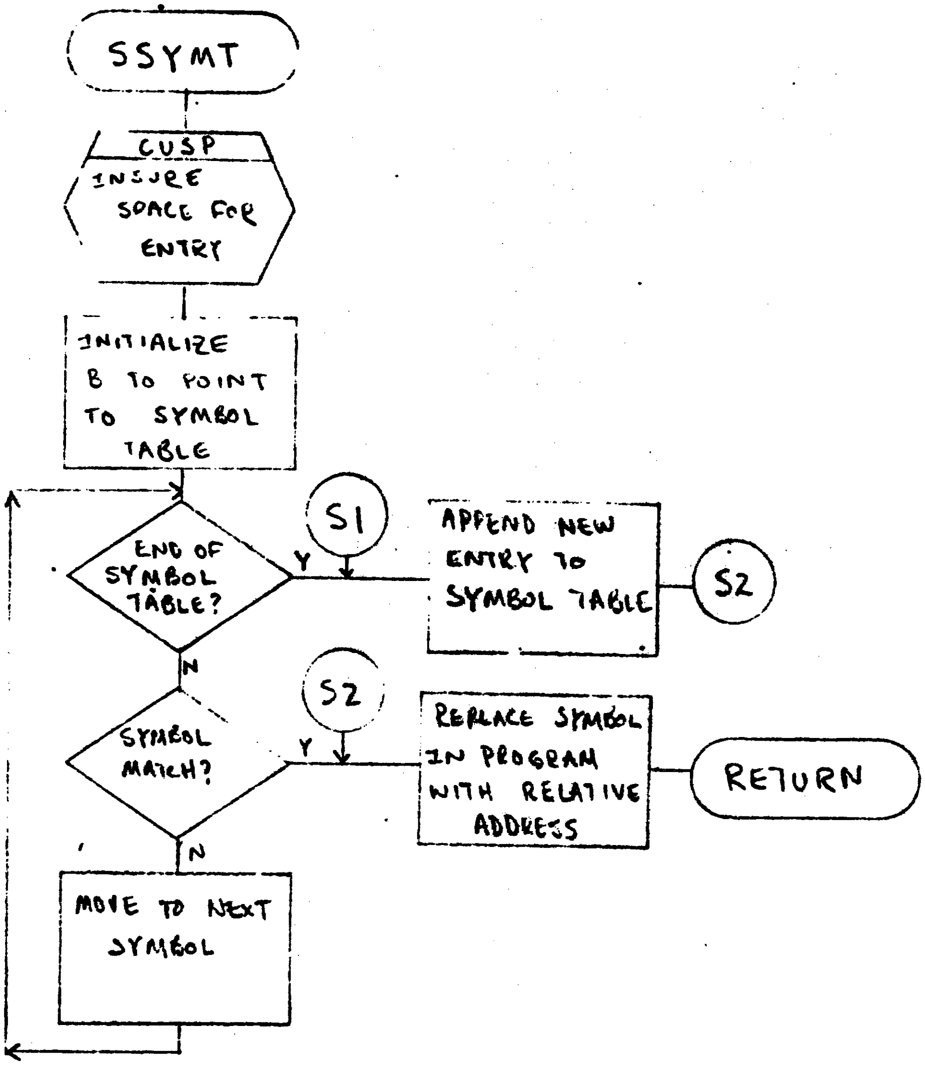
PRNST (1 of 2)


PRNST
SET EXIT FOR END-OF-STATEMENT
FILCT ← -5
COMSN ← USESN ← 0
FILPT ← DFILT
B ← SPROG
P1
PRNST EXITS TO HERE ON THE END-OF STATEMENT CONDITION DURING DCMPL
LNUM ← STATEMENT #
NSPTR ⇒ NEXT STATEMENT
A ← OPERATOR
COM?
Y
NON-COM STATEMENT SEEN?
DCMPL
DECOMPILE PROGRAM
ERROR EXIT
N
COMSN ≠ 0 TO SAY NON-COM STATEMENT SEEN
REM, DATA OR IMAGE?
Y
P3
N
FILES?
Y
TOO MANY FILES STATEMENTS ?
Y
N
SAVE POINTER TO FILES STATEMENT
SKIP OVER REST OF STATEMENT
P3
N
MAT?
N
PRINT?
Y
NEXT OPERATOR USING?
Y
INFST ← -1 (USING STATEMENT USESN ← -1 (USING SEEN)
P2
Y
GET NEXT OPERATOR
N
N
INFST ≠ -1 (NOT USING STATEMENT)
P2

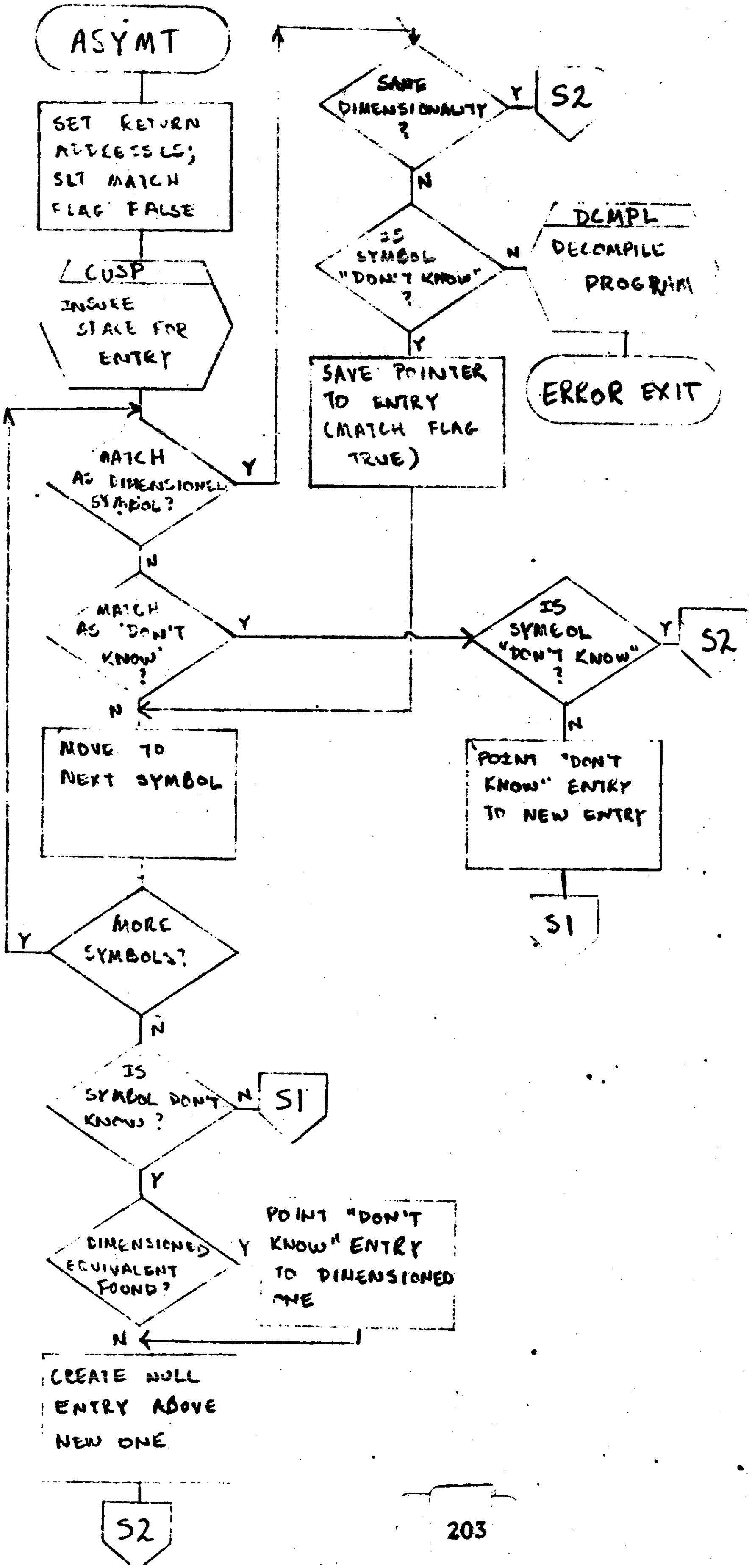
PRNST (2 of 2)


P2
MOVE TO NEXT WORD OF STATEMENT
P3
DECOMPILE FINISHED?
Y
D3
N
END OF STATEMENT?
Y
END-OF-STATEMENT EXIT
N
GET OPERATOR
QUOTE?
Y
SPACE OVER STRING CONSTANT
P2
N
SET E=1 FOR NON-VARIABLE OPERAND
NUMBER?
Y
SPACE OVER FLOATING POINT CONSTANT
P2
N
RETURN

SSYMT

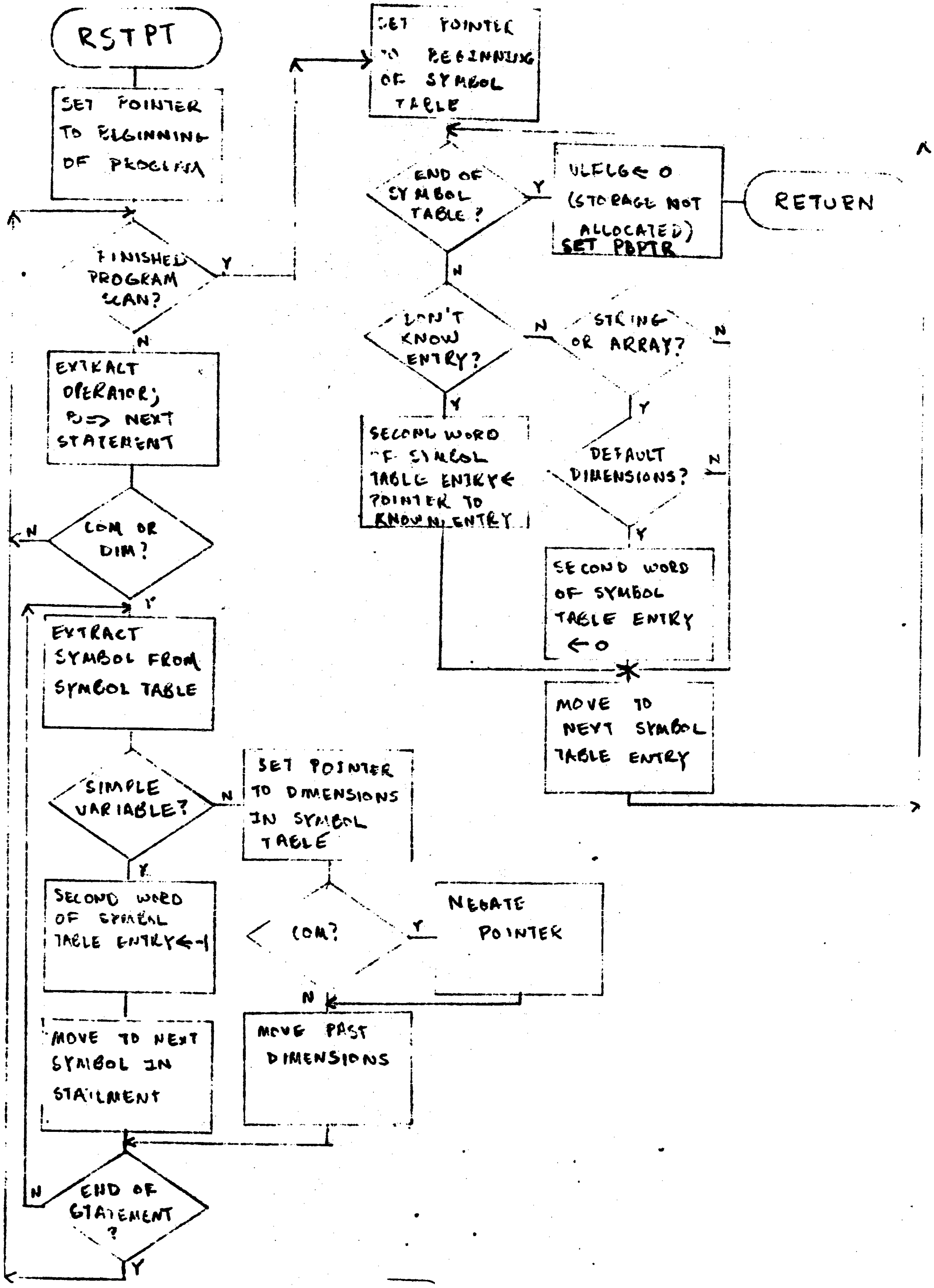

SSYMT
CUSP
INSURE SPACE FOR ENTRY
INITIALIZE B TO POINT TO SYMBOL TABLE
END OF SYMBOL TABLE?
Y
N
S1
APPEND NEW ENTRY TO SYMBOL TABLE
S2
SYMBOL MATCH?
Y
N
S2
REPLACE SYMBOL IN PROGRAM WITH RELATIVE ADDRESS
RETURN
MOVE TO NEXT SYMBOL

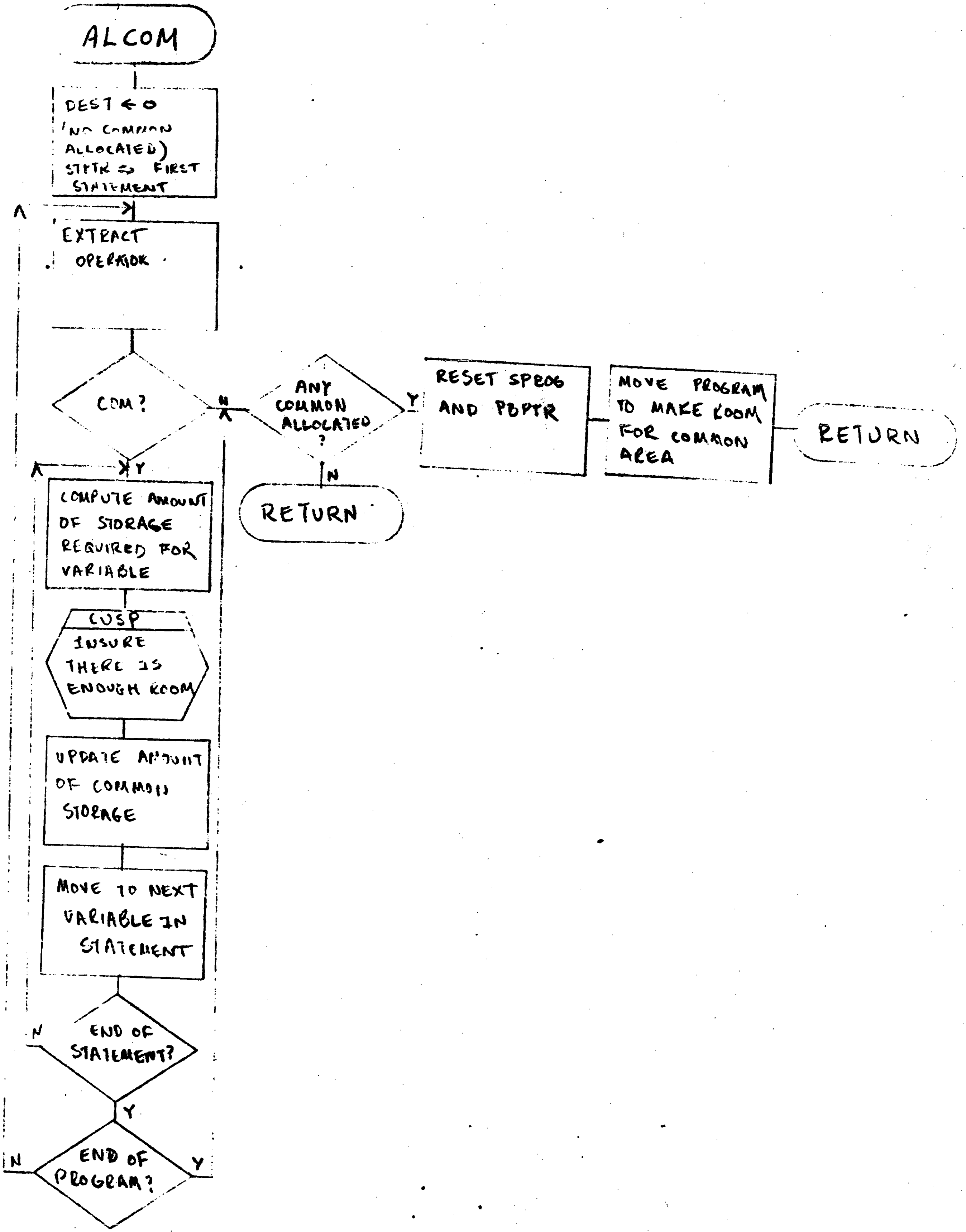
ASYMT



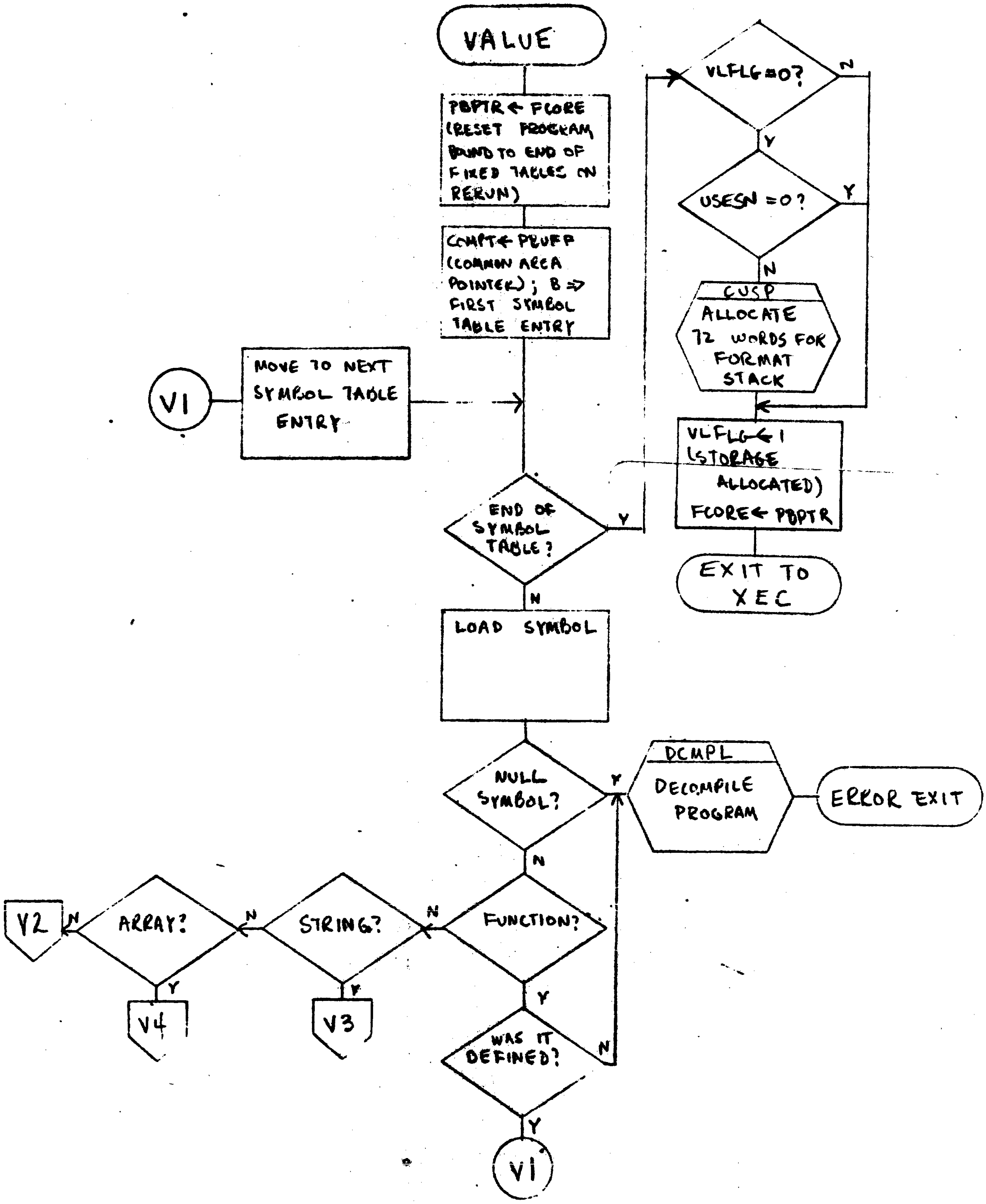
ASYMT
SET RETURN ADDRESSES; SET MATCH FLAG FALSE
CUSP
INSURE SPACE FOR ENTRY
MATCH AS DIMENSIONED SYMBOL?
Y
N
MATCH AS "DON'T KNOW"?
Y
N
MOVE TO NEXT SYMBOL
MORE SYMBOLS?
Y
N
IS SYMBOL DON'T KNOW?
N
S1
Y
DIMENSIONED EQUIVALENT FOUND?
Y
POINT "DON'T KNOW" ENTRY TO DIMENSIONED ONE
N
CREATE NULL ENTRY ABOVE NEW ONE
S2
SAME DIMENSIONALITY?
Y
S2
N
IS SYMBOL "DON'T KNOW"?
N
DCMPL
DECOMPILE PROGRAM
ERROR EXIT
Y
SAVE POINTER TO ENTRY (MATCH FLAG TRUE)
IS SYMBOL "DON'T KNOW"?
Y
S2
N
POINT "DON'T KNOW" ENTRY TO NEW ENTRY
S1

RSTPT

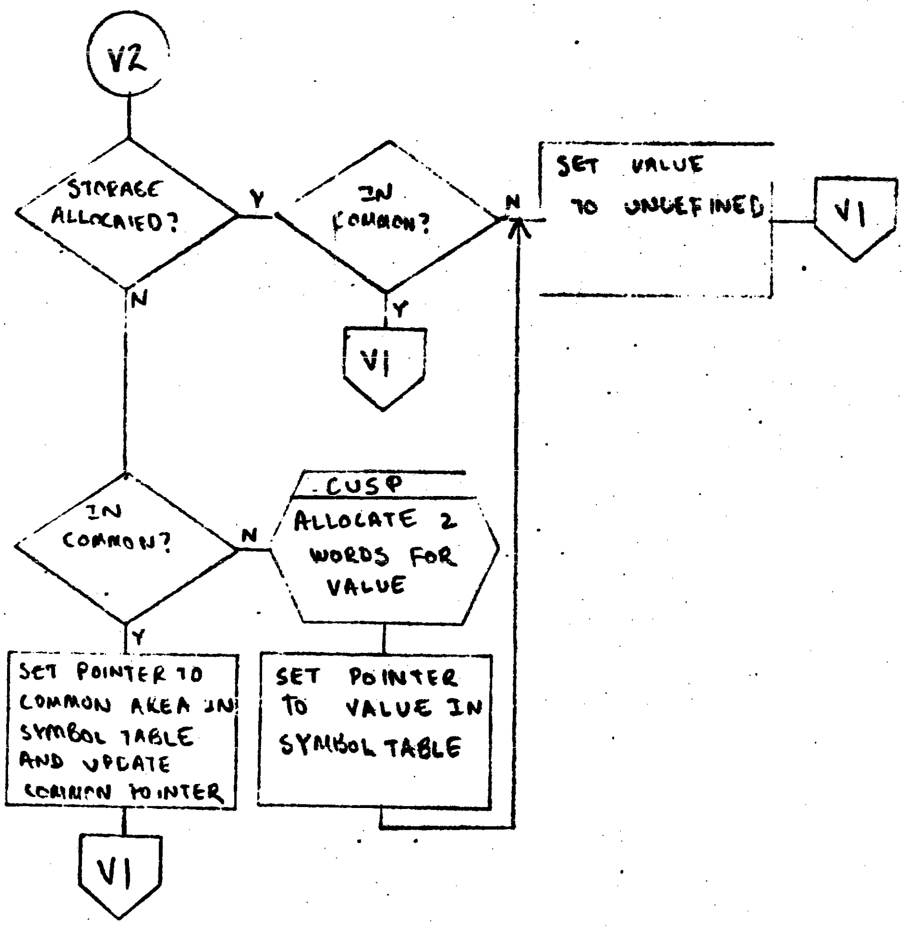

RSTPT
SET POINTER TO BEGINNING OF PROGRAM
FINISHED PROGRAM SCAN?
Y
N
EXTRACT OPERATOR; P=> NEXT STATEMENT
COM OR DIM?
N
Y
EXTRACT SYMBOL FROM SYMBOL TABLE
SIMPLE VARIABLE?
N
Y
SECOND WORD OF SYMBOL TABLE ENTRY←-1
MOVE TO NEXT SYMBOL IN STATEMENT
SET POINTER TO DIMENSIONS IN SYMBOL TABLE
COM?
Y
NEGATE POINTER
N
MOVE PAST DIMENSIONS
END OF STATEMENT ?
N
Y
SET POINTER TO BEGINNING OF SYMBOL TABLE
END OF SYMBOL TABLE ?
Y
ULFLG← 0 (STORAGE NOT ALLOCATED) SET PBPTR
RETURN
N
DON'T KNOW ENTRY?
N
Y
SECOND WORD OF SYMBOL TABLE ENTRY← POINTER TO KNOWN ENTRY
STRING OR ARRAY?
N
Y
DEFAULT DIMENSIONS?
N
Y
SECOND WORD OF SYMBOL TABLE ENTRY ←0
MOVE TO NEXT SYMBOL TABLE ENTRY


ALCOM
DEST ← 0 (NO COMMON ALLOCATED) STPTR ⇒ FIRST STATEMENT
EXTRACT OPERATOR
COM?
N
Y
ANY COMMON ALLOCATED ?
Y
N
RESET SPROG AND PBPTR
MOVE PROGRAM TO MAKE ROOM FOR COMMON AREA
RETURN
RETURN
COMPUTE AMOUNT OF STORAGE REQUIRED FOR VARIABLE
CUSP
INSURE THERE IS ENOUGH ROOM
UPDATE AMOUNT OF COMMON STORAGE
MOVE TO NEXT VARIABLE IN STATEMENT
END OF STATEMENT?
N
Y
END OF PROGRAM?
N
Y

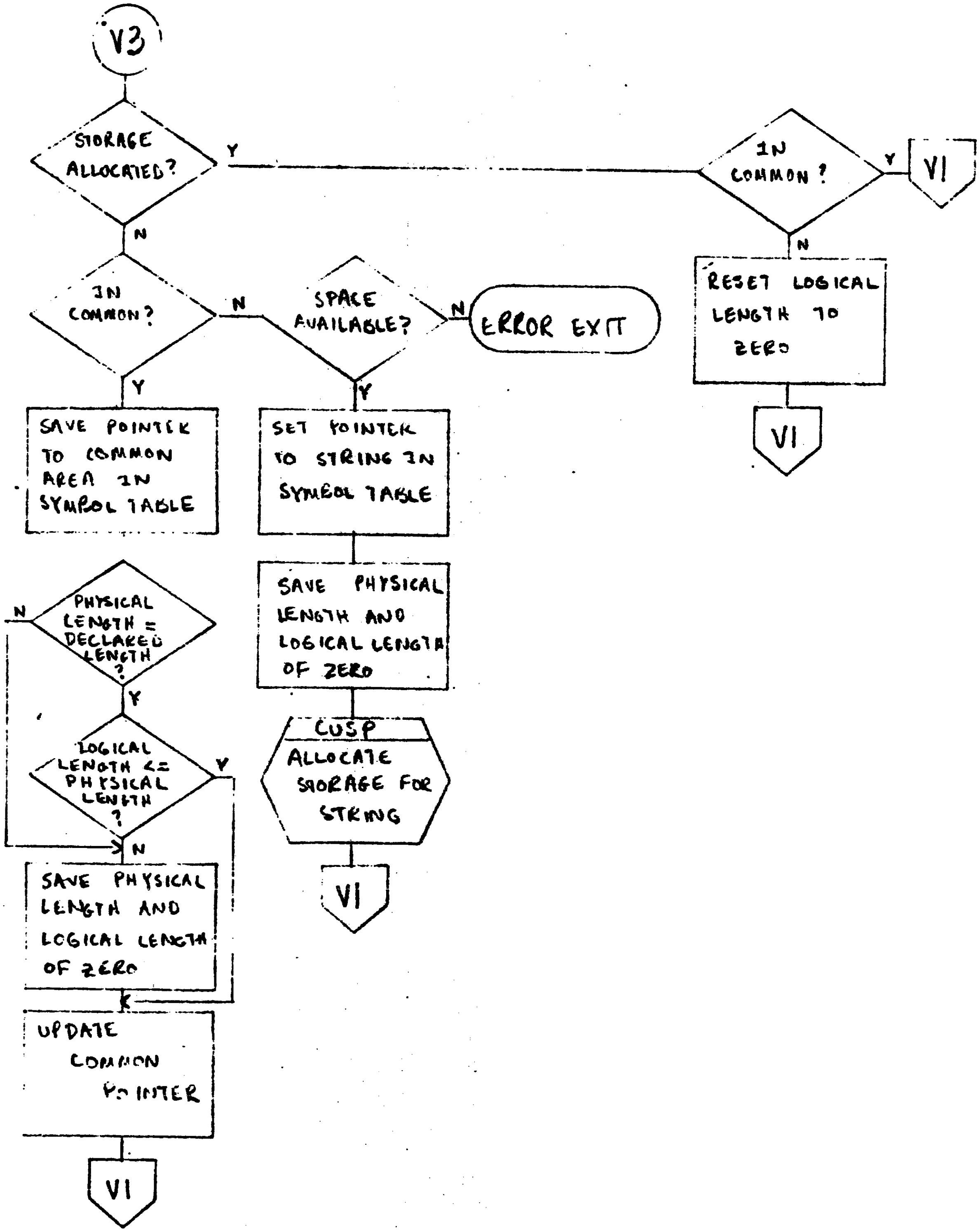
VALUE (1 of 5)

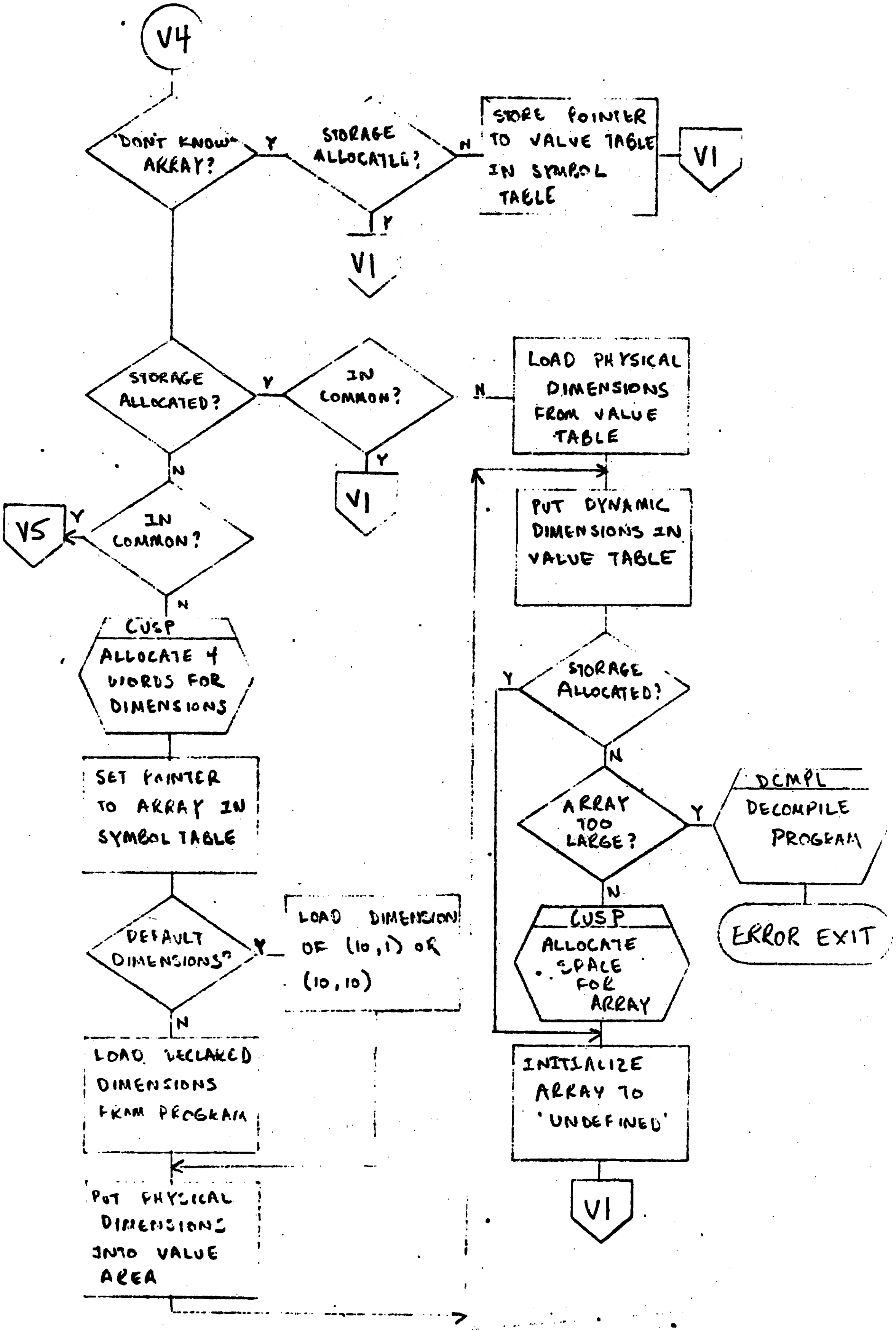

VALUE
PBPTR ← FCORE (RESET PROGRAM BOUND TO END OF FIXED TABLES ON RERUN)
COMPT ← PBUFF (COMMON AREA POINTER); B ⇒ FIRST SYMBOL TABLE ENTRY
V1
MOVE TO NEXT SYMBOL TABLE ENTRY
END OF SYMBOL TABLE?
Y
N
VLFLG=0?
N
Y
USESN =0?
Y
N
CUSP
ALLOCATE 72 WORDS FOR FORMAT STACK
VLFLG ← 1 (STORAGE ALLOCATED)
FCORE ← PBPTR
EXIT TO XEC
LOAD SYMBOL
NULL SYMBOL?
Y
N
DCMPL
DECOMPILE PROGRAM
ERROR EXIT
FUNCTION?
N
Y
STRING?
N
Y
ARRAY?
N
Y
V2
V3
V4
WAS IT DEFINED?
N
Y
V1

VALUE (2 of 5)

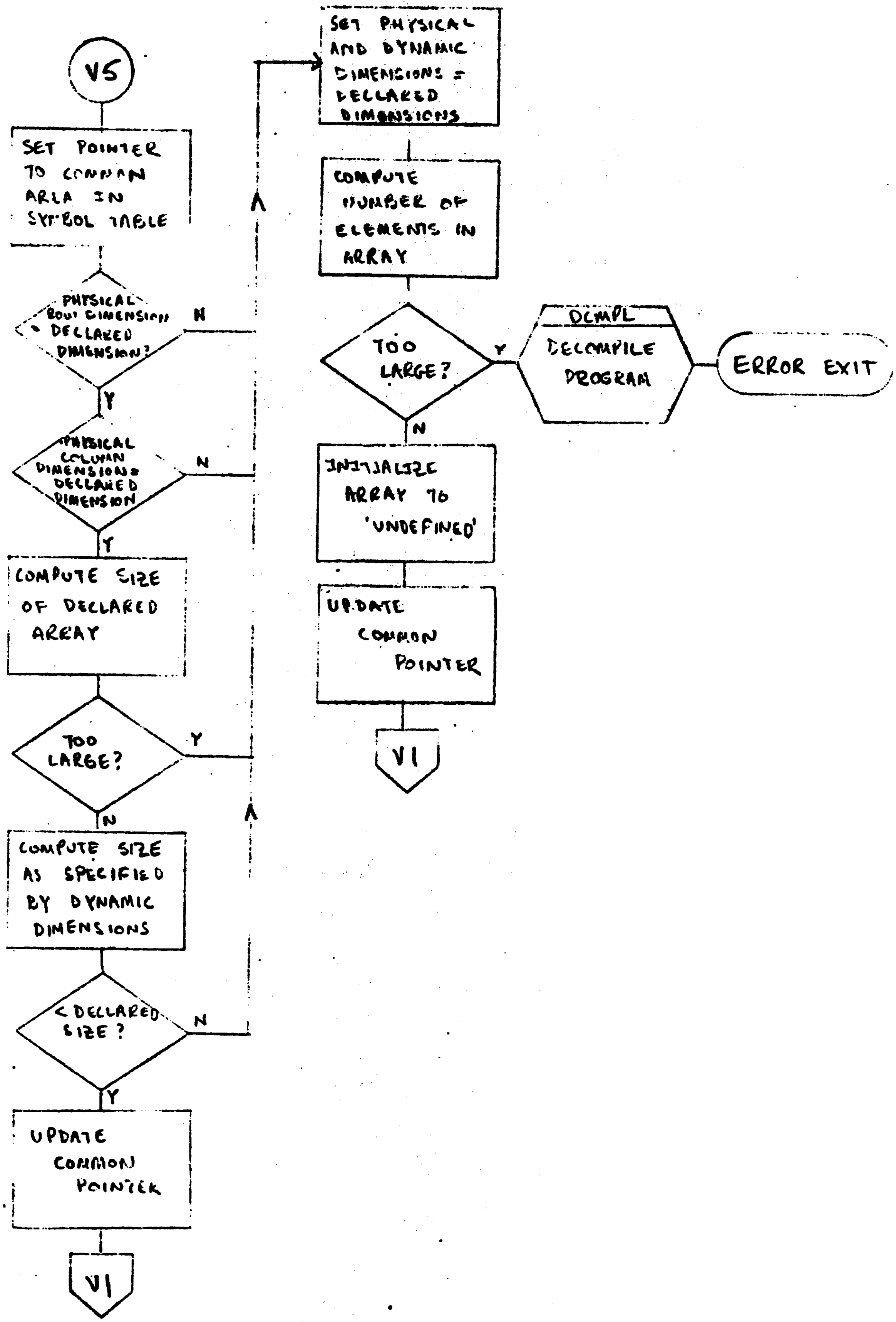

V2
STORAGE ALLOCATED?
Y
N
IN COMMON?
N
Y
V1
SET VALUE TO UNDEFINED
V1
IN COMMON?
N
Y
CUSP
ALLOCATE 2 WORDS FOR VALUE
SET POINTER TO COMMON AREA IN SYMBOL TABLE AND UPDATE COMMON POINTER
SET POINTER TO VALUE IN SYMBOL TABLE
V1

VALUE (3 of 5)

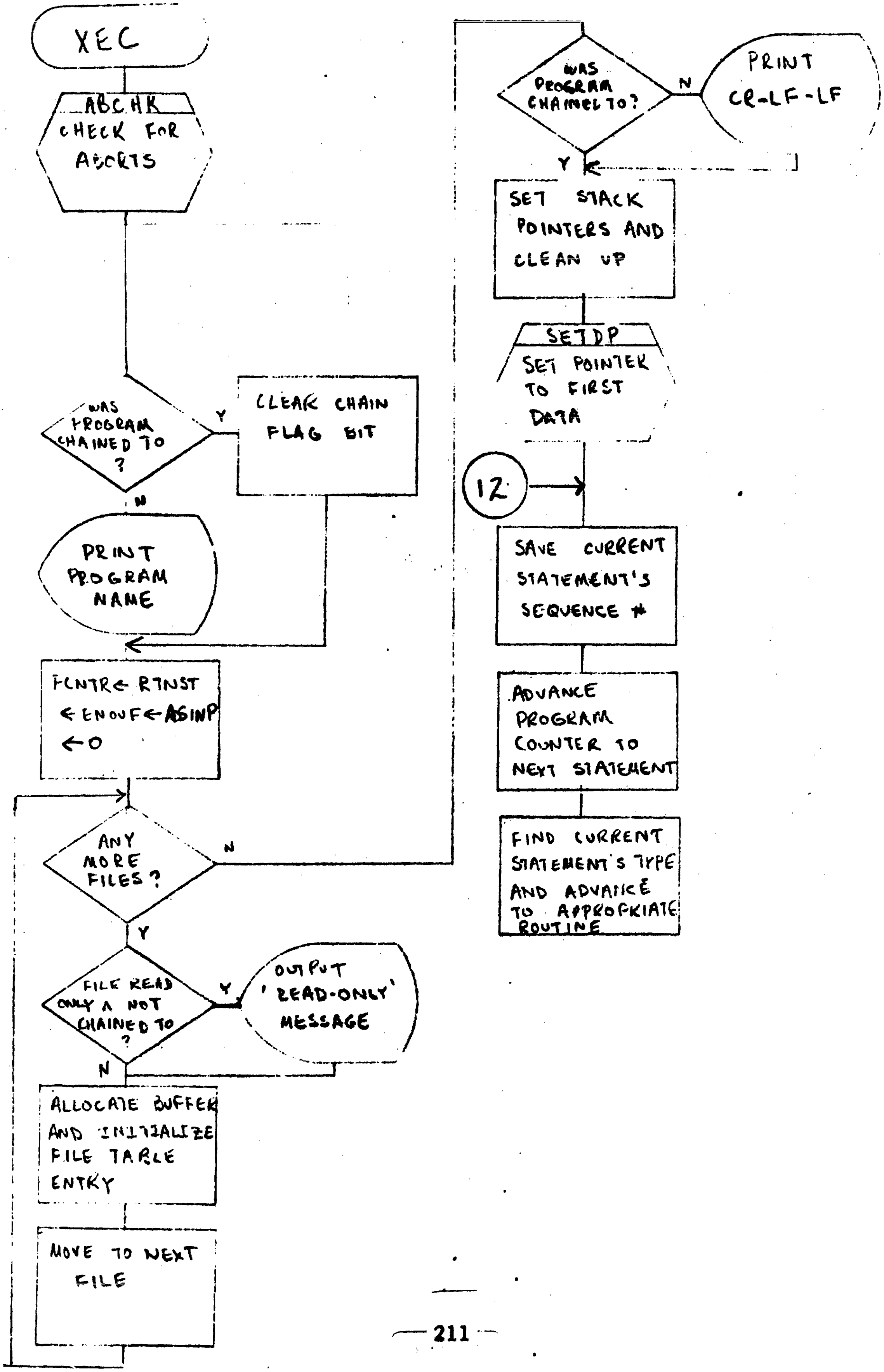

V3
STORAGE ALLOCATED?
Y
N
IN COMMON ?
Y
VI
N
RESET LOGICAL LENGTH TO ZERO
VI
IN COMMON?
N
SPACE AVAILABLE?
N
ERROR EXIT
Y
Y
SAVE POINTER TO COMMON AREA IN SYMBOL TABLE
SET POINTER TO STRING IN SYMBOL TABLE
SAVE PHYSICAL LENGTH AND LOGICAL LENGTH OF ZERO
CUSP
ALLOCATE STORAGE FOR STRING
VI
N
PHYSICAL LENGTH = DECLARED LENGTH ?
Y
LOGICAL LENGTH <= PHYSICAL LENGTH ?
Y
N
SAVE PHYSICAL LENGTH AND LOGICAL LENGTH OF ZERO
UPDATE COMMON POINTER
VI


V4
"DON'T KNOW" ARRAY?
Y
STORAGE ALLOCATED?
N
STORE POINTER TO VALUE TABLE IN SYMBOL TABLE
V1
Y
V1
STORAGE ALLOCATED?
Y
IN COMMON?
N
LOAD PHYSICAL DIMENSIONS FROM VALUE TABLE
Y
V1
N
V5
Y
IN COMMON?
N
PUT DYNAMIC DIMENSIONS IN VALUE TABLE
CUSP
ALLOCATE 4 WORDS FOR DIMENSIONS
Y
STORAGE ALLOCATED?
N
SET POINTER TO ARRAY IN SYMBOL TABLE
ARRAY TOO LARGE?
Y
DCMPL
DECOMPILE PROGRAM
N
ERROR EXIT
CUSP
ALLOCATE SPACE FOR ARRAY
DEFAULT DIMENSIONS?
Y
LOAD DIMENSION OF (10,1) OR (10,10)
N
LOAD DECLARED DIMENSIONS FROM PROGRAM
INITIALIZE ARRAY TO 'UNDEFINED'
V1
PUT PHYSICAL DIMENSIONS INTO VALUE AREA

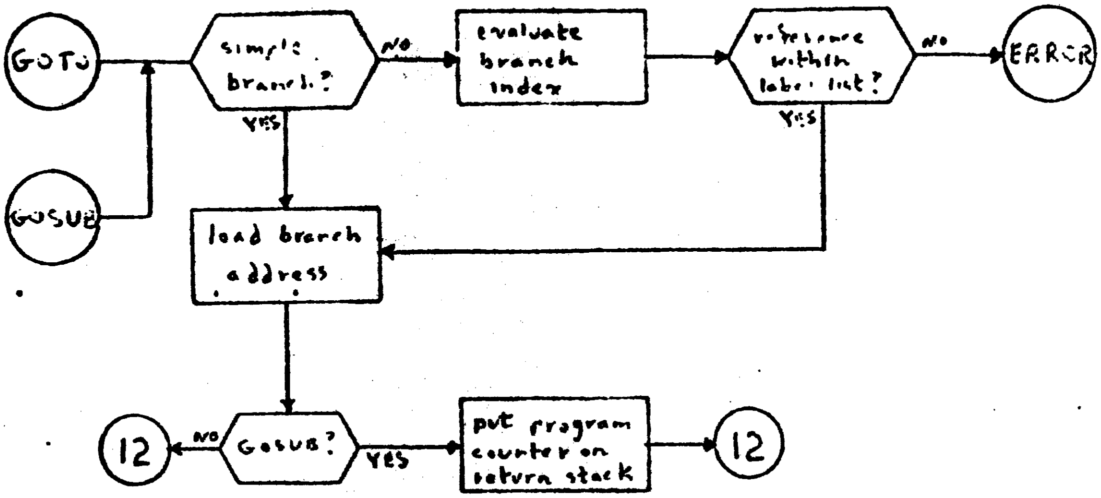
VALUE (5 of 5)

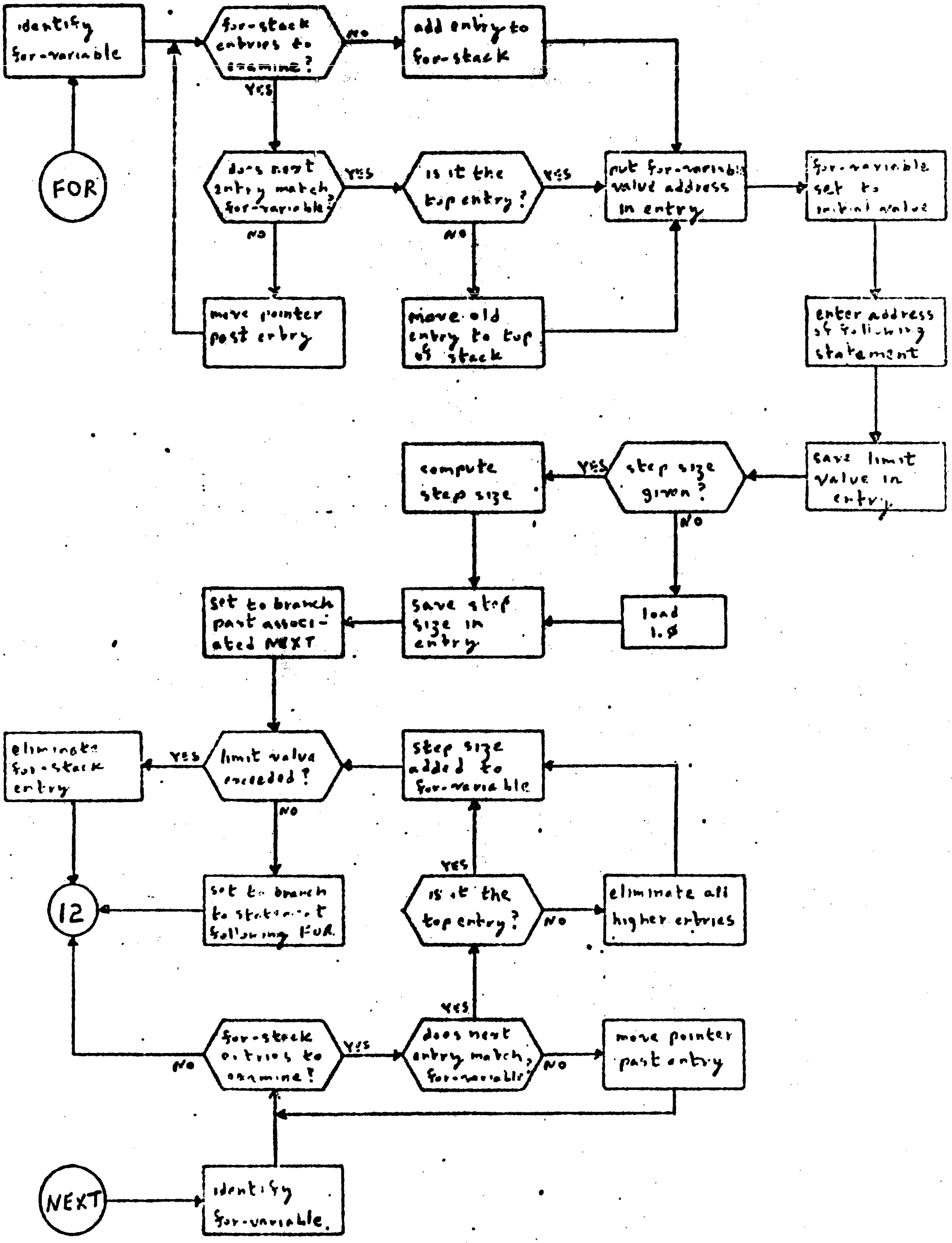

VS
SET POINTER TO COMMON AREA IN SYMBOL TABLE
PHYSICAL ROW DIMENSION = DECLARED DIMENSION?
N
Y
PHYSICAL COLUMN DIMENSION = DECLARED DIMENSION
N
Y
COMPUTE SIZE OF DECLARED ARRAY
TOO LARGE?
Y
N
COMPUTE SIZE AS SPECIFIED BY DYNAMIC DIMENSIONS
< DECLARED SIZE?
N
Y
UPDATE COMMON POINTER
VI
SET PHYSICAL AND DYNAMIC DIMENSIONS = DECLARED DIMENSIONS
COMPUTE NUMBER OF ELEMENTS IN ARRAY
TOO LARGE?
Y
DCMPL
DECOMPILE PROGRAM
ERROR EXIT
N
INITIALIZE ARRAY TO 'UNDEFINED'
UPDATE COMMON POINTER
VI

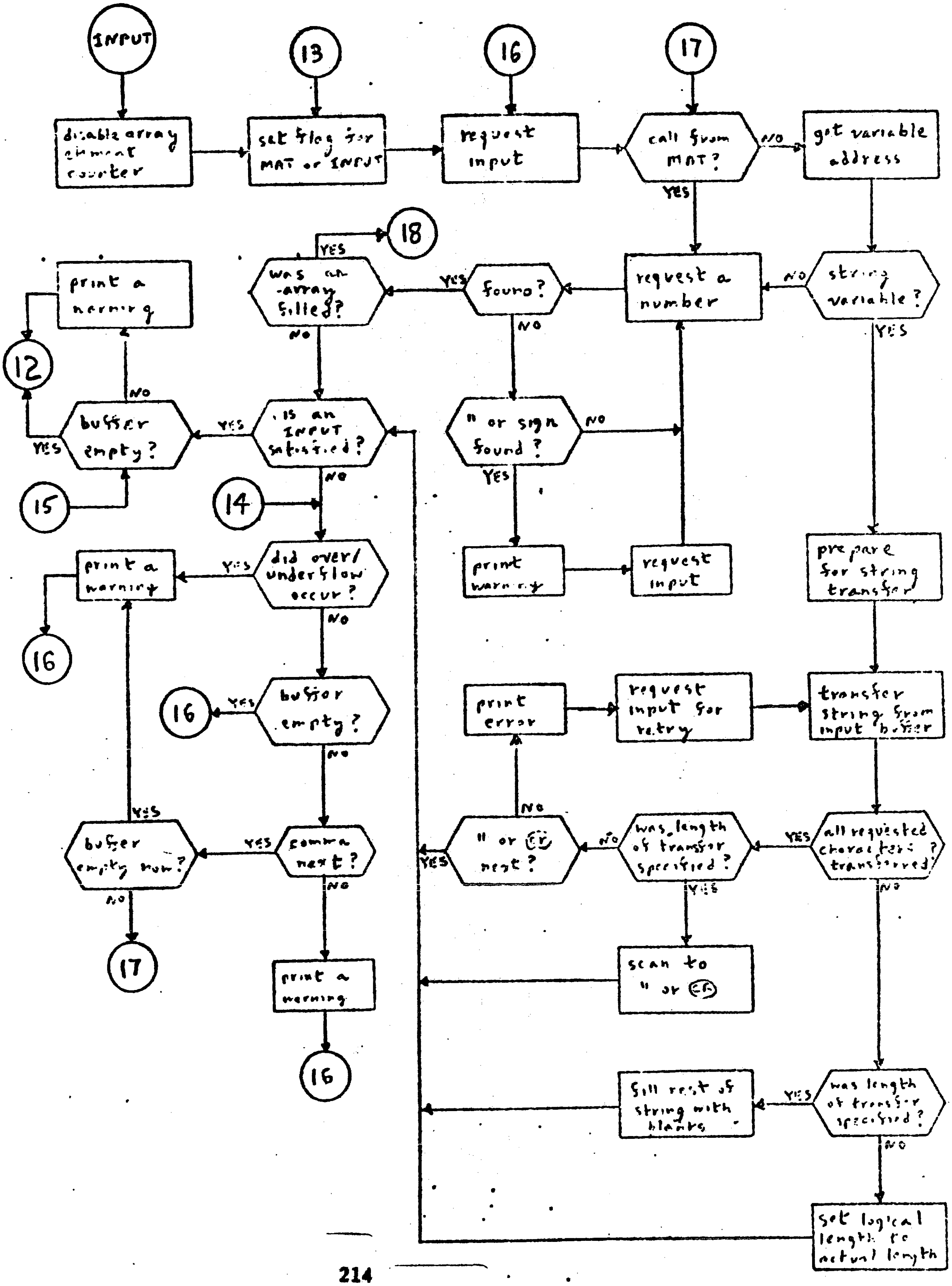
A. Main Loop

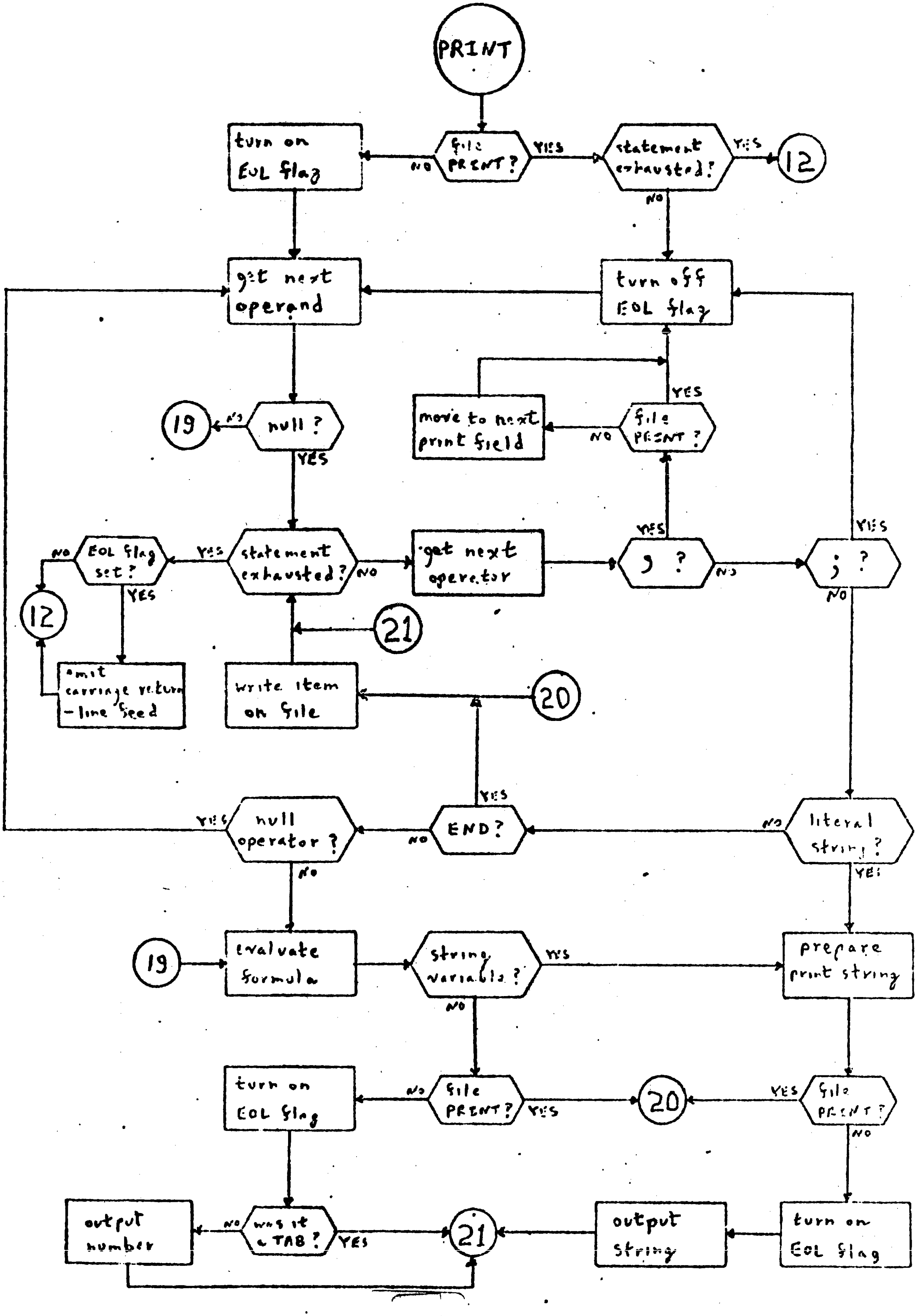

XEC
ABCHK
CHECK FOR ABORTS
WAS PROGRAM CHAINED TO ?
Y
CLEAR CHAIN FLAG BIT
N
PRINT PROGRAM NAME
FCNTR← RTNST ← ENOUF ← ASINP ← 0
ANY MORE FILES ?
N
Y
FILE READ ONLY & NOT CHAINED TO ?
Y
OUTPUT 'READ-ONLY' MESSAGE
N
ALLOCATE BUFFER AND INITIALIZE FILE TABLE ENTRY
MOVE TO NEXT FILE
WAS PROGRAM CHAINED TO?
N
PRINT CR-LF-LF
Y
SET STACK POINTERS AND CLEAN UP
SETDP
SET POINTER TO FIRST DATA
12
SAVE CURRENT STATEMENT'S SEQUENCE #
ADVANCE PROGRAM COUNTER TO NEXT STATEMENT
FIND CURRENT STATEMENT'S TYPE AND ADVANCE TO APPROPRIATE ROUTINE

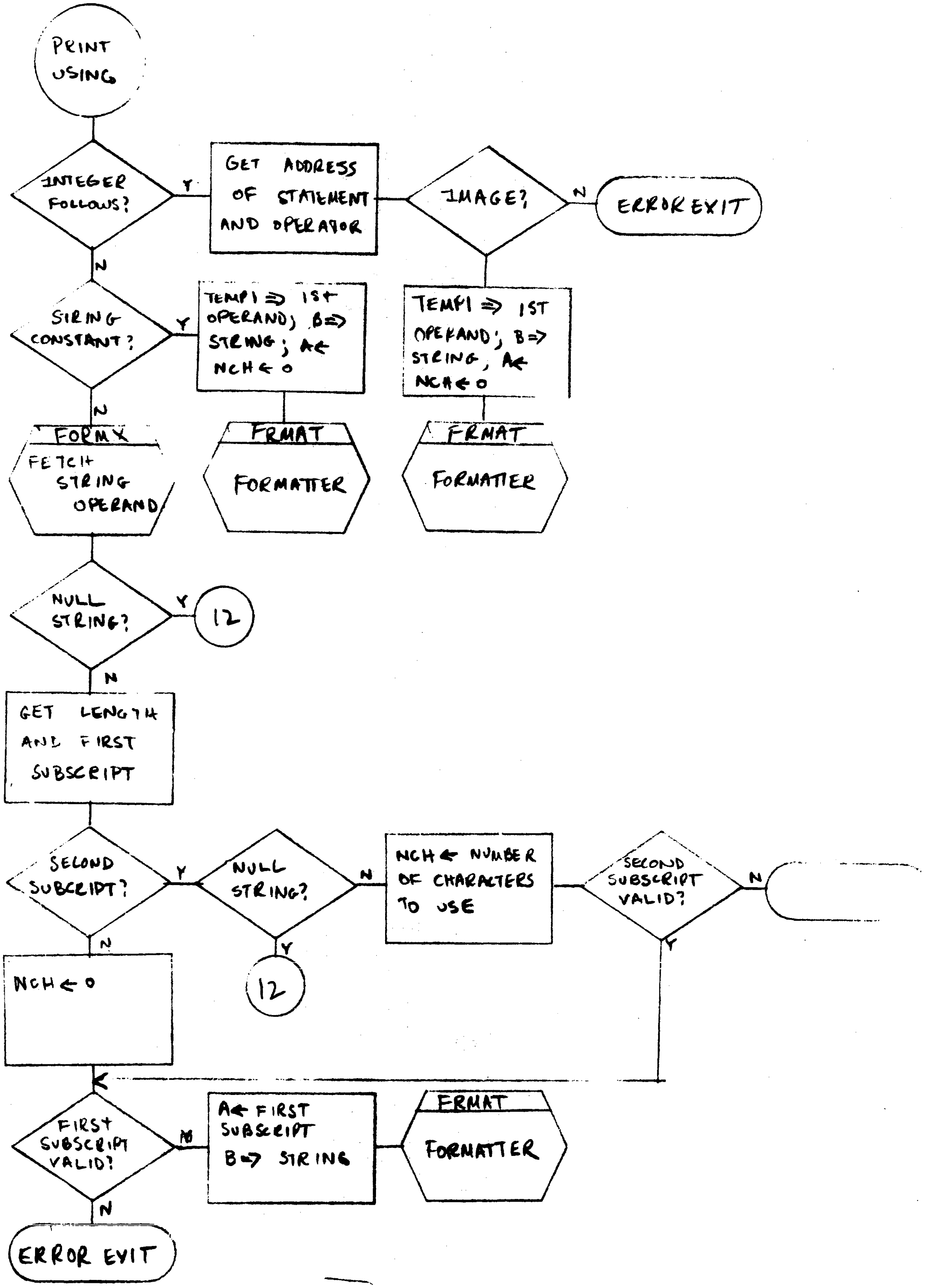
B. Selected Statement Types

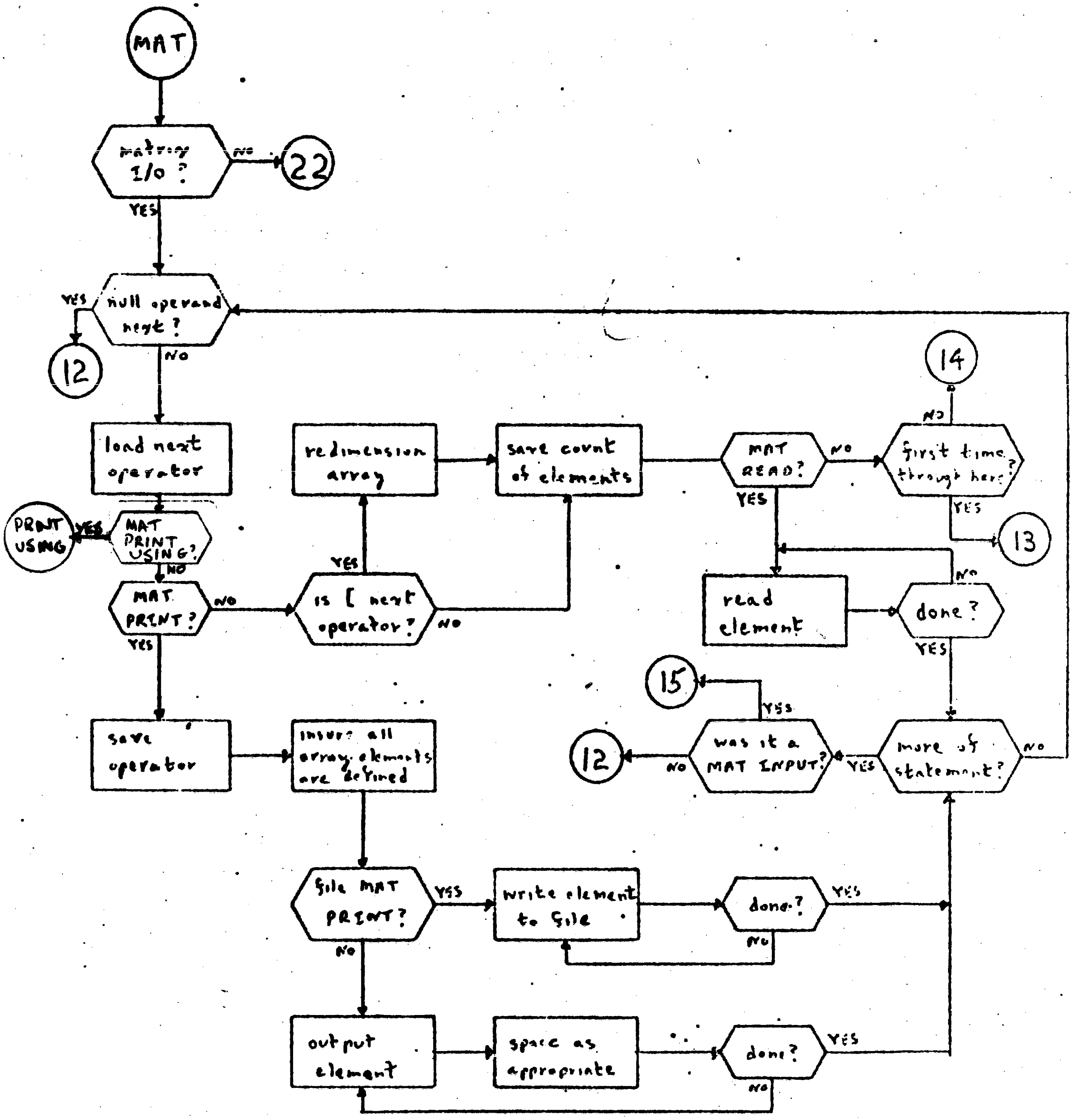


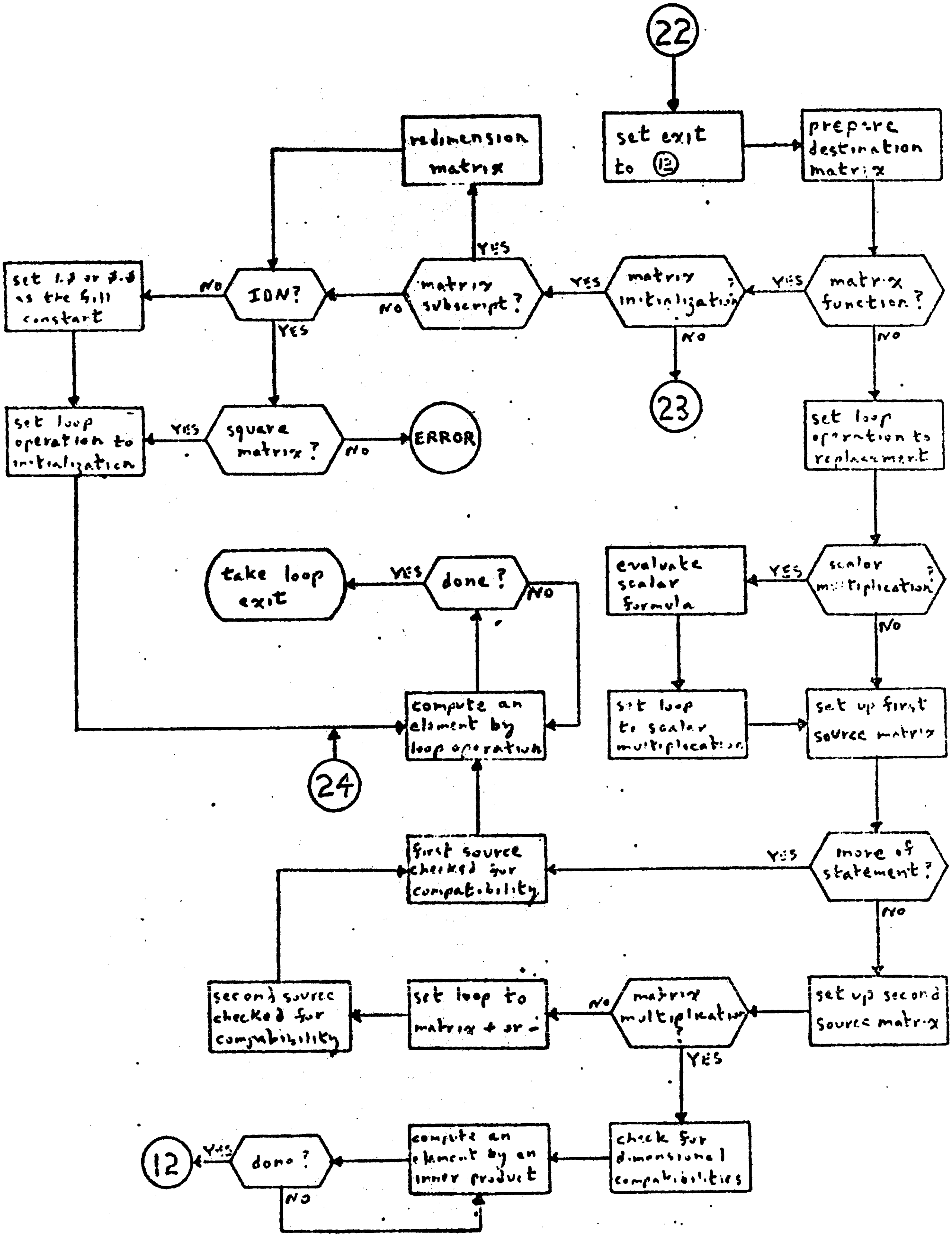

identify for-variable
FOR
for-stack entries to examine?
NO
YES
add entry to for-stack
does next entry match for-variable?
YES
is it the top entry?
YES
put for-variable value address in entry
for-variable set to initial value
NO
NO
move pointer past entry
move old entry to top of stack
enter address of following statement
save limit value in entry
step size given?
YES
NO
compute step size
load 1.Ø
save step size in entry
set to branch past associated NEXT
limit value exceeded?
YES
NO
eliminate for-stack entry
step size added to for-variable
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.

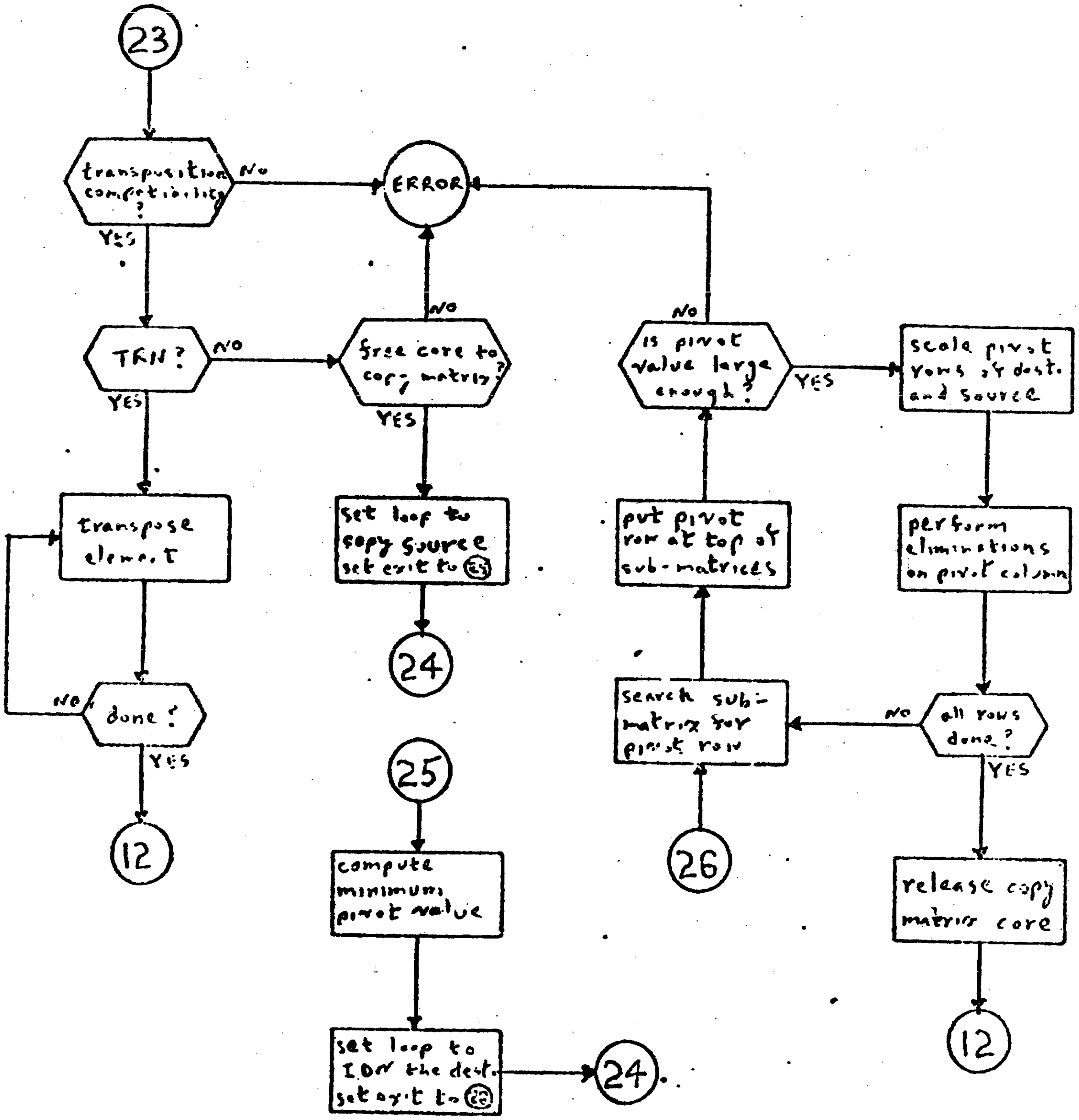

INPUT
13
16
17
disable array element counter
set flag for MAT or INPUT
request input
call from MAT?
NO
YES
get variable address
18
YES
was an array filled?
NO
YES
found?
NO
request a number
NO
string variable?
YES
print a warning
12
NO
buffer empty?
YES
YES
is an INPUT satisfied?
NO
" or sign found?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

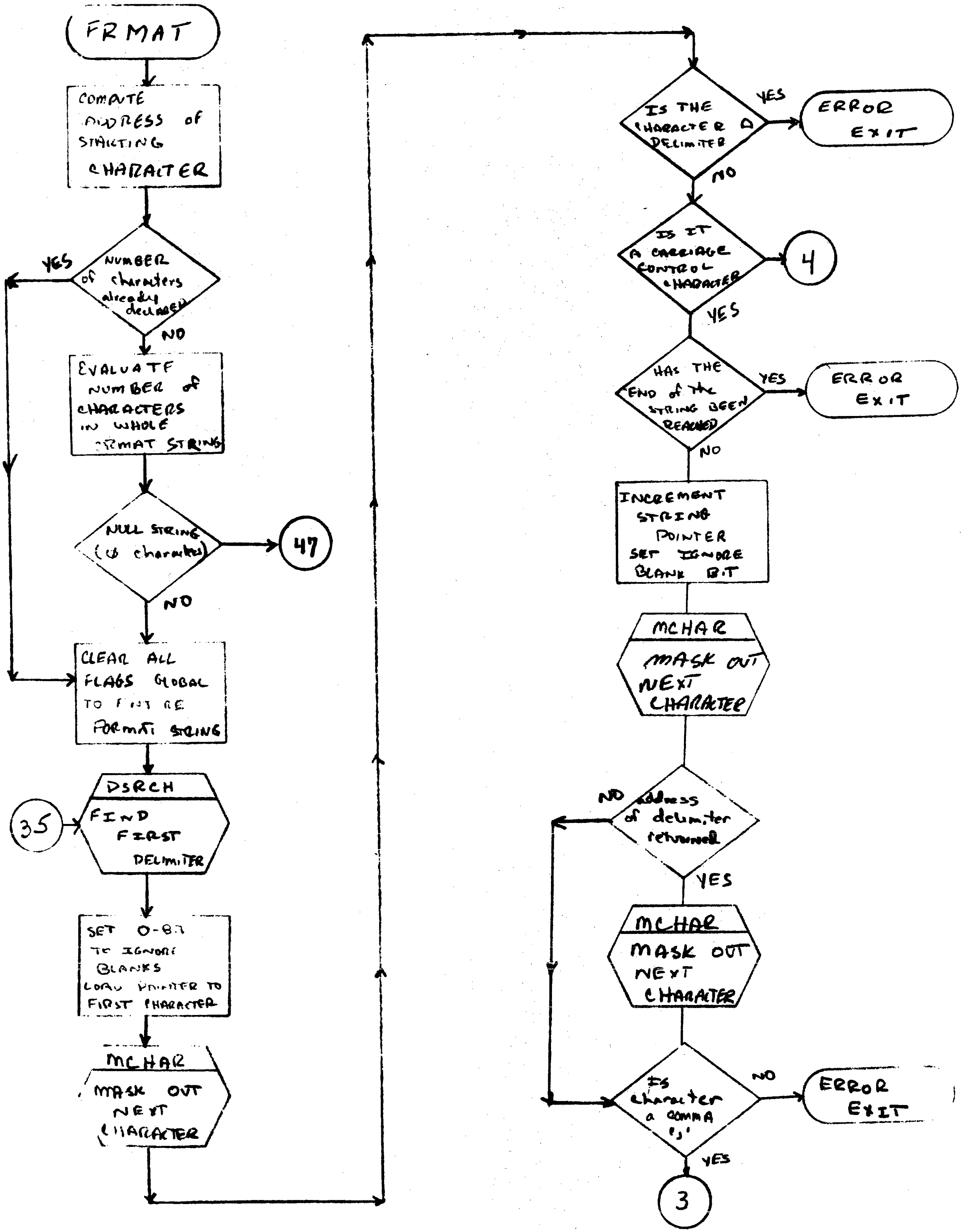

PRINT
file PRINT?
YES
NO
turn on EOL flag
statement exhausted?
YES
12
NO
get next operand
turn off EOL flag
YES
move to next print field
NO
file PRINT?
19
NO
null?
YES
YES
YES
NO
EOL flag set?
YES
statement exhausted?
NO
get next operator
, ?
NO
; ?
NO
YES
12
21
omit carriage return - line feed
write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

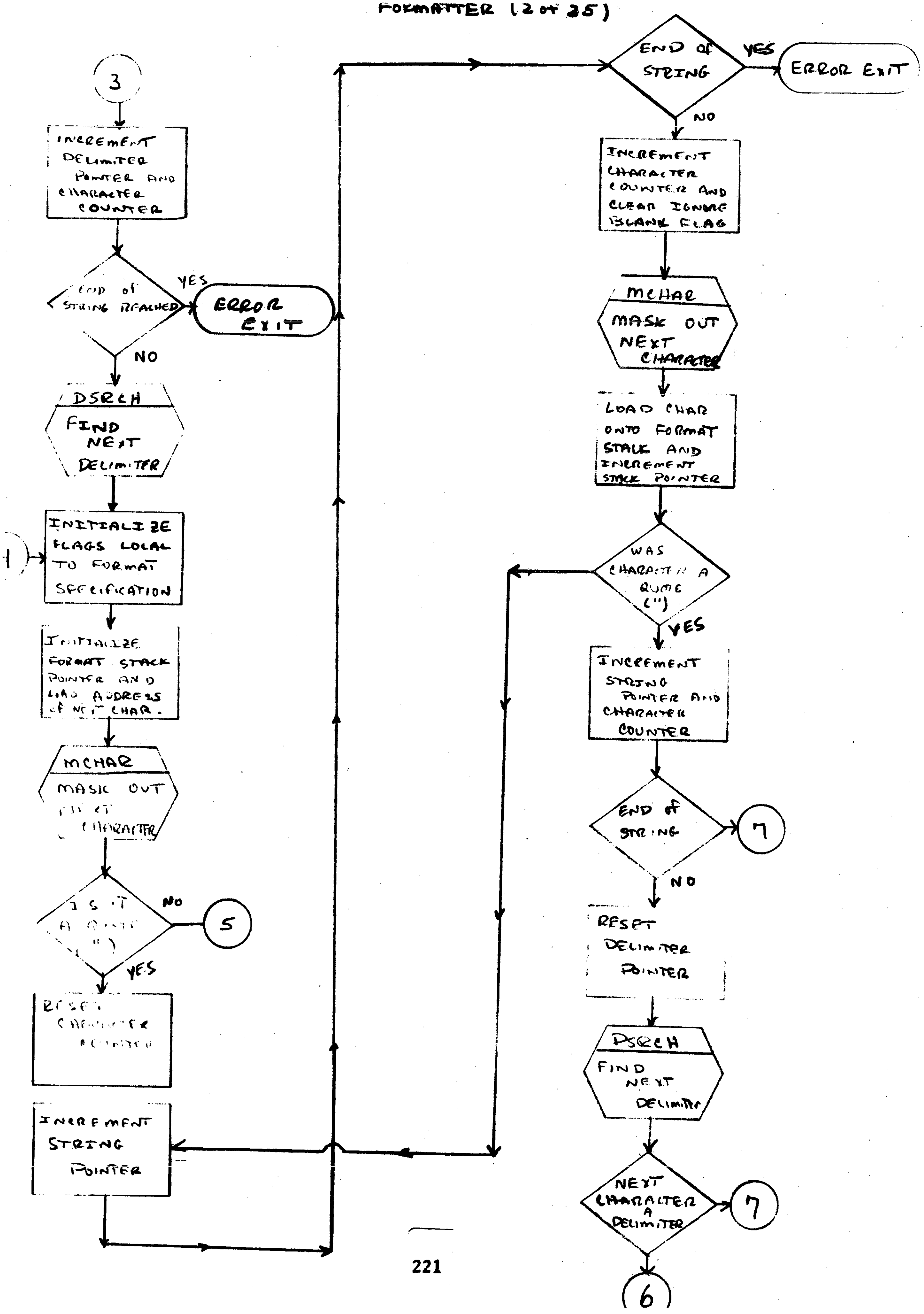

PRINT USING
INTEGER FOLLOWS?
Y
GET ADDRESS OF STATEMENT AND OPERATOR
IMAGE?
N
ERROR EXIT
N
STRING CONSTANT?
Y
TEMP1 ⇒ 1st OPERAND; B⇒ STRING; A← NCH ← 0
TEMP1 ⇒ 1ST OPERAND; B⇒ STRING, A← NCH ← 0
N
FORMX
FETCH STRING OPERAND
FRMAT
FORMATTER
FRMAT
FORMATTER
NULL STRING?
Y
12
N
GET LENGTH AND FIRST SUBSCRIPT
SECOND SUBSCRIPT?
Y
NULL STRING?
N
NCH ← NUMBER OF CHARACTERS TO USE
SECOND SUBSCRIPT VALID?
N
Y
Y
12
N
NCH ← 0
FIRST SUBSCRIPT VALID?
N
A← FIRST SUBSCRIPT B⇒ STRING
FRMAT
FORMATTER
N
ERROR EXIT

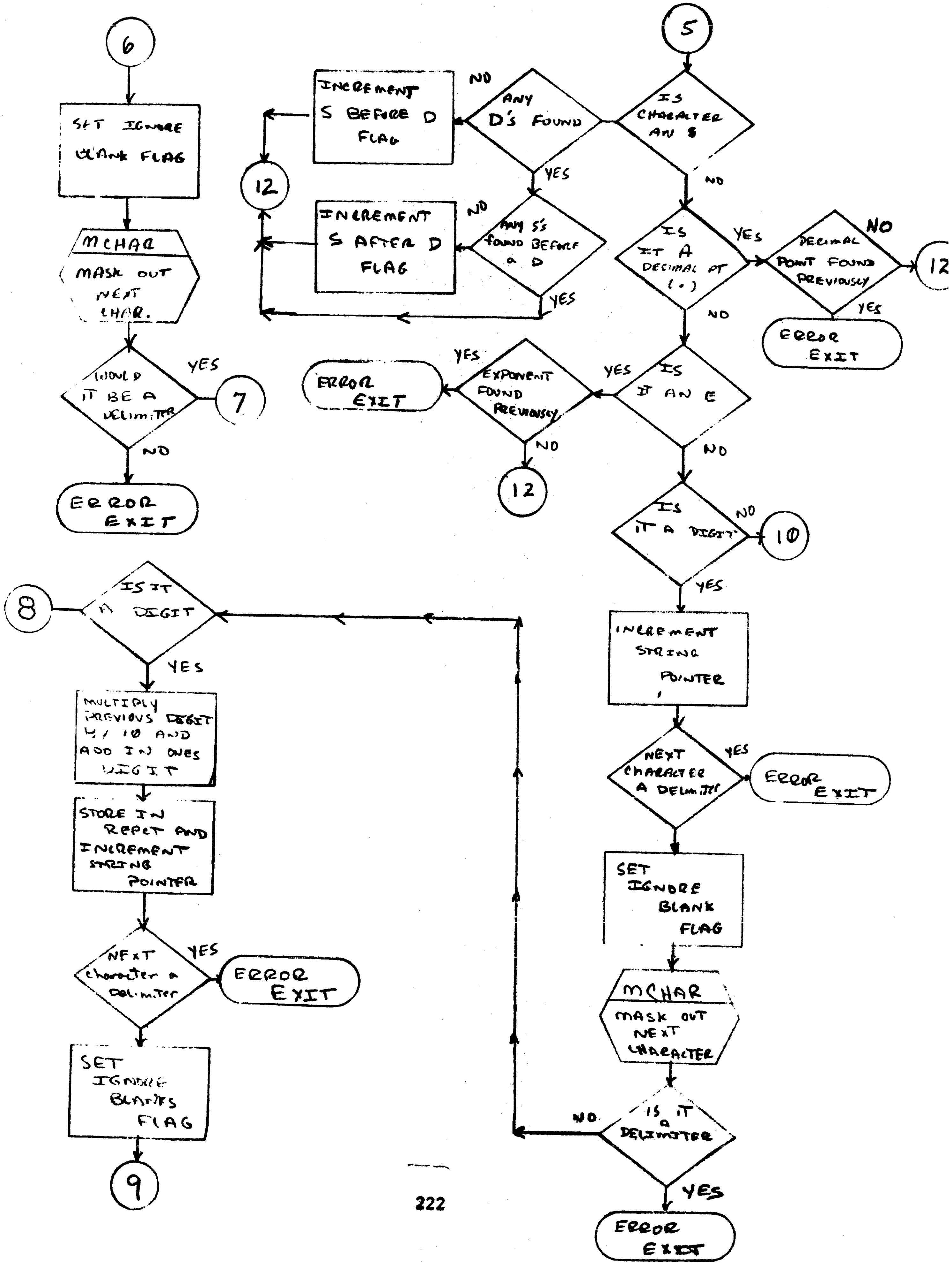

MAT
matrix I/O ?
NO
22
YES
null operand next ?
YES
12
NO
load next operator
PRINT USING
YES
MAT PRINT USING?
NO
MAT PRINT?
YES
NO
save operator
redimension array
YES
is [next operator?
NO
save count of elements
insure all array elements are defined
file MAT PRINT?
YES
NO
write element to file
done?
YES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

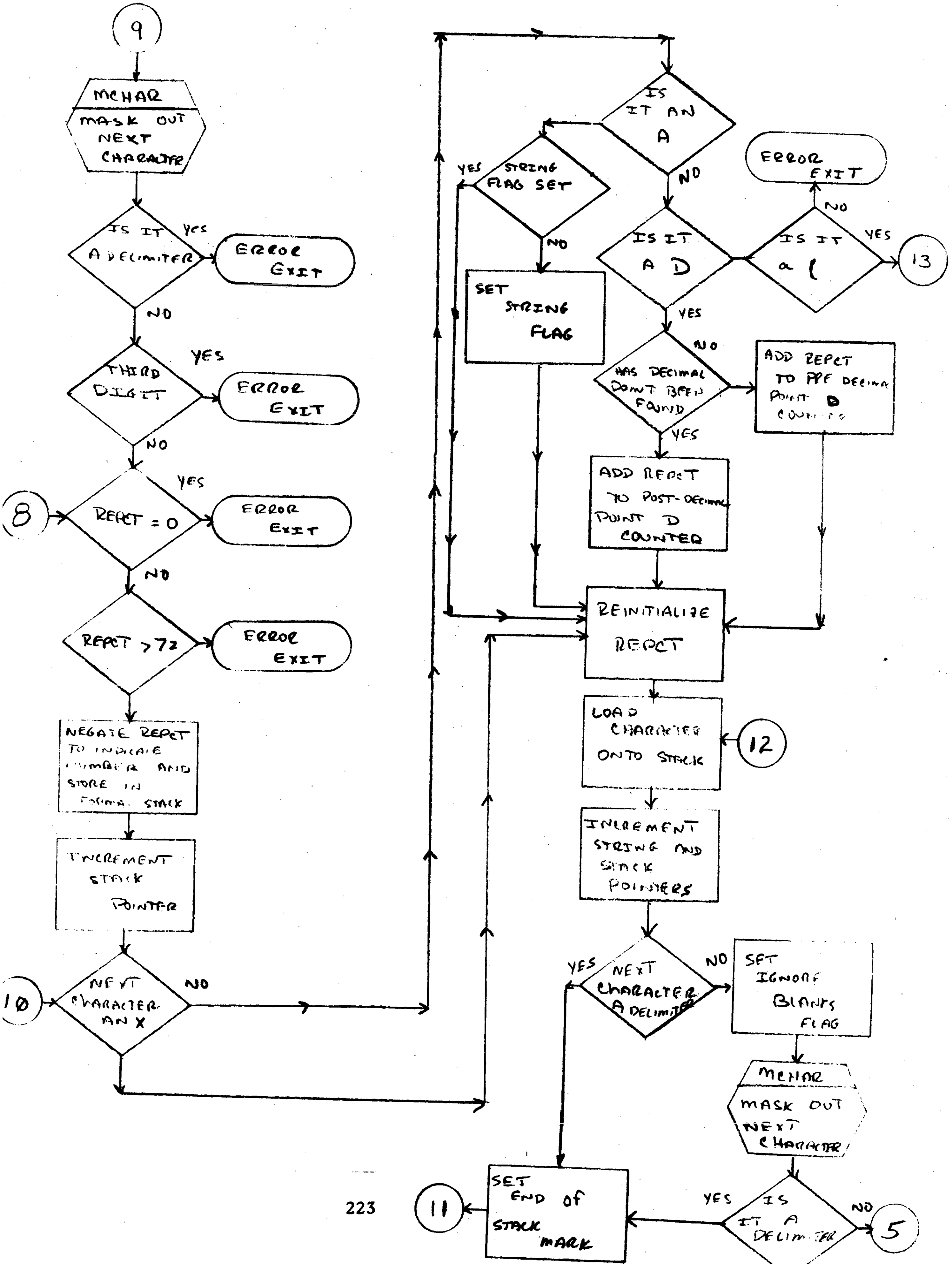

22
set exit to 12
prepare destination matrix
redimension matrix
YES
set 1.Ø or Ø.Ø as the fill constant
NO
IDN?
YES
matrix subscript?
NO
YES
matrix initialization?
YES
matrix function?
NO
NO
23
set loop operation to initialization
YES
square matrix?
NO
ERROR
set loop operation to replacement
take loop exit
YES
done?
NO
evaluate scalar formula
YES
scalar multiplication?
NO
compute an element by loop operation
set loop to scalar multiplication
set up first source matrix
24
first source checked for compatibility
YES
more of statement?
NO
second source checked for compatibility
set loop to matrix + or -
NO
matrix multiplication?
YES
set up second source matrix
12
YES
done?
NO
compute an element by an inner product
check for dimensional compatibilities


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

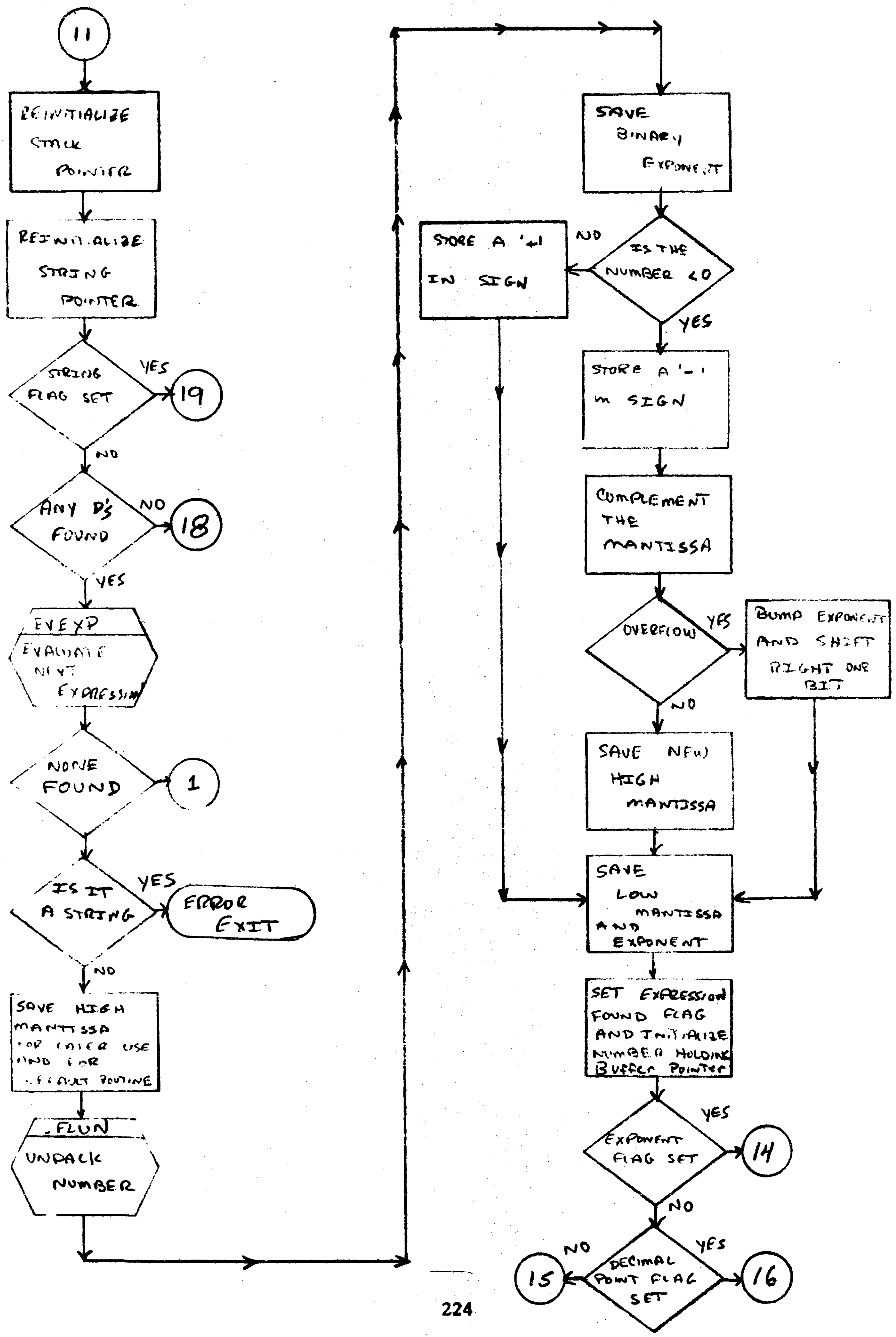

FRMAT
COMPUTE ADDRESS OF STARTING CHARACTER
NUMBER of characters already declared
YES
NO
EVALUATE NUMBER of CHARACTERS IN WHOLE RMAT STRING
NULL STRING (0 characters)
47
NO
CLEAR ALL FLAGS GLOBAL TO FIT RE FORMAT STRING
DSRCH
FIND FIRST DELIMITER
35
SET 0-B7 TO IGNORE BLANKS LOAD POINTER TO FIRST CHARACTER
MCHAR
MASK OUT NEXT CHARACTER
IS THE CHARACTER A DELIMITER
YES
ERROR EXIT
NO
IS IT A CARRIAGE CONTROL CHARACTER
4
YES
HAS THE END of the STRING BEEN REACHED
YES
ERROR EXIT
NO
INCREMENT STRING POINTER SET IGNORE BLANK BIT
MCHAR
MASK OUT NEXT CHARACTER
NO
address of delimiter returned
YES
MCHAR
MASK OUT NEXT CHARACTER
Is character a COMMA ','
NO
ERROR EXIT
YES
3


3
INCREMENT DELIMITER POINTER AND CHARACTER COUNTER
END OF STRING REACHED
YES
ERROR EXIT
NO
DSRCH
FIND NEXT DELIMITER
1
INITIALIZE FLAGS LOCAL TO FORMAT SPECIFICATION
INITIALIZE FORMAT STACK POINTER AND LOAD ADDRESS OF NEXT CHAR.
MCHAR
MASK OUT NEXT CHARACTER
IS IT A QUOTE (")
NO
5
YES
RESET CHARACTER COUNTER
INCREMENT STRING POINTER
END OF STRING
YES
ERROR EXIT
NO
INCREMENT CHARACTER COUNTER AND CLEAR IGNORE BLANK FLAG
MCHAR
MASK OUT NEXT CHARACTER
LOAD CHAR ONTO FORMAT STACK AND INCREMENT STACK POINTER
WAS CHARACTER A QUOTE (")
YES
INCREMENT STRING POINTER AND CHARACTER COUNTER
END OF STRING
7
NO
RESET DELIMITER POINTER
DSRCH
FIND NEXT DELIMITER
NEXT CHARACTER A DELIMITER
7
6

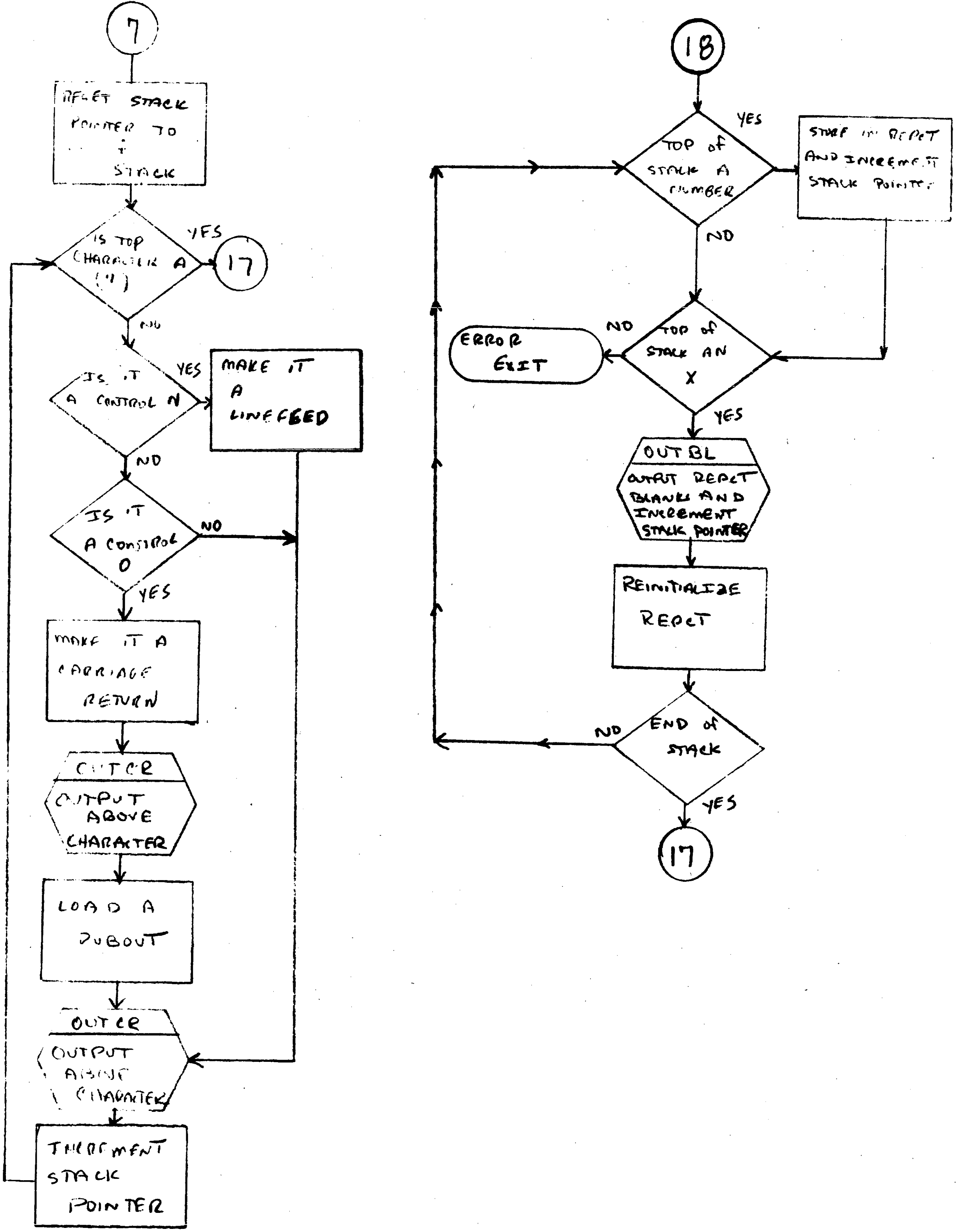

6
SET IGNORE BLANK FLAG
MCHAR
MASK OUT NEXT CHAR.
WOULD IT BE A DELIMITER
YES
7
NO
ERROR EXIT
5
IS CHARACTER AN $
NO
ANY D'S FOUND
NO
INCREMENT S BEFORE D FLAG
YES
ANY S'S FOUND BEFORE A D
NO
INCREMENT S AFTER D FLAG
YES
12
IS IT A DECIMAL PT (.)
YES
DECIMAL POINT FOUND PREVIOUSLY
NO
12
YES
ERROR EXIT
NO
IS IT AN E
YES
EXPONENT FOUND PREVIOUSLY
YES
ERROR EXIT
NO
12
NO
IS IT A DIGIT
NO
10
YES
INCREMENT STRING POINTER
NEXT CHARACTER A DELIMITER
YES
ERROR EXIT
SET IGNORE BLANK FLAG
MCHAR
MASK OUT NEXT CHARACTER
IS IT A DELIMITER
NO
YES
ERROR EXIT
8
IS IT A DIGIT
YES
MULTIPLY PREVIOUS DIGIT BY 10 AND ADD IN ONES DIGIT
STORE IN REPET AND INCREMENT STRING POINTER
NEXT CHARACTER A DELIMITER
YES
ERROR EXIT
SET IGNORE BLANKS FLAG
9


9
MCHAR
MASK OUT NEXT CHARACTER
IS IT A DELIMITER
YES
ERROR EXIT
NO
THIRD DIGIT
YES
ERROR EXIT
NO
8
REPCT = 0
YES
ERROR EXIT
NO
REPCT > 72
ERROR EXIT
NEGATE REPCT TO INDICATE NUMBER AND STORE IN FORMAT STACK
INCREMENT STACK POINTER
10
NEXT CHARACTER AN X
NO
IS IT AN A
YES
STRING FLAG SET
NO
SET STRING FLAG
NO
IS IT A D
ERROR EXIT
NO
IS IT a (
YES
13
YES
HAS DECIMAL POINT BEEN FOUND
NO
ADD REPCT TO PRE DECIMAL POINT D COUNTER
YES
ADD REPCT TO POST-DECIMAL POINT D COUNTER
REINITIALIZE REPCT
LOAD CHARACTER ONTO STACK
12
INCREMENT STRING AND STACK POINTERS
YES
NEXT CHARACTER A DELIMITER
NO
SET IGNORE BLANKS FLAG
MCHAR
MASK OUT NEXT CHARACTER
SET END OF STACK MARK
11
YES
IS IT A DELIMITER
NO
5

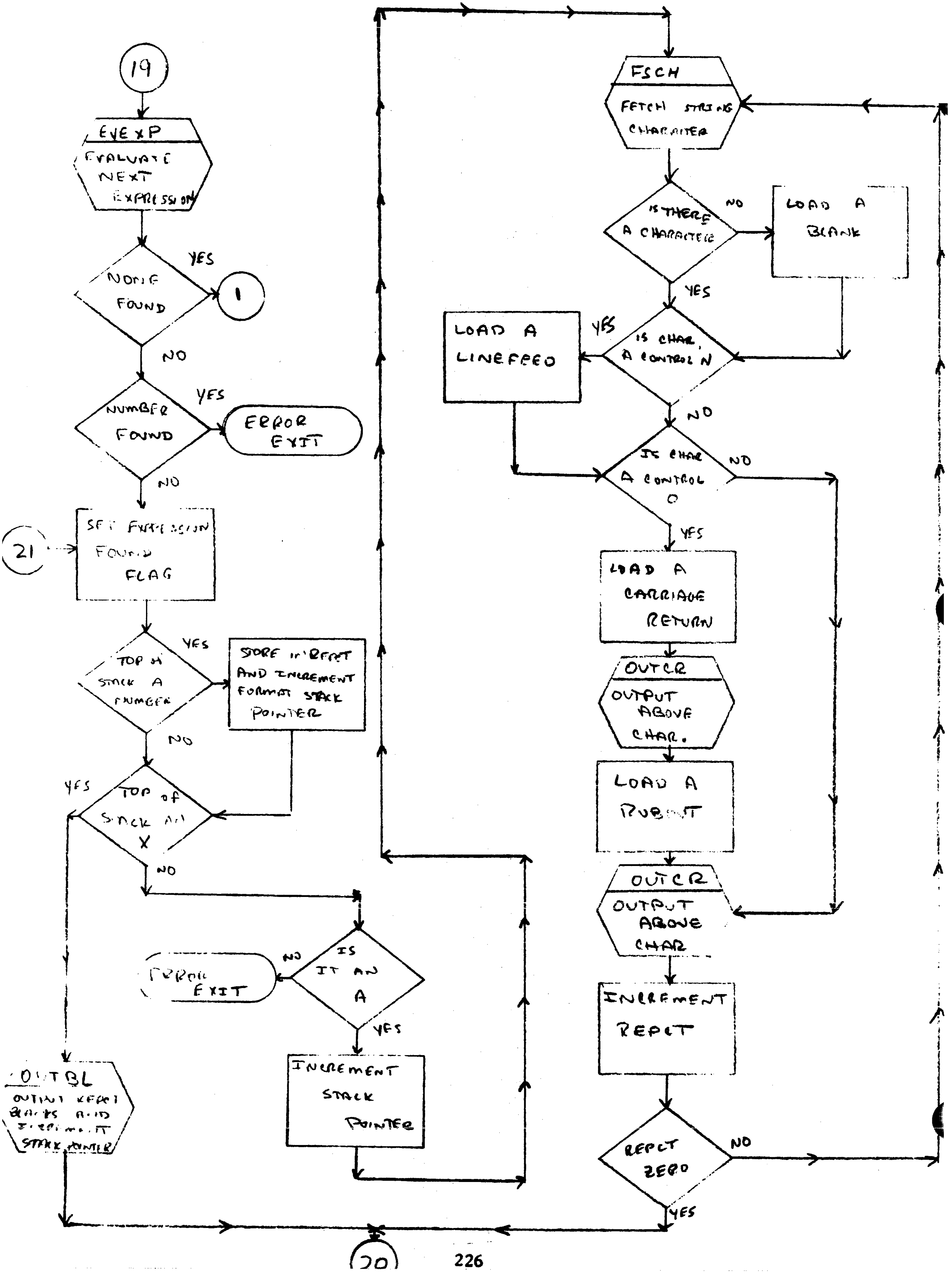
FORMATTER (5 # 25)



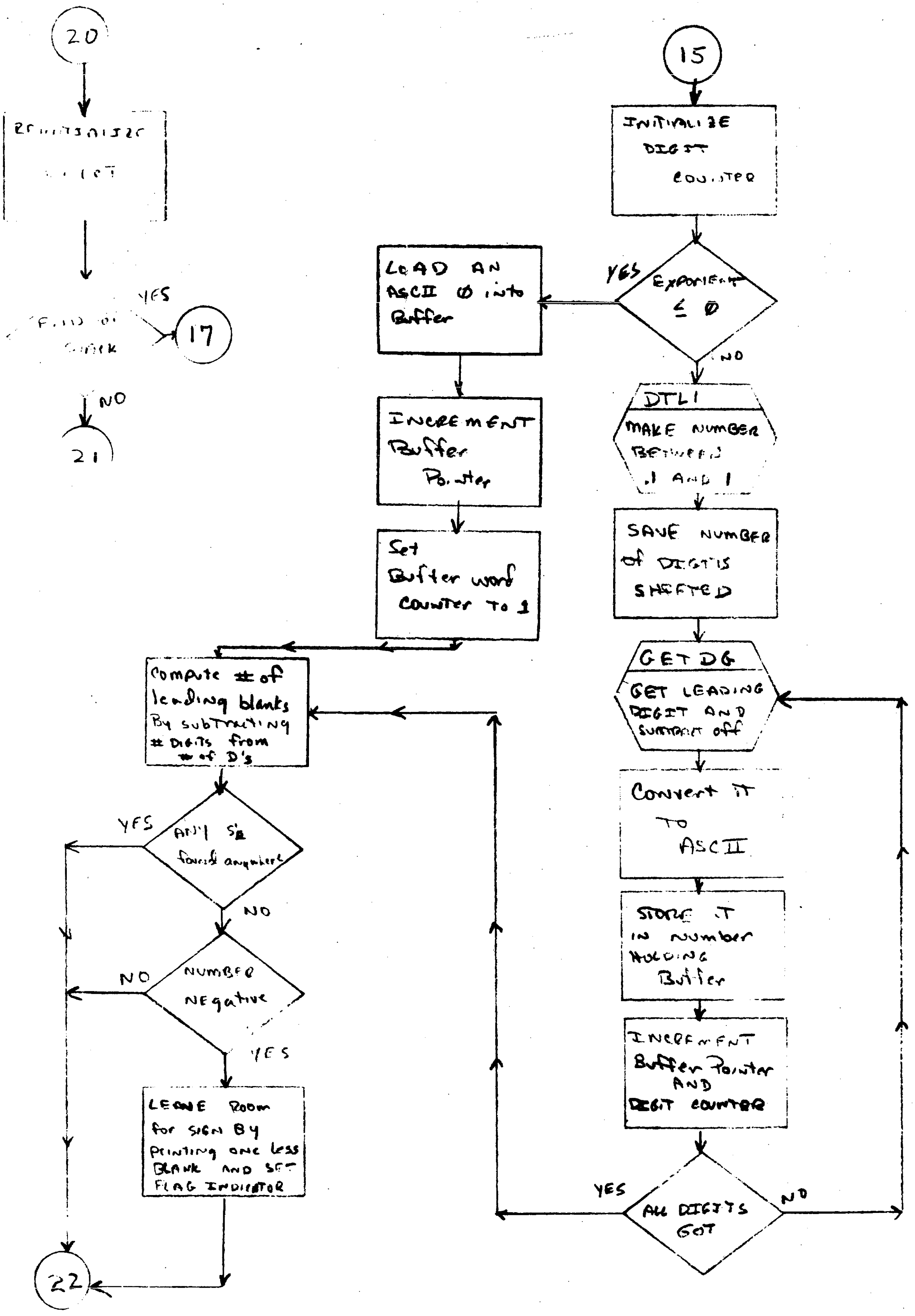
FORMATTER (6 OF 25)



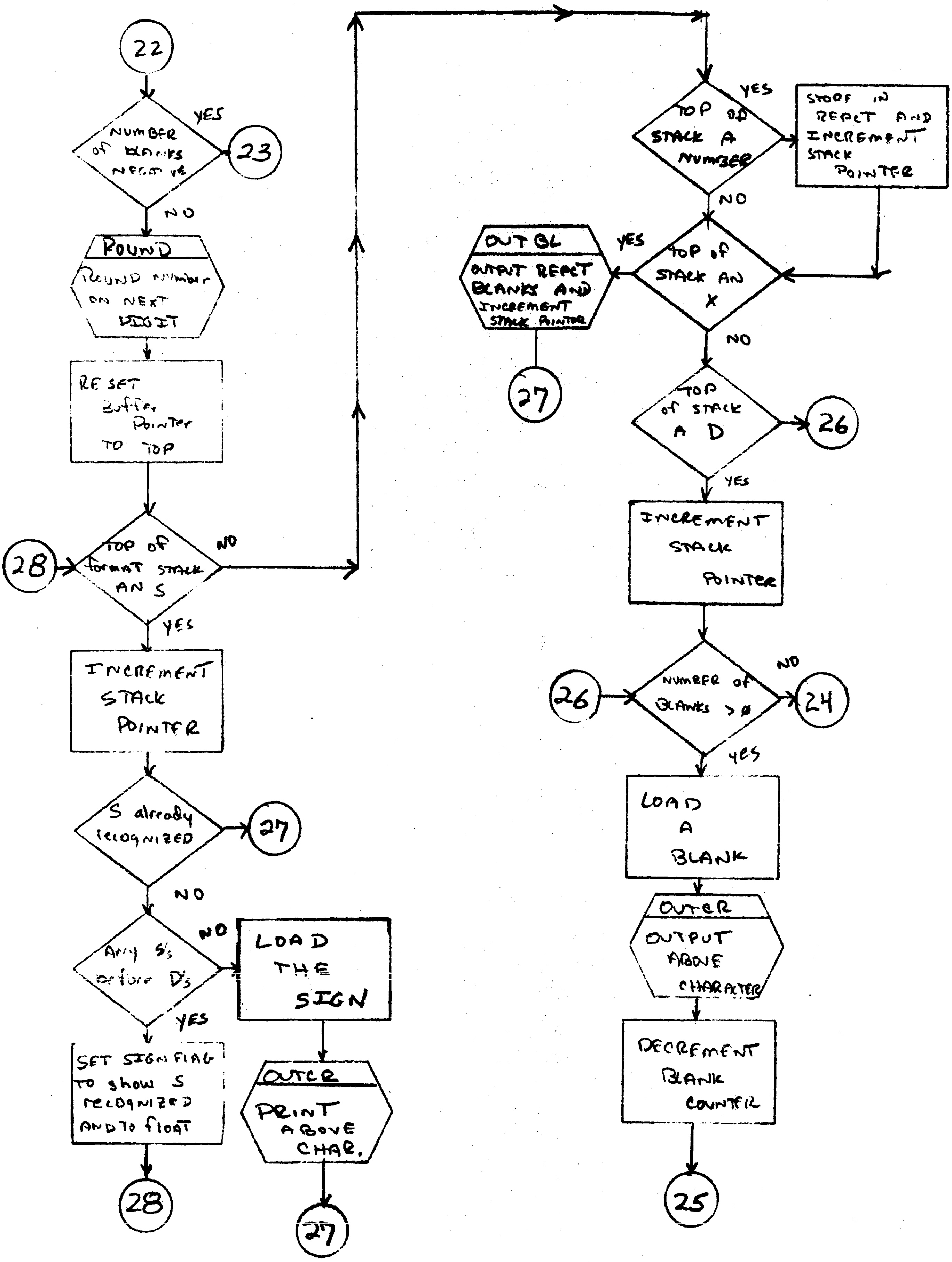
FORMATTER (7 of 25)



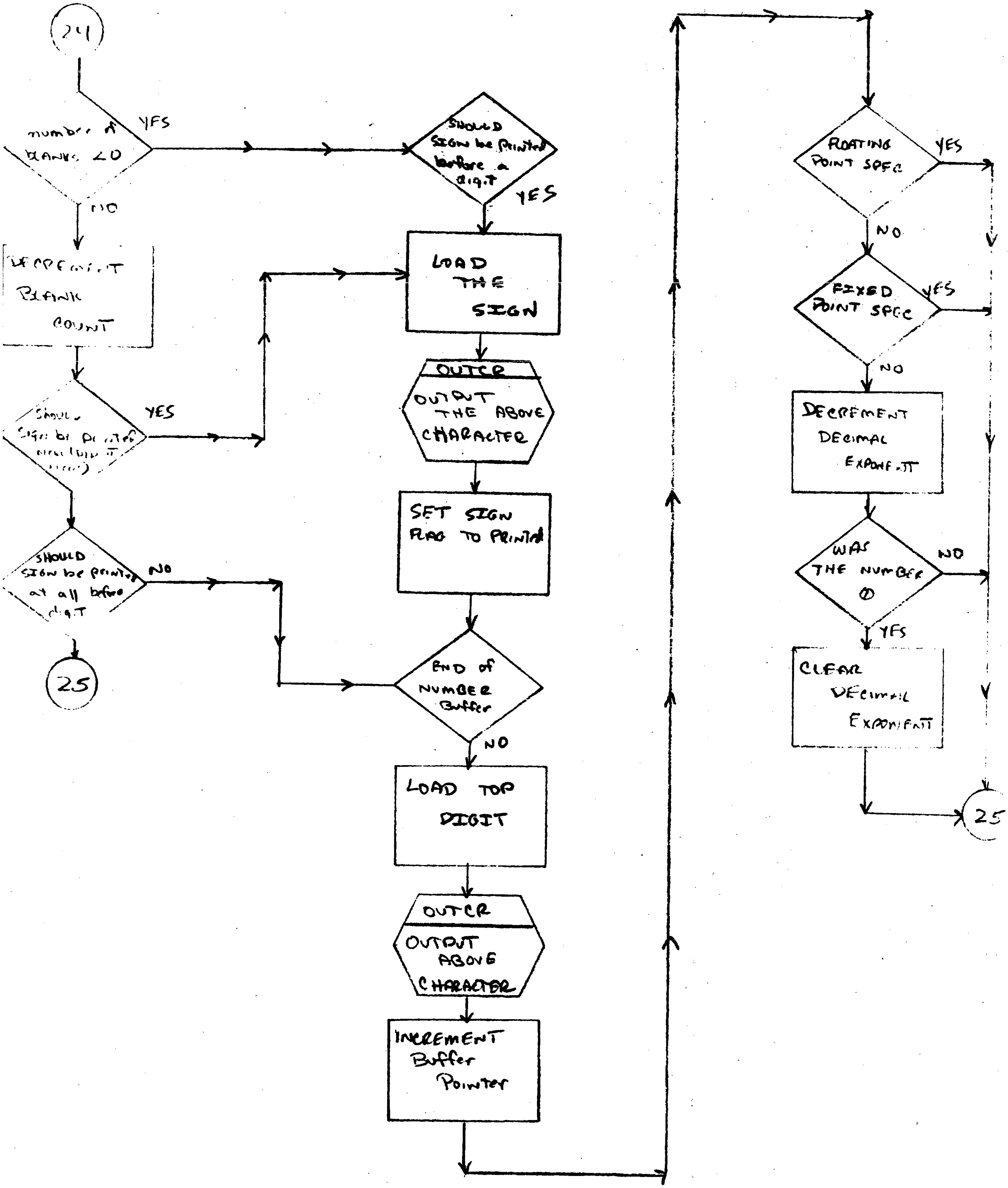
FORMATTER (8 of 25)

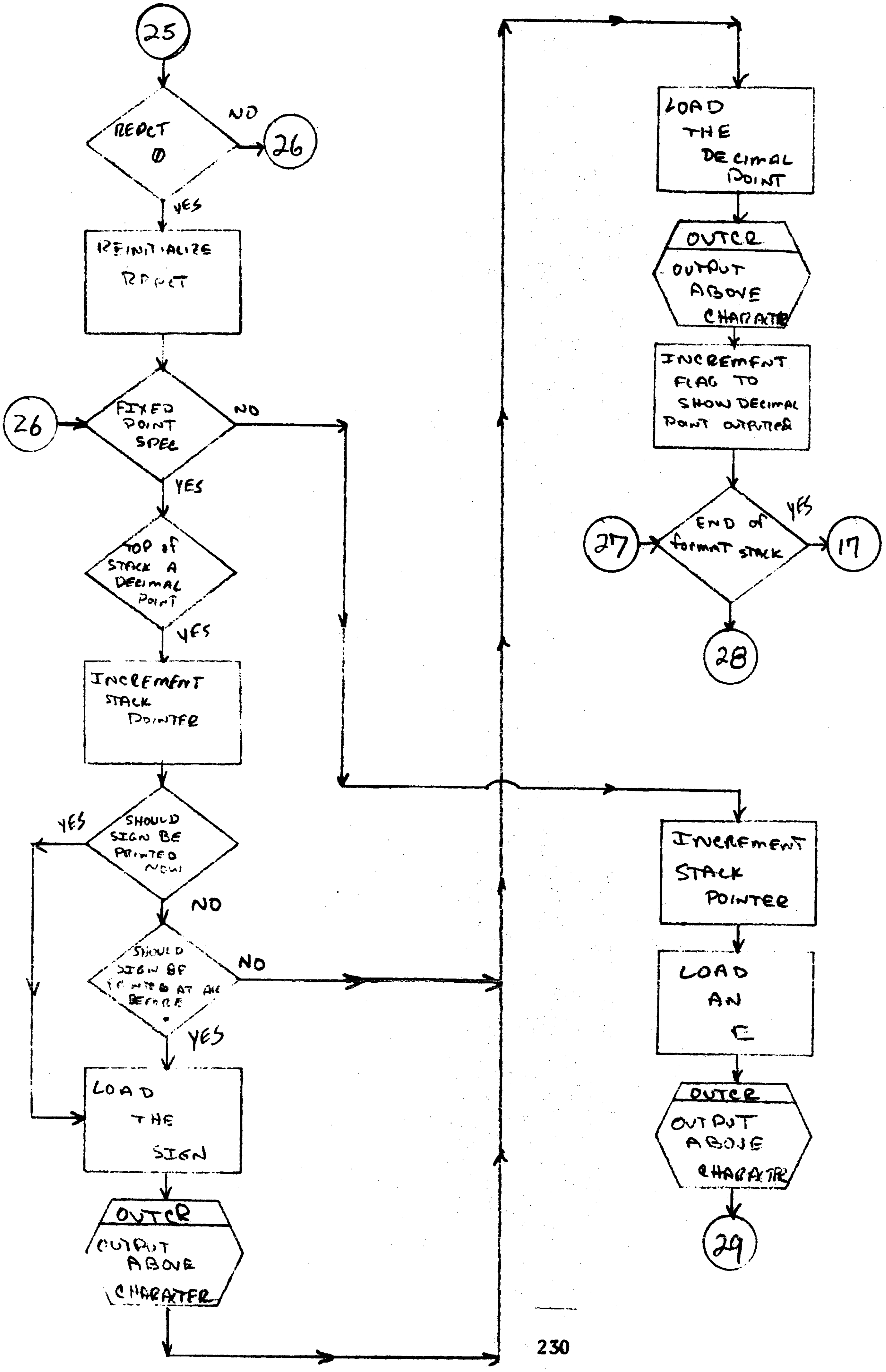

20
REINITIALIZE
END OF STACK
YES
17
NO
21
15
INITIALIZE DIGIT COUNTER
EXPONENT ≤ 0
YES
NO
LOAD AN ASCII 0 into Buffer
INCREMENT Buffer Pointer
Set Buffer word Counter To 1
DTL1
MAKE NUMBER BETWEEN .1 AND 1
SAVE NUMBER of DIGITS SHIFTED
GETDG
GET LEADING DIGIT AND SUBTRACT off
Convert IT TO ASCII
STORE IT IN Number HOLDING Buffer
INCREMENT Buffer Pointer AND DIGIT COUNTER
ALL DIGITS GOT
YES
NO
Compute # of leading blanks By subtracting # DIGITS from # of D's
ANY S's found anywhere
YES
NO
NUMBER NEgative
NO
YES
LEAVE Room for SIGN BY Printing one less BLANK AND SET FLAG INDICATOR
22

FORMATTER (9 of 25)

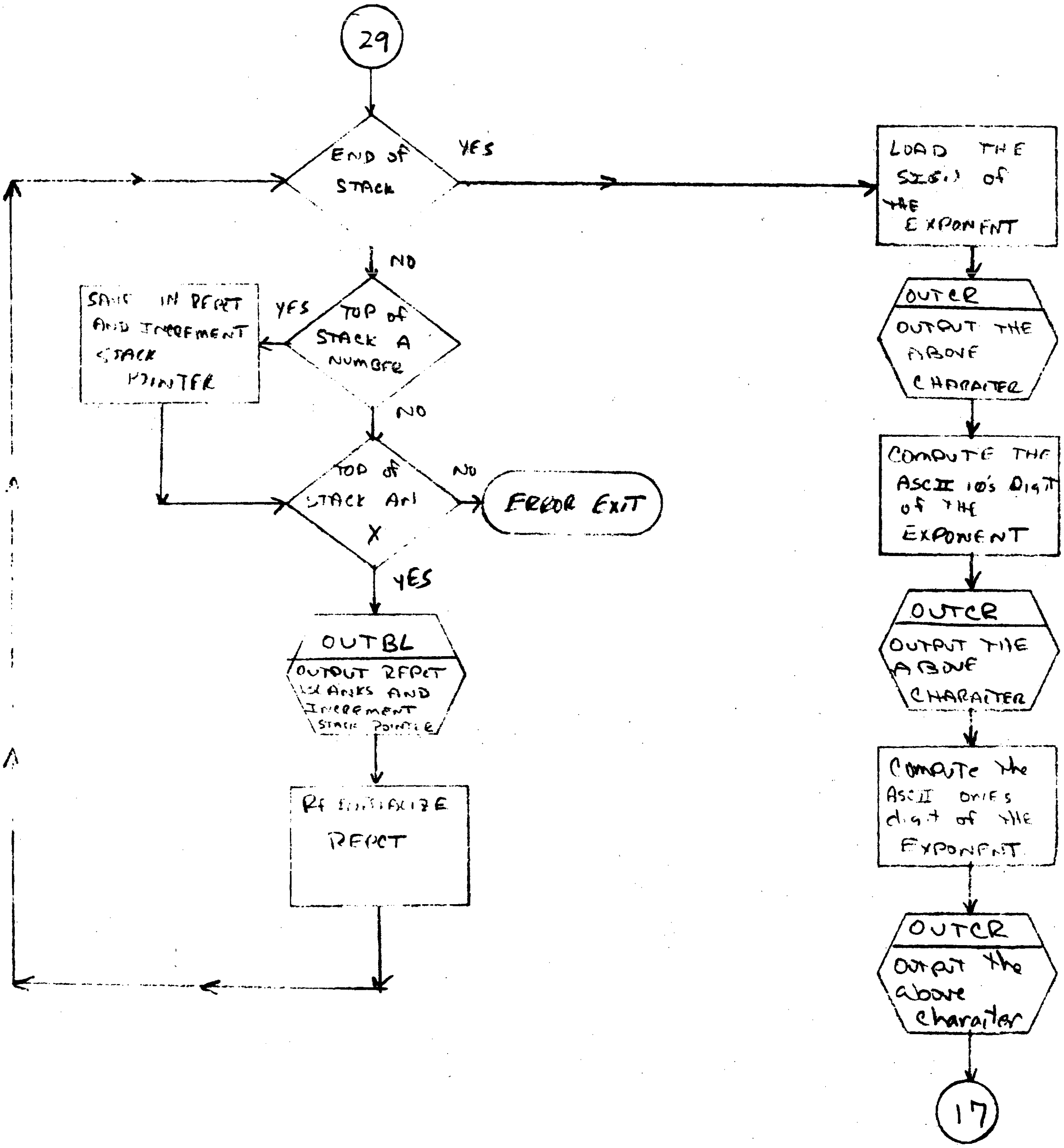


FORMATTER (1Ø of 25)



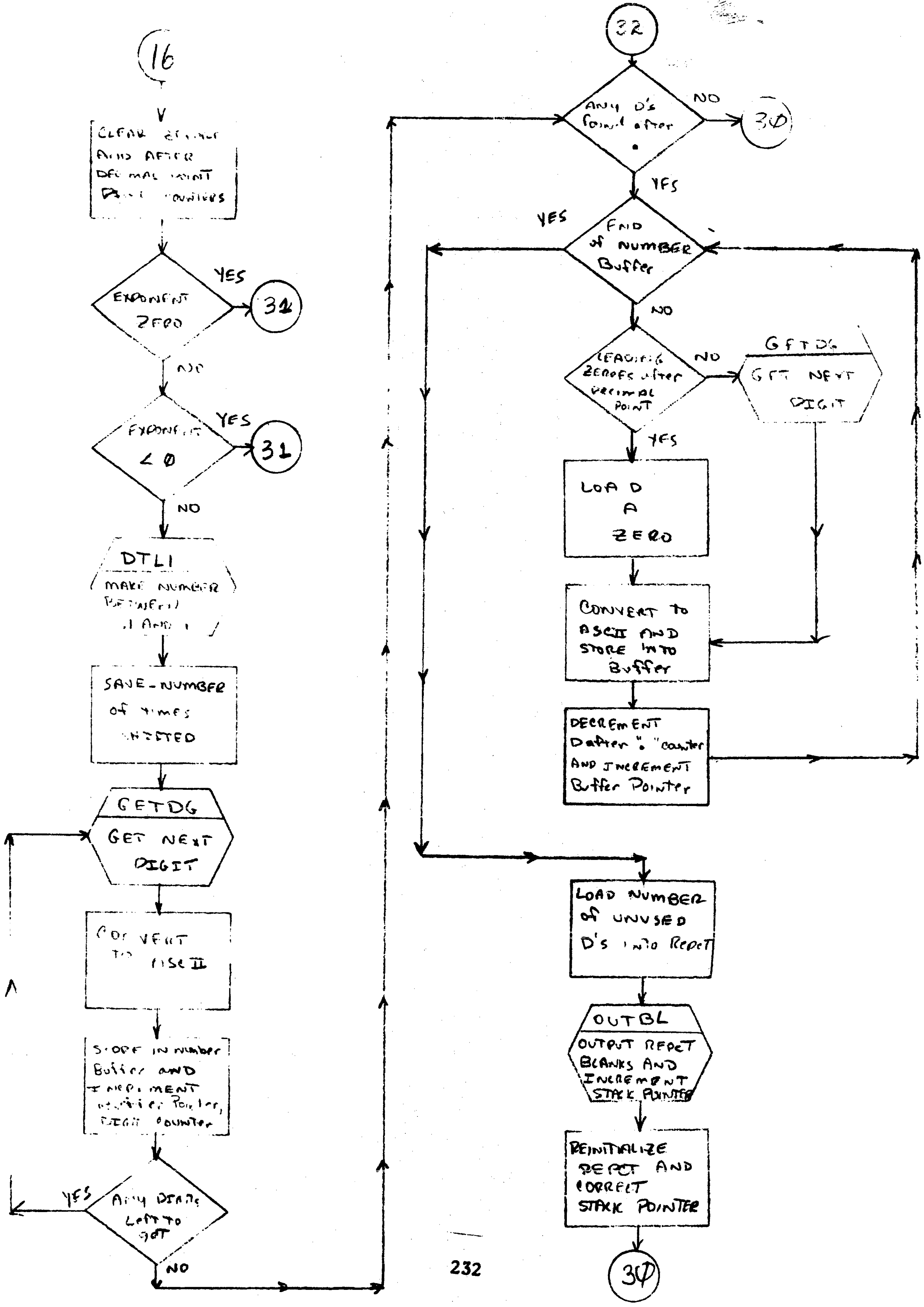

25
REPCT
0
NO
26
YES
REINITIALIZE
REPCT
26
FIXED
POINT
SPEC
NO
YES
TOP of
STACK A
DECIMAL
POINT
YES
INCREMENT
STACK
POINTER
YES
SHOULD
SIGN BE
PRINTED
NOW
NO
SHOULD
SIGN BE
PRINTED AT AN
BEFORE
.
NO
YES
LOAD
THE
SIGN
OUTCR
OUTPUT
ABOVE
CHARACTER
LOAD
THE
DECIMAL
POINT
OUTCR
OUTPUT
ABOVE
CHARACTER
INCREMENT
FLAG TO
SHOW DECIMAL
POINT OUTPUTTED
27
END of
format STACK
YES
17
28
INCREMENT
STACK
POINTER
LOAD
AN
E
OUTCR
OUTPUT
ABOVE
CHARACTER
29

FORMATTER (12 of 25)



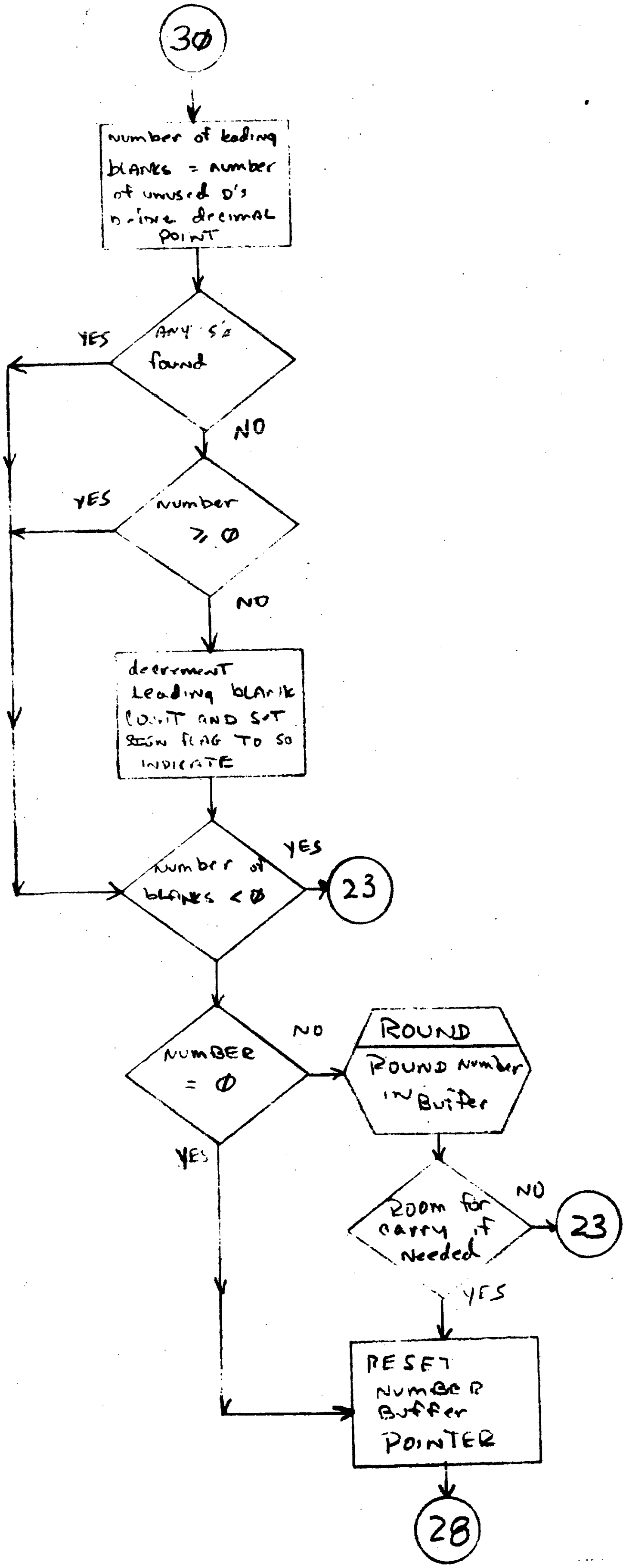
29
END of STACK
YES
NO
LOAD THE SIGN of THE EXPONENT
OUTCR
OUTPUT THE ABOVE CHARACTER
SAVE IN REPCT AND INCREMENT STACK POINTER
YES
TOP of STACK A NUMBER
NO
TOP of STACK AN X
NO
ERROR EXIT
YES
COMPUTE THE ASCII 10's DIGIT of THE EXPONENT
OUTCR
OUTPUT THE ABOVE CHARACTER
OUTBL
OUTPUT REPCT BLANKS AND INCREMENT STACK POINTER
Compute the ASCII ONES digit of the EXPONENT
REINITIALIZE REPCT
OUTCR
output the above charater
17

FORMATTER (13 of 25)

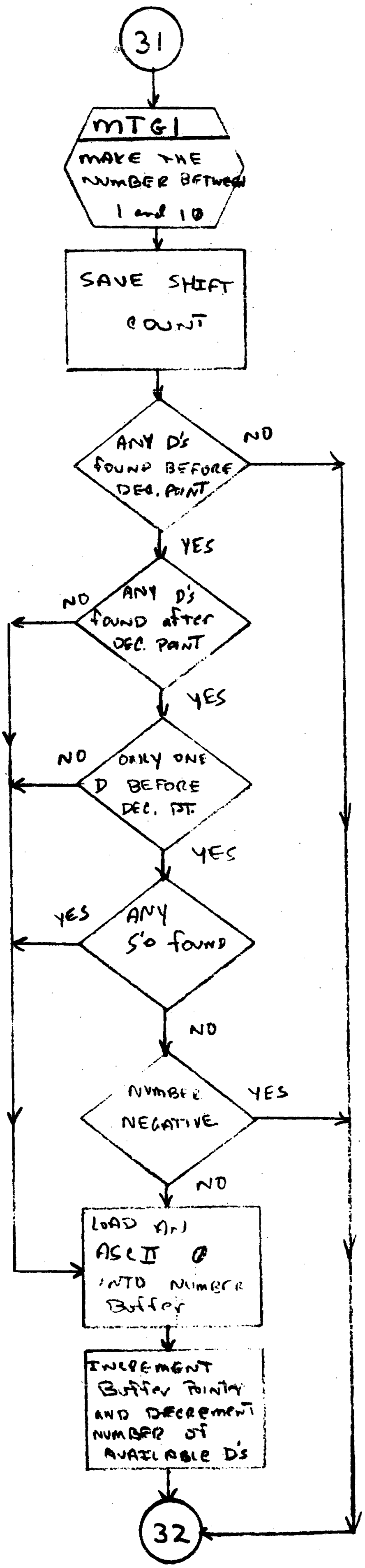


16
CLEAR BEFORE AND AFTER DECIMAL POINT DIGIT COUNTERS
EXPONENT ZERO
YES
32
NO
EXPONENT < 0
YES
31
NO
DTLI
MAKE NUMBER BETWEEN .1 AND 1
SAVE NUMBER OF TIMES SHIFTED
GETDG
GET NEXT DIGIT
CONVERT TO ASCII
STORE IN number Buffer AND INCREMENT number Pointer, DIGIT COUNTER
Any DIGITS LEFT TO GET
YES
NO
32
Any D's found after .
NO
30
YES
End of NUMBER Buffer
YES
NO
LEADING ZEROES after DECIMAL POINT
NO
GETDG
GET NEXT DIGIT
YES
LOAD A ZERO
CONVERT TO ASCII AND STORE INTO Buffer
DECREMENT D after "." counter AND INCREMENT Buffer Pointer
LOAD NUMBER OF UNUSED D's into REPCT
OUTBL
OUTPUT REPCT BLANKS AND INCREMENT STACK POINTER
REINITIALIZE REPCT AND CORRECT STACK POINTER
30

FORMATTER (14 of 25)

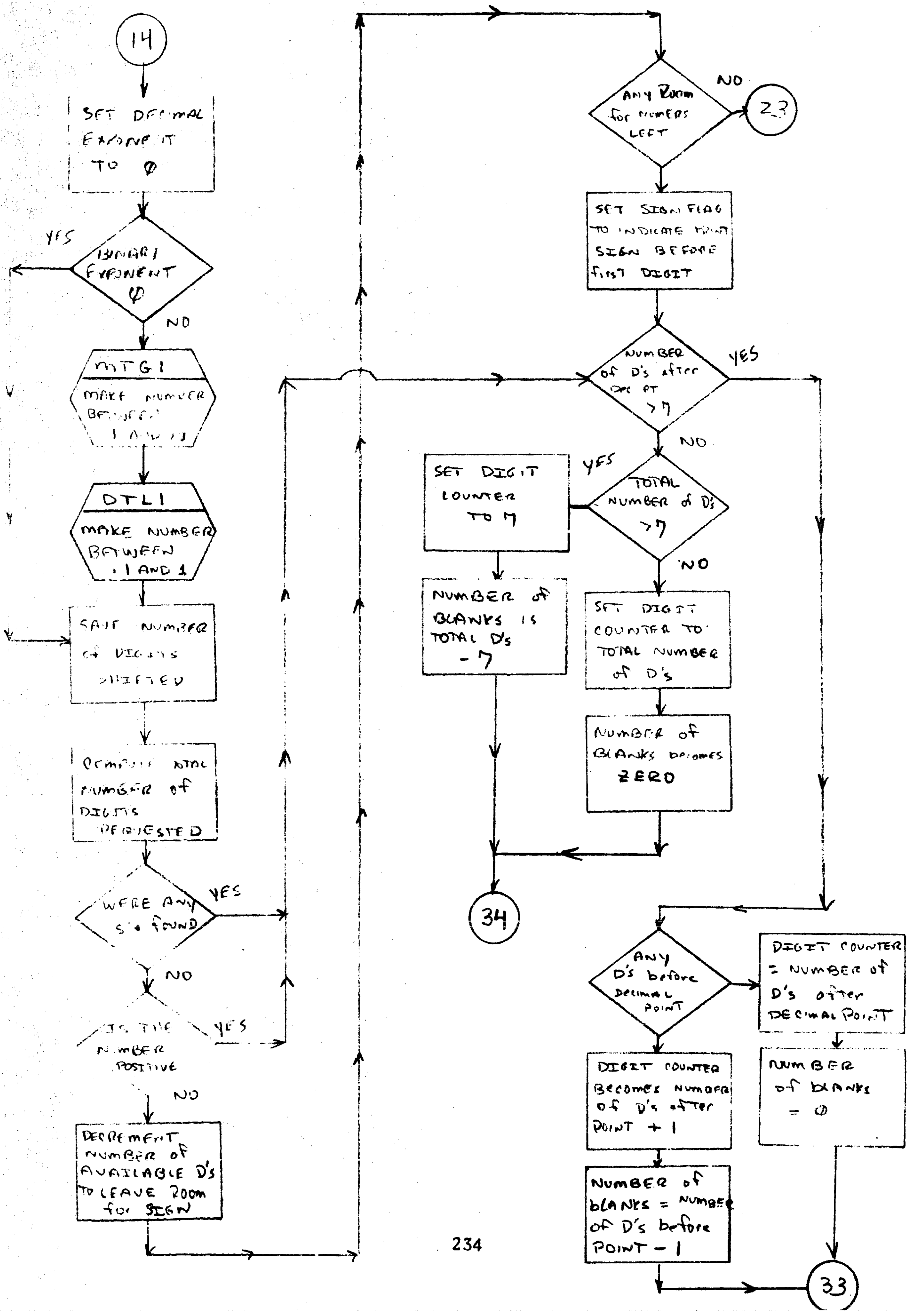


30
number of leading blanks = number of unused D's before decimal point
Any S's found
YES
NO
number ≥ Ø
YES
NO
decrement Leading blank count and set sign flag to so indicate
number of blanks < Ø
YES
23
NUMBER = Ø
NO
YES
ROUND
ROUND Number in Buffer
Room for carry if Needed
NO
23
YES
RESET NUMBER Buffer POINTER
28

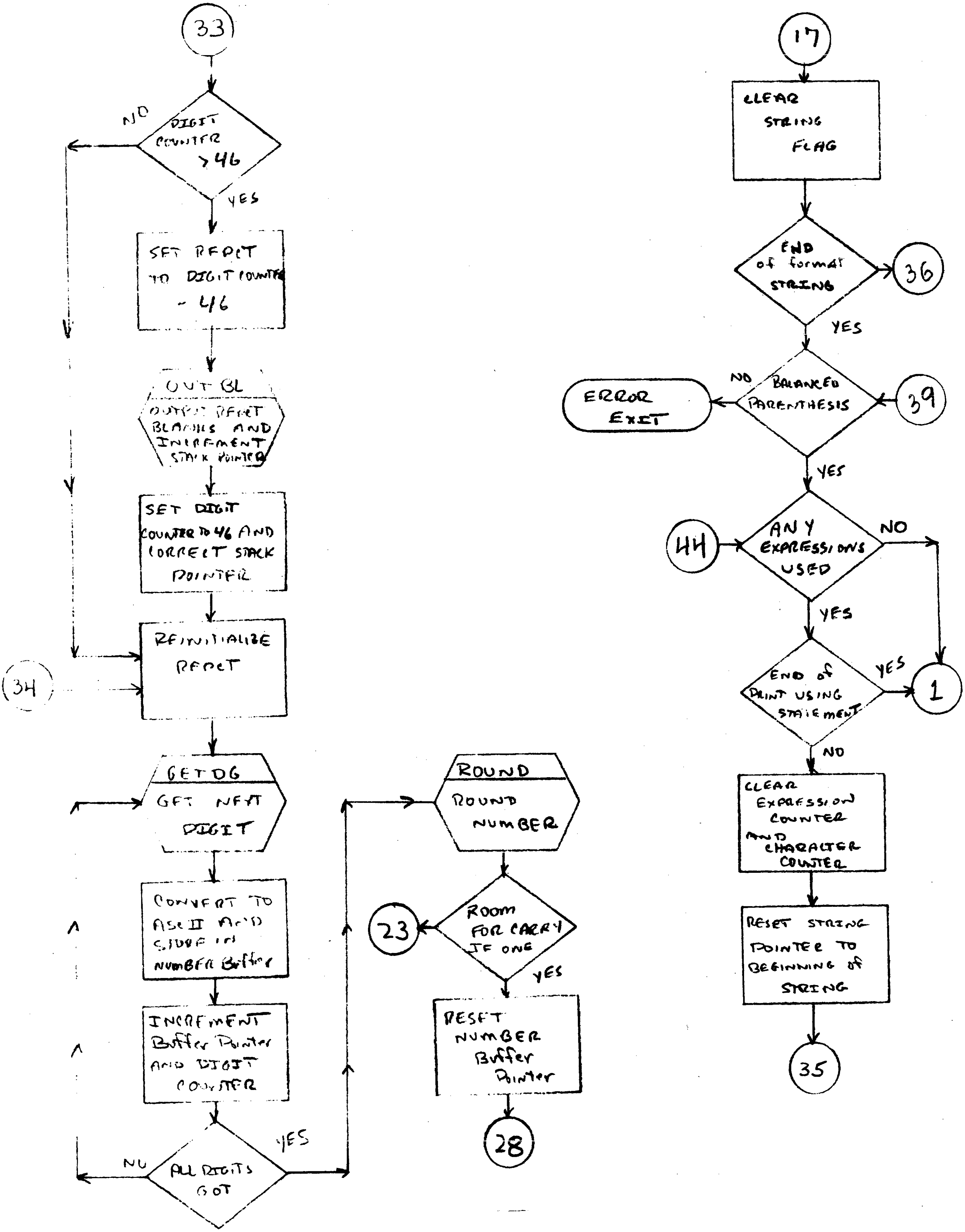


31
MTG1
MAKE THE NUMBER BETWEEN 1 and 10
SAVE SHIFT COUNT
ANY D's FOUND BEFORE DEC. POINT
NO
YES
ANY D's found after DEC. POINT
NO
YES
ONLY ONE D BEFORE DEC. PT.
NO
YES
ANY S's found
YES
NO
NUMBER NEGATIVE
YES
NO
LOAD AN ASCII Ø INTO NUMBER Buffer
INCREMENT Buffer Pointer AND DECREMENT NUMBER of AVAILABLE D's
32

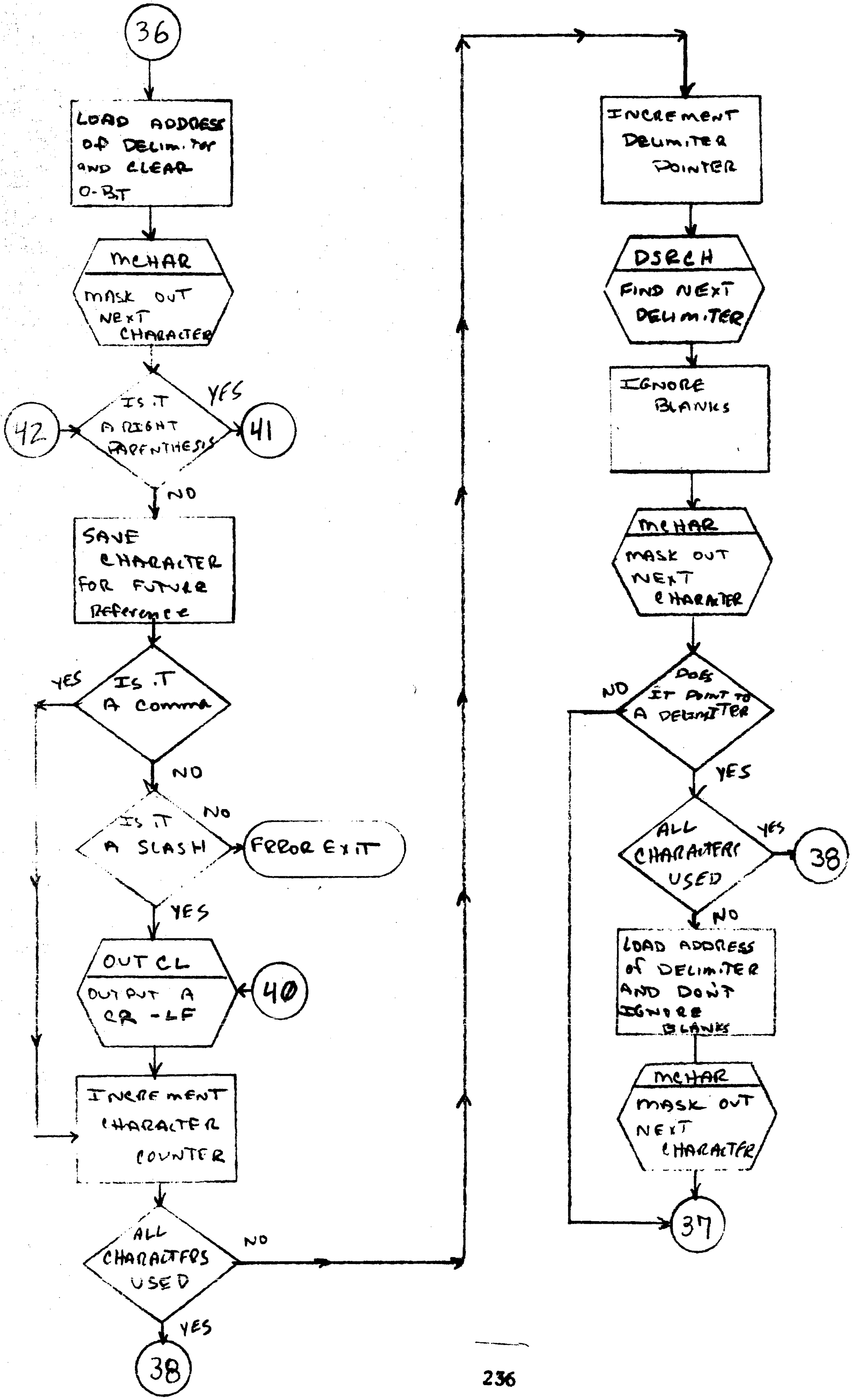
FORMATTER (15 of 25)


14
SET DECIMAL EXPONENT TO Ø
BINARY EXPONENT Ø
YES
NO
MTG1
MAKE NUMBER BETWEEN 1 AND 10
DTL1
MAKE NUMBER BETWEEN .1 AND 1
SAVE NUMBER of DIGITS SHIFTED
COMPUTE TOTAL NUMBER of DIGITS REQUESTED
WERE ANY S's FOUND
YES
NO
IS THE NUMBER POSITIVE
YES
NO
DECREMENT NUMBER of AVAILABLE D's TO LEAVE ROOM for SIGN
ANY ROOM for NUMBERS LEFT
NO
23
SET SIGN FLAG TO INDICATE PRINT SIGN BEFORE first DIGIT
NUMBER of D's after DEC PT >7
YES
NO
TOTAL NUMBER of D's >7
YES
NO
SET DIGIT COUNTER TO 7
NUMBER of BLANKS IS TOTAL D's -7
SET DIGIT COUNTER TO TOTAL NUMBER of D's
NUMBER of BLANKS BECOMES ZERO
34
ANY D's before DECIMAL POINT
DIGIT COUNTER = NUMBER of D's after DECIMAL POINT
NUMBER of BLANKS = Ø
DIGIT COUNTER BECOMES NUMBER of D's after POINT + 1
NUMBER of BLANKS = NUMBER of D's before POINT - 1
33

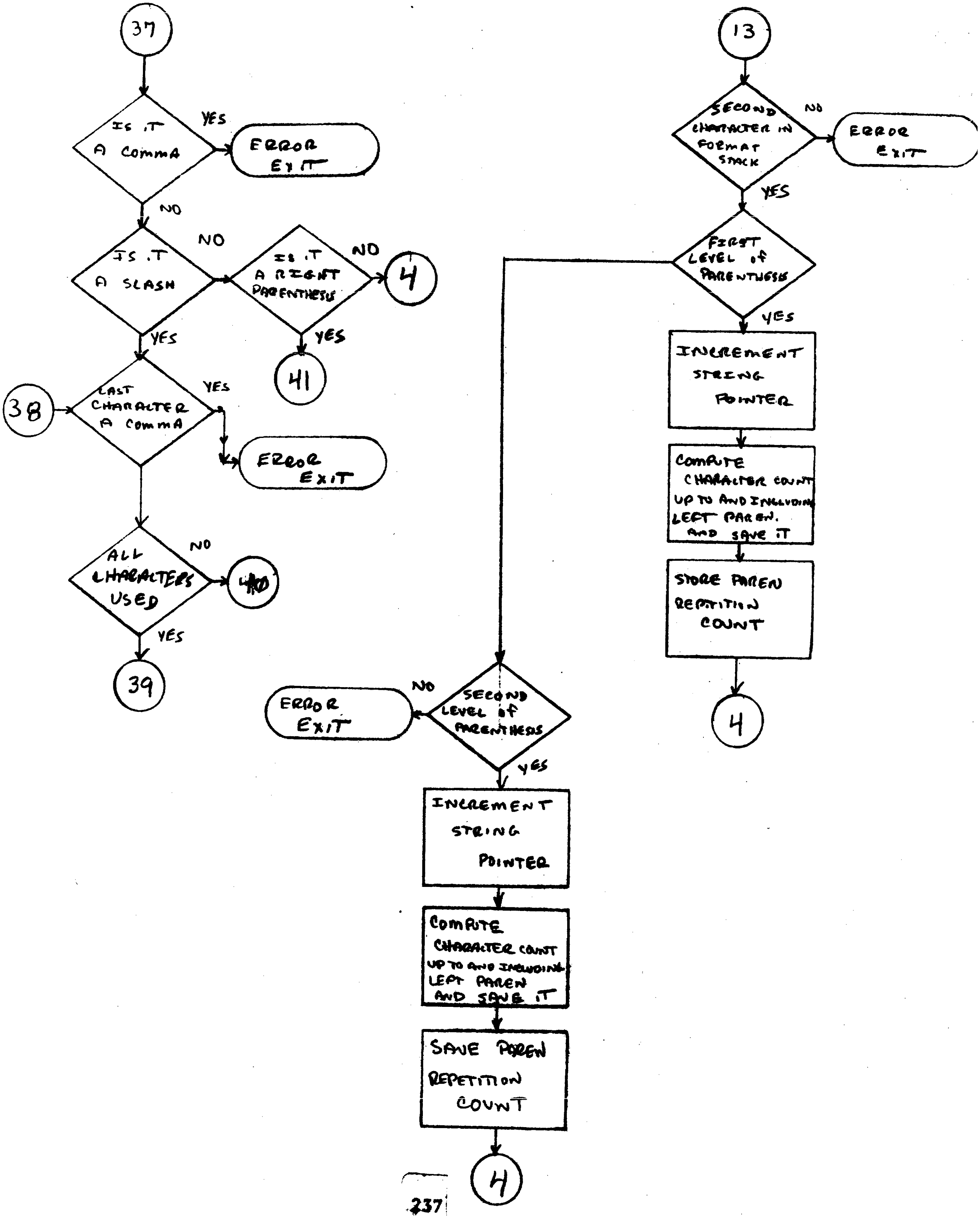
FORMATTER (16 of 25)


33
NO
DIGIT COUNTER >46
YES
SET REPET TO DIGIT COUNTER - 46
OUT BL
OUTPUT REPET BLANKS AND INCREMENT STACK POINTER
SET DIGIT COUNTER TO 46 AND CORRECT STACK POINTER
REINITIALIZE REPET
34
GETDG
GET NEXT DIGIT
CONVERT TO ASCII AND STORE IN NUMBER Buffer
INCREMENT Buffer Pointer AND DIGIT COUNTER
NO
ALL DIGITS GOT
YES
ROUND
ROUND NUMBER
23
ROOM FOR CARRY IF ONE
YES
RESET NUMBER Buffer Pointer
28
17
CLEAR STRING FLAG
END of format STRING
36
YES
NO
ERROR EXIT
BALANCED PARENTHESIS
39
YES
44
ANY EXPRESSIONS USED
NO
YES
END of PRINT USING STATEMENT
YES
1
NO
CLEAR EXPRESSION COUNTER AND CHARACTER COUNTER
RESET STRING POINTER TO BEGINNING of STRING
35

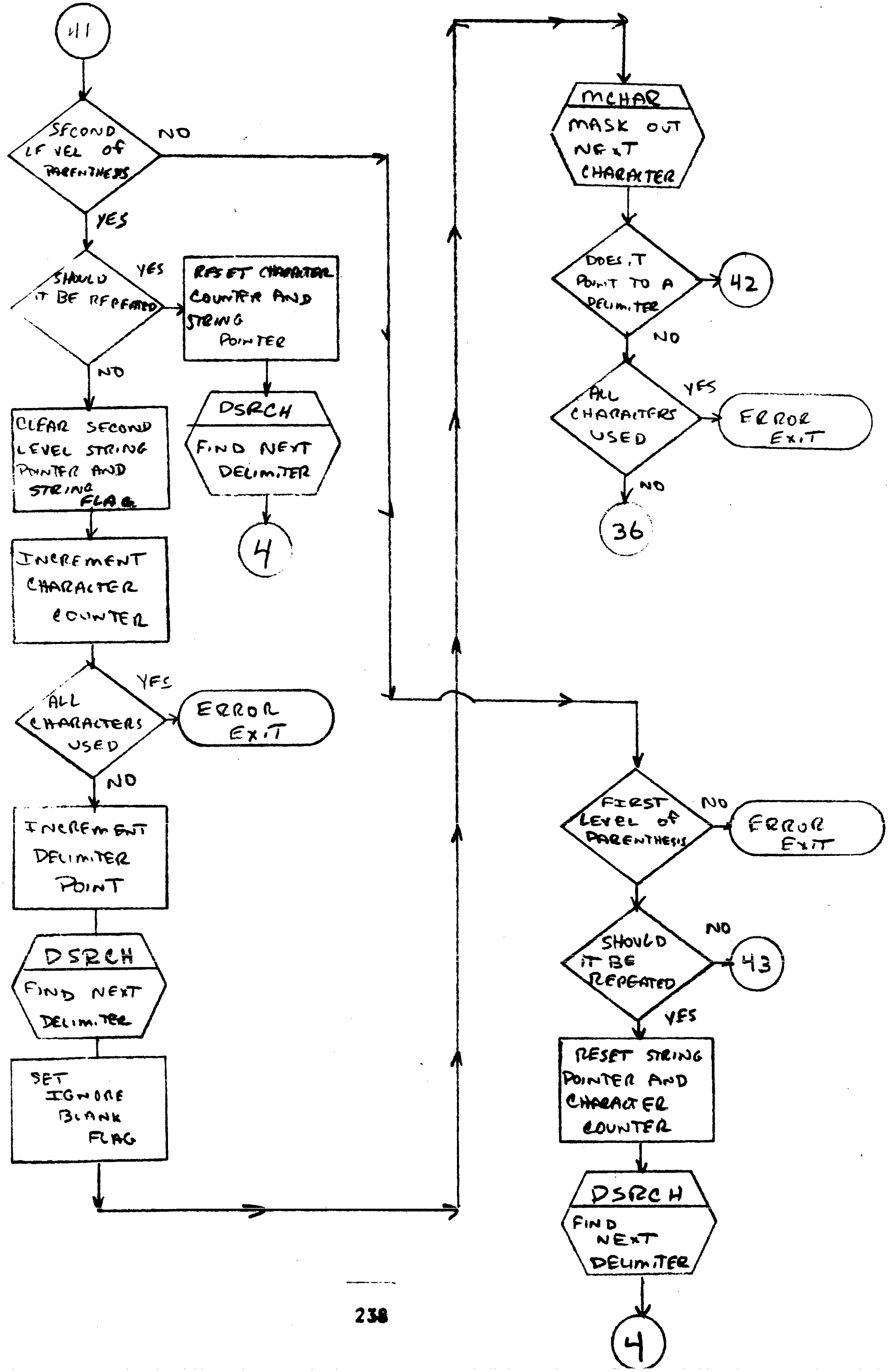
FORMATTER (17 of 25)

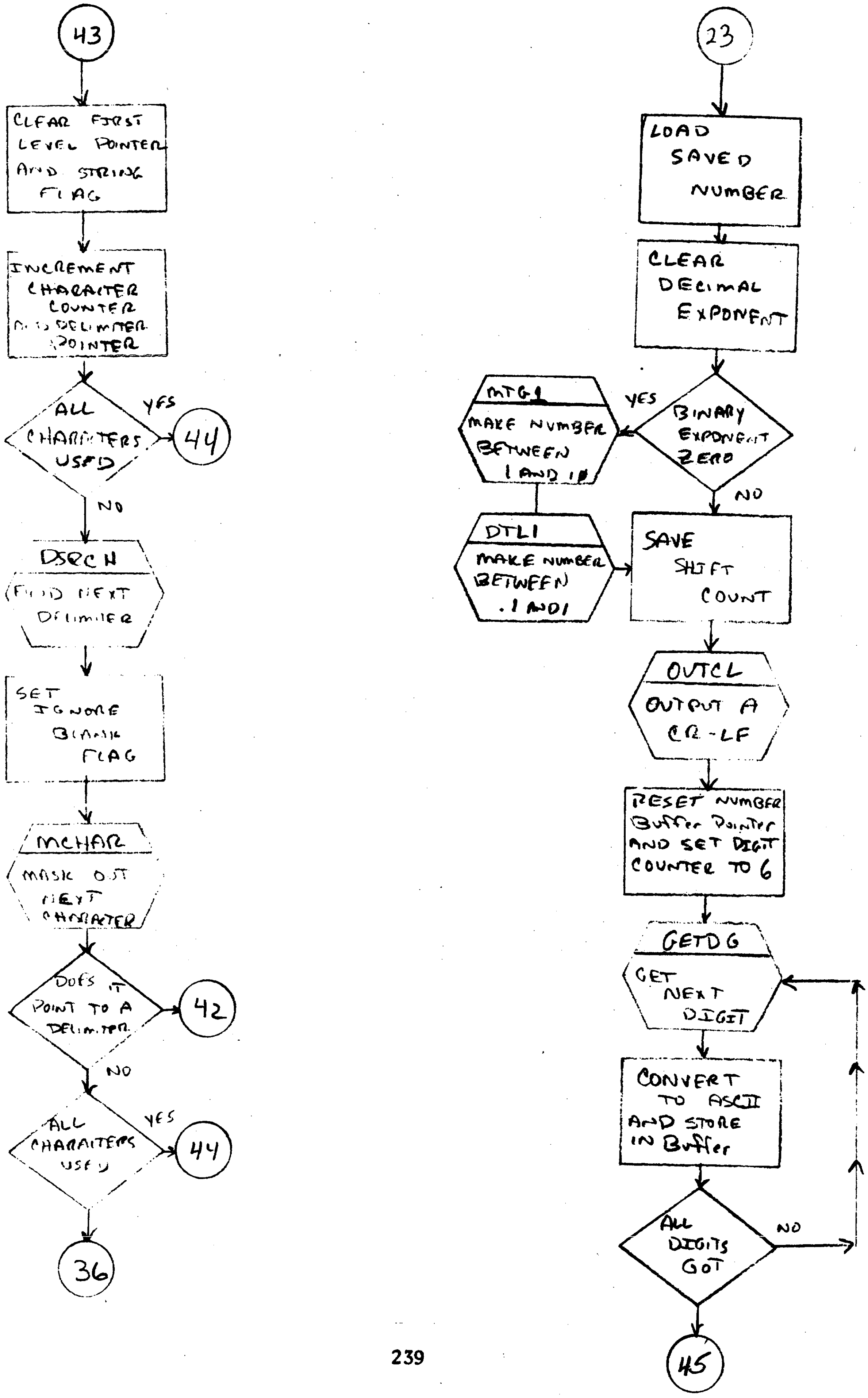


FORMATTER (18 of 25)

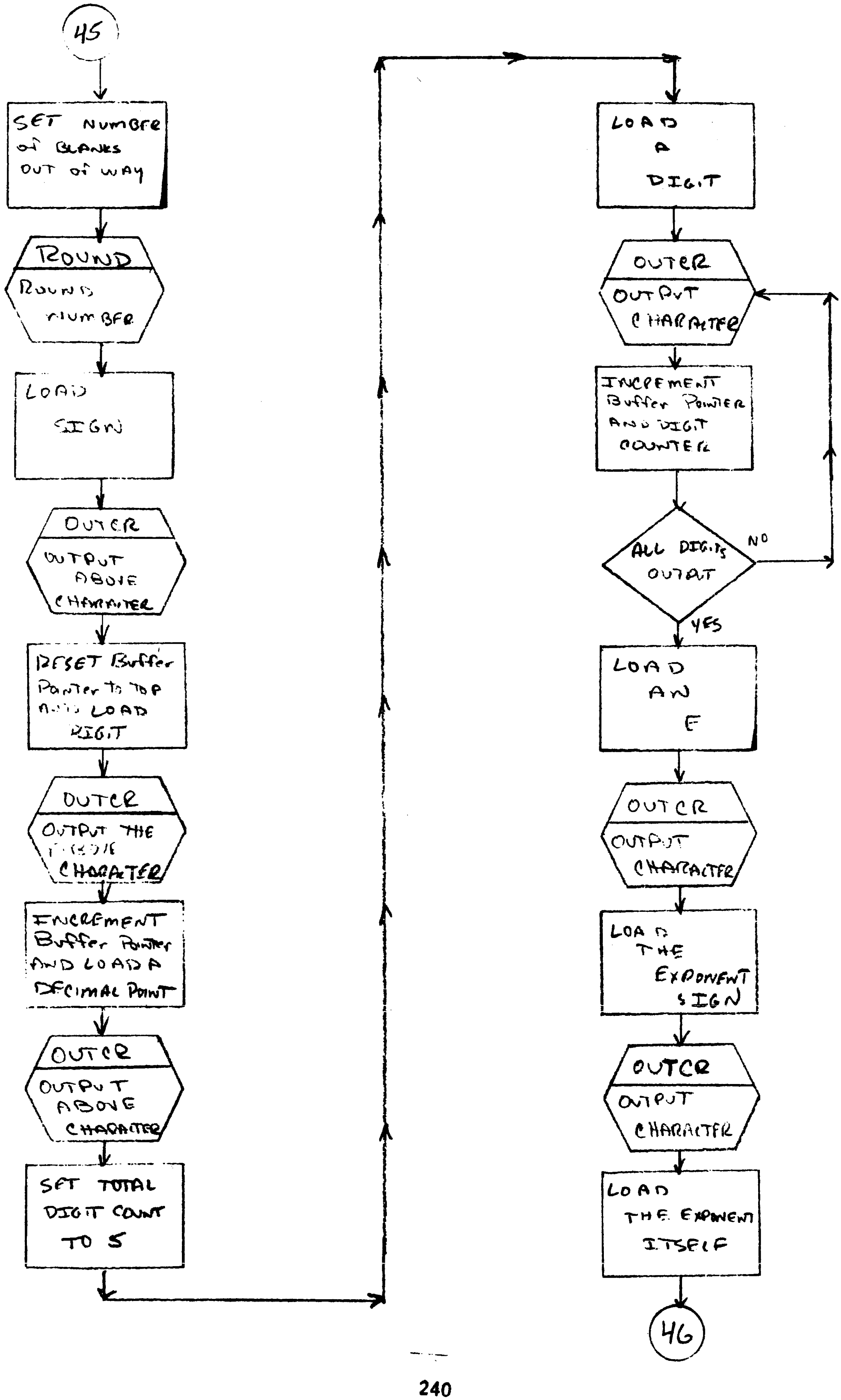

37
IS IT A COMMA
YES
ERROR EXIT
NO
IS IT A SLASH
NO
IS IT A RIGHT PARENTHESIS
NO
4
YES
41
YES
38
LAST CHARACTER A COMMA
YES
ERROR EXIT
ALL CHARACTERS USED
NO
40
YES
39
13
SECOND CHARACTER IN FORMAT STACK
NO
ERROR EXIT
YES
FIRST LEVEL of PARENTHESIS
YES
INCREMENT STRING POINTER
COMPUTE CHARACTER COUNT UP TO AND INCLUDING LEFT PAREN. AND SAVE IT
STORE PAREN REPITITION COUNT
4
SECOND LEVEL of PARENTHESIS
NO
ERROR EXIT
YES
INCREMENT STRING POINTER
COMPUTE CHARACTER COUNT UP TO AND INCLUDING LEFT PAREN AND SAVE IT
SAVE PAREN REPETITION COUNT
4

FORMATTER (19 of 25)

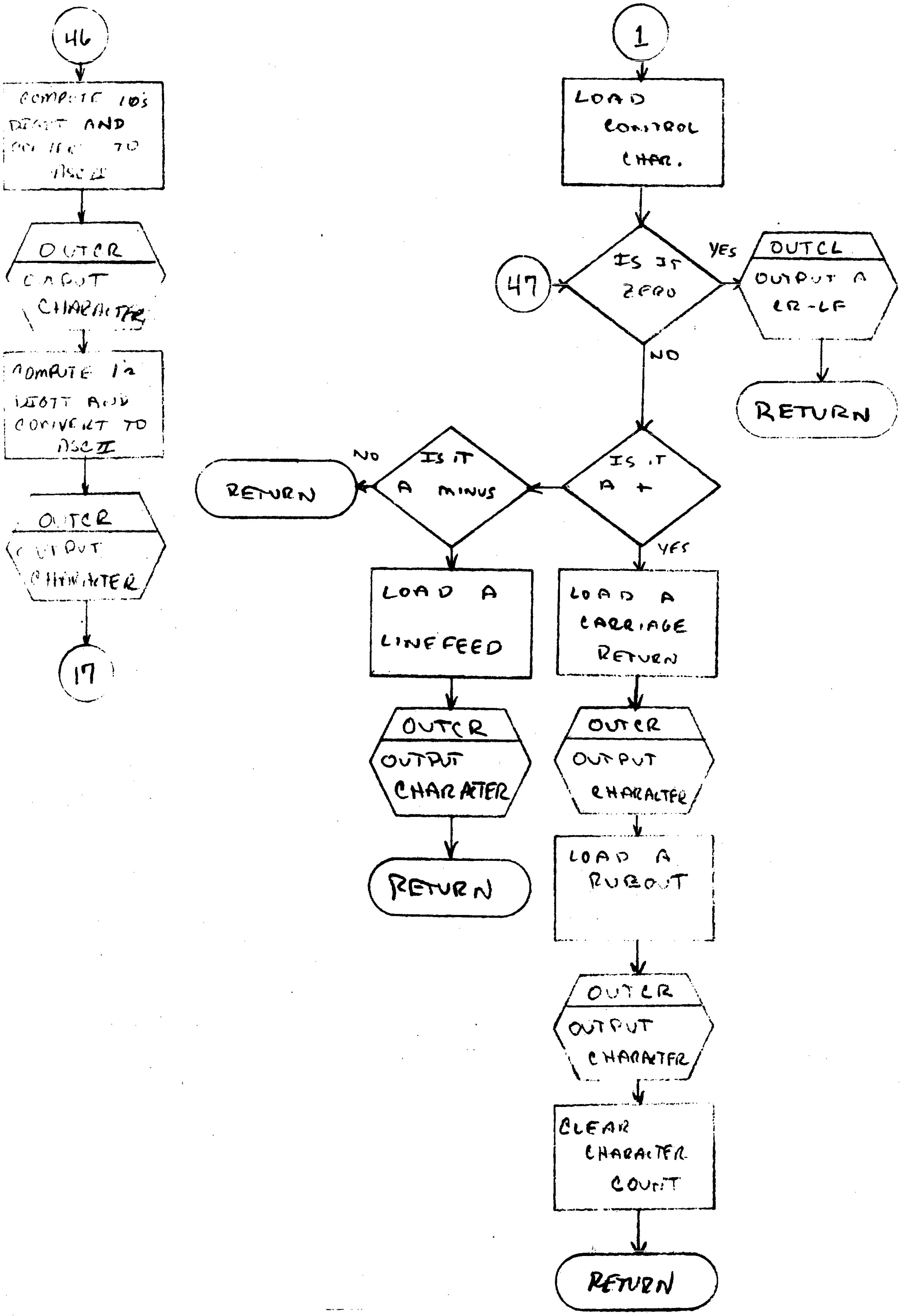


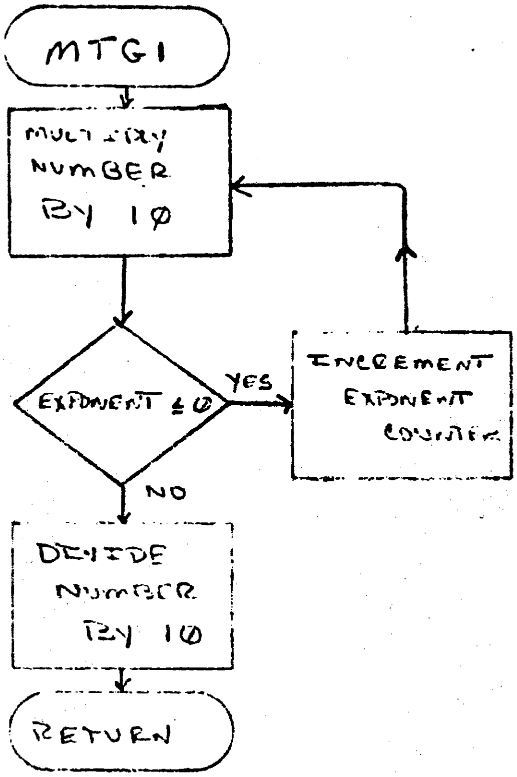

43
CLEAR FIRST LEVEL POINTER AND STRING FLAG
INCREMENT CHARACTER COUNTER AND DELIMITER POINTER
ALL CHARACTERS USED
YES
44
NO
DSRCH
(FIND NEXT DELIMITER
SET IGNORE BLANK FLAG
MCHAR
MASK OUT NEXT CHARACTER
DOES IT POINT TO A DELIMITER
42
NO
ALL CHARACTERS USED
YES
44
36
23
LOAD SAVED NUMBER
CLEAR DECIMAL EXPONENT
BINARY EXPONENT ZERO
YES
MTG1
MAKE NUMBER BETWEEN 1 AND 10
NO
DTL1
MAKE NUMBER BETWEEN .1 AND 1
SAVE SHIFT COUNT
OUTCL
OUTPUT A CR-LF
RESET NUMBER BUFFER POINTER AND SET DIGIT COUNTER TO 6
GETDG
GET NEXT DIGIT
CONVERT TO ASCII AND STORE IN BUFFER
ALL DIGITS GOT
NO
45

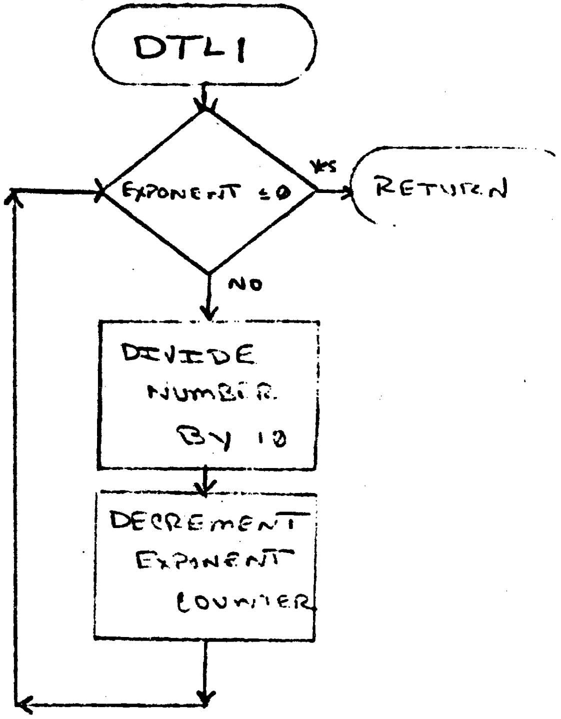
FORMATTER (21 of 25)

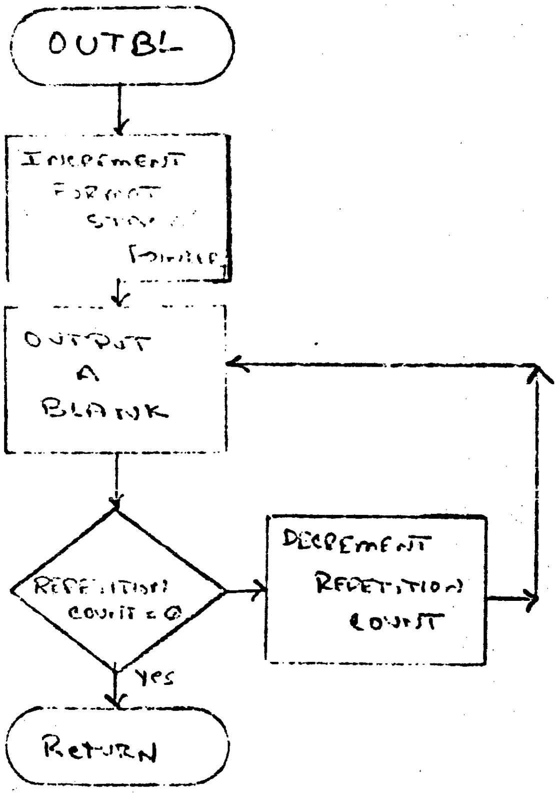


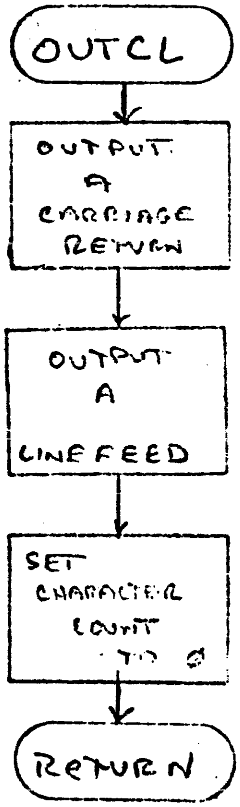
FORMATTER (22 of 25)

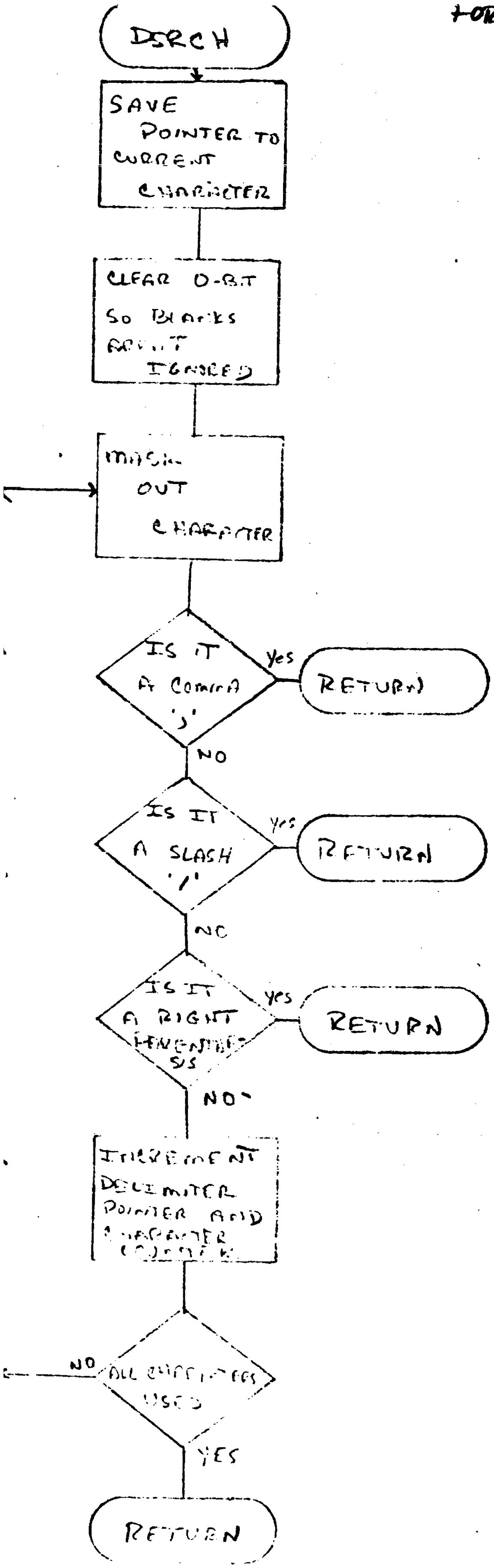


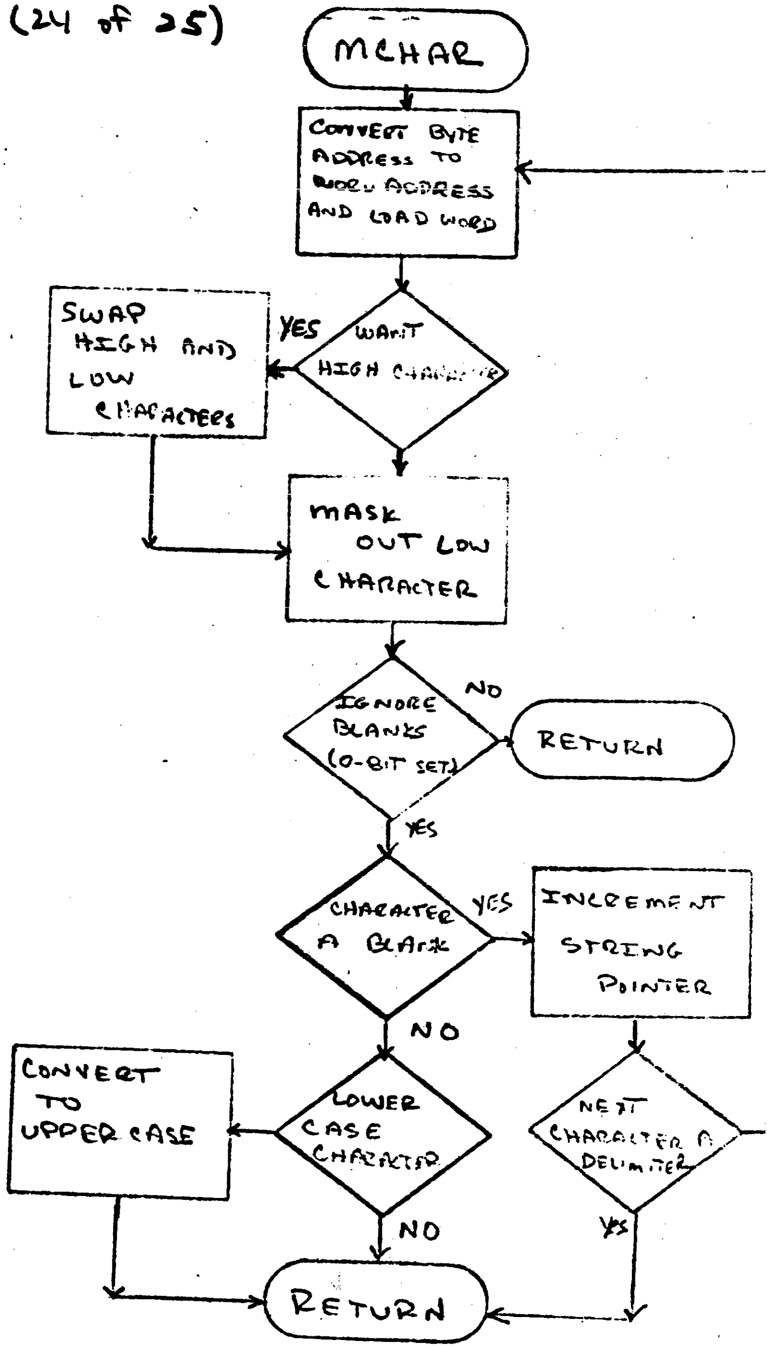

MTGI
MULTIPLY NUMBER BY 10
EXPONENT ≤ 0
YES
INCREMENT EXPONENT COUNTER
NO
DIVIDE NUMBER BY 10
RETURN

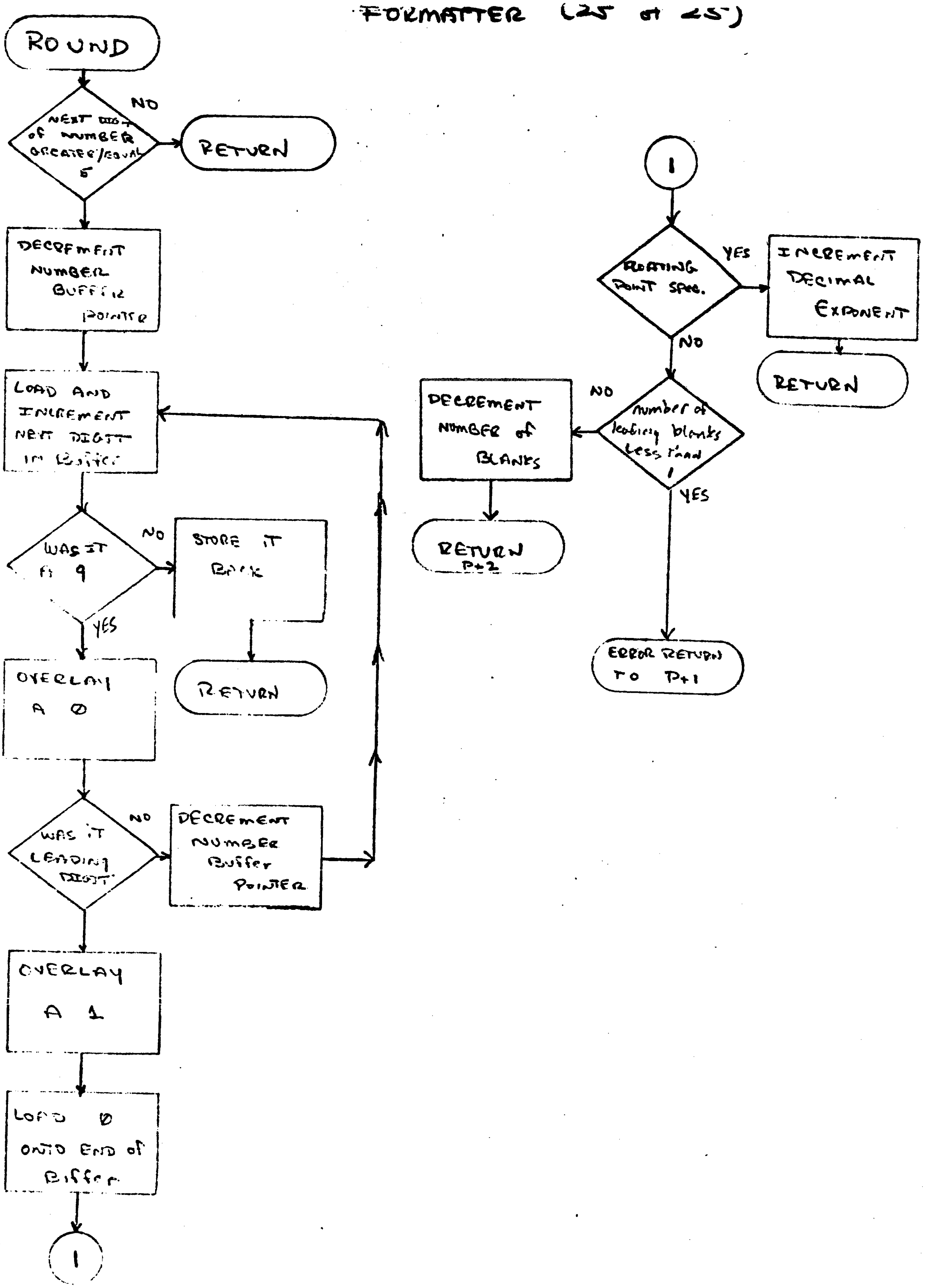

DTL1
EXPONENT ≤ 0
YES
RETURN
NO
DIVIDE NUMBER BY 10
DECREMENT EXPONENT COUNTER

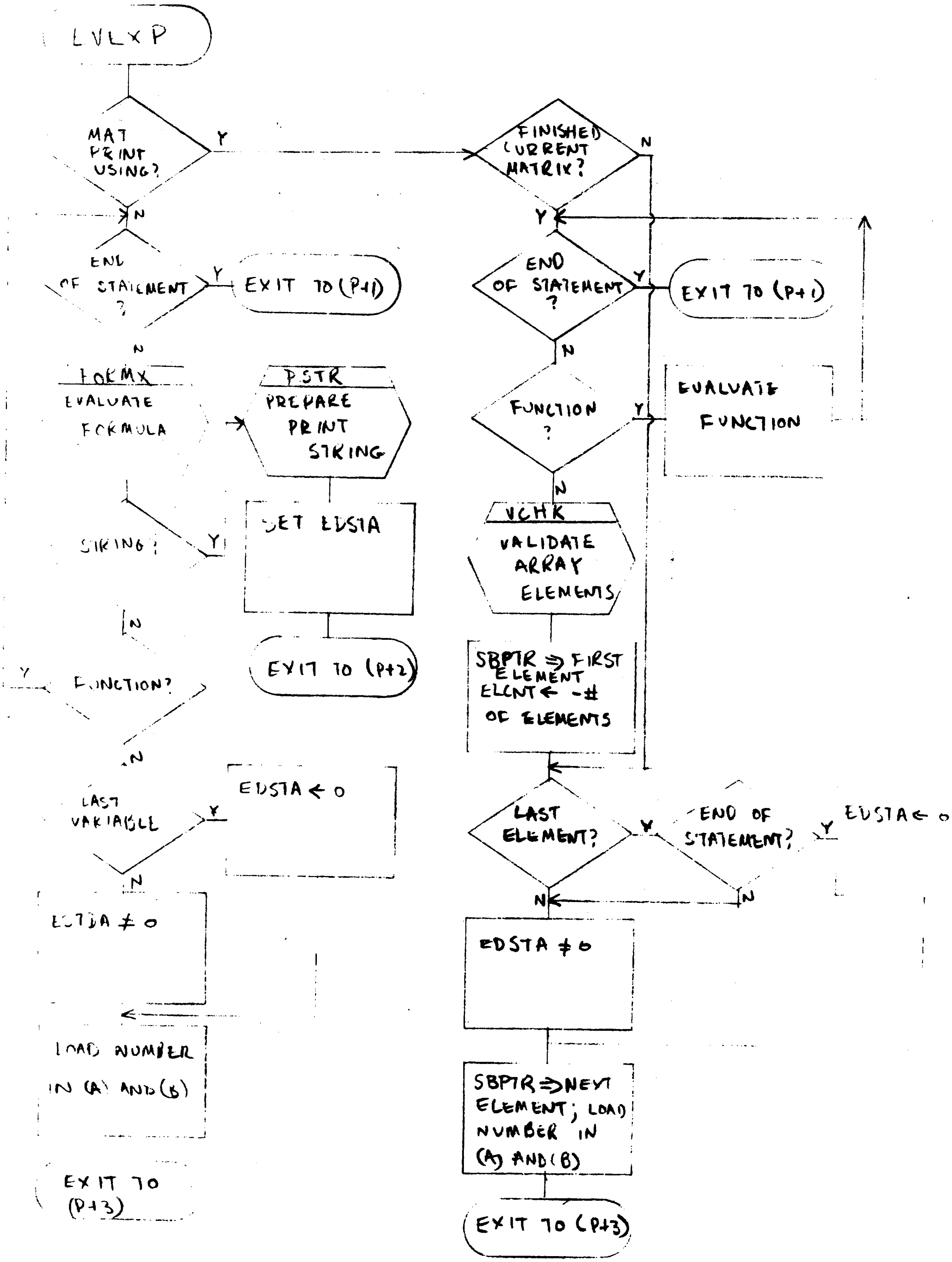

OUTBL
INCREMENT FORMAT STRING POINTER
OUTPUT A BLANK
REPETITION COUNT ≤ 0
DECREMENT REPETITION COUNT
YES
RETURN


OUTCL
OUTPUT A CARRIAGE RETURN
OUTPUT A LINEFEED
SET CHARACTER COUNT TO 0
RETURN


DSRCH
SAVE POINTER TO CURRENT CHARACTER
CLEAR O-BIT SO BLANKS AREN'T IGNORED
MASK OUT CHARACTER
IS IT A COMMA ','
YES
RETURN
NO
IS IT A SLASH '/'
YES
RETURN
NO
IS IT A RIGHT PARENTHESIS
YES
RETURN
NO
INCREMENT DELIMITER POINTER AND CHARACTER COUNTER
NO
ALL CHARACTERS USED
YES
RETURN


MCHAR
CONVERT BYTE ADDRESS TO WORD ADDRESS AND LOAD WORD
YES
WANT HIGH CHARACTER
SWAP HIGH AND LOW CHARACTERS
MASK OUT LOW CHARACTER
IGNORE BLANKS (O-BIT SET)
NO
RETURN
YES
CHARACTER A BLANK
YES
INCREMENT STRING POINTER
NO
CONVERT TO UPPER CASE
LOWER CASE CHARACTER
NEXT CHARACTER A DELIMITER
NO
YES
RETURN


ROUND
NEXT DIGIT of NUMBER GREATER/EQUAL 5
NO
RETURN
DECREMENT NUMBER BUFFER POINTER
LOAD AND INCREMENT NEXT DIGIT IN BUFFER
WAS IT A 9
NO
STORE IT BACK
RETURN
YES
OVERLAY A 0
WAS IT LEADING DIGIT
NO
DECREMENT NUMBER BUFFER POINTER
OVERLAY A 1
LOAD 0 ONTO END OF BUFFER
1
1
FLOATING POINT SPEC.
YES
INCREMENT DECIMAL EXPONENT
RETURN
NO
number of leading blanks less than 1
NO
DECREMENT NUMBER OF BLANKS
RETURN P+2
YES
ERROR RETURN TO P+1


LVLXP
MAT PRINT USING?
Y
N
END OF STATEMENT ?
Y
EXIT TO (P+1)
N
FORMX
EVALUATE FORMULA
PSTR
PREPARE PRINT STRING
STRING?
Y
SET EDSTA
N
EXIT TO (P+2)
Y
FUNCTION?
N
LAST VARIABLE
Y
EDSTA ← 0
N
EDSTA ≠ 0
LOAD NUMBER IN (A) AND (B)
EXIT TO (P+3)
FINISHED CURRENT MATRIX?
N
Y
END OF STATEMENT ?
Y
EXIT TO (P+1)
N
FUNCTION ?
Y
EVALUATE FUNCTION
N
VCHK
VALIDATE ARRAY ELEMENTS
SBPTR ⇒ FIRST ELEMENT
ELCNT ← -# OF ELEMENTS
LAST ELEMENT?
Y
END OF STATEMENT?
Y
EDSTA ← 0
N
N
EDSTA ≠ 0
SBPTR ⇒ NEXT ELEMENT; LOAD NUMBER IN (A) AND (B)
EXIT TO (P+3)

2000C Loader

The 2000C TSB Loader is a separate program which runs on the system computer. It is explicitly loaded by the operator to perform the following functions:

1) Generate a new system from paper tapes.

2) Update an existing disc resident system using new paper tapes.

3) Reload a system from magnetic tape.

4) Restore the drum resident bootstrap routines.

5) Resuscitate a system that has crashed because of certain hardware or software failures.

The loader is implicitly loaded when the operator requests any of the following functions:

6) A normal load from disc of a slept system.

7) A selective load.

8) A selective dump.

9) A sleep or hibernate.

In addition, the loader contains the moving head disc driver for the system, which remains in core at all times.

The operation of the loader for each of these uses may be understood by using the following brief descriptions, the following flowcharts, and the listing.

1) The loader generates the system tables by asking the configuration option questions. Discs are checked for labels and the drum resident tables are written out. The first system record is read, and, using it, enough disc space is reserved for the system. The remainder of the system is read from paper tape and written to disc.

The system library routines are handled by the SYSLB routine, which puts them on the disc. The remainder of the system is written out afterwords. The user swap areas are initialized and the DATE-TIME sequence is entered.

2) Paper tape update uses only the paper tape load section of the load sequence. It does not change any disc area other than the system code.

3) Magnetic tape load is similar to paper tape load except that certain tables are read from tape rather than being initialized. Also, after the system has been written to disc, the mag tape loading section is entered to load the users library.

4) The loader may be entered at 14000B to restore the bootstrap loaders to the drum. The bootstrap sequence is entered after they have been written.

5) When the system tables are intact (both in core and on the drum) but the system cannot be slept normally, the loader may be loaded and started at 3000B to force the sleep procedure to occur.

6) The bootstrap procedure calls in the loader which copies system information to the drum, reads in the system, and enters the DATE-TIME sequence.

7) If the selective load option is selected while loading, the mag tape load section performs the load and continues the load process as if it were a reload from mag tape.

8) Selective dump uses the mag tape dump section of the loader to generate a tape containing specified user library entries.

9) The sleep process first copies all of the system tables and sanctified files from the drum and core to the disc. Then, if sleep or dump has been specified, the system and system tables are dumped in their entirety to mage tape and the mag tape dump section is entered to dump the user library. If sleep, only entries changed since the last hibernate are dumped. If hibernate, all entries are dumped.

A different version of the 2000C Loader exists for each different type of moving head disc supported. Except for the disc driver and several configuration options default values, these loaders are identical.

This section contains brief descriptions of minor routines in the loader and flowcharts for the more complicated ones.

TTY 35

The loader teletype driver is a non-interrupt driver which provides output and input capabilities on the console (usually an ASR 35). On output requests, printing is completed before control is returned to the caller. On input requests, control is not returned until the operator has typed a carriage return. Character backup (using the left arrow) and line cancellation (using the escape key) are handled internally to the driver.

DRUMP

The loader drum driver is a non-interrupt driver which provides output and input capabilities on drums. The driver decodes the specified drum address into the proper select code, track, and sector numbers using the ?TBL part of the equipment table. Any drum errors are retried by the driver up to 100 times. Continued failure causes a halt (if DISC8 is zero) or a skip return (if DISC8 is an RSS) with an error message printed.

DISCZ

This routine provides the necessary environment for the moving head disk driver and transfers input/output/seek requests to the actual driver.

Since the interrupt system must be enabled for the disk driver, this is done in DISCZ.

In conjunction with the GMQBD routine, DISCZ provides a buffer for disk driver generated error messages and prints any such errors that occur.

JDSE

This short routine sets up all interrupt locations associated with moving head disks to call the driver interrupt processor.

SELCP

Enter with the first character in (A). routine finds a two digit octal number in the range [SELCD,I, SELCD+1,I]. errors are printed, exit to P+3. otherwise return to P+4 with the integer in (B) and the next character in (A).

GTDNO

This routine searches the input record for "-" followed by a disk number followed by a comma. If A = -1 when called, a check is made to ensure that the specified unit exists. Errors are printed, exit to P+1. Otherwise, return with number in (A) to P+2.

SETDS

The SETDS routine updates the drum table and trax tables given the logical drum number and the new select code.

STDIS

This routine updates the disk table and interrupt locations given the logical disk number and the new select code and unit number.

GTTRK

This routine searches the ADT for a full drum track. The starting location is given in (A) (0 sez full search). No find causes an error (insufficient drum space) and loading is terminated. The entry is removed from the ADT.

FAPD

This routine searches the ADT for a piece of drum space of specified size. No find causes no skip return. On find, the space is removed and the return is single skip.

FNZSC

This routine advances TEMP3 to the next entry in the disk EQT that contains a valid disk. No find sets pointer to first entry, no skip return. A find sets other pointers and gives a single skip return.

RDLBL and WTLBL

These routines calculate the address of a specified disk label and call DISCZ to read or write the label.

SDADT

This routine is called by FSDAD to remove space from the ADT in core.

CDATE

This routine is used to allocate disk space for the tables. 8K is allocated for each track of a table.

GNDAT

GNDAT is used when system space is being allocated. It writes out the current disk ADT and reads in another. If no other exists, exit is to the out of storage error and loading is terminated.

BUADT

BUADT uses the trax table to generate the ADT for all of the drum space.

RTADT

This routine is used by CLDT to return any unused quarter tracks to the in core ADT.

RCOP

RCOP asks "configuration options" and waits for an answer of yes or no. The variable COFLG is set to 0 for no and non-zero for yes. Return is single skip for yes.

RQINT

This routine packs a one or two digit decimal integer in a specified range. An invalid input prints "ILLEGAL INPUT" and returns to P+1. A simple CR returns to P+2. Otherwise, return to P+3 with the integer in (A).

CFFW

CFFW converts the first 4 words of a directory entry (ID and NAME) into printable ascii format in a specified buffer.

GMQBD

This routine provides a placebo for the disk driver when it asks for an error message buffer. Either this routine or DISCZ will print out the message.

WDLTD

This routine asks the question: "DISK OPERATING SYSTEM PRESENT?" (unless no configuration options) and uses this bit of information to place the two drum bootstraps onto the drum. If a properly configures DOS exists, its bootstrap will not be affected; however, it is still the operator's responsibility to lock any other tracks used by DOS.

WDLTE

This routine writes the final disk bootstrap onto each of the moving head disks.

BSBSO

BSBSO is the standard RTE, DOS and TSB bootstrap for drum sector zero. If switch Ø is up when it is run, it reads in sector 2, otherwise sector 1, and jumps to the first location of that sector.

BSLDR

BSLDR is the TSB bootstrap that resides on sector 1 (sector 2, also, if no DØS). It reads the final disk bootstrap from block 1 of the moving head disk in sel code 17 unit Ø. The code in this loader was originally meant to be read in at an origin of 100B. (BSORG EQU 100B). Later versions of the loader load it at 14600B but the special construction remains.

RESU

This section of code is used for the resusitation process. It reads in the system segment table and the DAT and calls the sleep section.

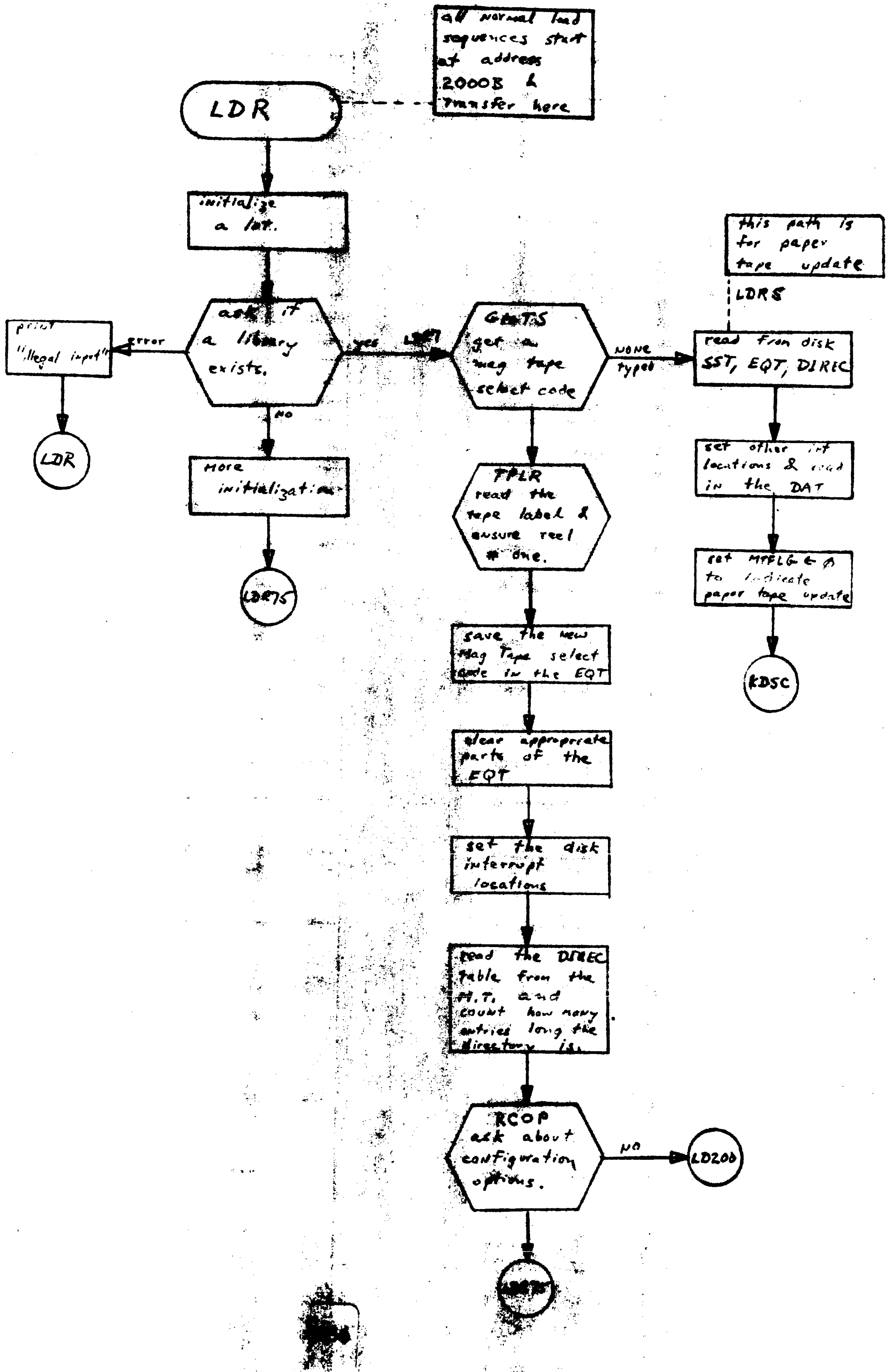
EOTCH

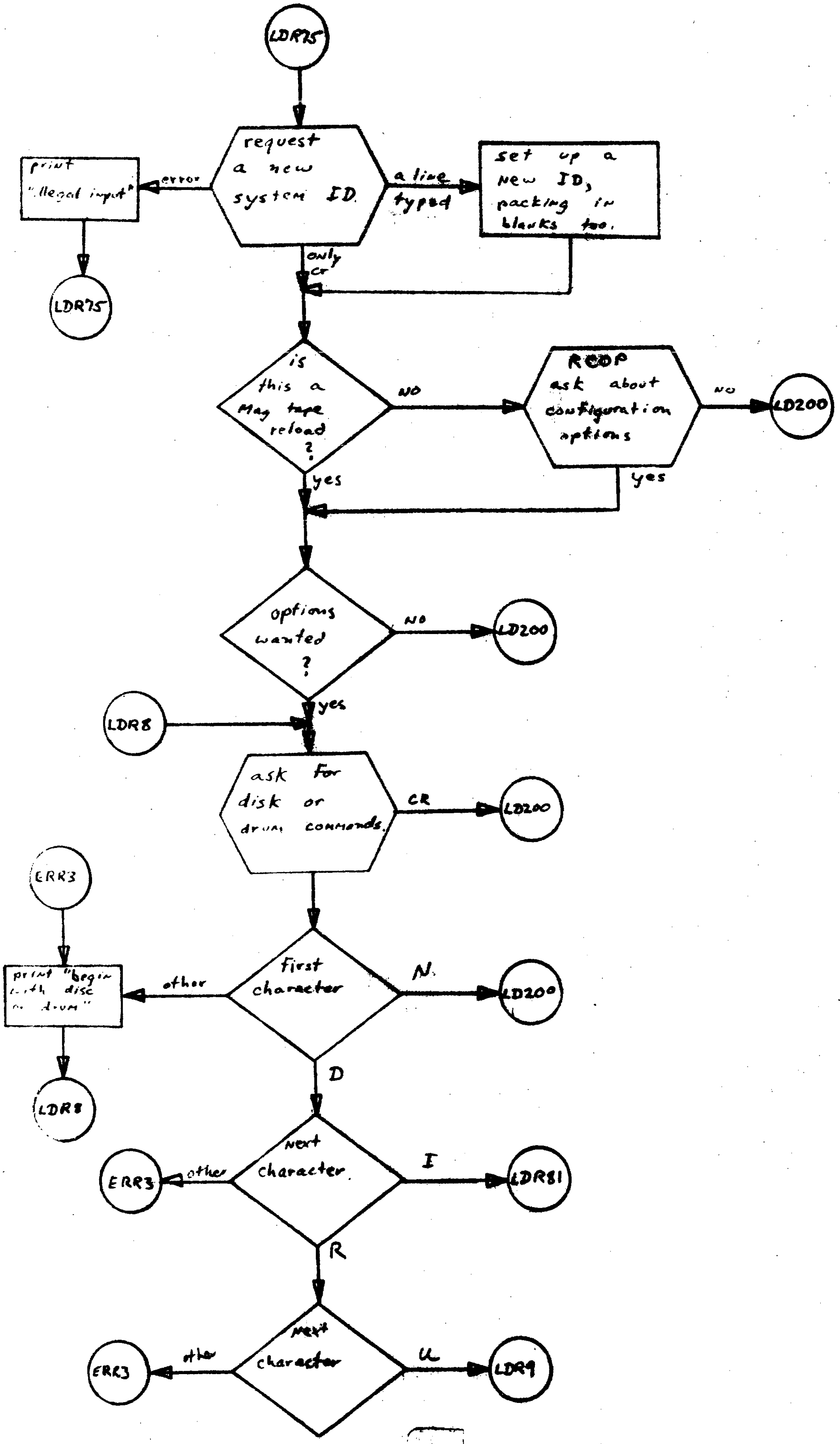
This routine checks for an end of tape (during Mag Tape Sleep operations). It is called when end of tape is not allowable (before first file mark). An end of tape prints a tape too short message and halts. Pressing run restarts the dump.

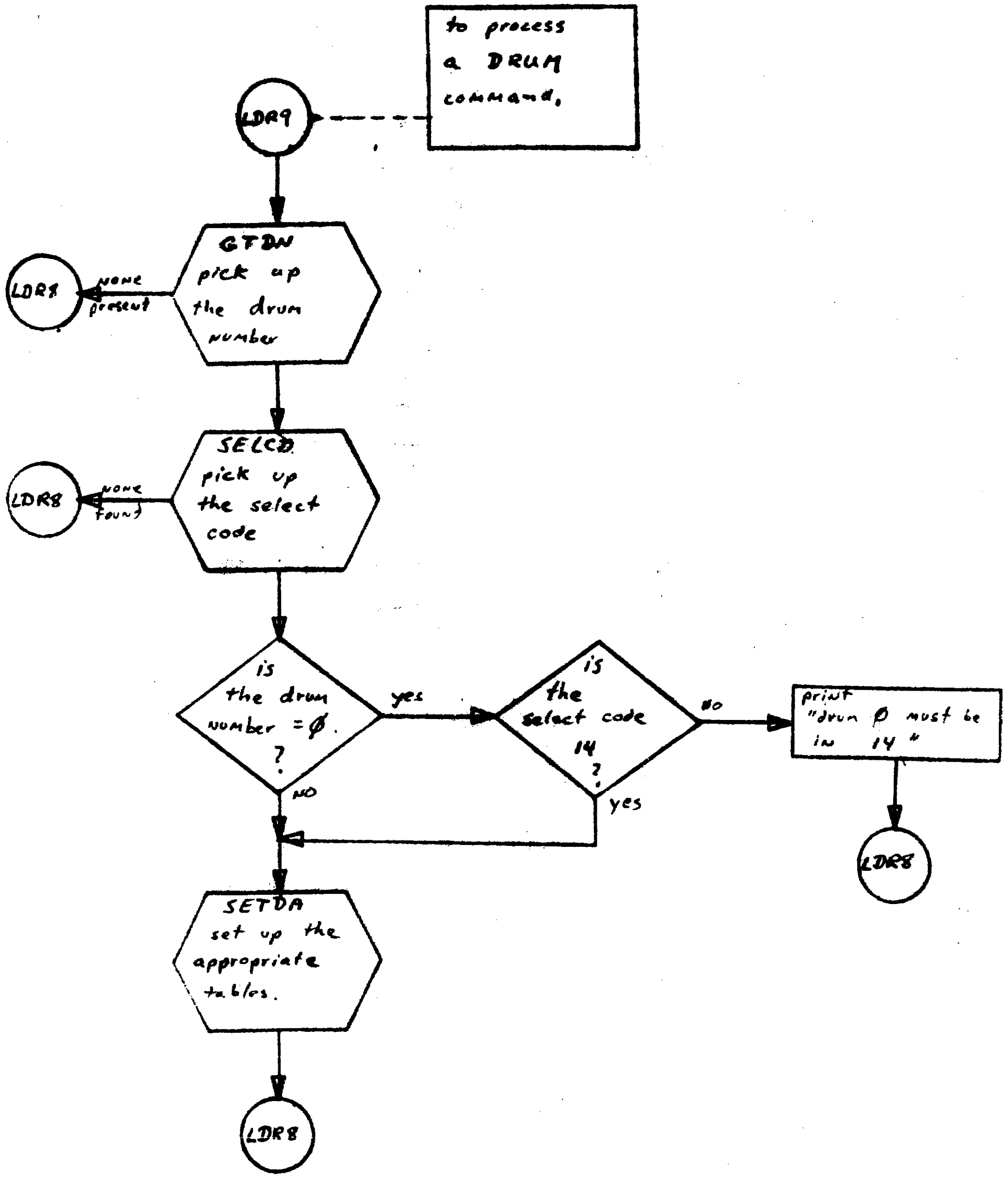
DREN - ENSU

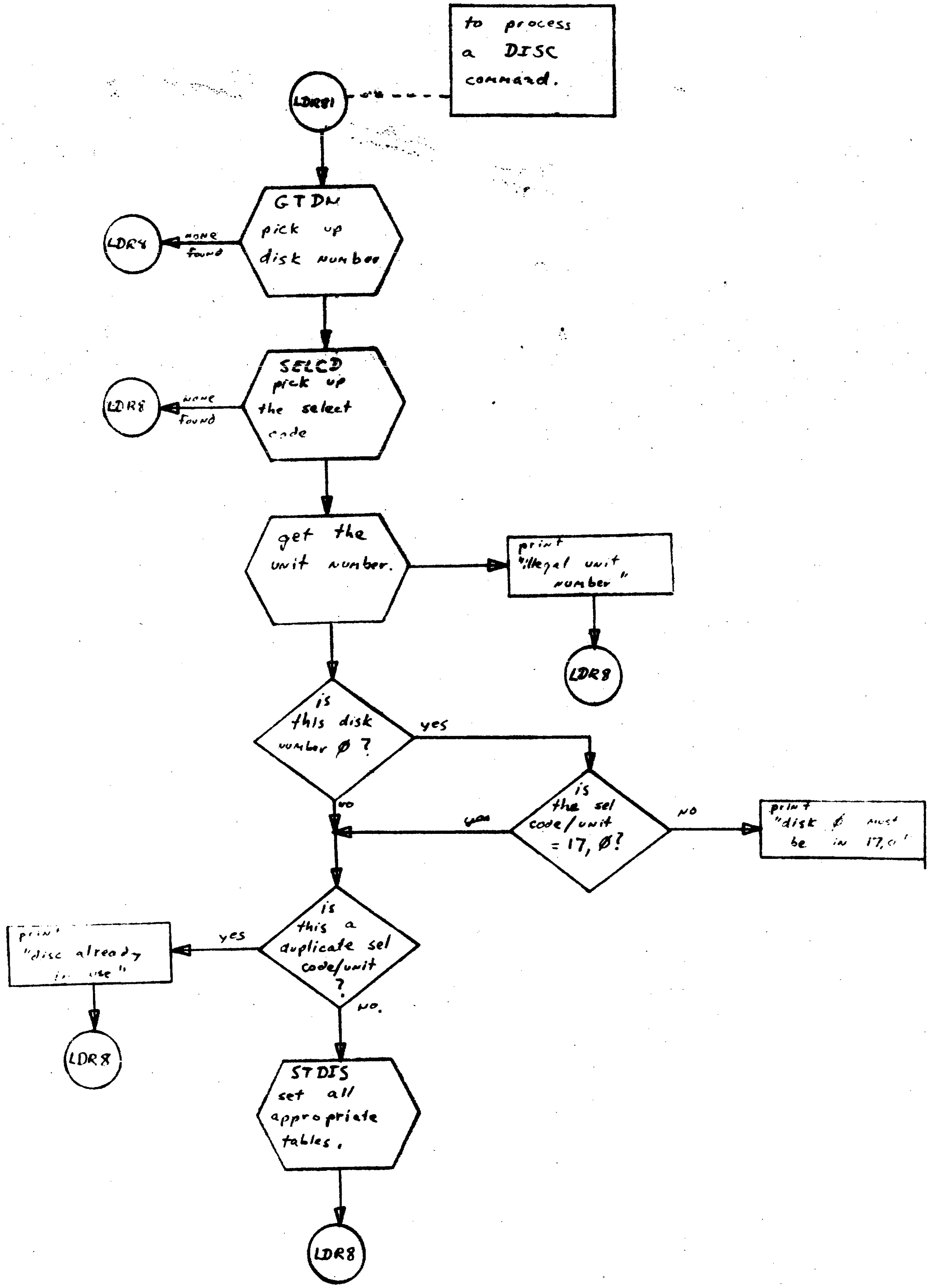
The DREN routine is used to insert a 12 word directory entry into the directory. If the proper directory track is full, ENSU is called to redistribute the dir. tracks. See the system documentation for supersave for a description of its operation.

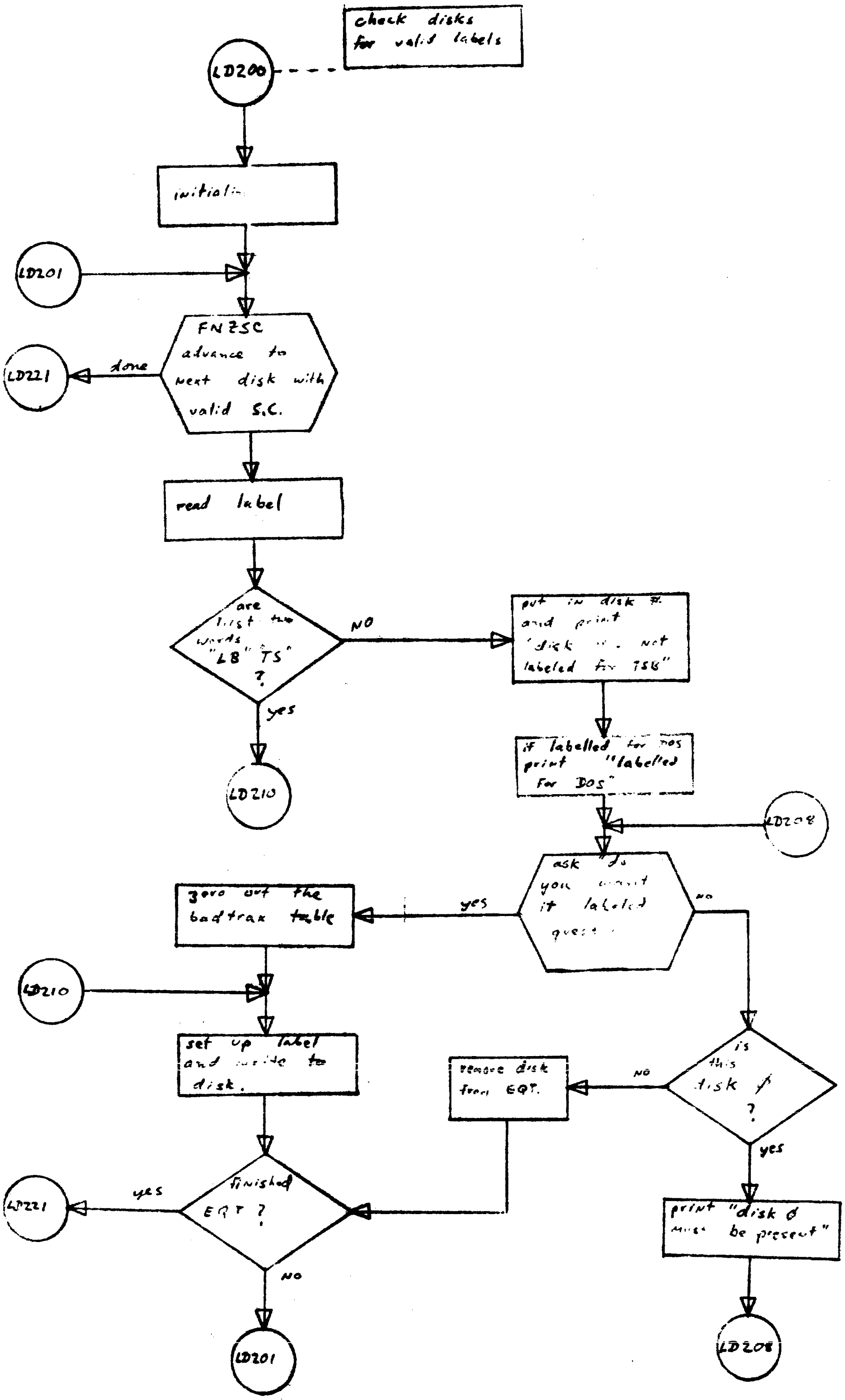
MLS

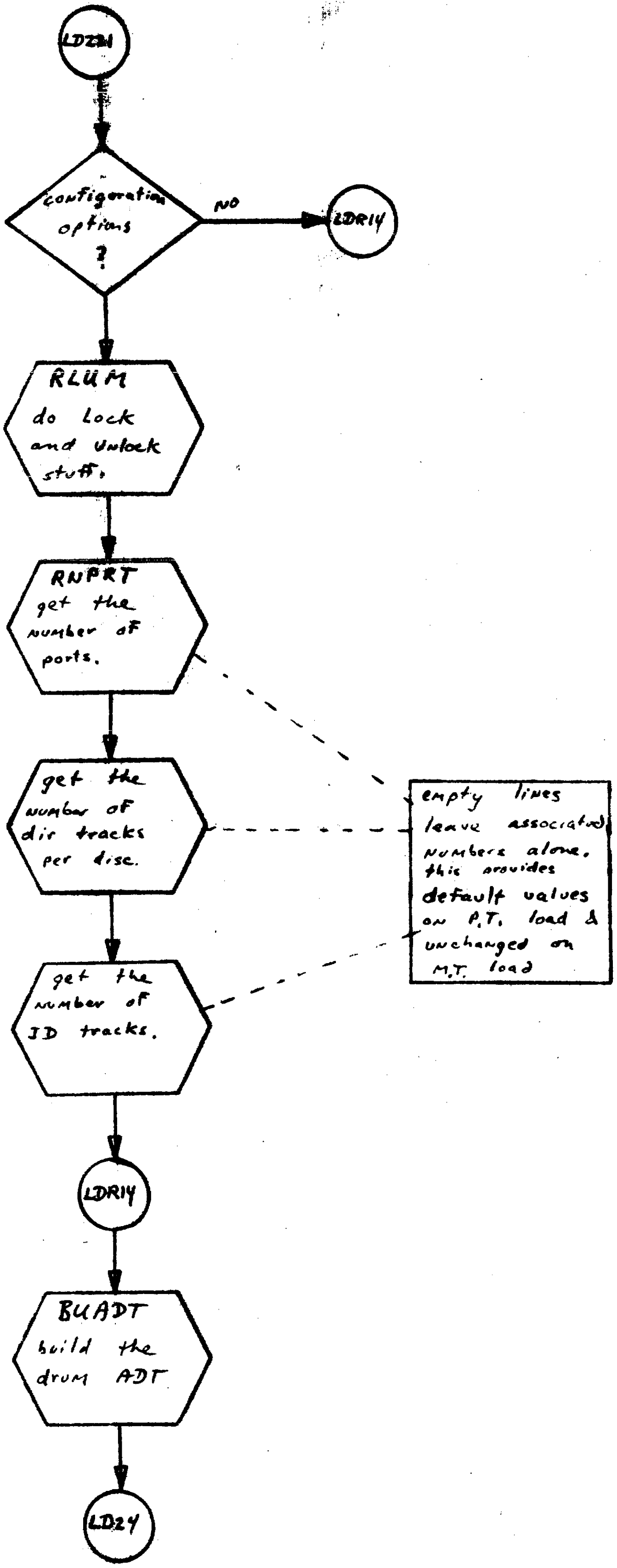

all normal load sequences start at address 2000B & transfer here
LDR
initialize a lot.
ask if a library exists.
error
print "illegal input"
LDR
yes
LDR1
NO
more initialization
LDR75
GETTS get a mag tape select code
NONE typed
this path is for paper tape update
LDRS
read from disk SST, EQT, DIREC
set other int locations & read in the DAT
set MTFLG ← 0 to indicate paper tape update
KDSC
TPLR read the tape label & ensure reel # one.
save the new Mag Tape select code in the EQT
clear appropriate parts of the EQT
set the disk interrupt locations
read the DIREC table from the M.T. and count how many entries long the Directory is.
RCOP ask about configuration options.
NO
LD200


LDR75
request a new system ID.
error
print "illegal input"
LDR75
a line typed
set up a new ID, packing in blanks too.
only cr
is this a Mag tape reload ?
NO
RCOP ask about configuration options
NO
LD200
yes
yes
options wanted ?
NO
LD200
yes
LDR8
ask for disk or drum commands.
CR
LD200
ERR3
print "begin with disc or drum"
LDR8
other
first character
N.
LD200
D
ERR3
other
next character.
I
LDR81
R
ERR3
other
next character
U
LDR9

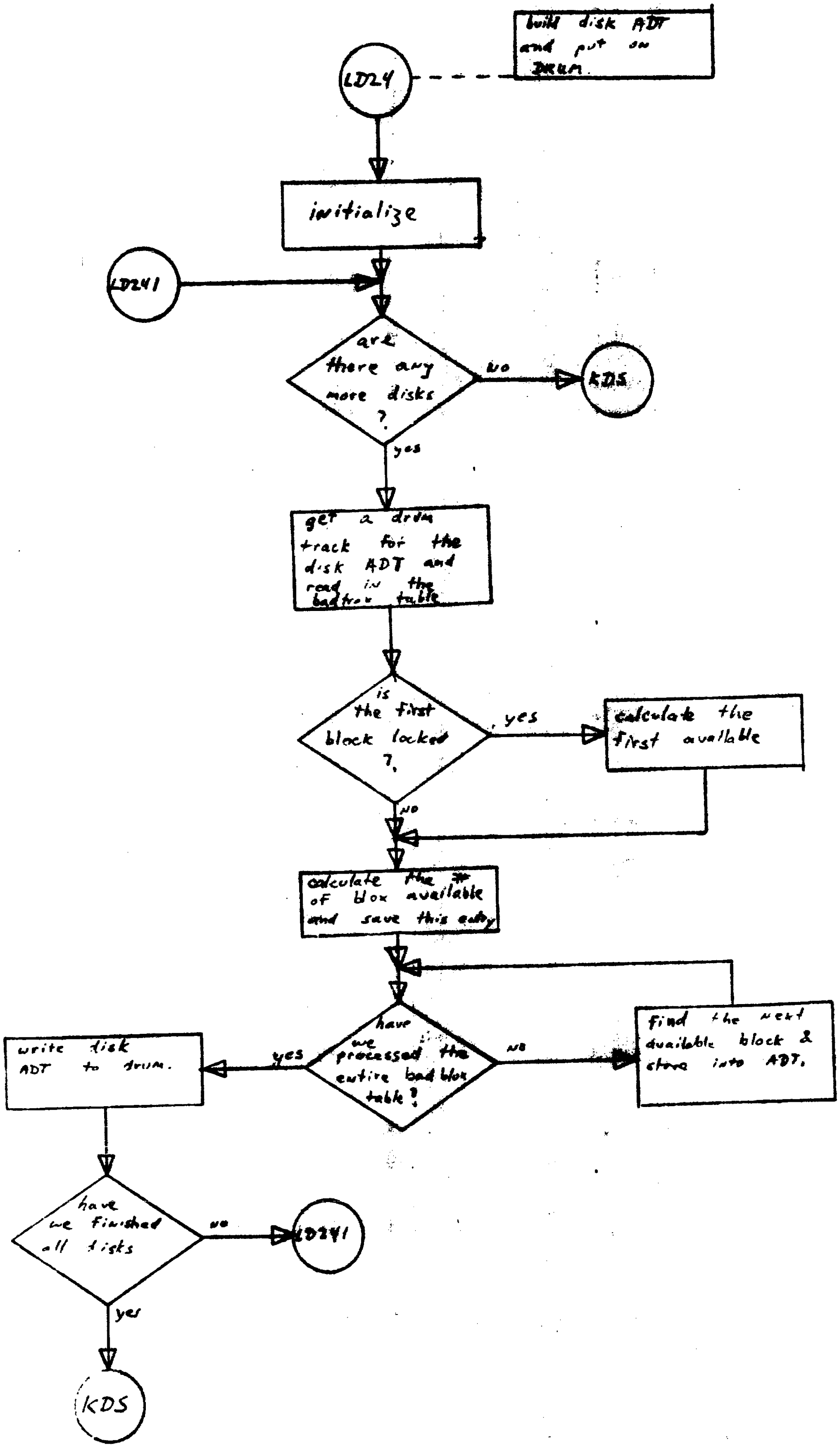

to process
a DRUM
command,
LDR9
GTDN
pick up
the drum
number
none
present
LDR8
SELCD
pick up
the select
code
none
found
LDR8
is
the drum
number = ∅.
?
yes
NO
is
the
select code
14
?
yes
no
print
"drum ∅ must be
in 14"
LDR8
SETDA
set up the
appropriate
tables.
LDR8

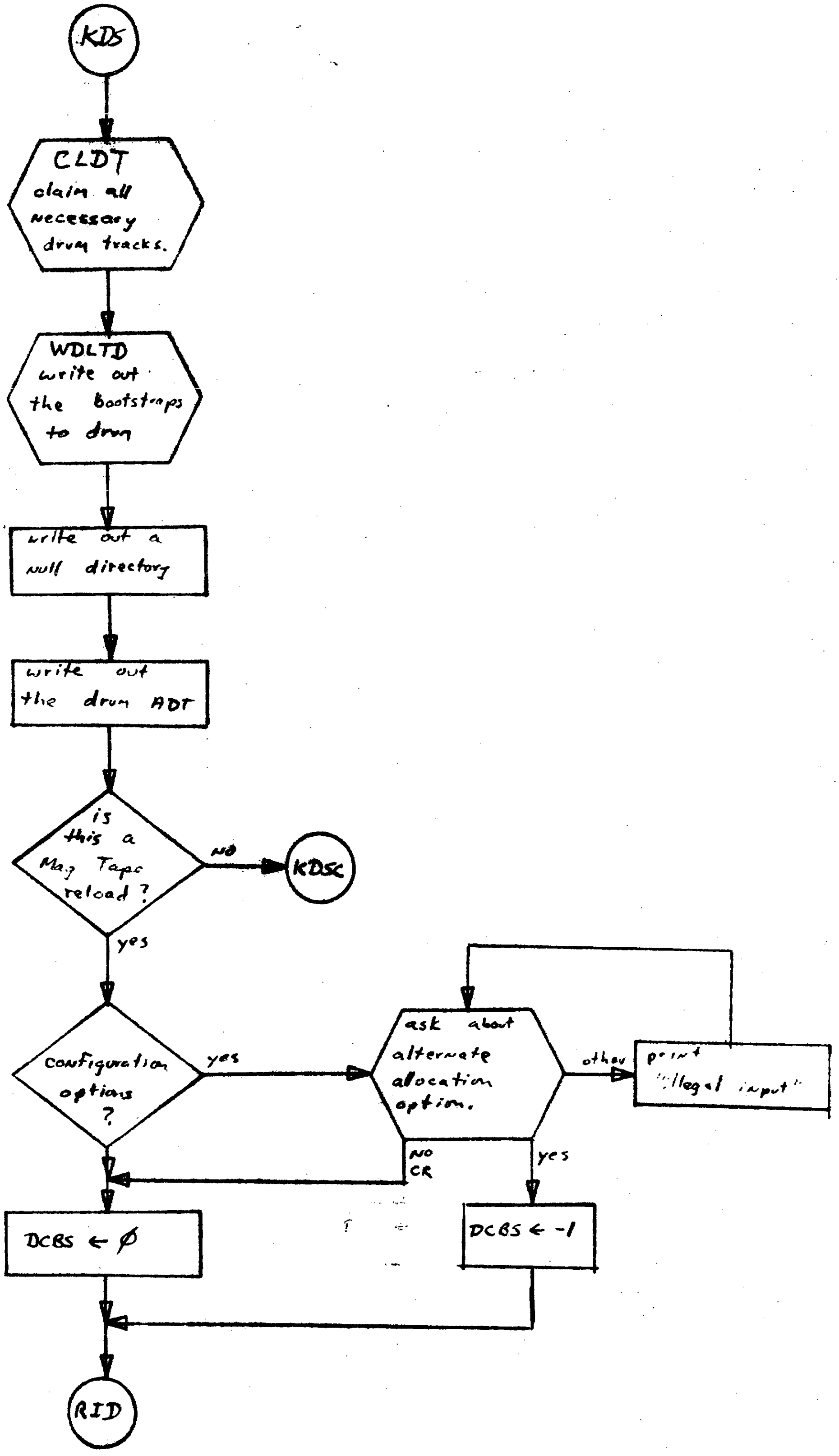

to process a DISC command.
LDR81
GTDN pick up disk number
none found
LDR8
SELCD pick up the select code
none found
LDR8
get the unit number.
print "illegal unit number"
LDR8
is this disk number Ø ?
yes
no
is the sel code/unit = 17, Ø?
yes
no
print "disk Ø must be in 17,0"
is this a duplicate sel code/unit ?
yes
no.
print "disc already in use"
LDR8
STDIS set all appropriate tables.
LDR8


check disks for valid labels
LD200
initialize
LD201
FNZSC advance to next disk with valid S.C.
done
LD221
read label
are first two words "LB" "TS" ?
NO
yes
LD210
put in disk #. and print "disk ... not labeled for TSB"
if labelled for DOS print "labelled for DOS"
LD208
ask "do you want it labeled question
yes
no
zero out the badtrax table
LD210
set up label and write to disk.
is this disk Ø ?
NO
yes
remove disk from EQT.
print "disk Ø must be present"
LD208
finished EQT ?
yes
LD221
NO
LD201

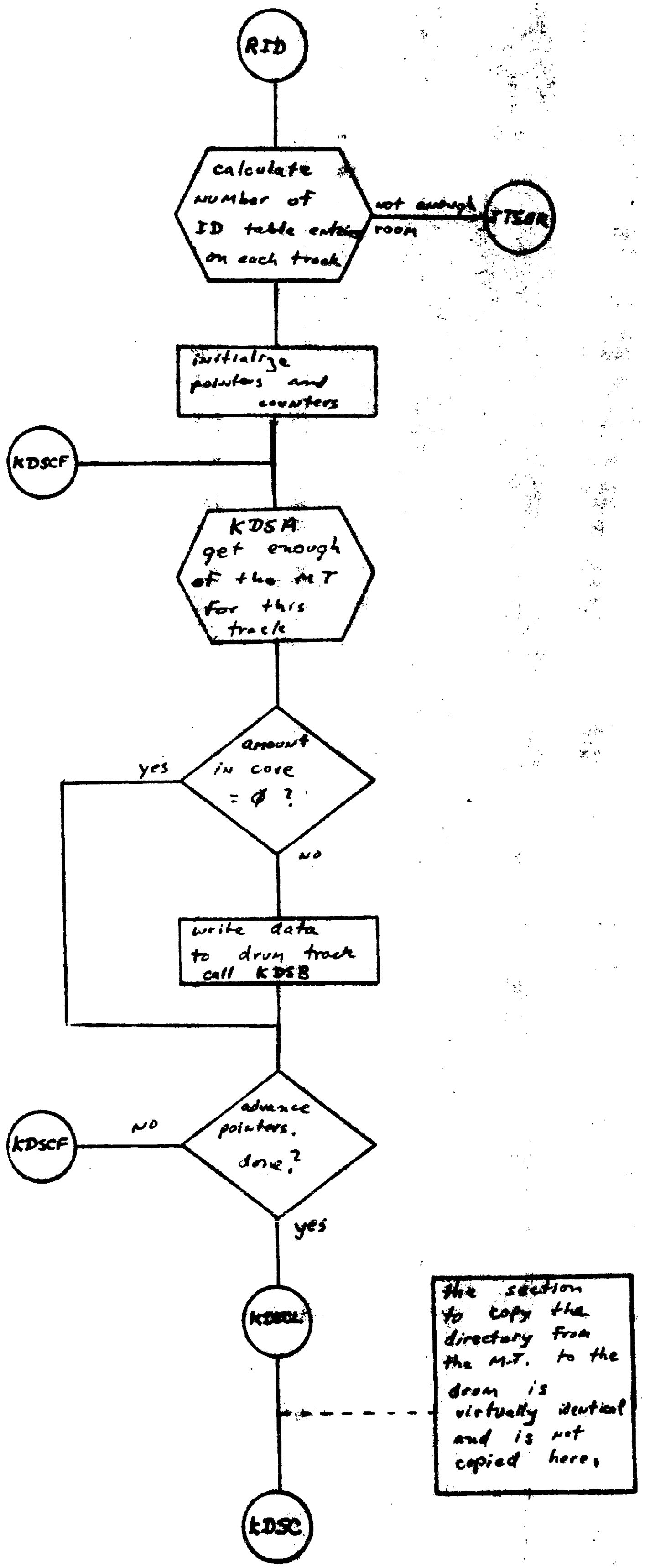

LD2N
configuration options ?
NO
LDR14
RLUM
do Lock and Unlock stuff.
RNPRT
get the number of ports.
get the number of dir tracks per disc.
get the number of ID tracks.
empty lines leave associated numbers alone. this provides default values on P.T. load & unchanged on M.T. load
LDR14
BUADT
build the drum ADT
LD24

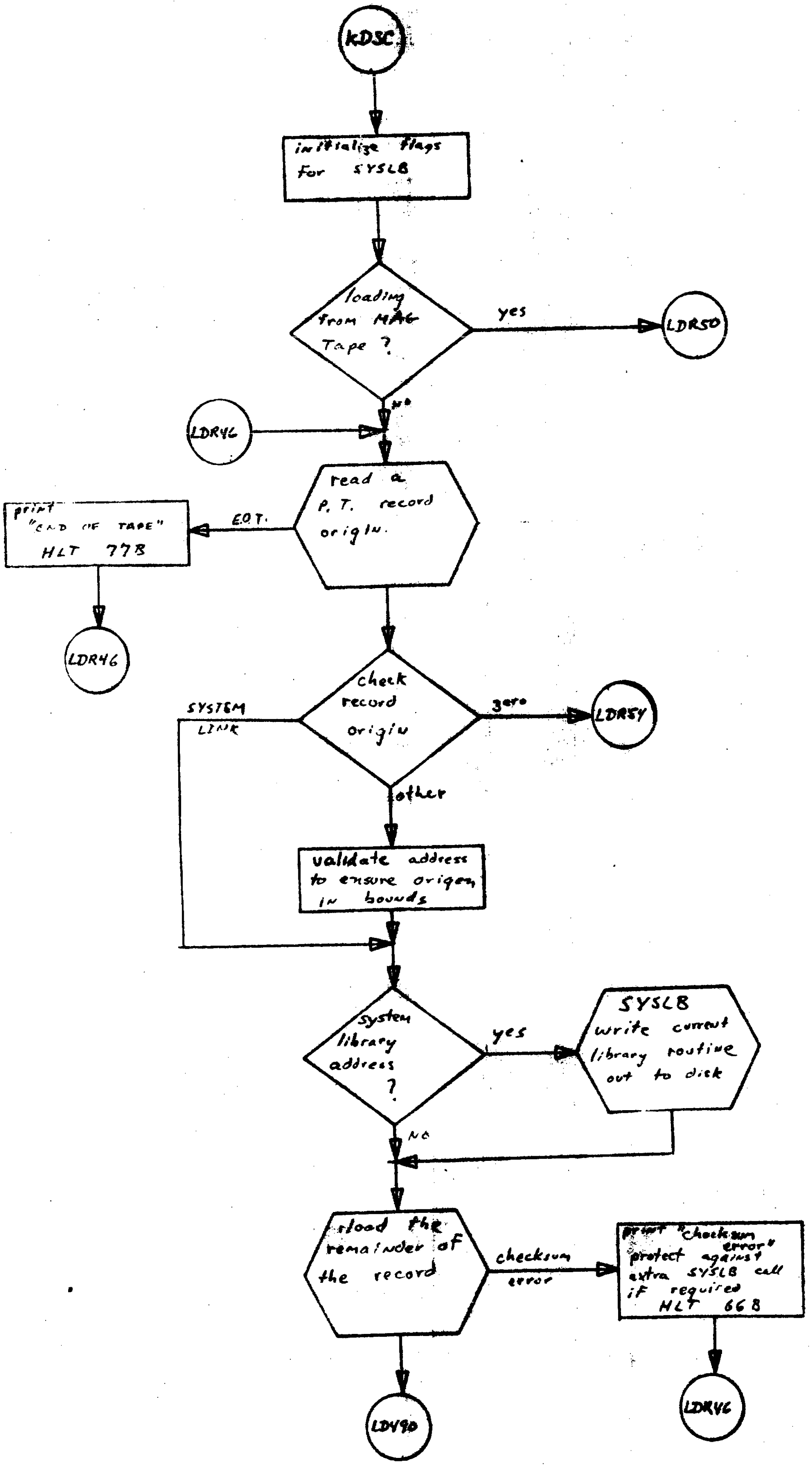
MLS


build disk ADT and put on DRUM.
LD24
initialize
LD241
are there any more disks ?
no
KDS
yes
get a drum track for the disk ADT and read in the badtrax table
is the first block locked ?
yes
calculate the first available
no
calculate the # of blox available and save this entry
have we processed the entire bad blox table ?
no
find the next available block & store into ADT.
yes
write disk ADT to drum.
have we finished all disks
no
LD241
yes
KDS

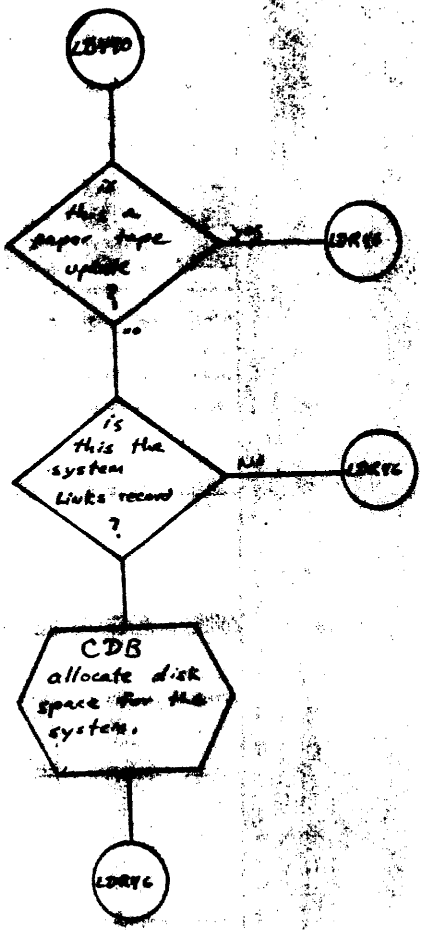

KDS
CLDT
claim all necessary drum tracks.
WDLTD
write out the Bootstraps to drum
write out a null directory
write out the drum ADT
is this a Mag Tape reload ?
no
KDSC
yes
configuration options ?
yes
ask about alternate allocation option.
other
print "illegal input"
NO CR
yes
DCBS ← ∅
DCBS ← -1
RID

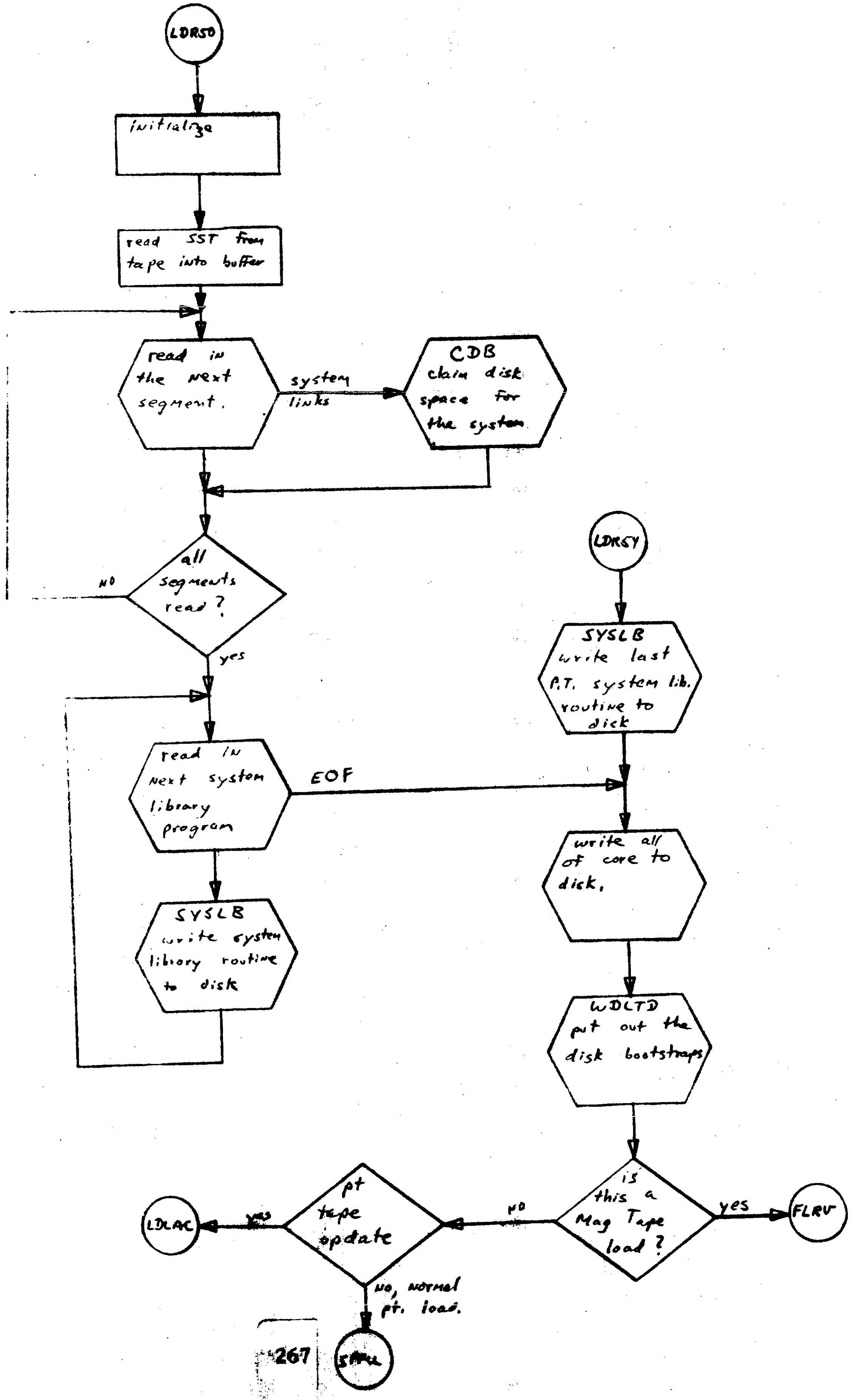
MLS


RID
calculate number of ID table entries on each track
not enough room
ITSER
initialize pointers and counters
KDSCF
KDSA get enough of the MT for this track
amount in core = Ø ?
yes
no
write data to drum track call KDSB
advance pointers, done?
no
KDSCF
yes
KDSCL
the section to copy the directory from the M.T. to the drum is virtually identical and is not copied here,
KDSC

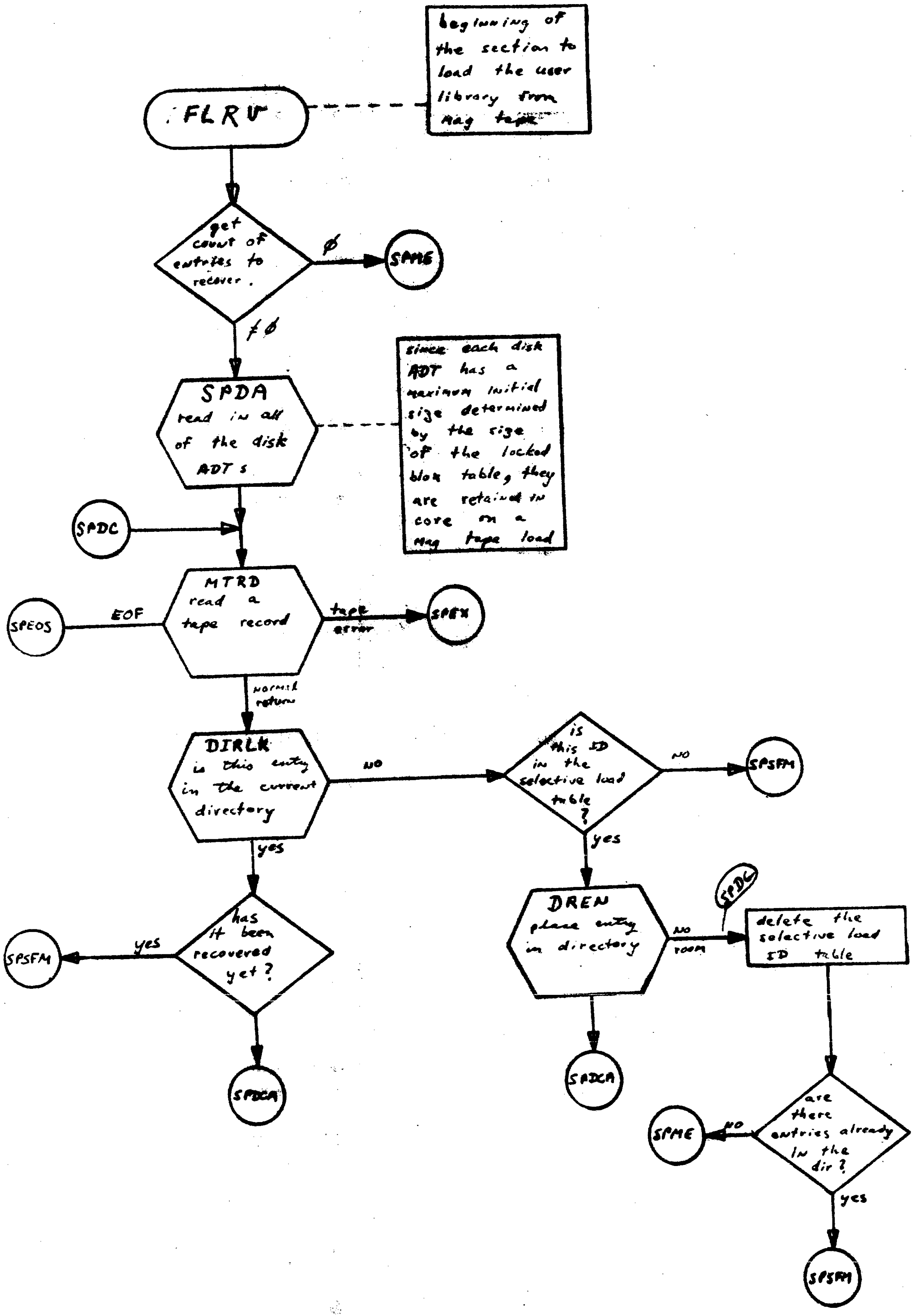

KDSC
initialize flags For SYSLB
loading from MAG Tape ?
yes
LDR50
NO
LDR46
read a P.T. record origin.
E.O.T.
print "END OF TAPE" HLT 77B
LDR46
check record origin
SYSTEM LINK
zero
LDR54
other
validate address to ensure origen in bounds
system library address ?
yes
SYSLB write current library routine out to disk
NO
load the remainder of the record
checksum error
print "checksum error" protect against extra SYSLB call if required HLT 66B
LD490
LDR46

MLS


LDR90
is this a paper tape update ?
YES
LDR96
NO
is this the system Links record ?
NO
LDR96
CDB allocate disk space for the system.
LDR96

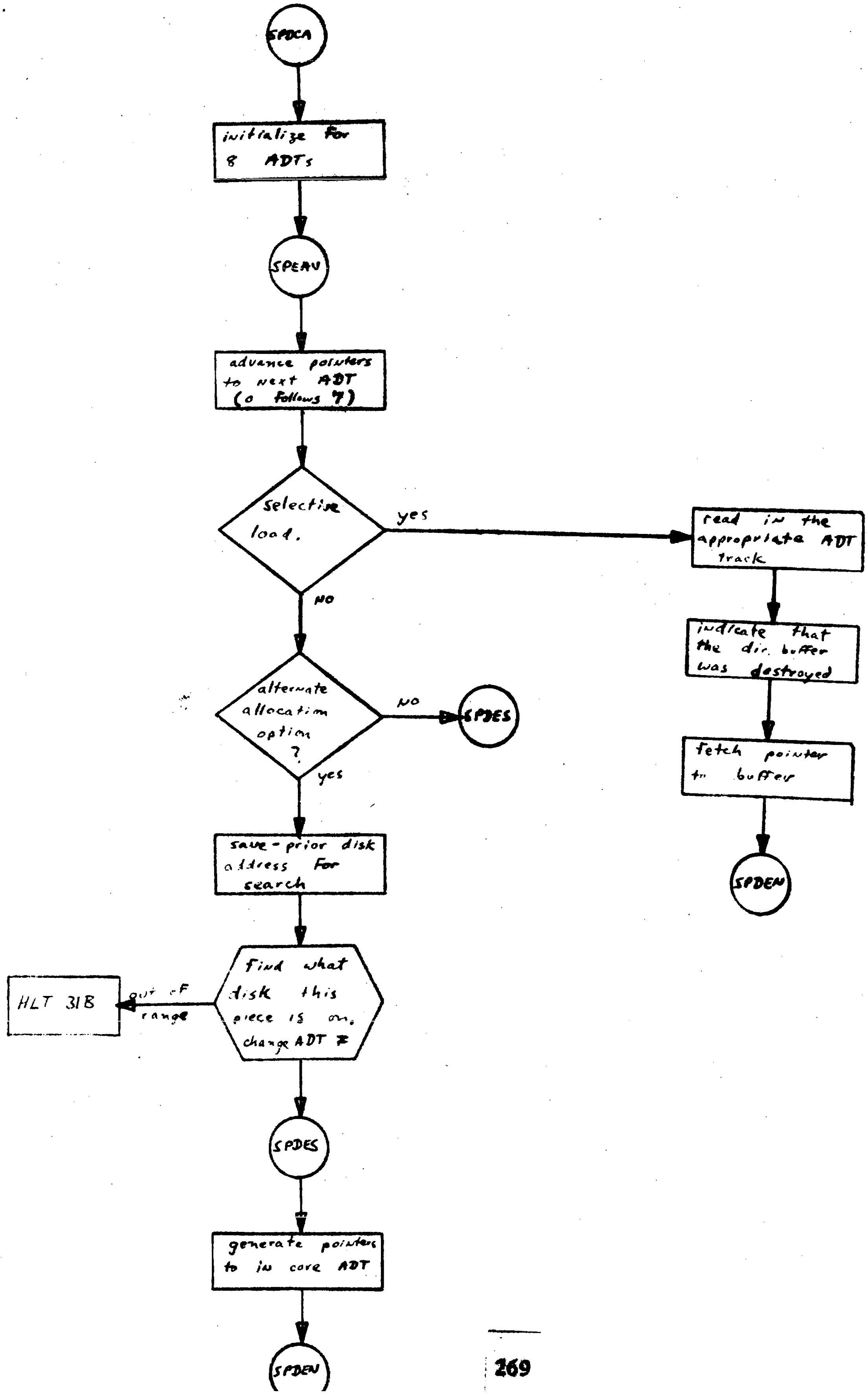

LDRSO
initialize
read SST from tape into buffer
read in the next segment.
system links
CDB claim disk space for the system
all segments read?
NO
yes
read IN next system library program
EOF
SYSLB write system library routine to disk
LDRSY
SYSLB write last P.T. system lib. routine to disk
write all of core to disk,
WDLTD put out the disk bootstraps
is this a Mag Tape load?
yes
FLRV
NO
pt tape update
yes
LDLAC
no, normal pt. load.
SMPL

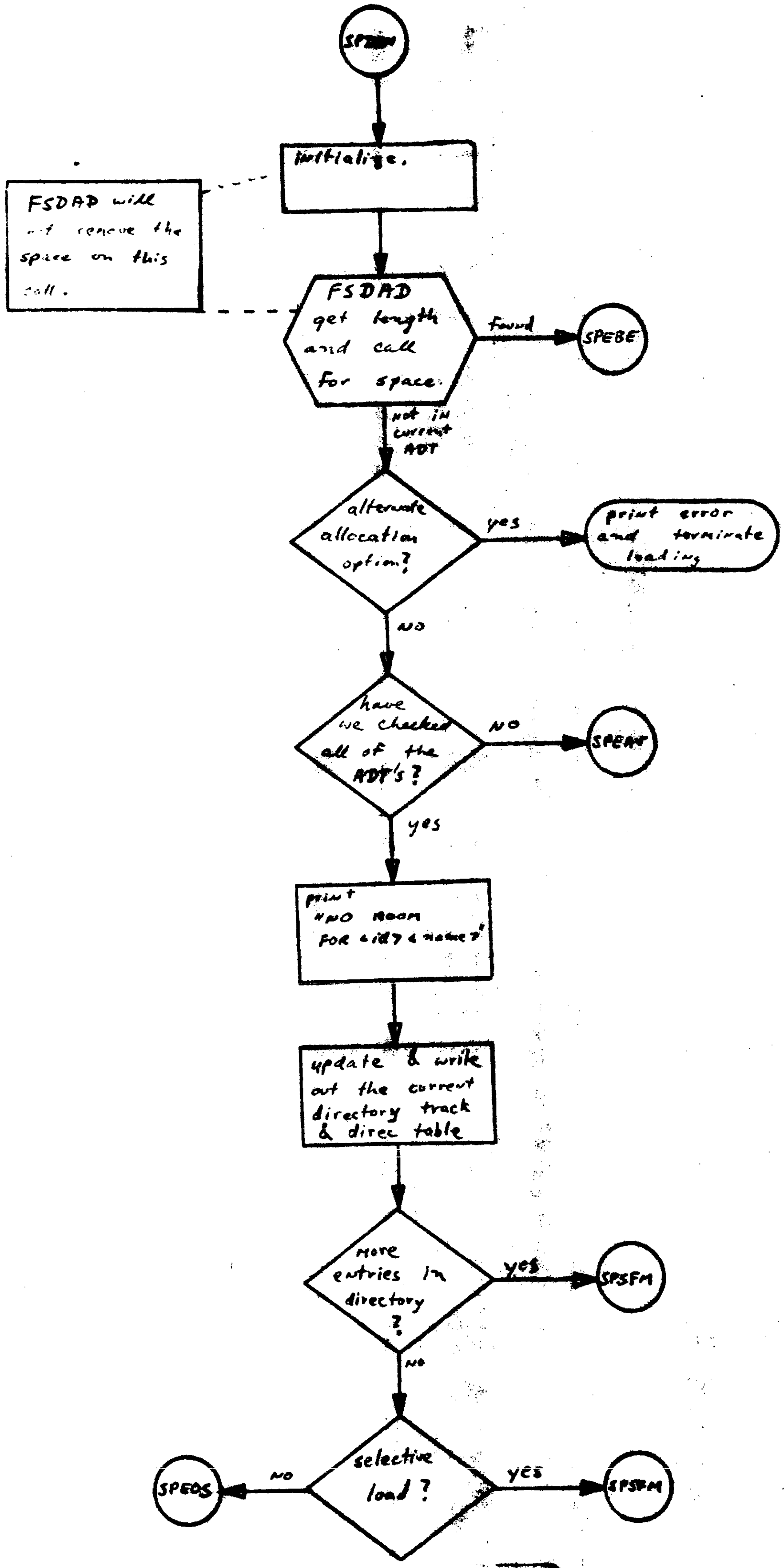
MLS



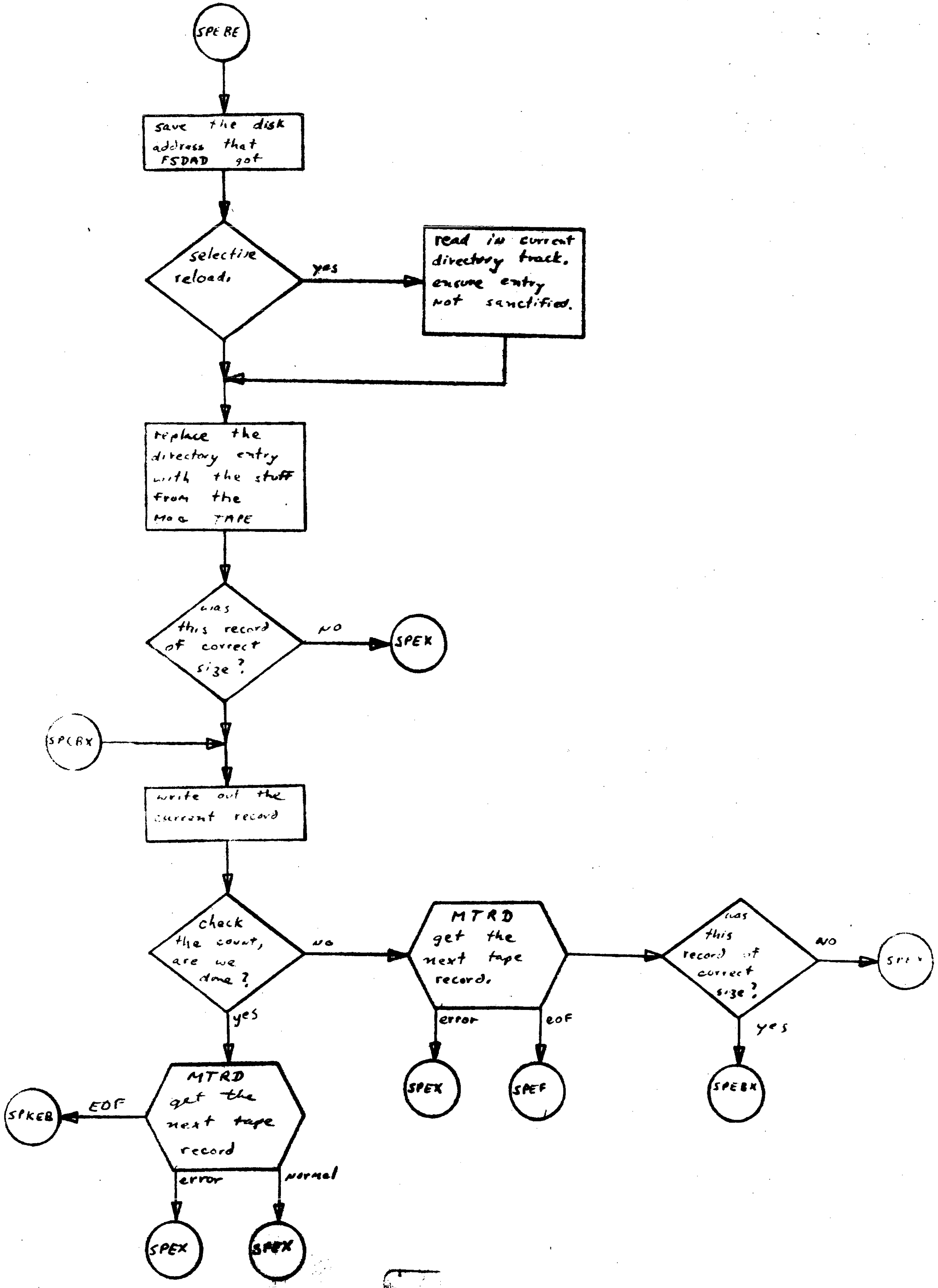
FLRV
beginning of the section to load the user library from mag ~~tape~~
get count of entries to recover.
∅
SPME
≠∅
SPDA
read in all of the disk ADTs
since each disk ADT has a maximum initial size determined by the size of the locked blok table, they are retained in core on a mag tape load
SPDC
MTRD
read a tape record
SPEOS
EOF
tape error
SPEX
normal return
DIRLK
is this entry in the current directory
no
is this SD in the selective load table ?
no
SPSFM
yes
yes
has it been recovered yet?
yes
SPSFM
SPDCA
DREN
place entry in directory
no room
SPDC
delete the selective load SD table
SPDCA
are there entries already in the dir?
no
SPME
yes
SPSFM

MLS

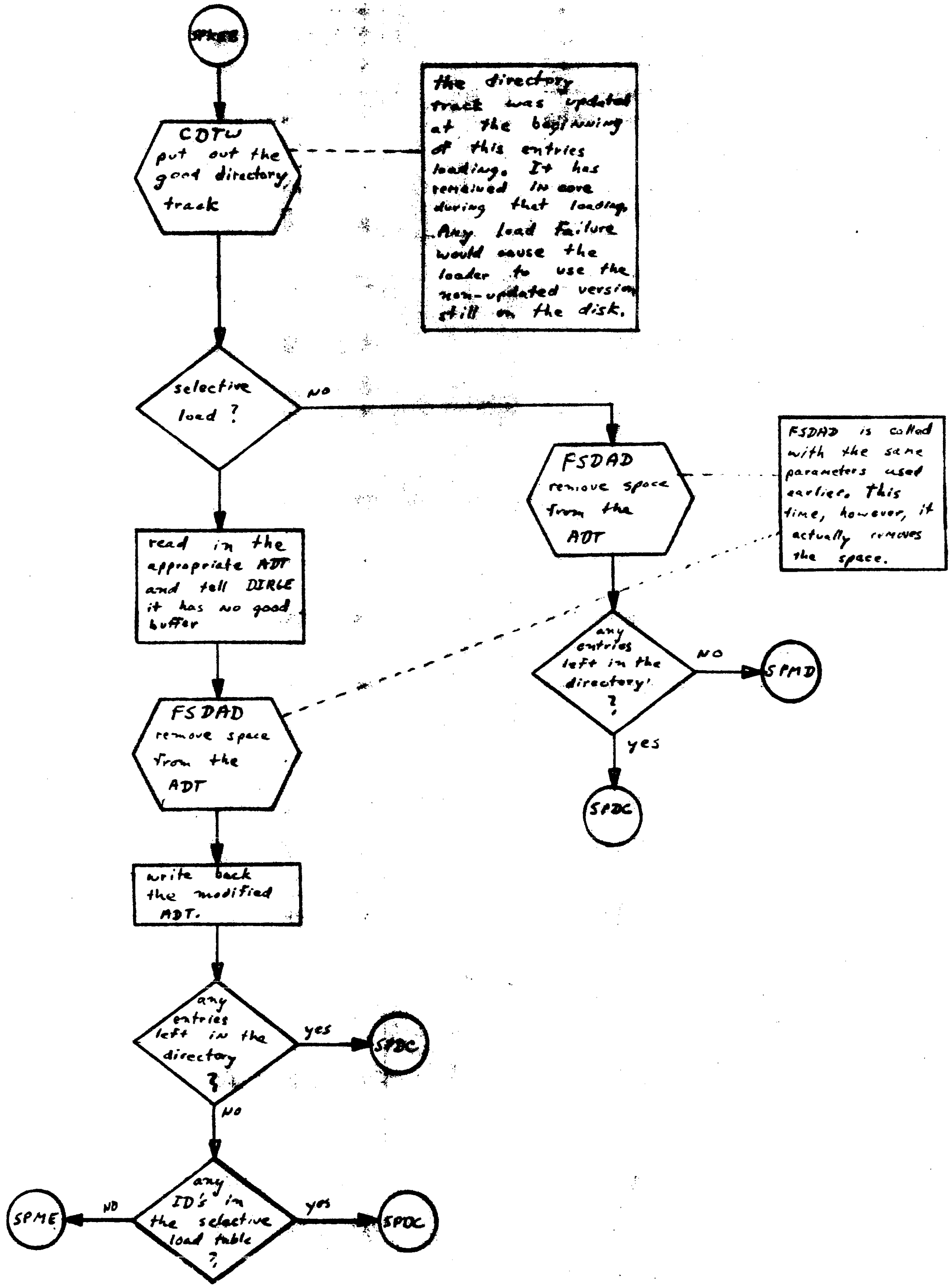

SPDCA
initialize for 8 ADTs
SPEAV
advance pointers to next ADT (0 follows 7)
selective load.
yes
read in the appropriate ADT track
indicate that the dir. buffer was destroyed
fetch pointer to buffer
SPDEN
NO
alternate allocation option ?
NO
SPDES
yes
save-prior disk address for search
Find what disk this piece is on, change ADT #
out of range
HLT 31B
SPDES
generate pointers to in core ADT
SPDEN

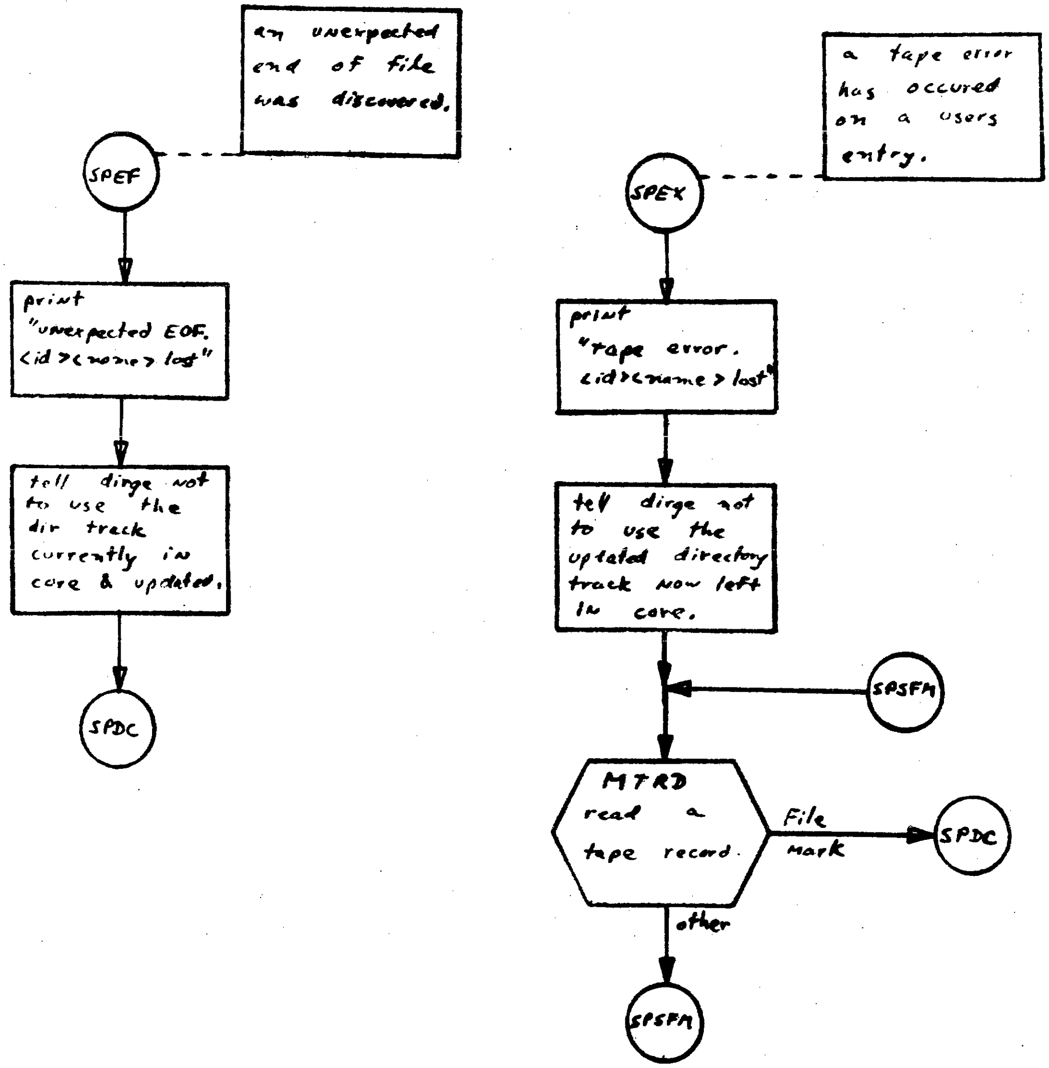

SPEAW
Initialize.
FSDAP will not remove the space on this call.
FSDAD get length and call for space.
found
SPEBE
not in current ADT
alternate allocation option?
yes
print error and terminate loading
NO
have we checked all of the ADT's?
NO
SPEAT
yes
print "NO ROOM FOR <id> <name>'
update & write out the current directory track & direc table
more entries in directory ?
yes
SPSFM
NO
selective load ?
NO
SPEDS
yes
SPSFM

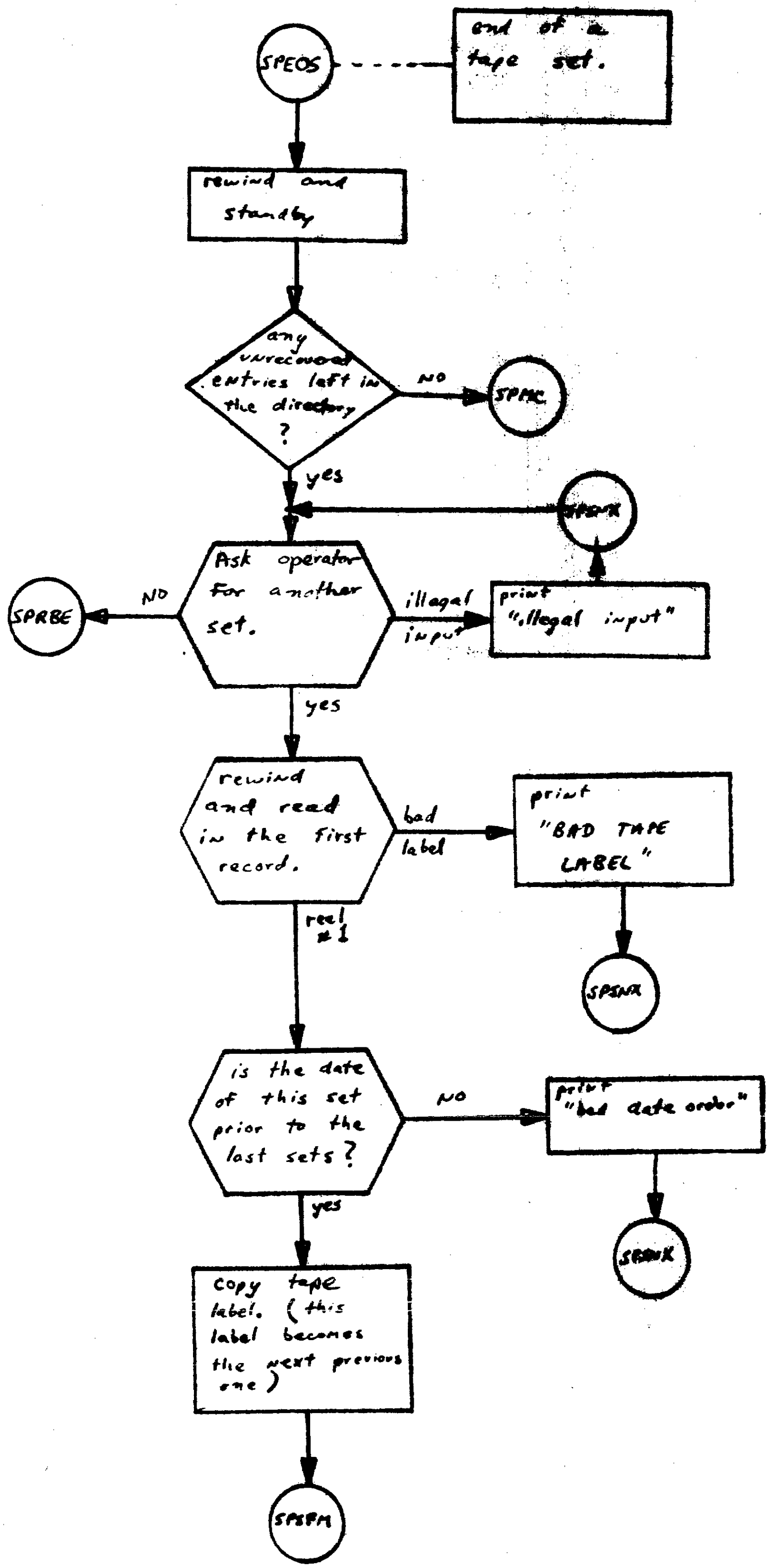
MLS

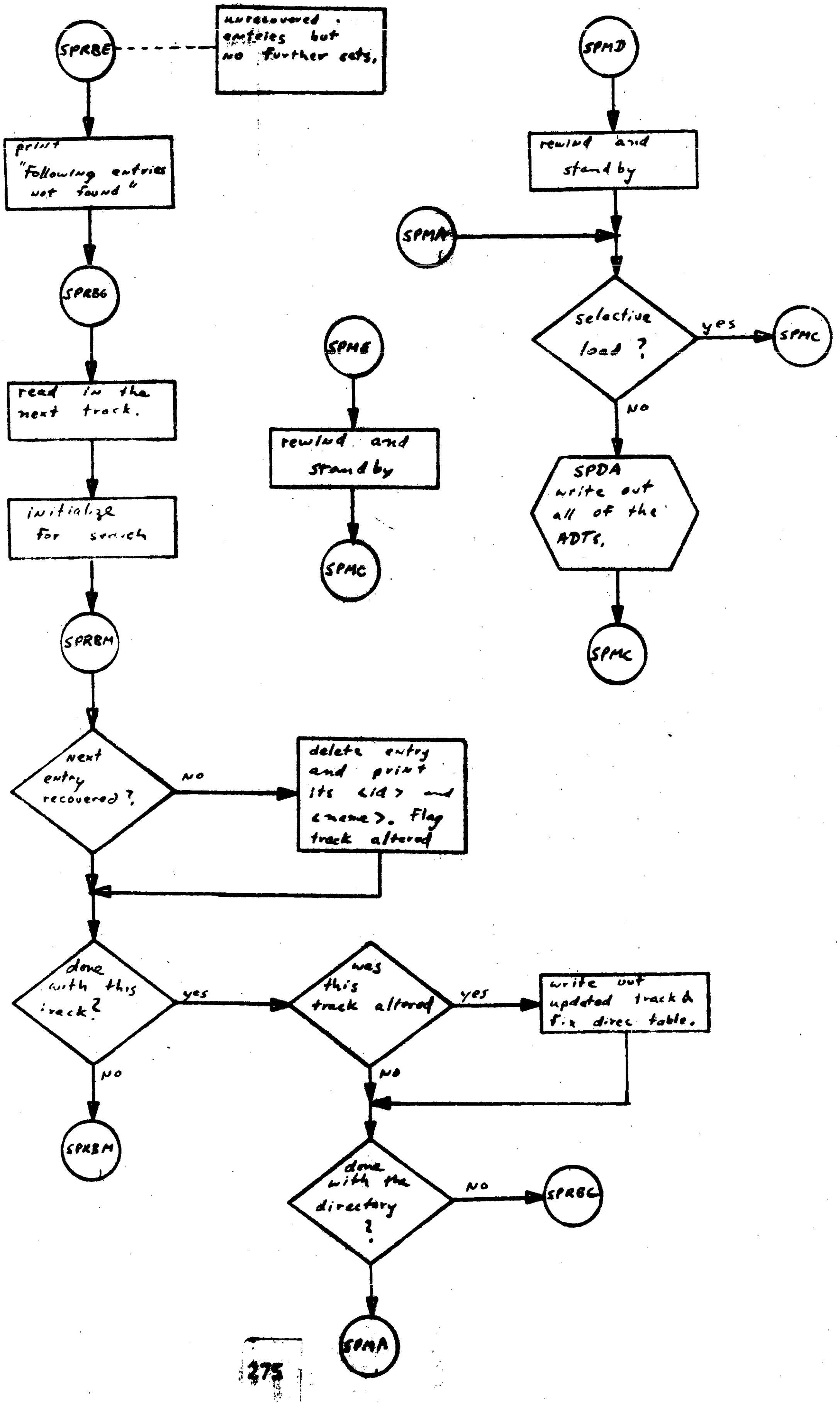

SPEBE
save the disk address that FSDAD got
selective reload,
yes
read in current directory track. ensure entry not sanctified.
replace the directory entry with the stuff from the Mag TAPE
was this record of correct size ?
NO
SPEX
SPCBX
write out the current record
check the count, are we done ?
no
yes
MTRD get the next tape record.
error
eof
SPEX
SPEF
was this record of correct size ?
NO
SPEX
yes
SPEBX
MTRD get the next tape record
EOF
SPKEB
error
normal
SPEX
SPEX

MLS

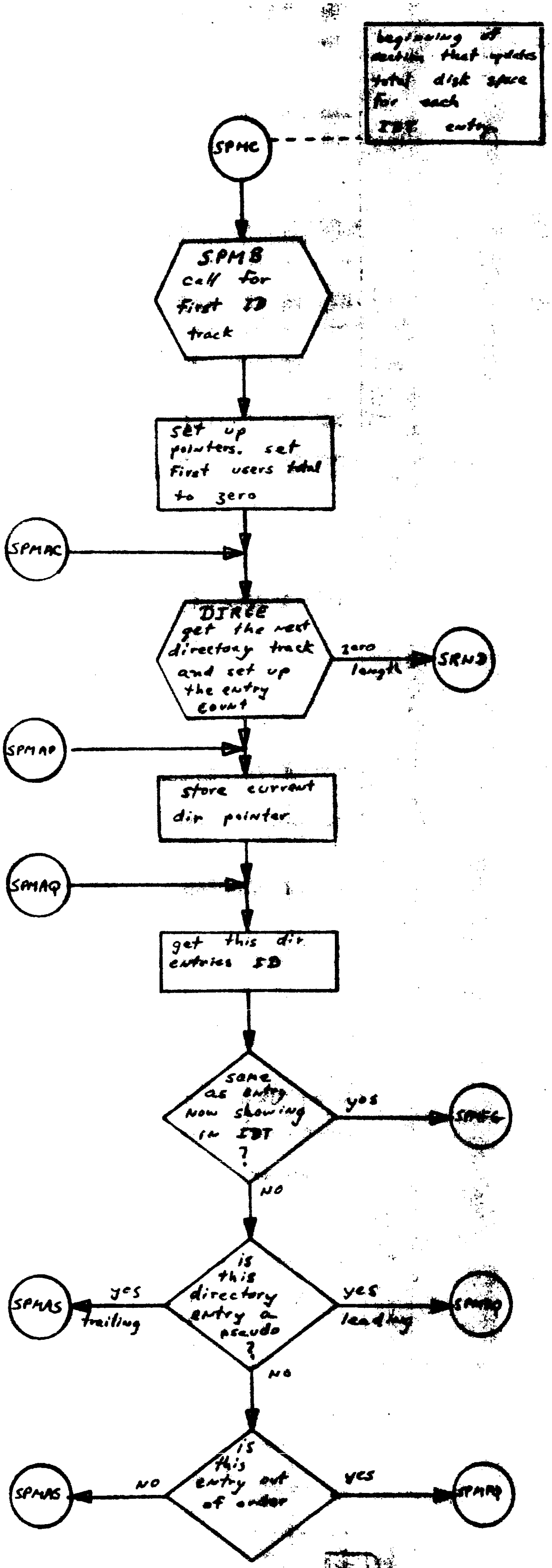

CDTW
put out the good directory track
the directory track was updated at the beginning of this entries loading. It has remained in core during that loading. Any load Failure would cause the loader to use the non-updated version still on the disk.
selective load ?
NO
read in the appropriate ADT and tell DIRLE it has no good buffer
FSDAD
remove space from the ADT
write back the modified ADT.
any entries left in the directory ?
yes
SPDC
NO
any ID's in the selective load table ?
NO
SPME
yes
SPDC
FSDAD
remove space from the ADT
FSDAD is called with the same parameters used earlier. This time, however, it actually removes the space.
any entries left in the directory ?
NO
SPMD
yes
SPDC


an unexpected end of file was discovered.
SPEF
print "unexpected EOF. <id><name> lost"
tell dirge not to use the dir track currently in core & updated.
SPDC
a tape error has occured on a users entry.
SPEX
print "tape error. <id><name> lost"
tell dirge not to use the uptated directory track now left in core.
SPSFM
MTRD
read a tape record.
File mark
SPDC
other
SPSFM

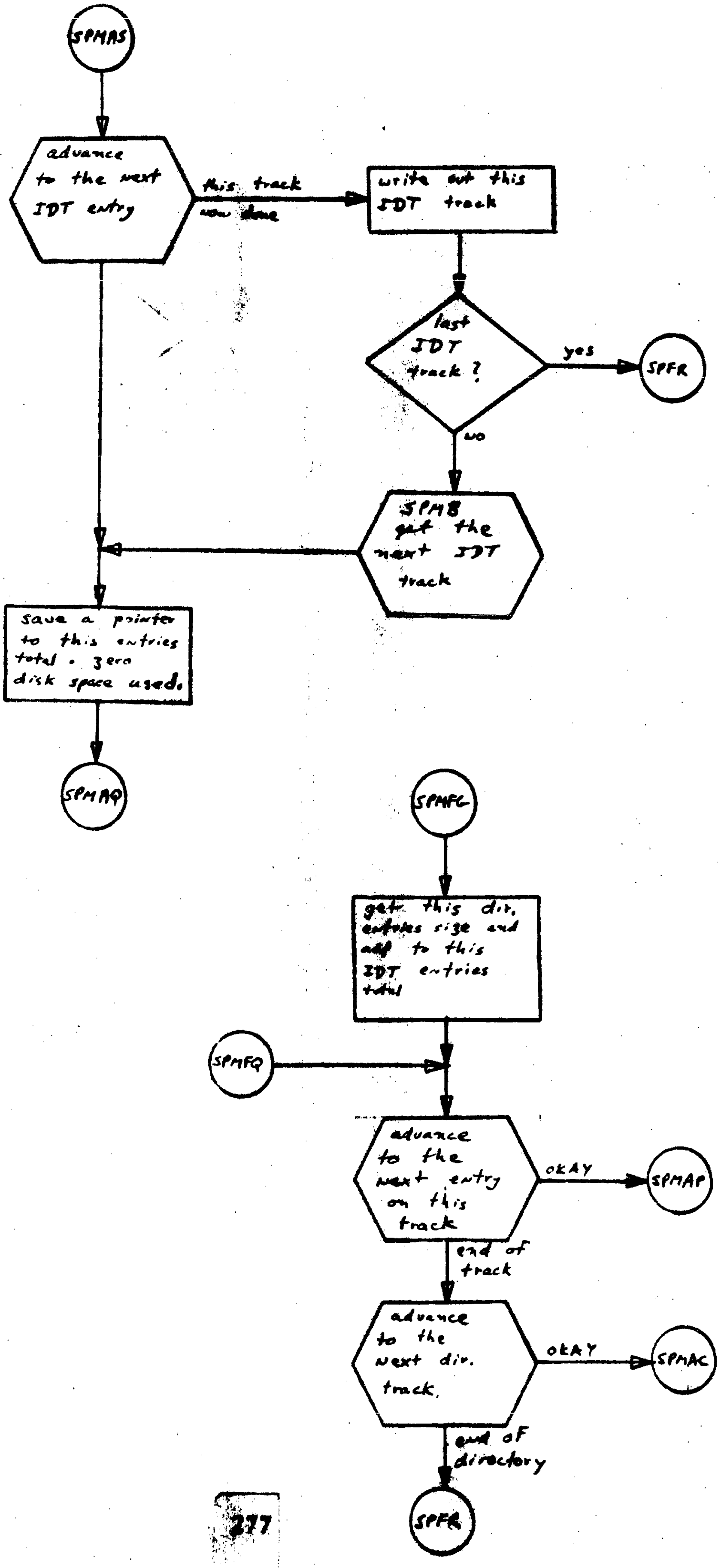

SPEOS
end of a tape set.
rewind and standby
any unrecovered entries left in the directory ?
NO
SPMC
yes
SPSNX
Ask operator for another set.
NO
SPRBE
illegal input
print "illegal input"
yes
rewind and read in the first record.
bad label
print "BAD TAPE LABEL"
SPSNX
reel #1
is the date of this set prior to the last sets ?
NO
print "bad date order"
SPSNX
yes
copy tape label. (this label becomes the next previous one)
SPSFM


SPRBE
unrecovered entries but no further cats,
print "Following entries not found"
SPRBG
read in the next track.
initialize for search
SPRBM
next entry recovered?
NO
delete entry and print its <id> and <name>. Flag track altered
done with this track?
yes
NO
SPRBM
was this track altered
yes
write out updated track & fix direc table.
NO
done with the directory?
NO
SPRBG
SPMA
SPME
rewind and stand by
SPMC
SPMD
rewind and stand by
SPMA
selective load?
yes
SPMC
NO
SPDA write out all of the ADTS,
SPMC

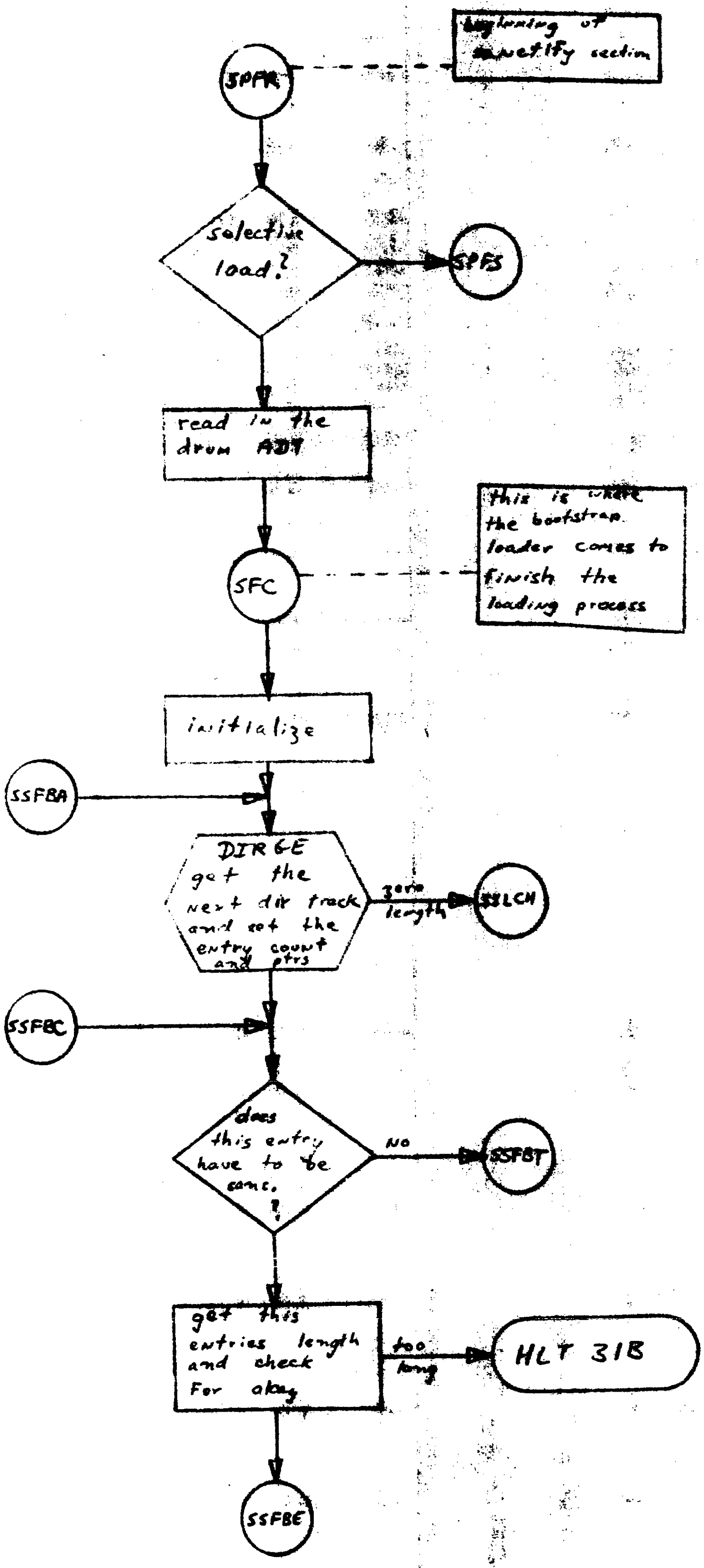
MLS


beginning of section that updates total disk space for each IBT entry
SPMC
SPMB
call for first ID track
set up pointers. set first users total to zero
SPMAC
DIREE
get the next directory track and set up the entry count
zero length
SRND
SPMAP
store current dir pointer
SPMAQ
get this dir entries ID
same as entry now showing in IBT ?
yes
SPMFG
NO
is this directory entry a pseudo ?
yes
trailing
SPMAS
yes
leading
SPMAQ
NO
is this entry out of order
NO
SPMAS
yes
SPMAQ

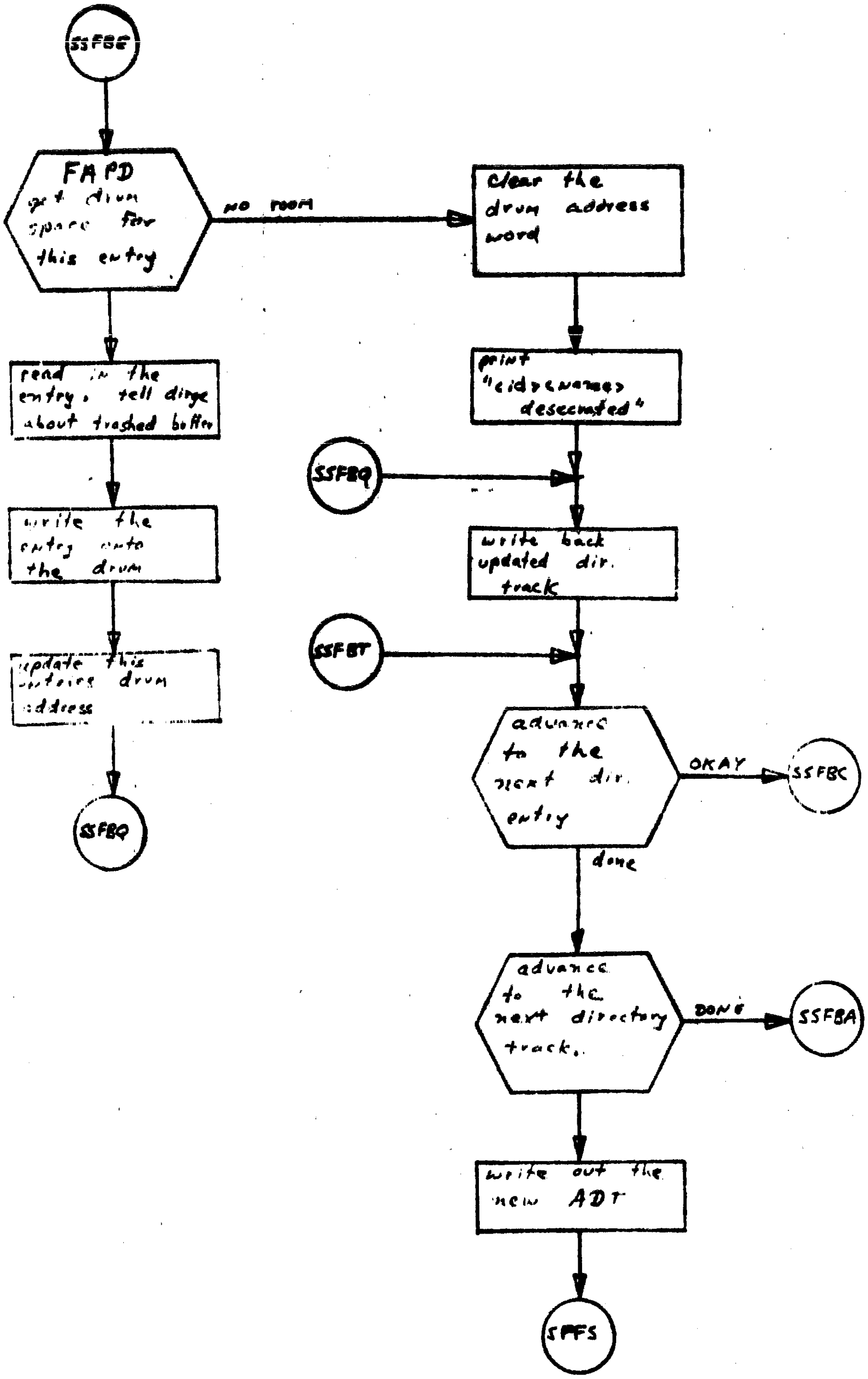
MLS


SPMAS
advance to the next IDT entry
this track now done
write out this IDT track
last IDT track ?
yes
SPFR
no
SPM8 get the next IDT track
save a pointer to this entries total . zero disk space used.
SPMAQ
SPMFC
get this dir. entries size and add to this IDT entries total
SPMFQ
advance to the next entry on this track
OKAY
SPMAP
end of track
advance to the next dir. track.
OKAY
SPMAC
end of directory
SPFR

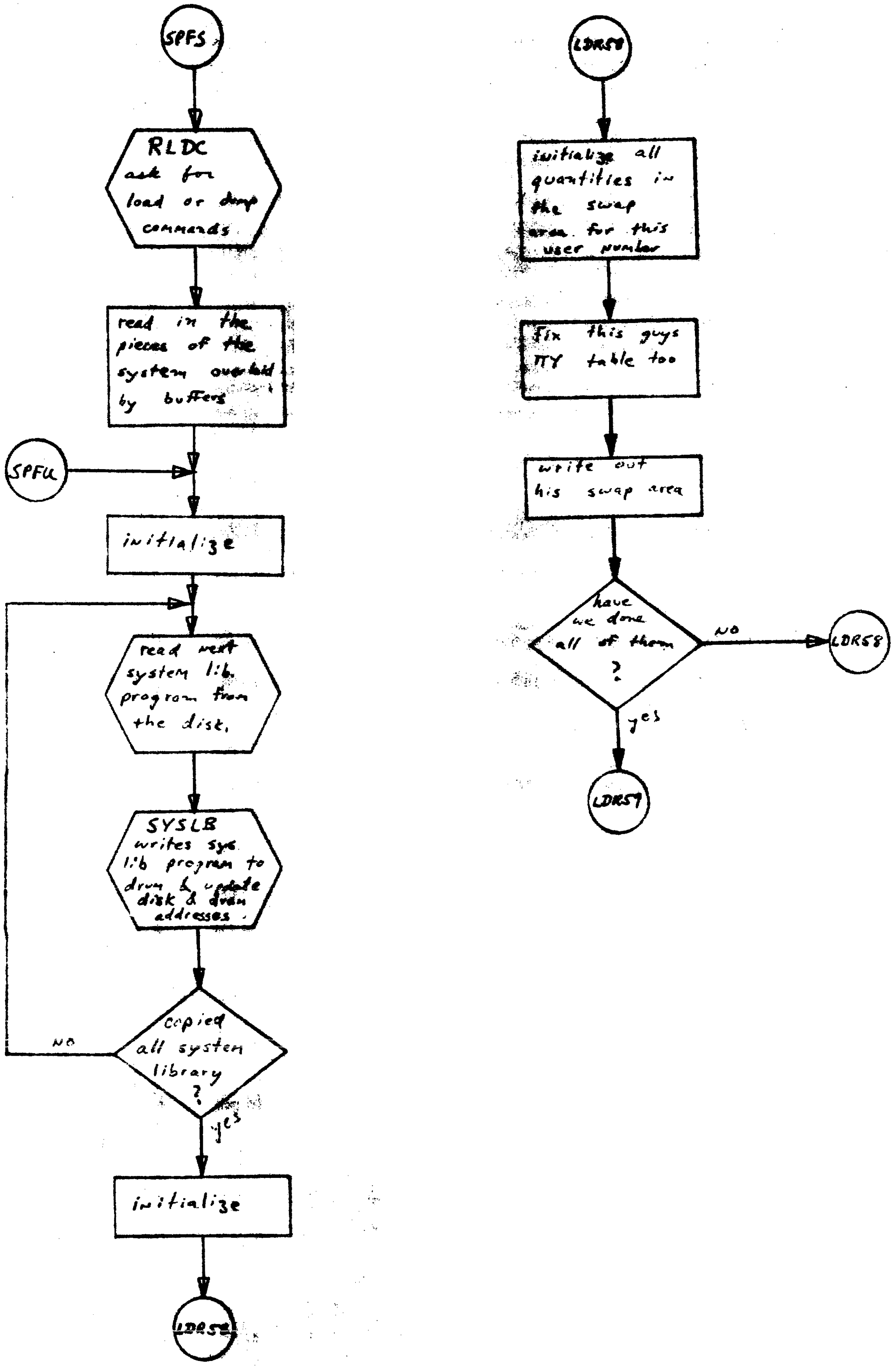
MLS


beginning of rectify section
SPFR
selective load?
SPFS
read in the drum ADT
this is where the bootstrap loader comes to finish the loading process
SFC
initialize
SSFBA
DIRGE get the next dir track and set the entry count and ptrs
zero length
SSLCH
SSFBC
does this entry have to be conc. ?
NO
SSFBT
get this entries length and check For okay
too long
HLT 31B
SSFBE

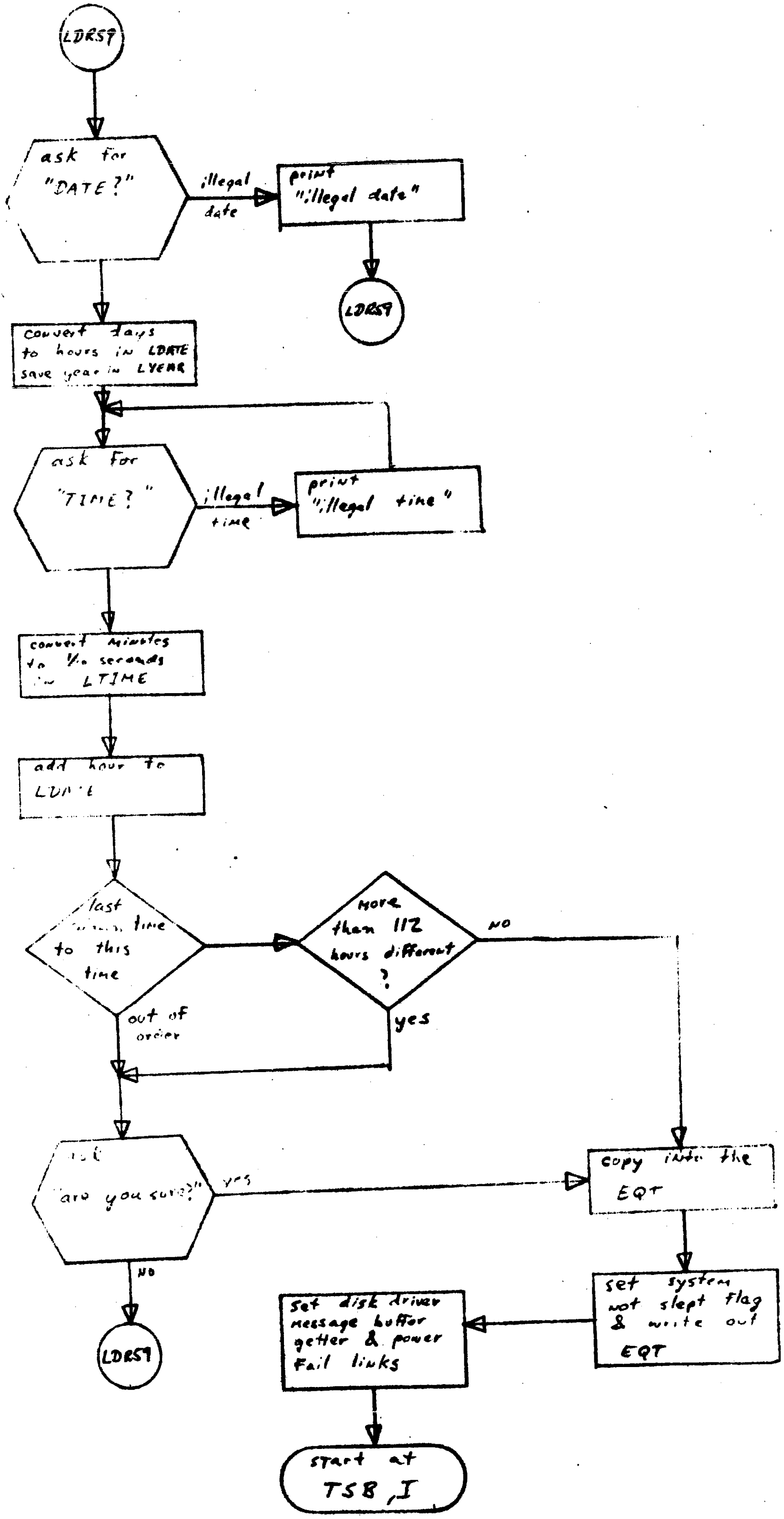
MLS


SSFBE
FAPD get drum space for this entry
no room
clear the drum address word
read in the entry, tell dirge about trashed buffer
print "cid>cnames> desecrated"
SSFBQ
write the entry onto the drum
write back updated dir. track
SSFBT
update this entries drum address
advance to the next dir. entry
OKAY
SSFBC
SSFBQ
done
advance to the next directory track.
DONE
SSFBA
write out the new ADT
SPFS

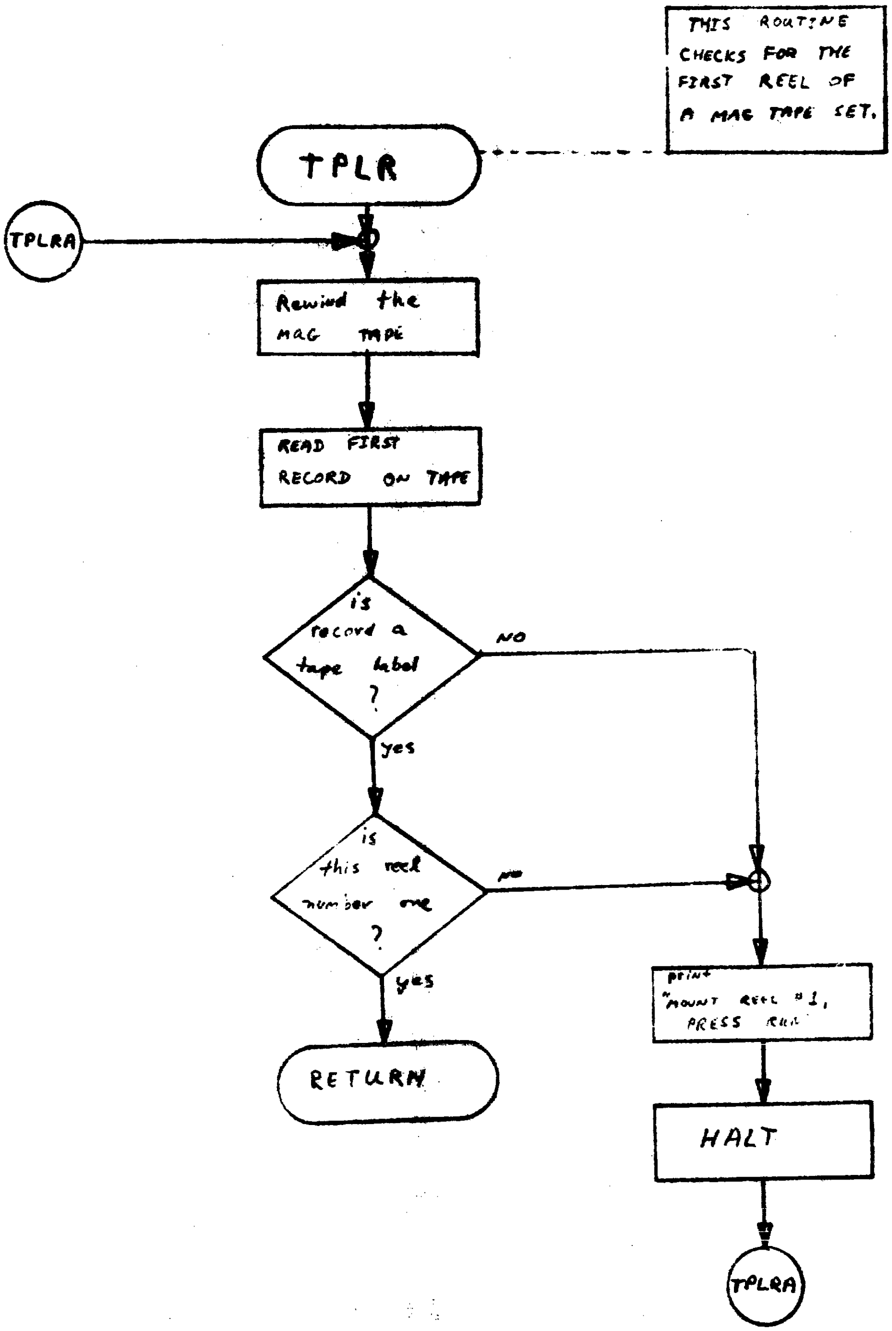
MLS

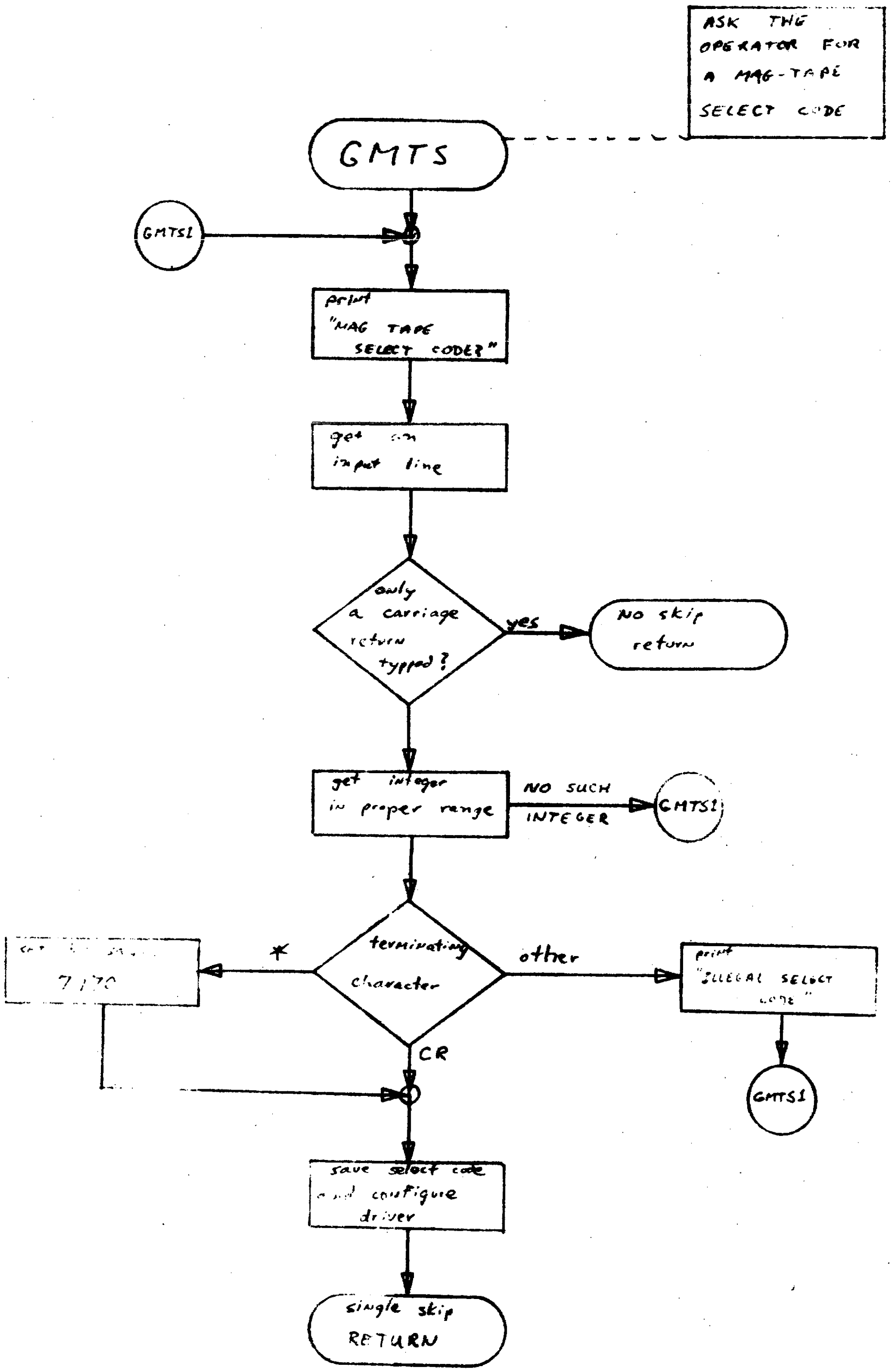

SPFS
RLDC ask for load or dump commands
read in the pieces of the system overlaid by buffers
SPFU
initialize
read next system lib. program from the disk.
SYSLB writes sys lib program to drum & update disk & drum addresses.
copied all system library ?
NO
yes
initialize
LDR58
LDR58
initialize all quantities in the swap area for this user number
Fix this guys TTY table too
write out his swap area
have we done all of them ?
NO
LDR58
yes
LDR59

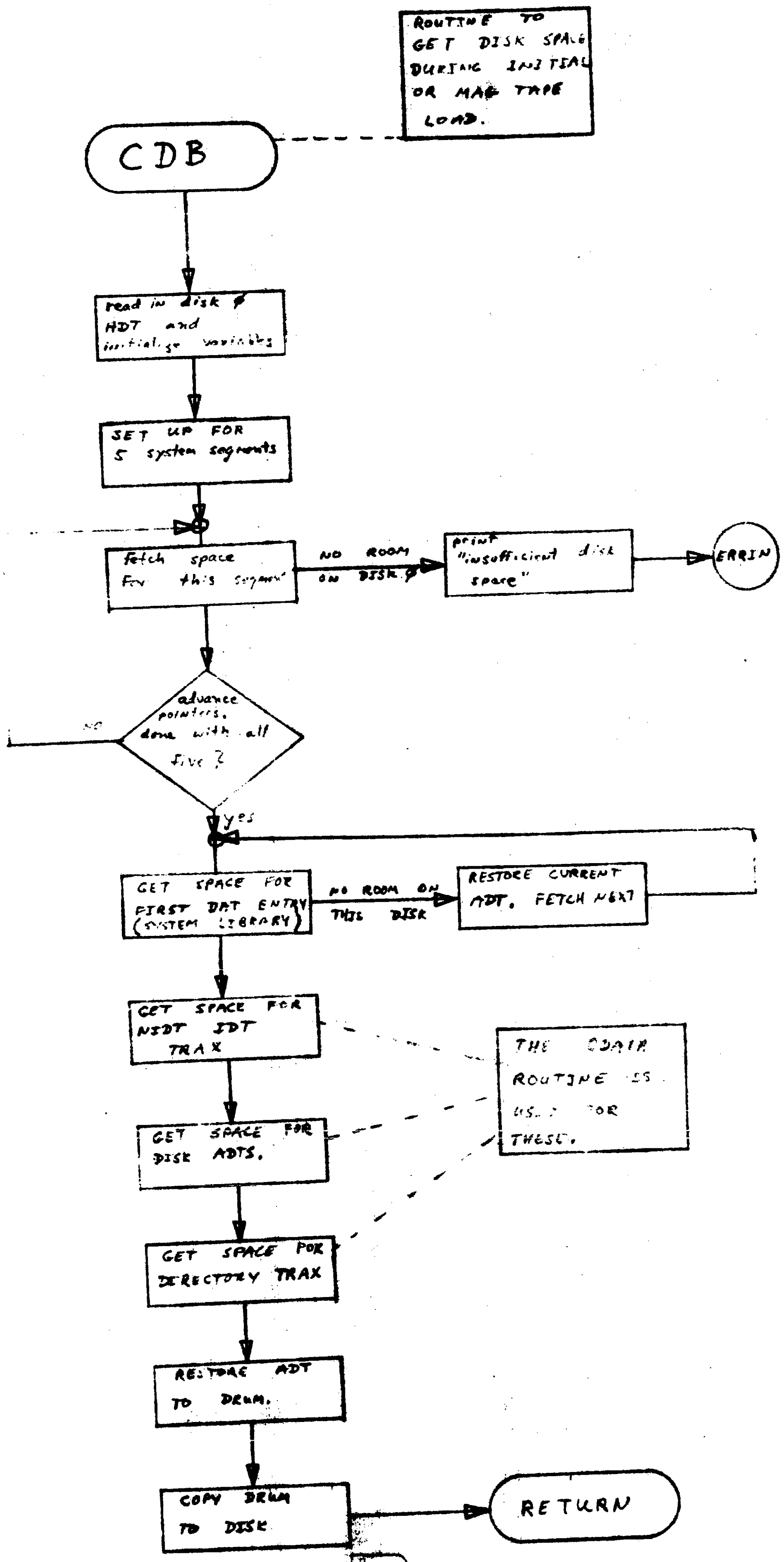
MLS

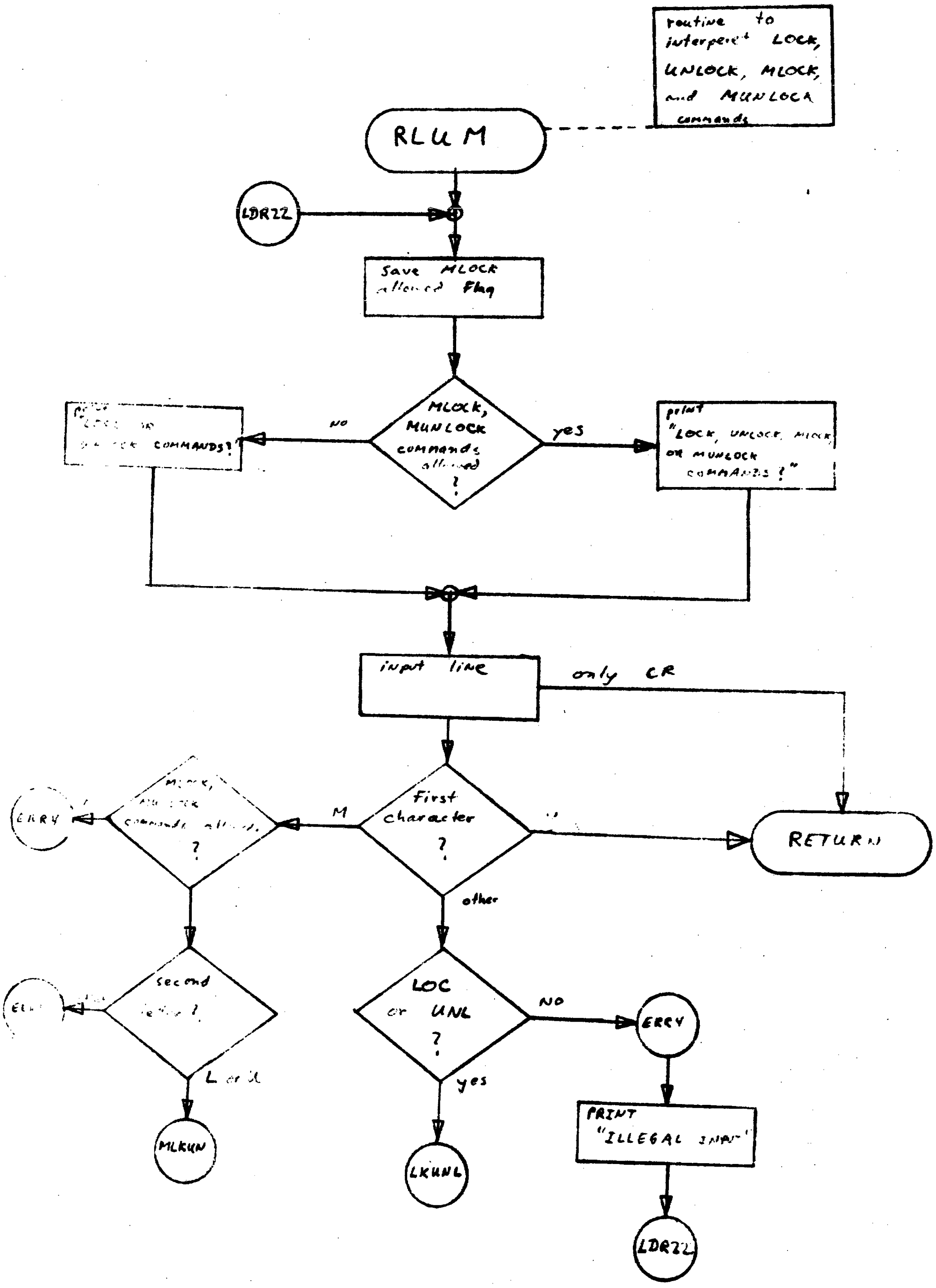


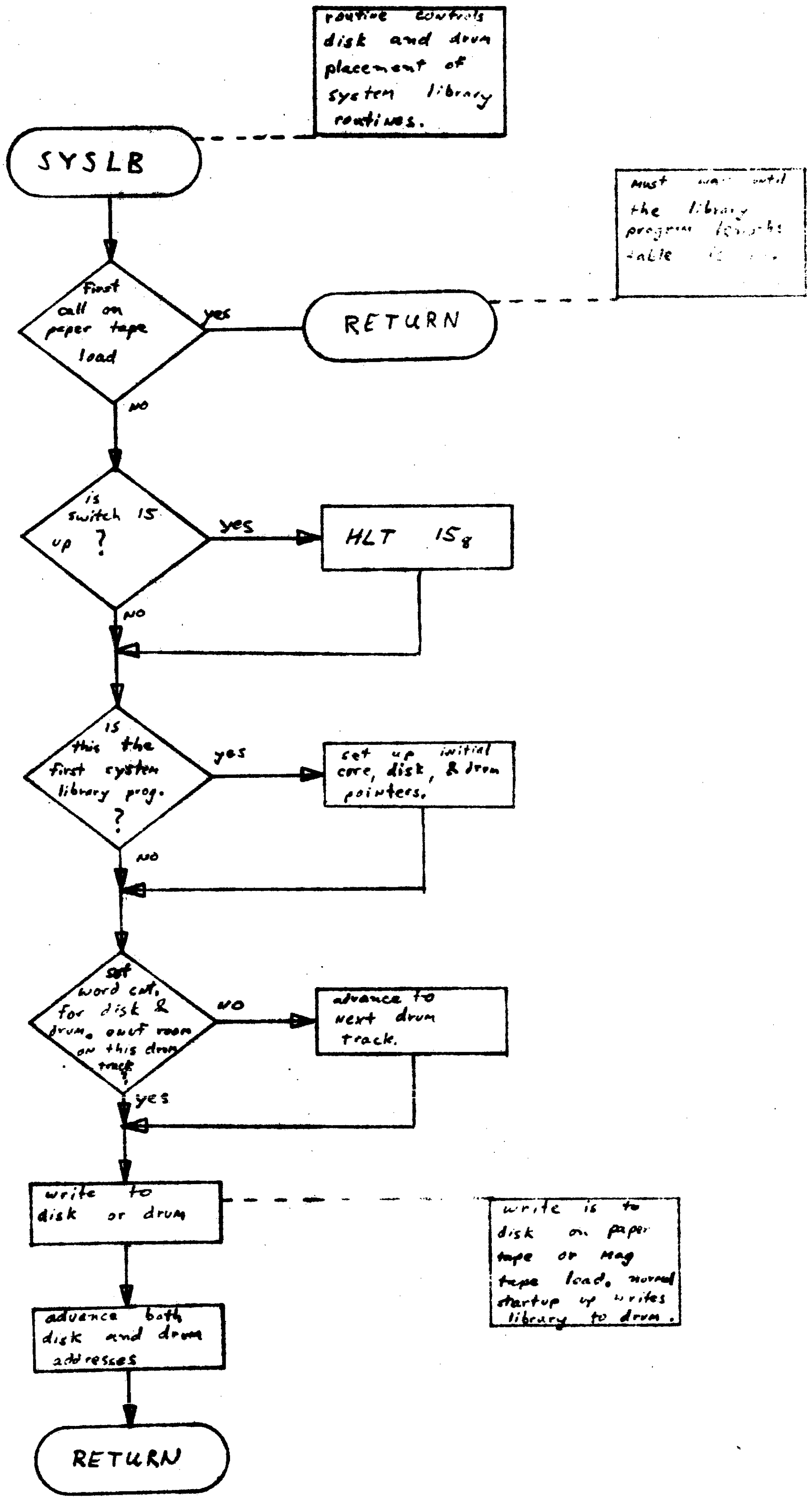
LDRS7
ask for "DATE?"
illegal date
print "illegal date"
LDRS7
convert days to hours in LDATE save year in LYEAR
ask for "TIME?"
illegal time
print "illegal time"
convert minutes to 1/10 seconds in LTIME
add hour to LDATE
last compare time to this time
out of order
more than 112 hours different ?
NO
yes
ask "are you sure?"
yes
NO
LDRS7
copy into the EQT
set system not slept flag & write out EQT
set disk driver message buffer getter & power fail links
start at TSB,I

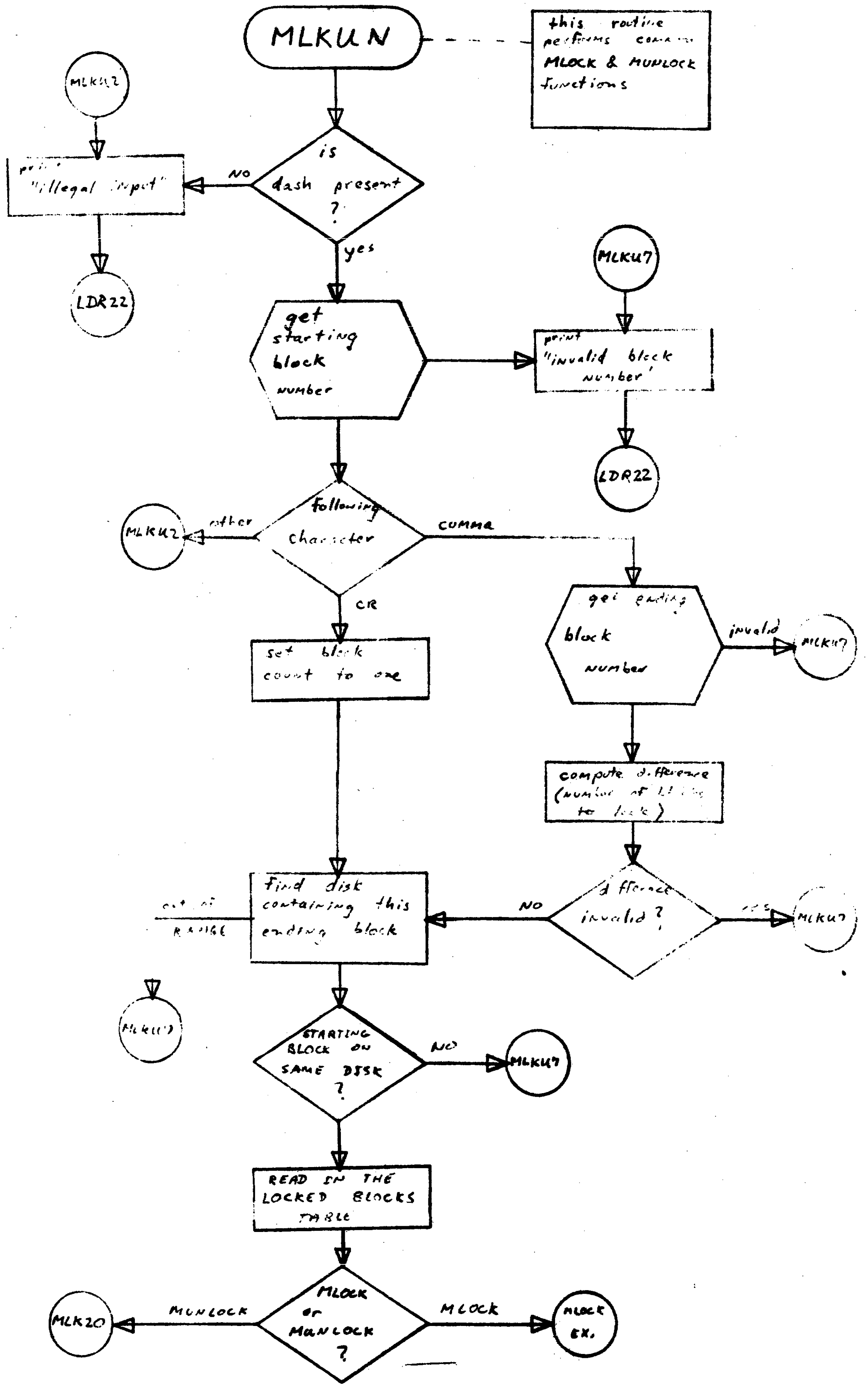

THIS ROUTINE CHECKS FOR THE FIRST REEL OF A MAG TAPE SET.
TPLR
TPLRA
Rewind the MAG TAPE
READ FIRST RECORD ON TAPE
is record a tape label ?
NO
yes
is this reel number one ?
no
yes
RETURN
print "MOUNT REEL #1, PRESS RUN"
HALT
TPLRA


ASK THE OPERATOR FOR A MAG-TAPE SELECT CODE
GMTS
GMTS1
print "MAG TAPE SELECT CODE?"
get an input line
only a carriage return typped?
yes
No skip return
get integer in proper range
NO SUCH INTEGER
GMTS1
terminating character
*
set [illegible]
7170
other
print "ILLEGAL SELECT CODE"
GMTS1
CR
save select code and configure driver
single skip RETURN

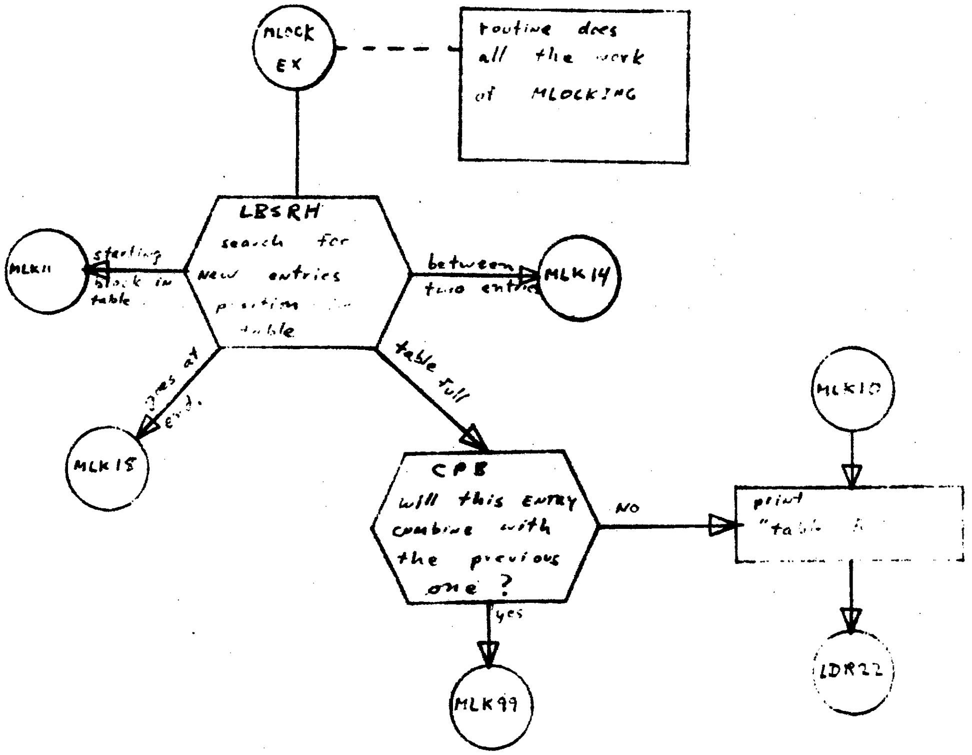

ROUTINE TO GET DISK SPACE DURING INITIAL OR MAG TAPE LOAD.
CDB
read in disk φ HDT and initialize variables
SET UP FOR 5 system segments
fetch space for this segment
NO ROOM ON DISK φ
print "insufficient disk space"
ERRIN
advance pointers. done with all five?
NO
yes
GET SPACE FOR FIRST DAT ENTRY (SYSTEM LIBRARY)
NO ROOM ON THIS DISK
RESTORE CURRENT ADT. FETCH NEXT
GET SPACE FOR NIDT IDT TRAX
GET SPACE FOR DISK ADTS.
GET SPACE FOR DIRECTORY TRAX
THE CDAIR ROUTINE IS USED FOR THESE.
RESTORE ADT TO DRUM.
COPY DRUM TO DISK
RETURN

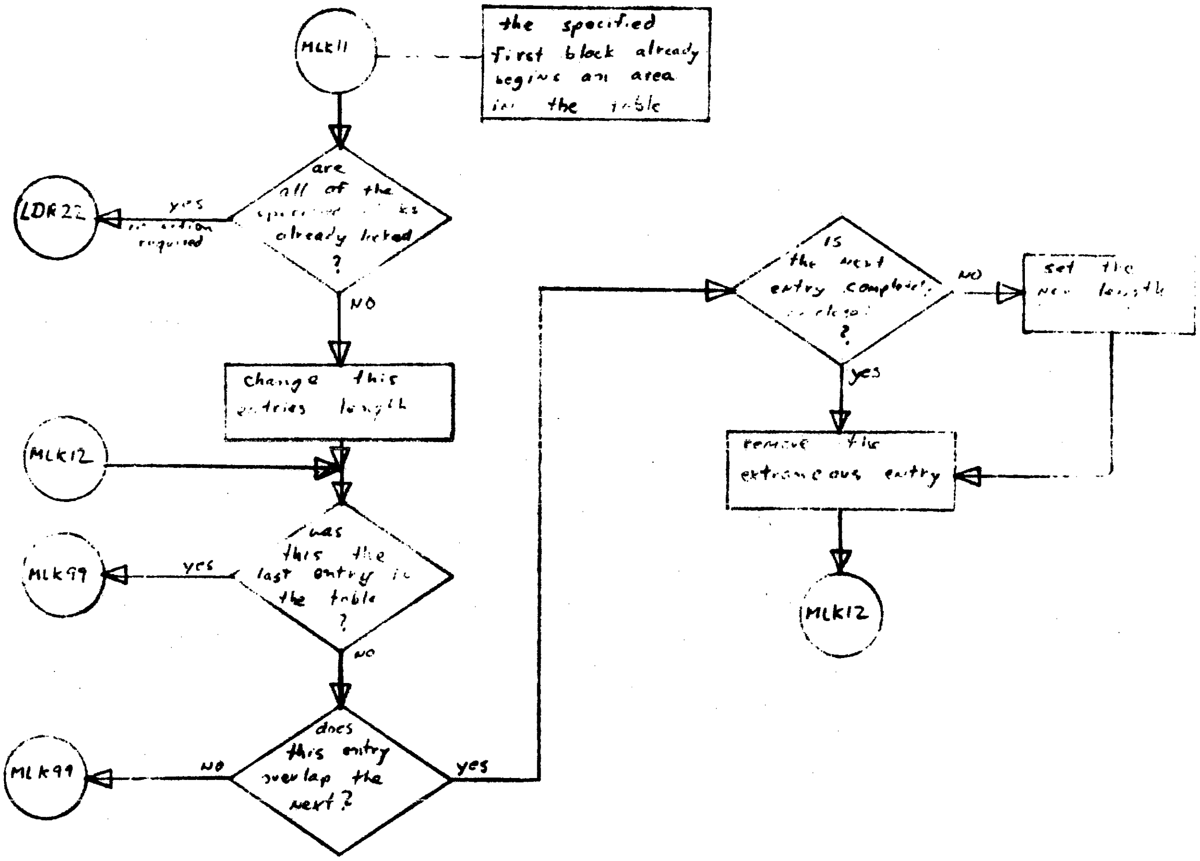

routine to interperet LOCK, UNLOCK, MLOCK, and MUNLOCK commands
RLUM
LDR22
Save MLOCK allowed Flag
MLOCK, MUNLOCK commands allowed ?
no
yes
print "LOCK OR UNLOCK COMMANDS?"
print "LOCK, UNLOCK, MLOCK, OR MUNLOCK COMMANDS ?"
input line
only CR
First character ?
M
other
RETURN
MLOCK, MUNLOCK commands allowed ?
ERRY
second letter ?
L or U
MLKUN
LOC or UNL ?
NO
yes
LKUNL
ERRY
PRINT "ILLEGAL INPUT"
LDR22

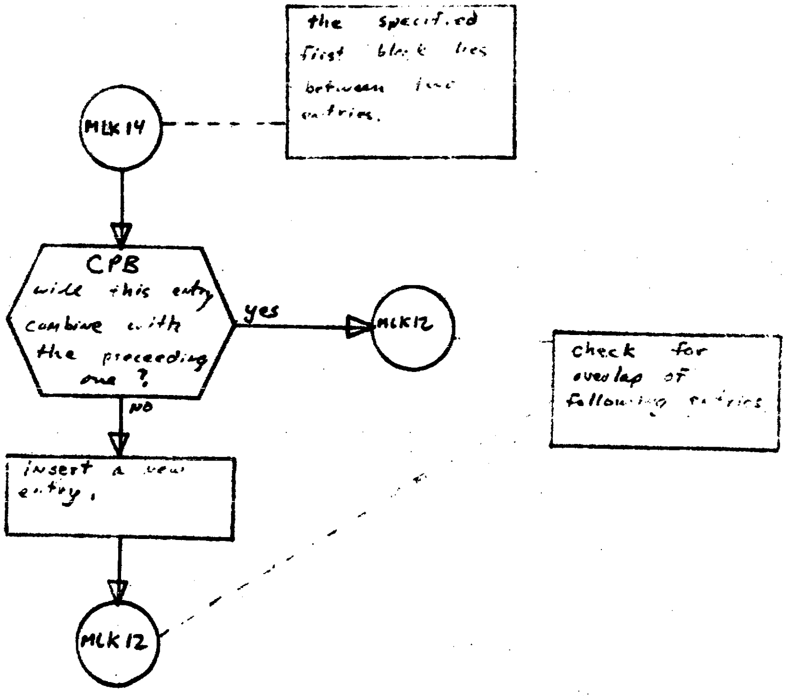

routine controls disk and drum placement of system library routines.
SYSLB
First call on paper tape load
yes
RETURN
must wait until the library program lengths table is [illegible]
no
is switch 15 up ?
yes
HLT 15₈
no
is this the first system library prog. ?
yes
set up initial core, disk, & drum pointers.
no
set word cnt. for disk & drum. out room on this drum track ?
no
advance to next drum track.
yes
write to disk or drum
write is to disk on paper tape or mag tape load, normal startup writes library to drum.
advance both disk and drum addresses
RETURN

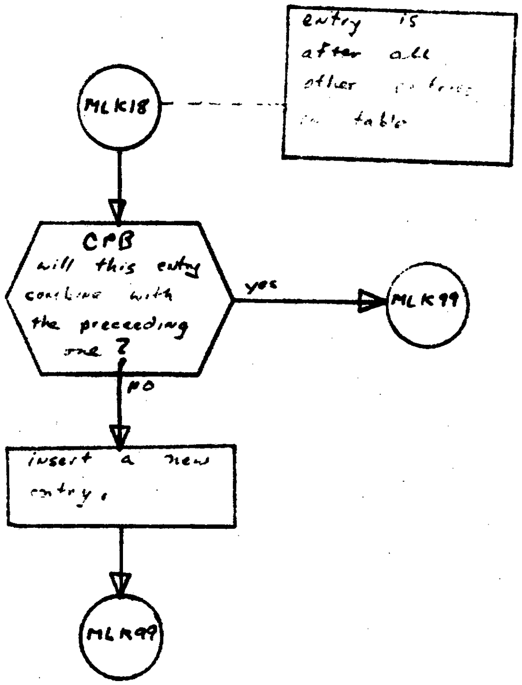

MLKUN
this routine performs common MLOCK & MUNLOCK functions
is dash present ?
No
yes
MLKU2
print "illegal input"
LDR22
get starting block number
MLKU7
print "invalid block number"
LDR22
following character
other
MLKU2
comma
CR
set block count to one
get ending block number
invalid
MLKU7
compute difference (number of blocks to lock)
difference invalid ?
NO
yes
MLKU7
find disk containing this ending block
out of RANGE
MLKU7
STARTING BLOCK ON SAME DISK ?
NO
MLKU7
READ IN THE LOCKED BLOCKS TABLE
MLOCK or MUNLOCK ?
MUNLOCK
MLK20
MLOCK
MLOCK EX.

MLOCK - MUNLOCK

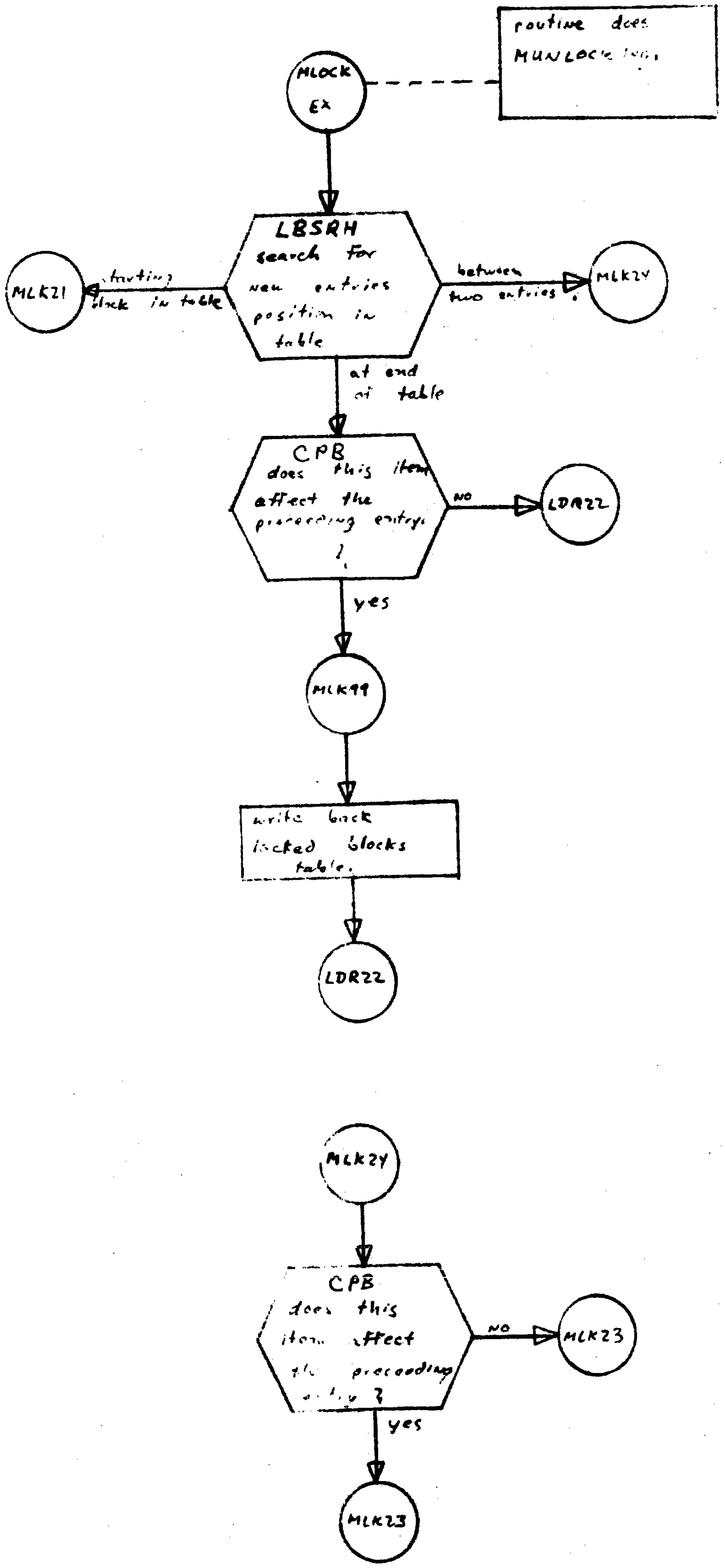

MLOCK EX
routine does all the work of MLOCKING
LBSRH search for new entries position in table
starting block in table
MLK11
between two entries
MLK14
yes at end.
MLK15
table full
CPB will this ENTRY combine with the previous one ?
NO
yes
MLK99
MLK10
print "table
LDR22


MLK11
the specified first block already begins an area in the table
are all of the specified blocks already locked ?
yes
no action required
LDR22
NO
Change this entries length
MLK12
was this the last entry in the table ?
yes
MLK99
no
does this entry overlap the next ?
NO
MLK99
yes
IS the next entry completely overlapped ?
NO
set the new length
yes
remove the extraneous entry
MLK12

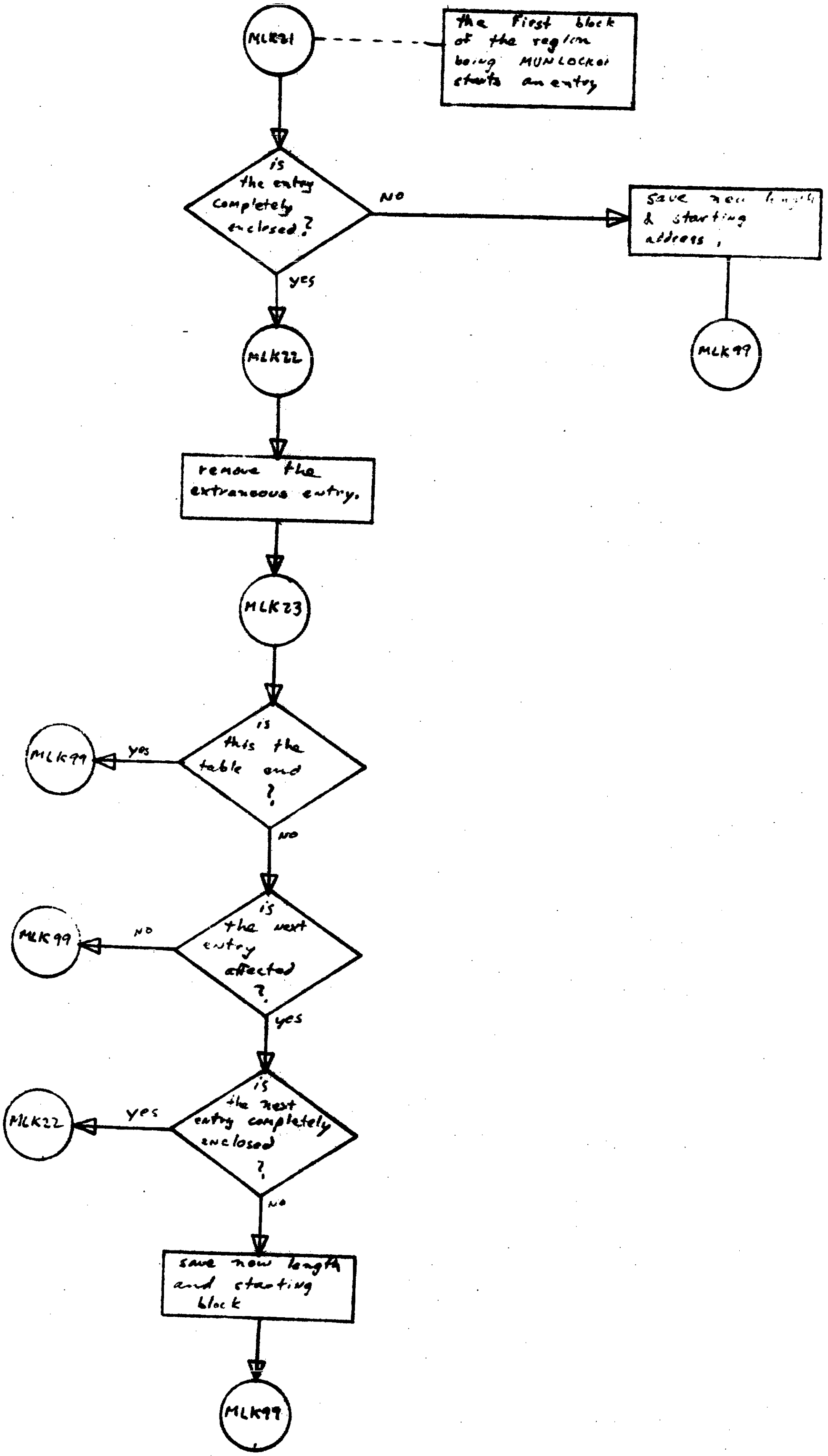

MLK14
the specified first block lies between two entries.
CPB
will this entry combine with the preceeding one ?
yes
MLK12
no
insert a new entry.
MLK12
check for overlap of following entries

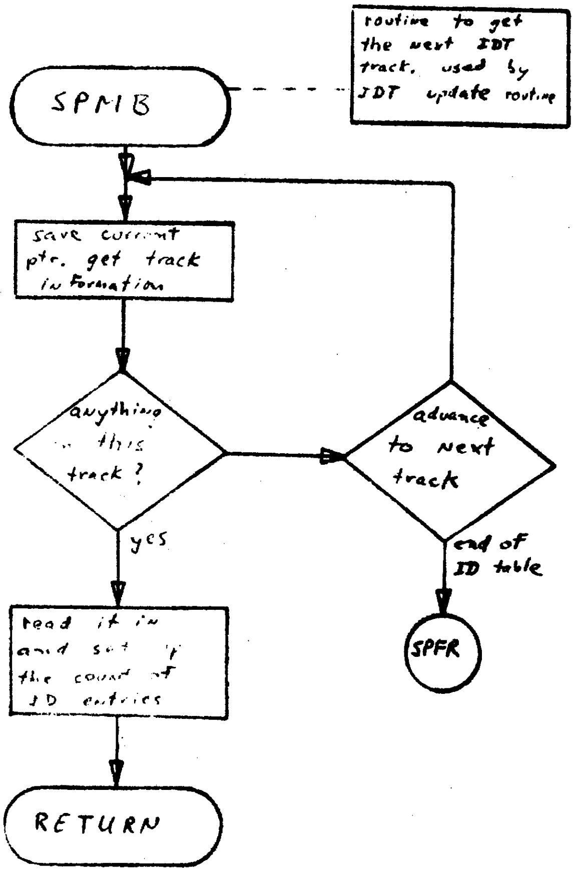

MLK18
entry is after all other entries in table
CPB
will this entry combine with the preceeding one ?
yes
MLK99
no
insert a new entry.
MLK99

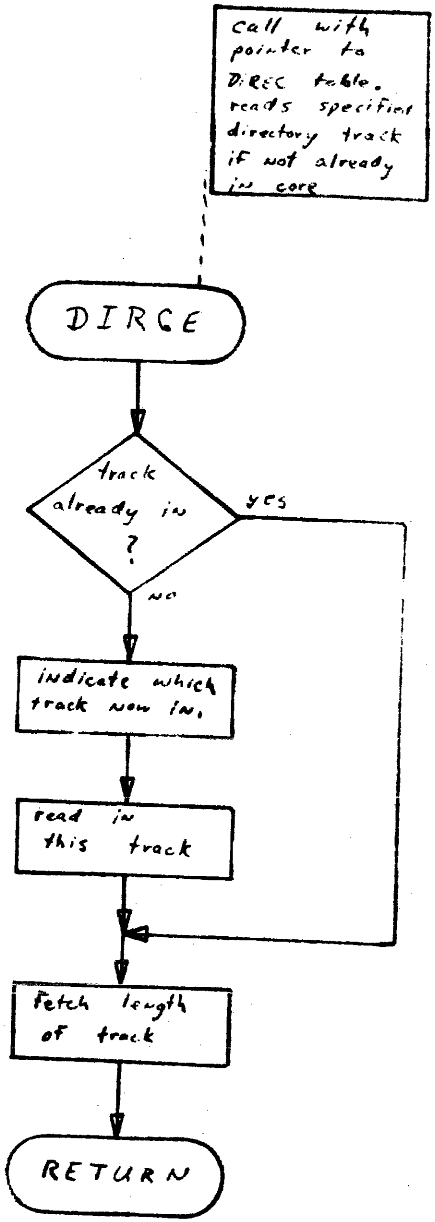
MLOCK - MUNLOCK page [illegible]

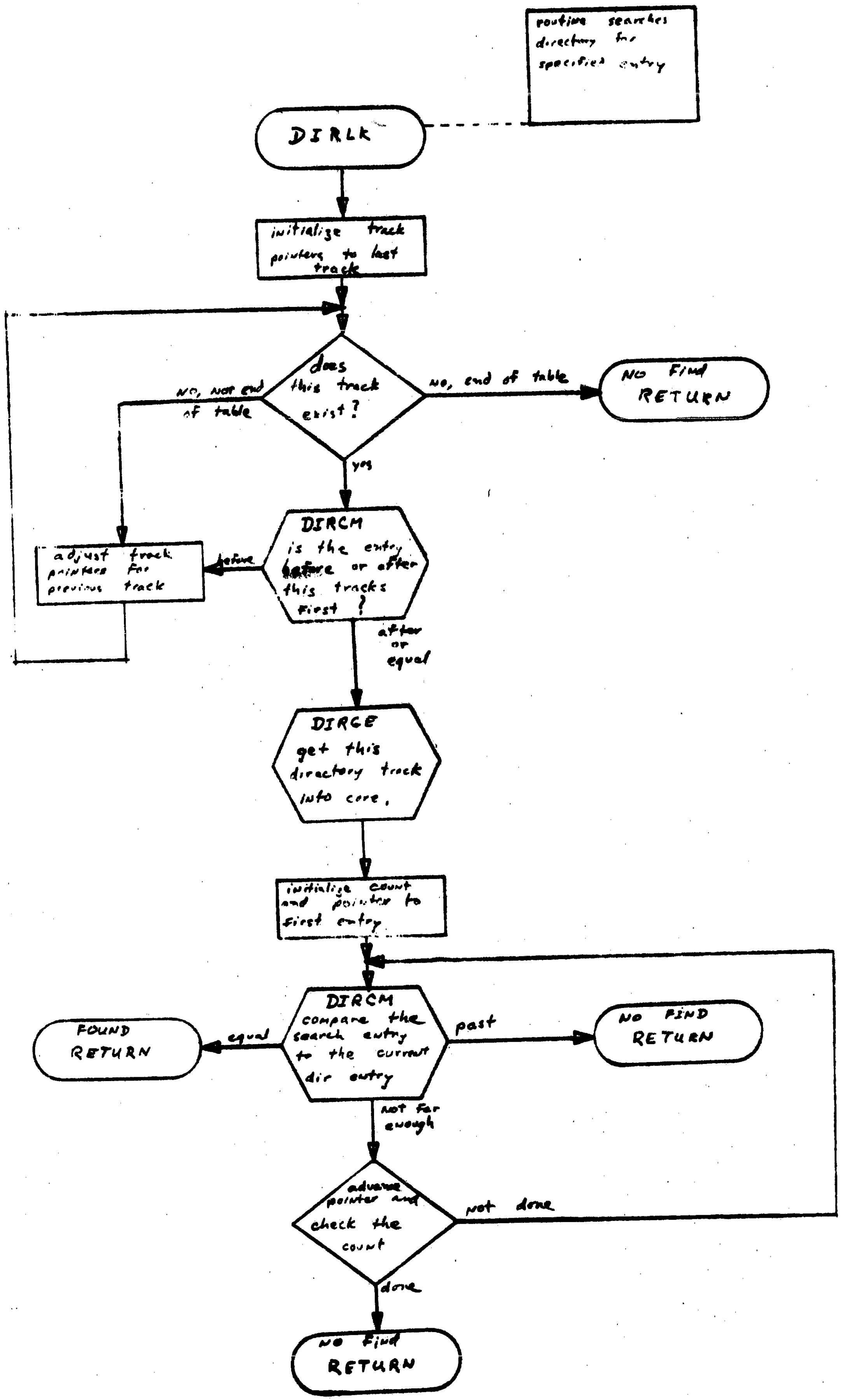

MLOCK EX
routine does MUNLOCK [illegible]
LBSRH search for new entries position in table
starting block in table
MLK21
between two entries
MLK24
at end of table
CPB does this item affect the preceeding entry ?
NO
LDR22
yes
MLK99
write back locked blocks table.
LDR22
MLK24
CPB does this item affect the preceeding entry ?
NO
MLK23
yes
MLK23

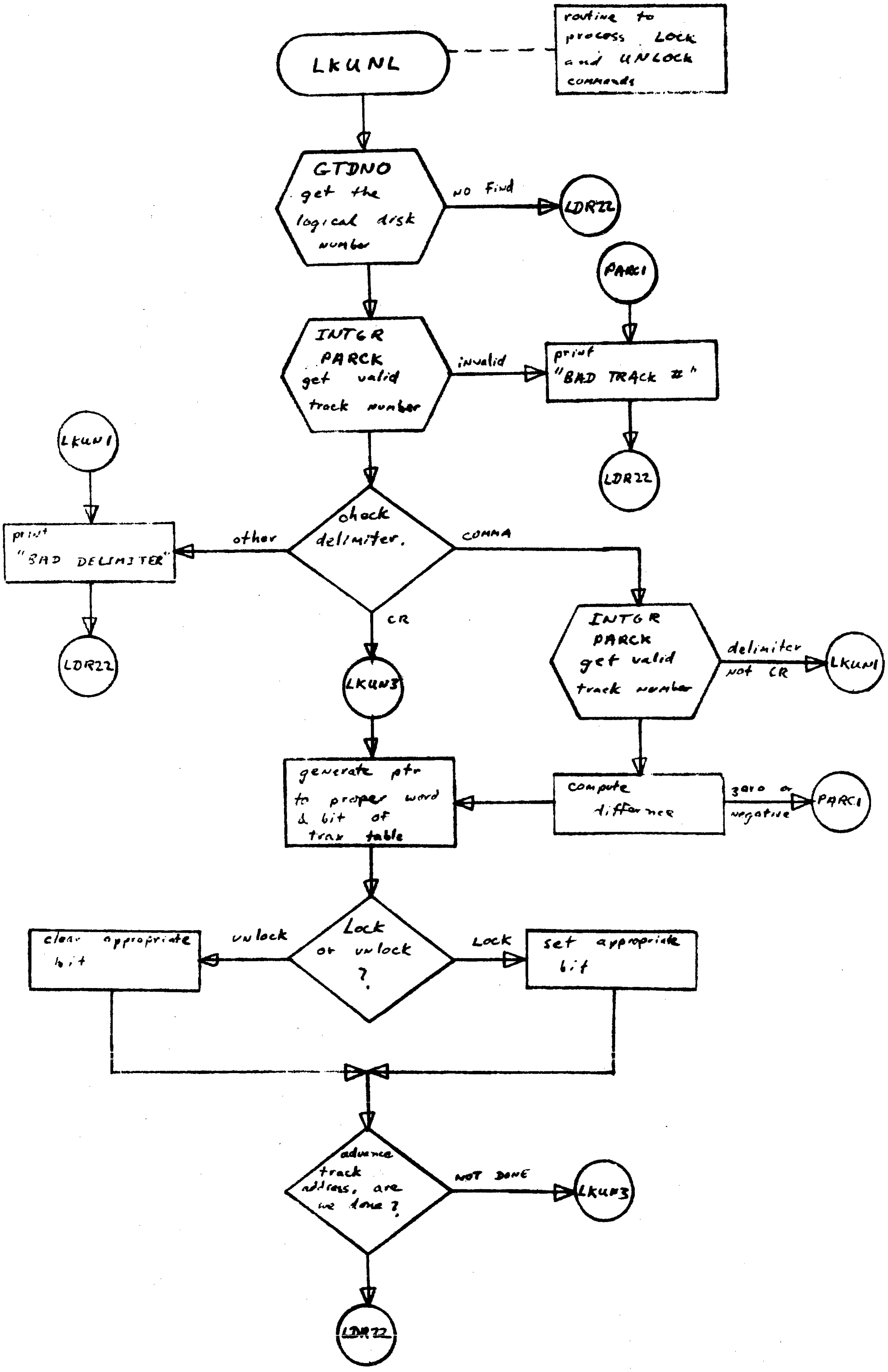
MLOCK - MUNLOCK [illegible]

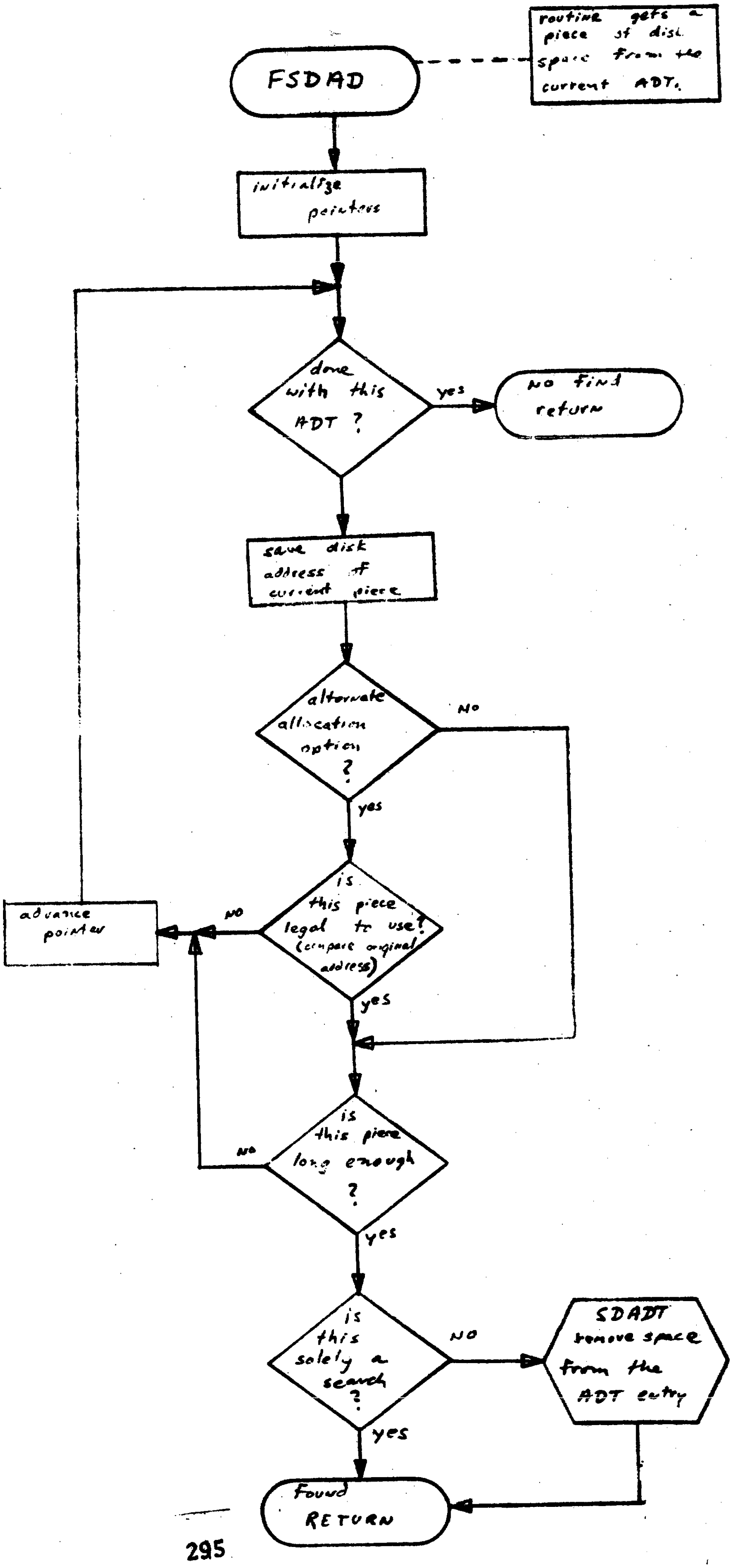

MLK21
the First block of the region being MUNLOCKed starts an entry
is the entry completely enclosed?
NO
save new length & starting address,
MLK99
yes
MLK22
remove the extraneous entry.
MLK23
is this the table end?
yes
MLK99
NO
is the next entry affected?
NO
MLK99
yes
is the next entry completely enclosed?
yes
MLK22
NO
save new length and starting block
MLK99

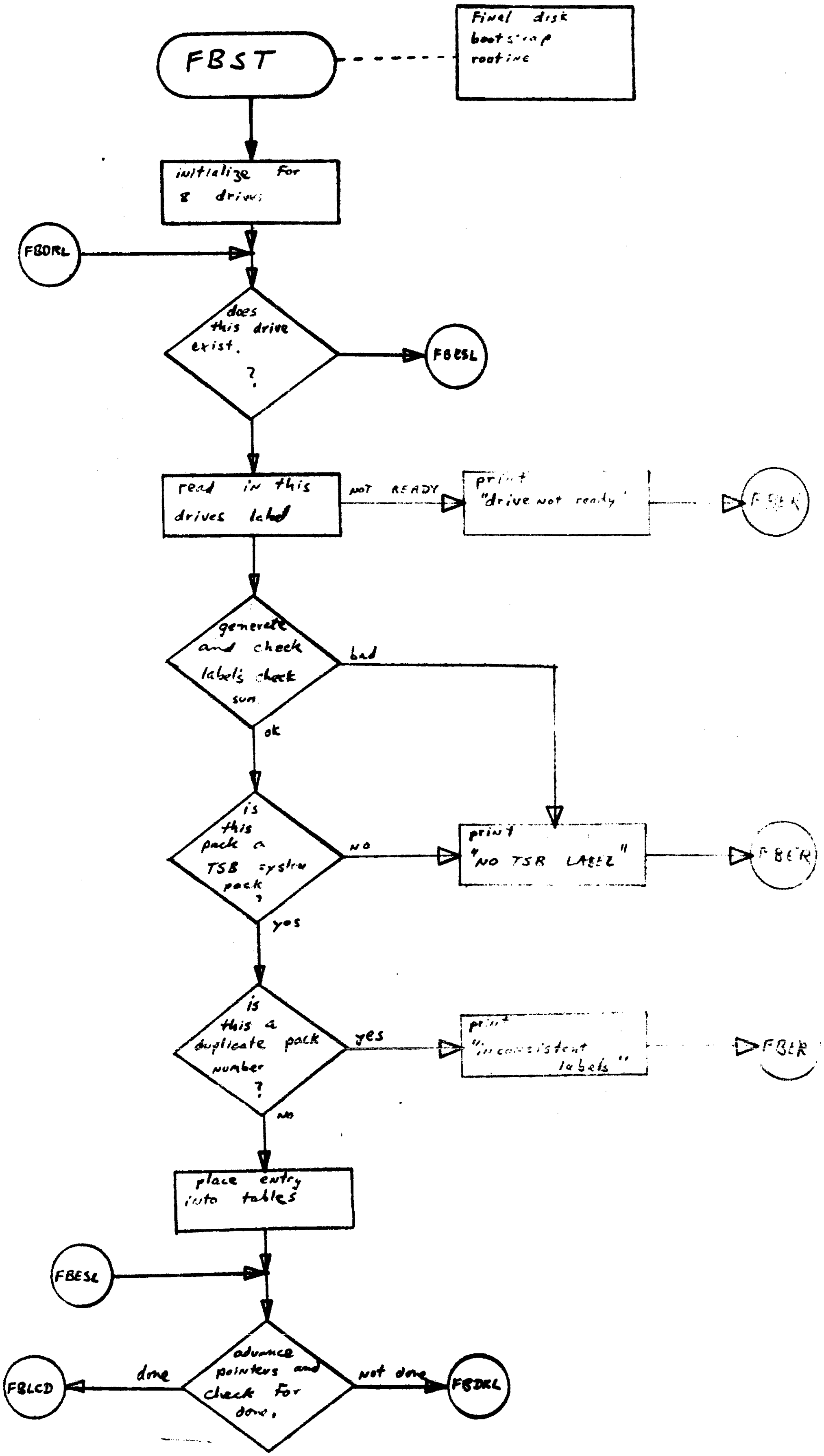

routine to get the next IDT track. used by IDT update routine
SPMB
save current ptr. get track information
anything in this track ?
yes
advance to next track
end of ID table
SPFR
read it in and set up the count of ID entries
RETURN

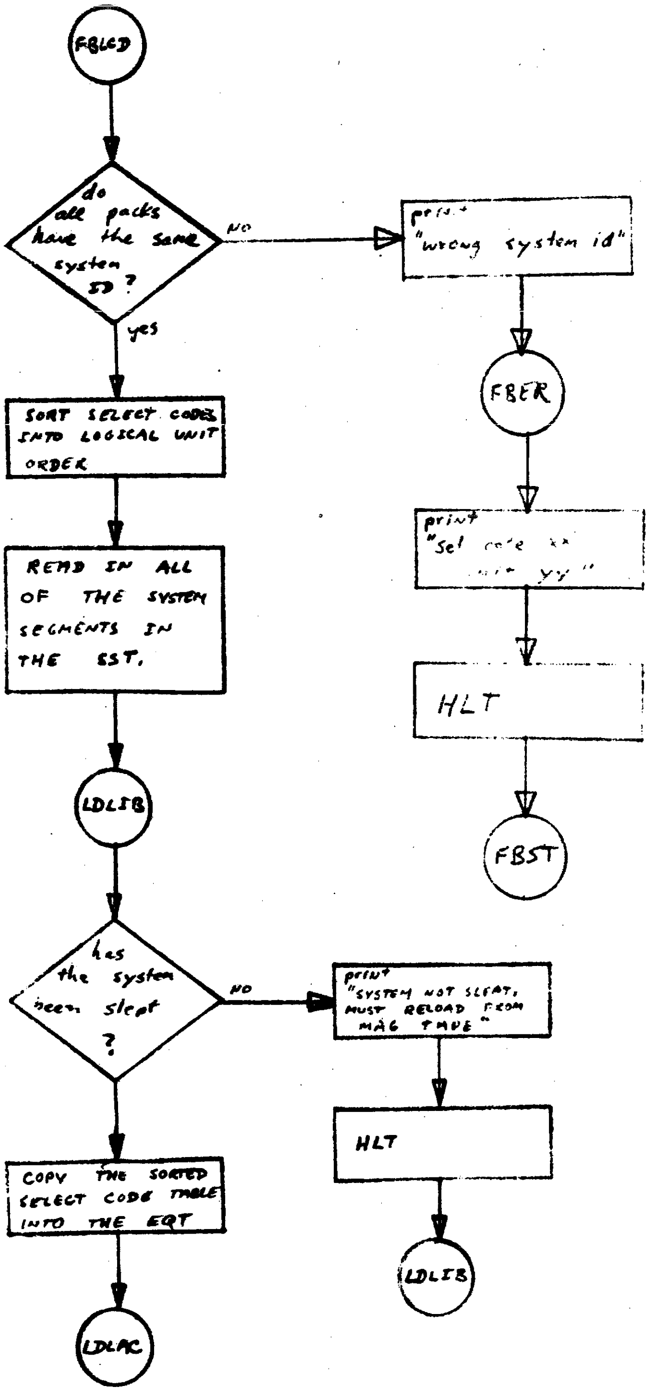

call with pointer to DIREC table. reads specified directory track if not already in core
DIRCE
track already in ?
yes
no
indicate which track now in.
read in this track
fetch length of track
RETURN

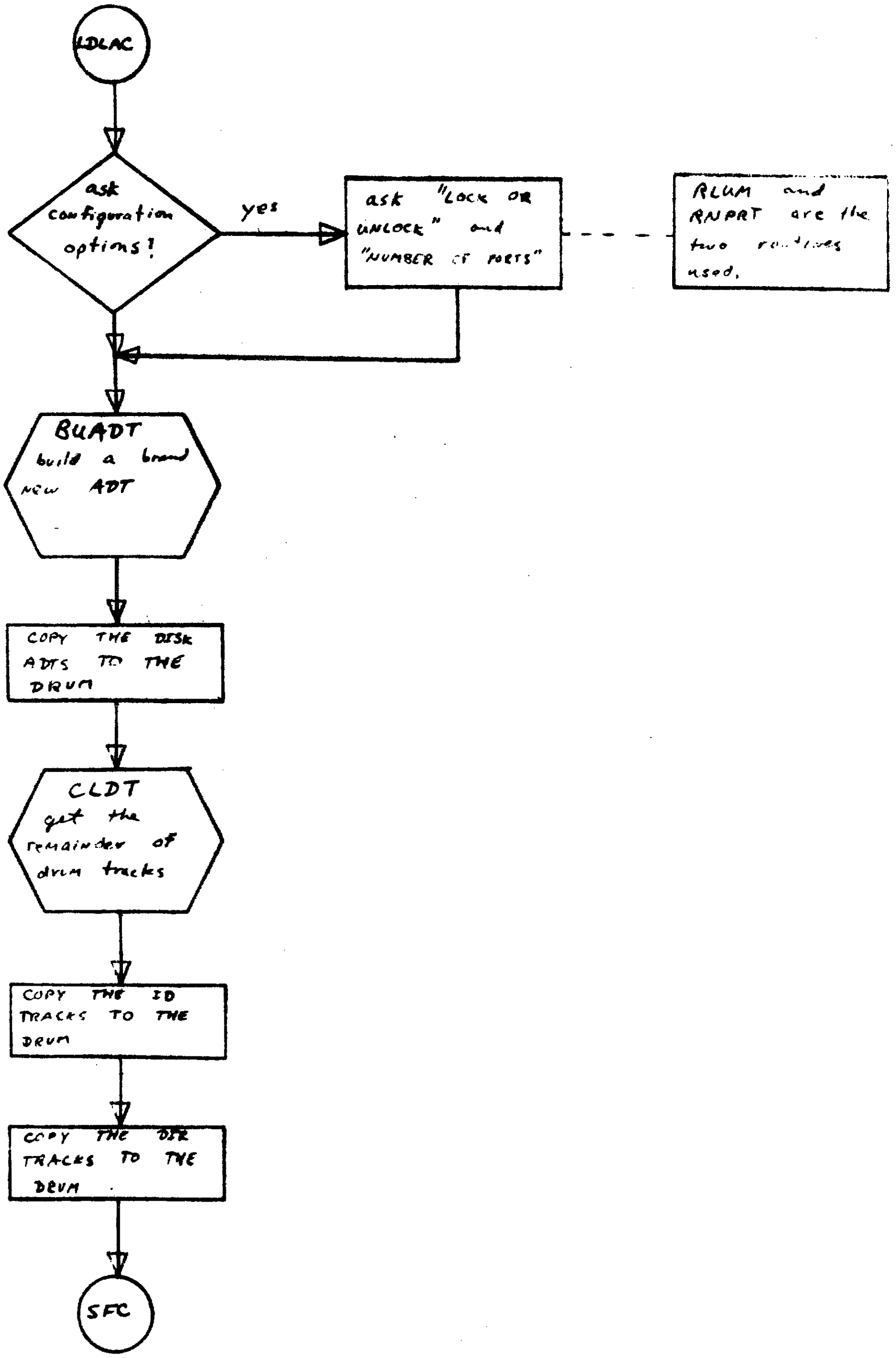

routine searches directory for specified entry
DIRLK
initialize track pointers to last track
does this track exist?
no, not end of table
no, end of table
NO FIND RETURN
yes
DIRCM
is the entry before or after this tracks first ?
before
adjust track pointers for previous track
after or equal
DIRGE
get this directory track into core,
initialize count and pointer to first entry
DIRCM
compare the search entry to the current dir entry
equal
FOUND RETURN
past
NO FIND RETURN
not far enough
advance pointer and check the count
not done
done
NO FIND RETURN

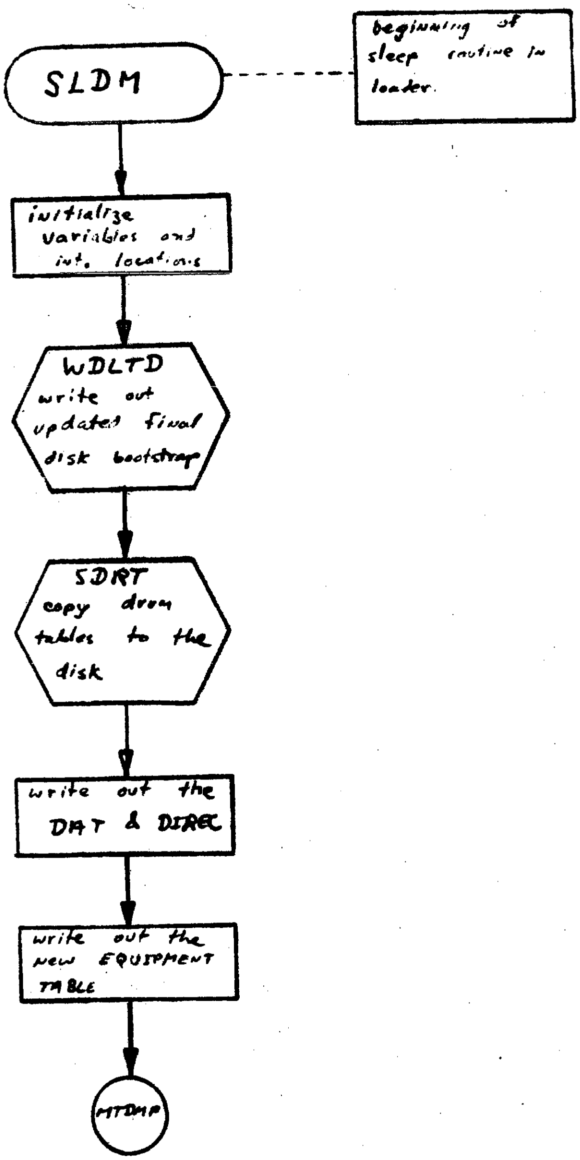

LKUNL
routine to process LOCK and UNLOCK commands
GTDNO get the logical disk number
no find
LDR22
PARC1
INTGR PARCK get valid track number
invalid
print "BAD TRACK #"
LDR22
LKUN1
print "BAD DELIMITER"
LDR22
other
check delimiter.
COMMA
CR
LKUN3
INTGR PARCK get valid track number
delimiter not CR
LKUN1
generate ptr to proper word & bit of trax table
compute difference
zero or negative
PARC1
clear appropriate bit
unlock
Lock or unlock ?
Lock
set appropriate bit
advance track address, are we done ?
NOT DONE
LKUN3
LDR22

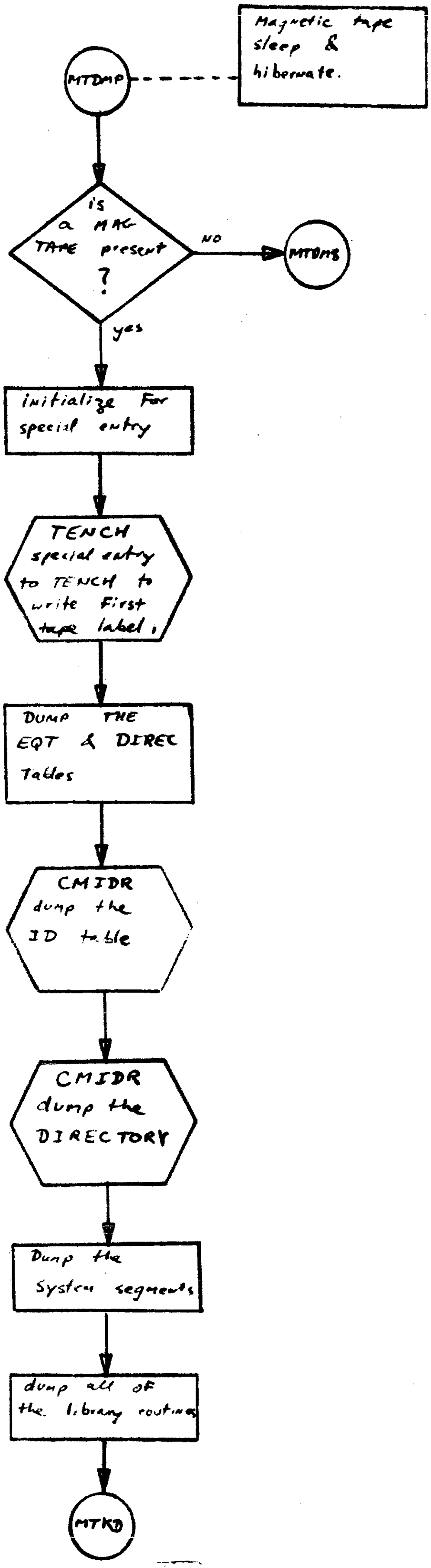

routine gets a piece of disk space from the current ADT.
FSDAD
initialize pointers
done with this ADT ?
yes
No find return
save disk address of current piece
alternate allocation option ?
NO
yes
is this piece legal to use? (compare original address)
NO
advance pointer
yes
is this piece long enough ?
NO
yes
is this solely a search ?
NO
SDADT remove space from the ADT entry
yes
Found RETURN

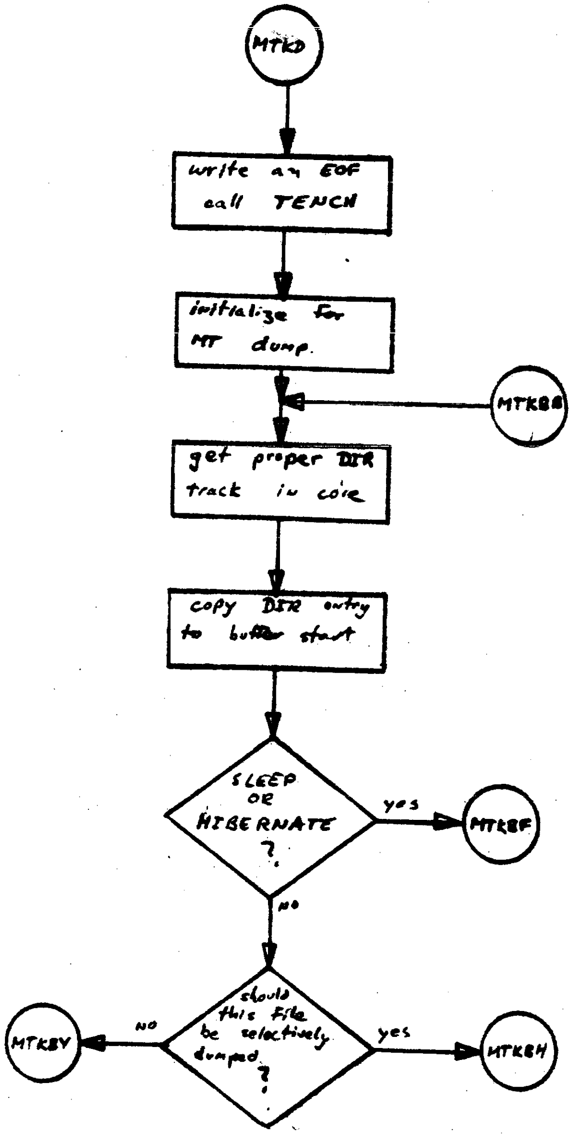

FBST
Final disk bootstrap routine
initialize for 8 drives
FBDRL
does this drive exist. ?
FBESL
read in this drives label
NOT READY
print "drive not ready"
FBER
generate and check label's check sum
bad
ok
is this pack a TSB system pack ?
NO
print "NO TSB LABEL"
FBER
yes
is this a duplicate pack number ?
yes
print "inconsistent labels"
FBER
no
place entry into tables
FBESL
advance pointers and check for done,
done
FBLCD
not done
FBDRL

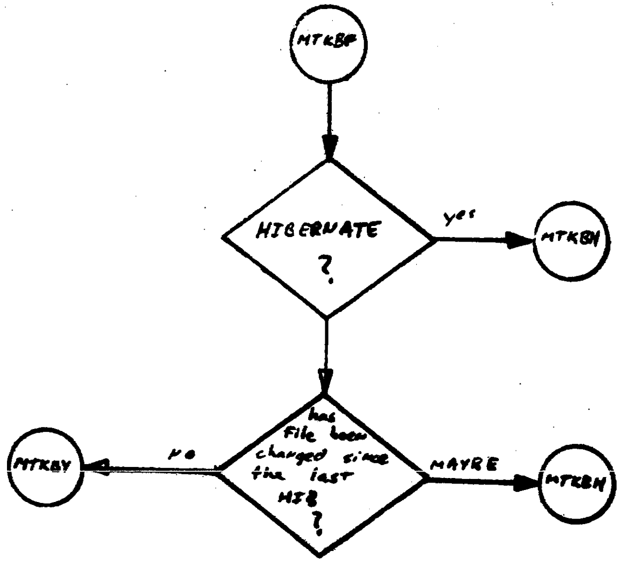

FBLCD
do all packs have the same system ID ?
NO
yes
print "wrong system id"
FBER
print "Sel code xx unit yy"
HLT
FBST
SORT SELECT CODES INTO LOGICAL UNIT ORDER
READ IN ALL OF THE SYSTEM SEGMENTS IN THE SST.
LDLIB
has the system been slept ?
NO
print "SYSTEM NOT SLEPT, MUST RELOAD FROM MAG TAPE"
HLT
LDLIB
COPY THE SORTED SELECT CODE TABLE INTO THE EQT
LDLPC

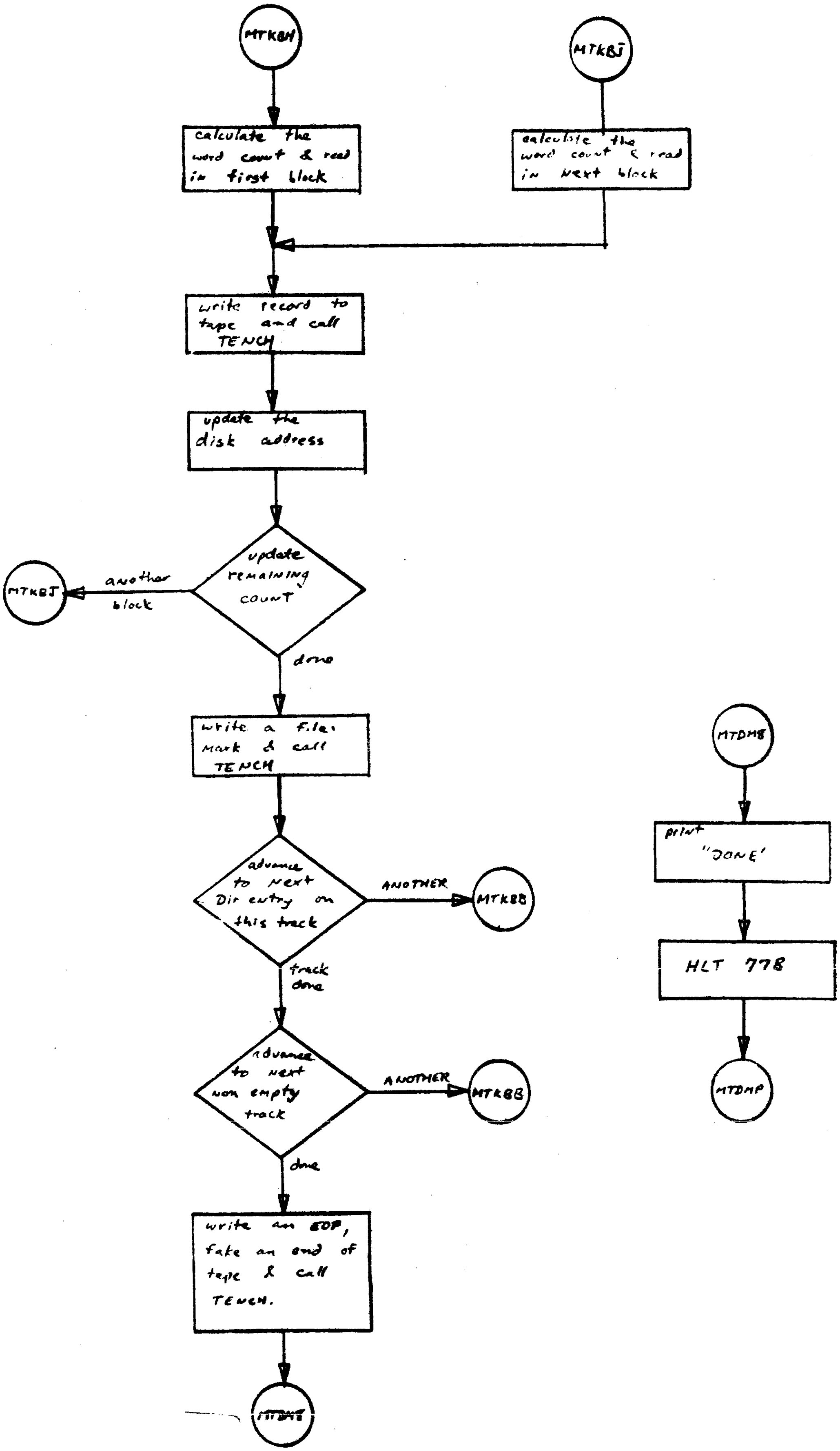

LDLAC
ask configuration options?
yes
ask "LOCK OR UNLOCK" and "NUMBER OF PORTS"
RLUM and RNPRT are the two routines used.
BUADT
build a brand new ADT
COPY THE DISK ADTS TO THE DRUM
CLDT
get the remainder of drum tracks
COPY THE ID TRACKS TO THE DRUM
COPY THE DIR TRACKS TO THE DRUM
SFC

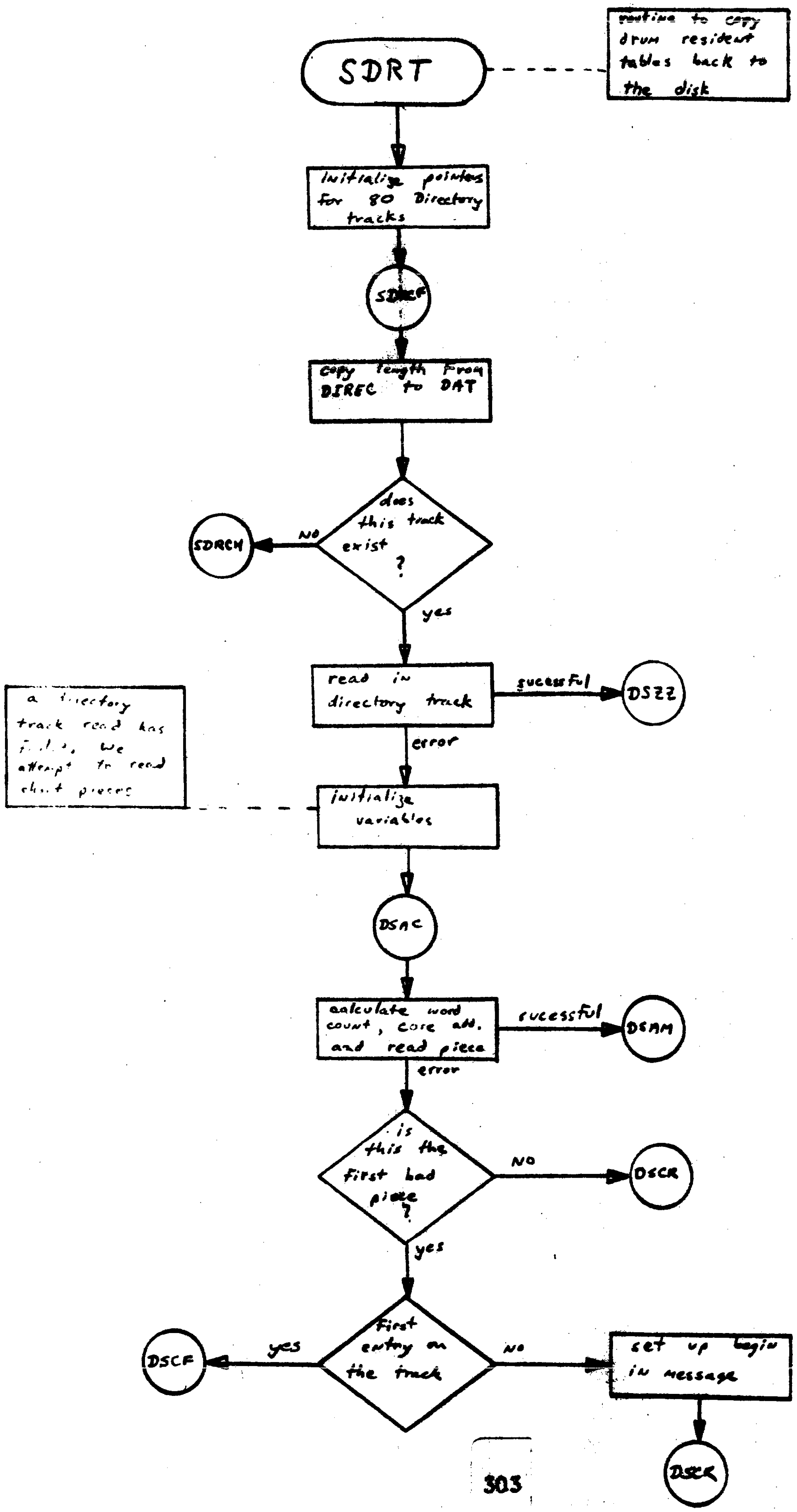

SLDM
beginning of sleep routine in loader.
initialize variables and int. locations
WDLTD
write out updated final disk bootstrap
SDRT
copy drum tables to the disk
write out the DAT & DIREC
write out the new EQUIPMENT TABLE
MTDMP

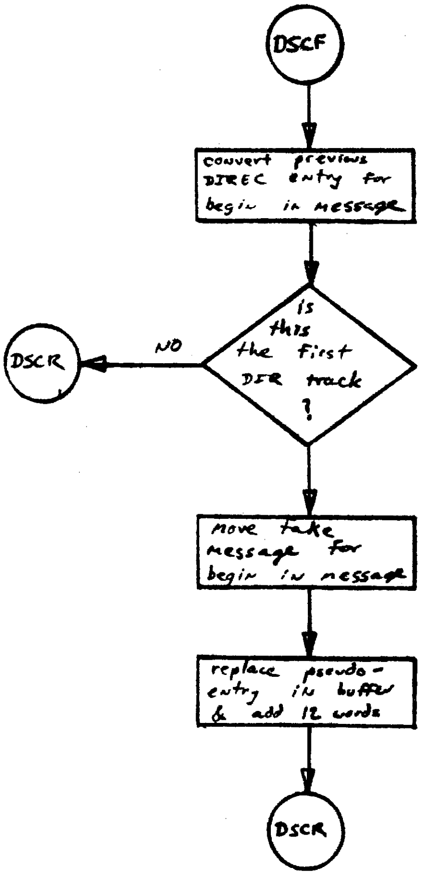

Magnetic tape sleep & hibernate.
MTDMP
is a MAG TAPE present ?
NO
MTDM8
yes
initialize For special entry
TENCH special entry to TENCH to write First tape label,
DUMP THE EQT & DIREC Tables
CMIDR dump the ID table
CMIDR dump the DIRECTORY
Dump the System segments
dump all of the library routines
MTKD

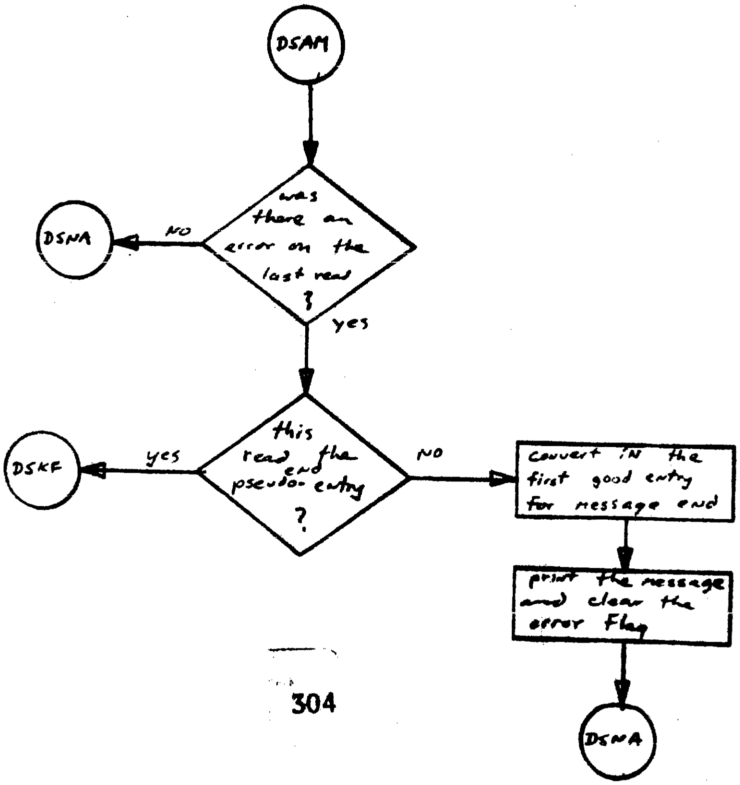

MTKD
write an EOF
call TENCH
initialize for
MT dump.
MTKBB
get proper DIR
track in core
copy DIR entry
to buffer start
SLEEP
OR
HIBERNATE
?
yes
MTKBF
NO
should
this file
be selectively
dumped
?
NO
MTKBY
yes
MTKBH

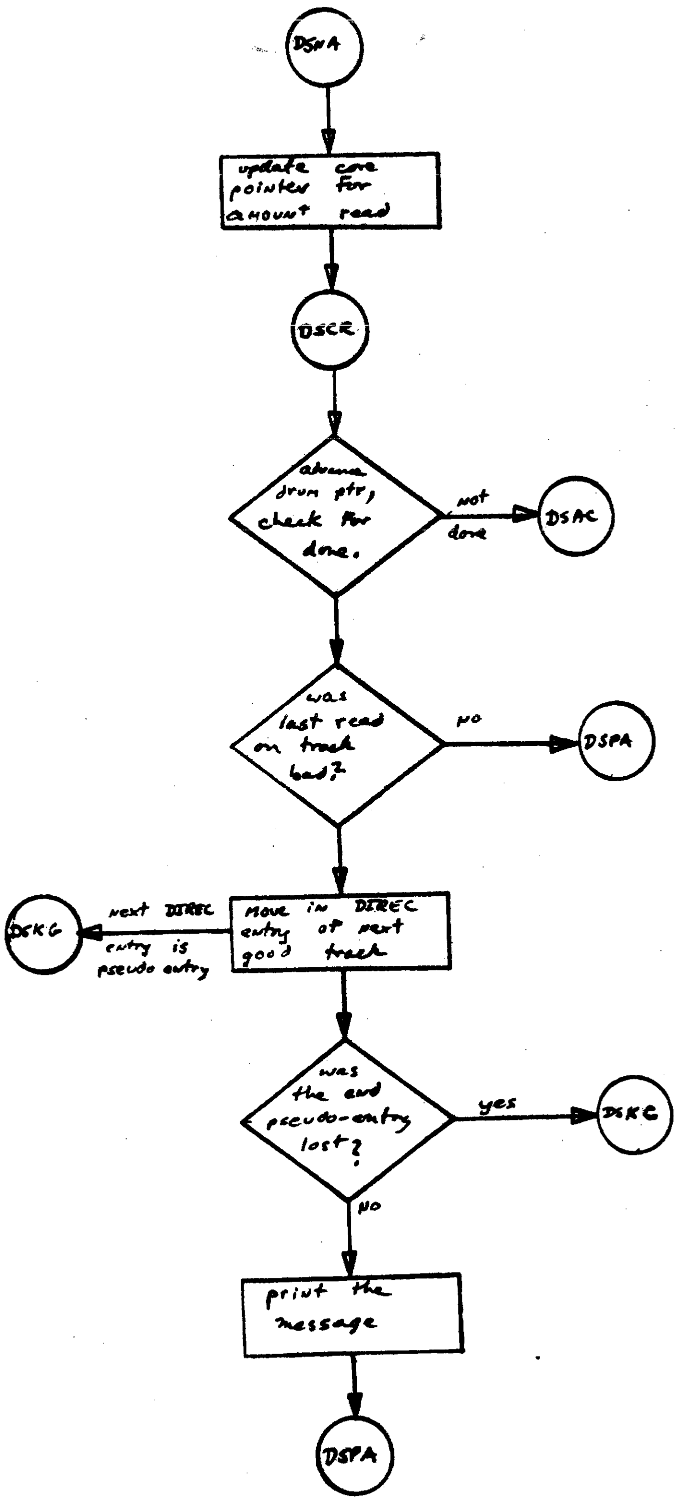

MTKBF
HIBERNATE
?
yes
MTKBH
has
File been
changed since
the last
HIB
?
NO
MTKBY
MAYBE
MTKBH

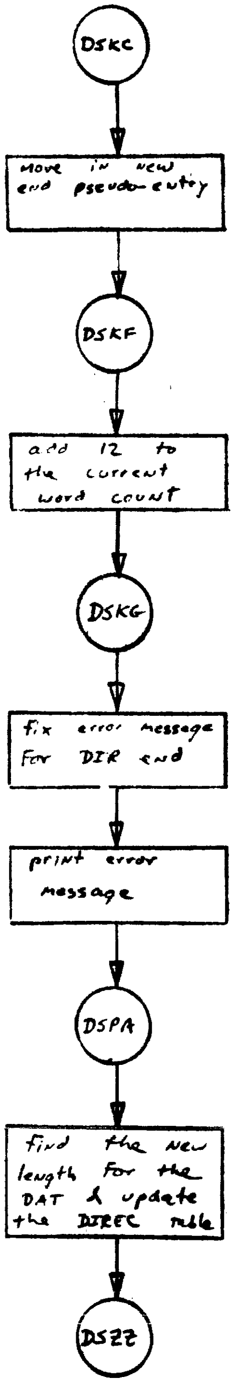

MTKBH
MTKBJ
calculate the word count & read in first block
calculate the word count & read in Next block
write record to tape and call TENCH
update the disk address
update remaining count
another block
MTKBJ
done
write a File Mark & call TENCH
advance to Next Dir entry on this track
ANOTHER
MTKBB
track done
advance to next non empty track
ANOTHER
MTKBB
done
write an EOF, fake an end of tape & call TENCH.
MTDM8
MTDM8
print "DONE'
HLT 778
MTDMP

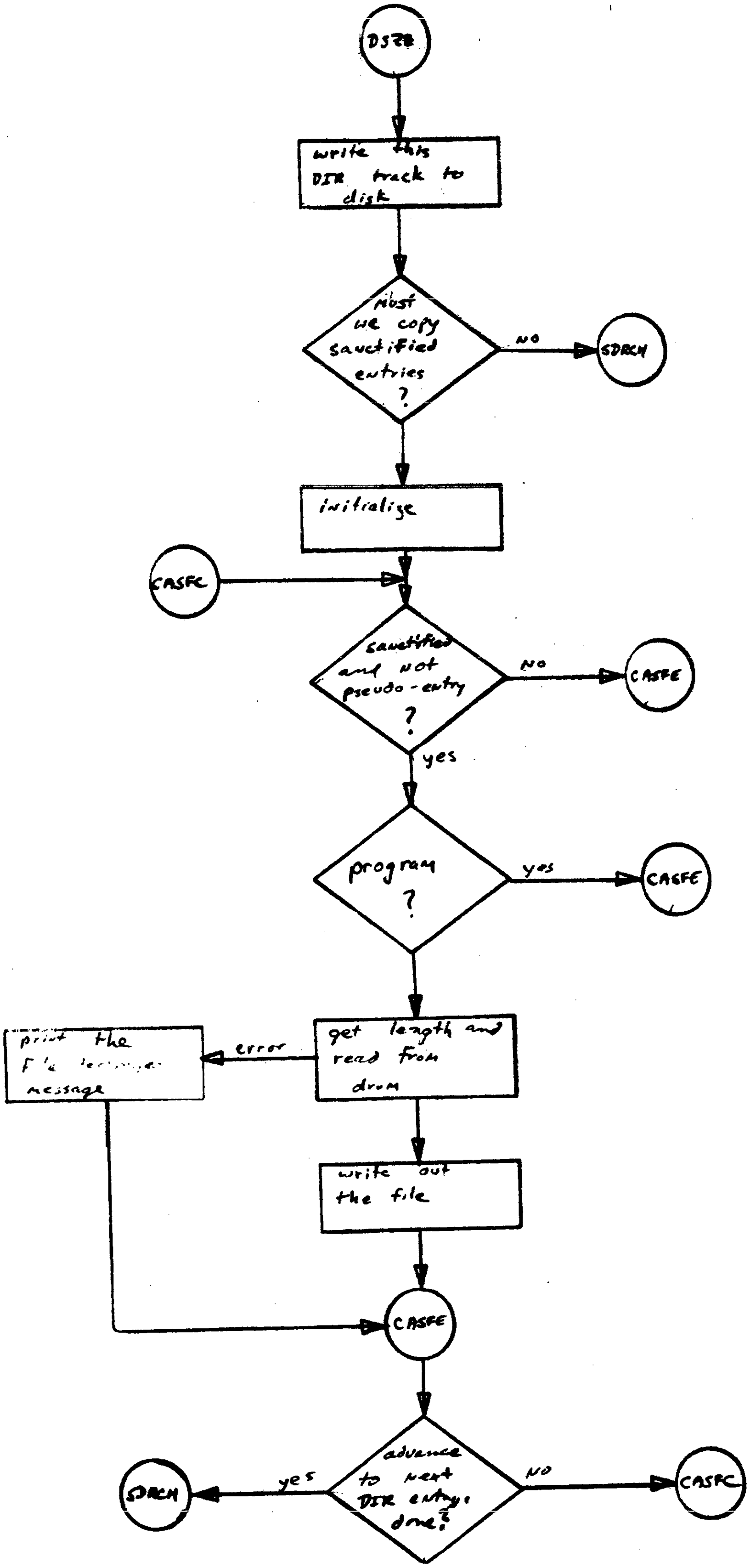

routine to copy drum resident tables back to the disk
SDRT
Initialize pointers for 80 Directory tracks
SDRCF
copy length From DIREC to DAT
does this track exist ?
NO
SDRCH
yes
read in directory track
successful
DSZZ
error
a directory track read has failed, we attempt to read short pieces
initialize variables
DSAC
calculate word count, core add. and read piece
sucessful
DSAM
error
is this the First bad piece ?
NO
DSCR
yes
First entry on the track
yes
DSCF
NO
set up begin in message
DSCR

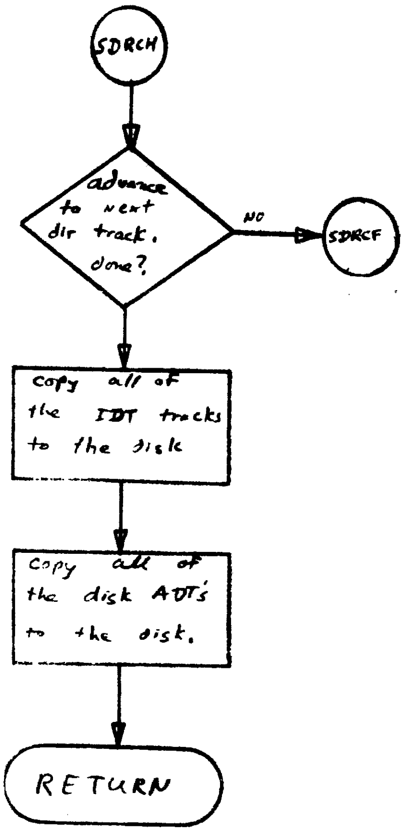

DSCF
convert previous DIREC entry for begin in message
is this the first DIR track ?
NO
DSCR
move take message for begin in message
replace pseudo-entry in buffer & add 12 words
DSCR

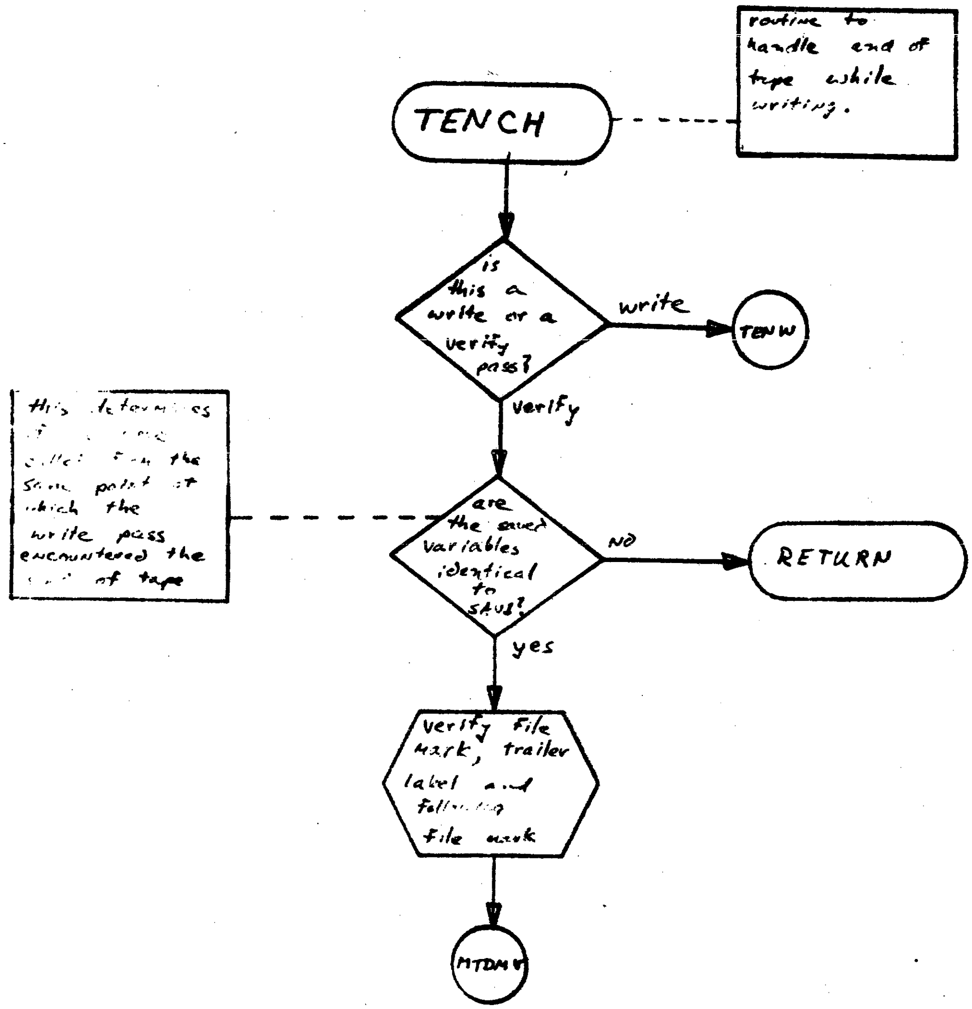

DSAM
was there an error on the last read ?
NO
DSNA
yes
this read the end pseudo-entry ?
yes
DSKF
NO
convert in the first good entry for message end
print the message and clear the error flag
DSNA


DSNA
update core pointer for amount read
DSCR
advance drum ptr, check for done.
not done
DSAC
was last read on track bad?
no
DSPA
Move in DIREC entry of next good track
next DIREC entry is pseudo entry
DSKG
was the end pseudo-entry lost?
yes
DSKG
no
print the message
DSPA

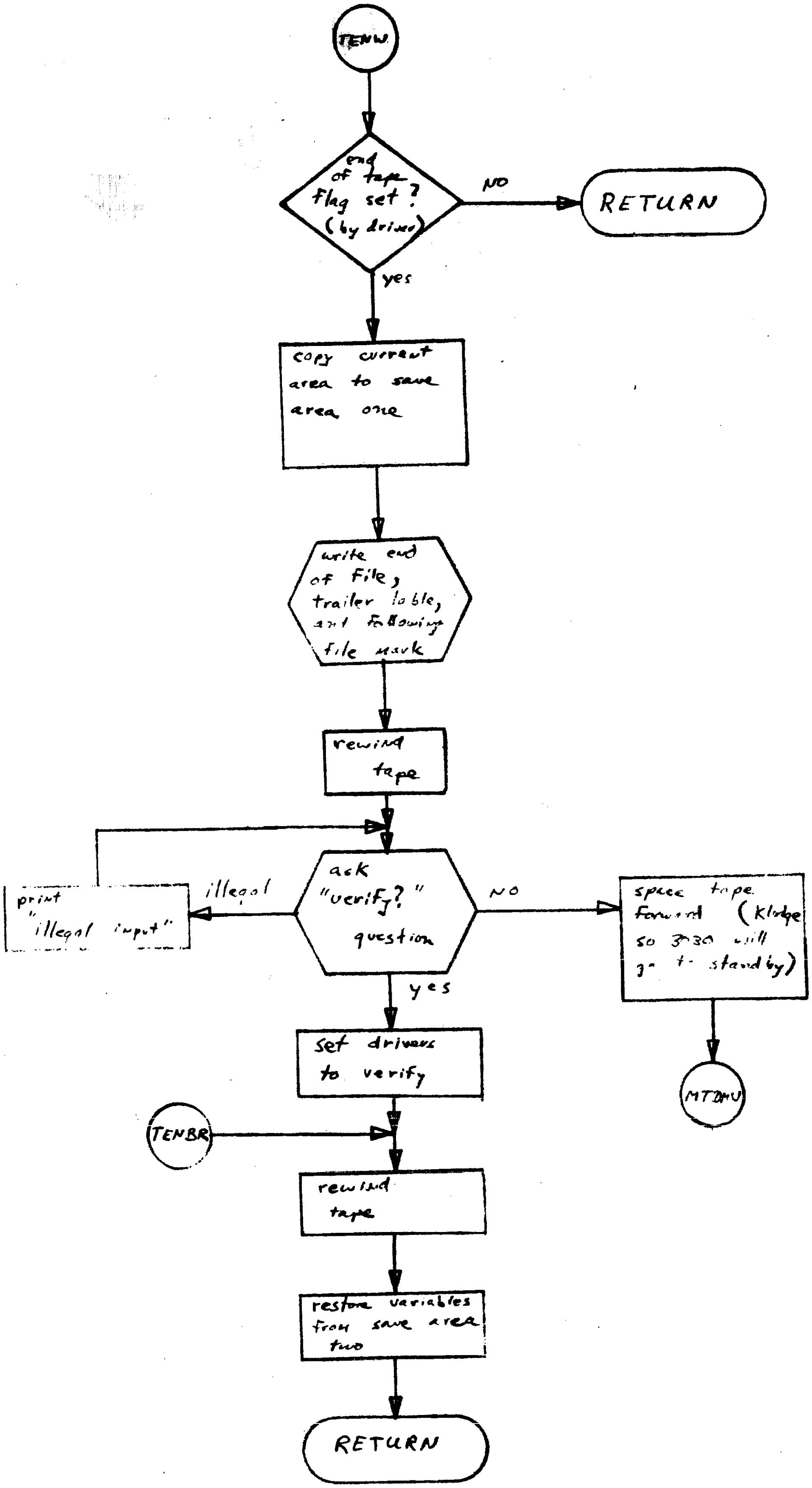

DSKC
move in new end pseudo-entry
DSKF
add 12 to the current word count
DSKG
fix error message for DIR end
print error message
DSPA
find the new length for the DAT & update the DIREC table
DSZZ

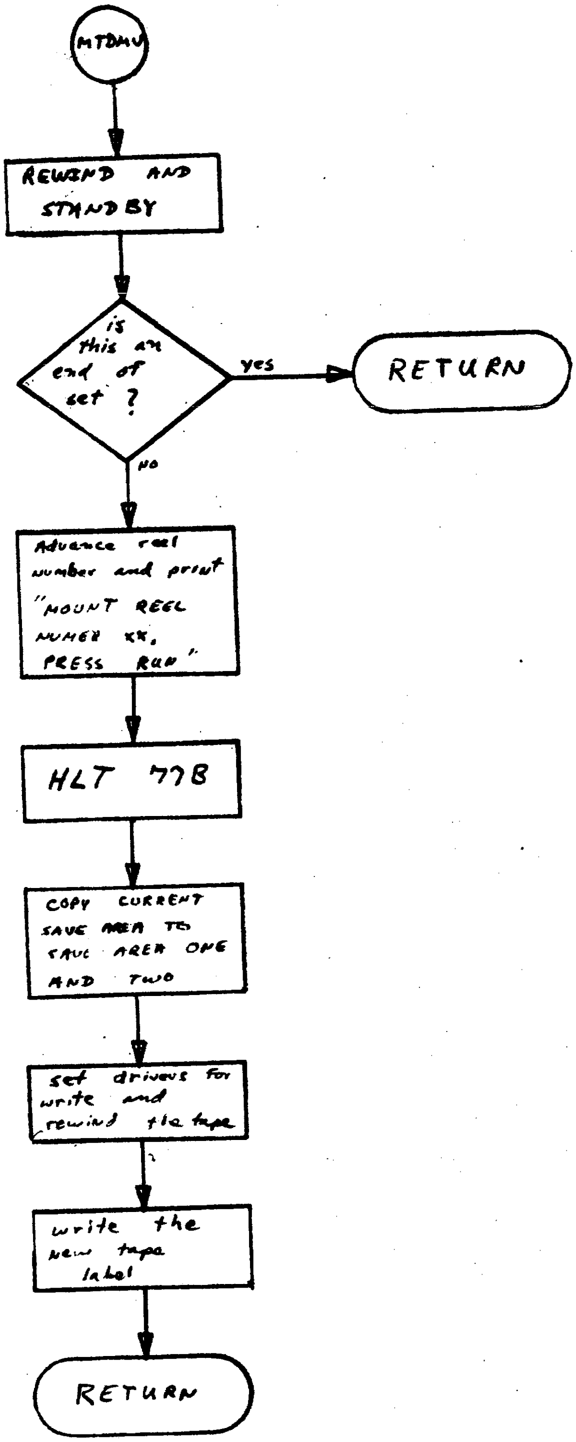

DSZB
write this DSR track to disk
Must we copy sanctified entries ?
NO
SDRCH
initialize
CASFC
sanctified and not pseudo-entry ?
NO
CASFE
yes
program ?
yes
CASFE
get length and read from drum
error
print the file [illegible] message
write out the file
CASFE
advance to next DSR entry; done?
yes
SDRCH
NO
CASFC

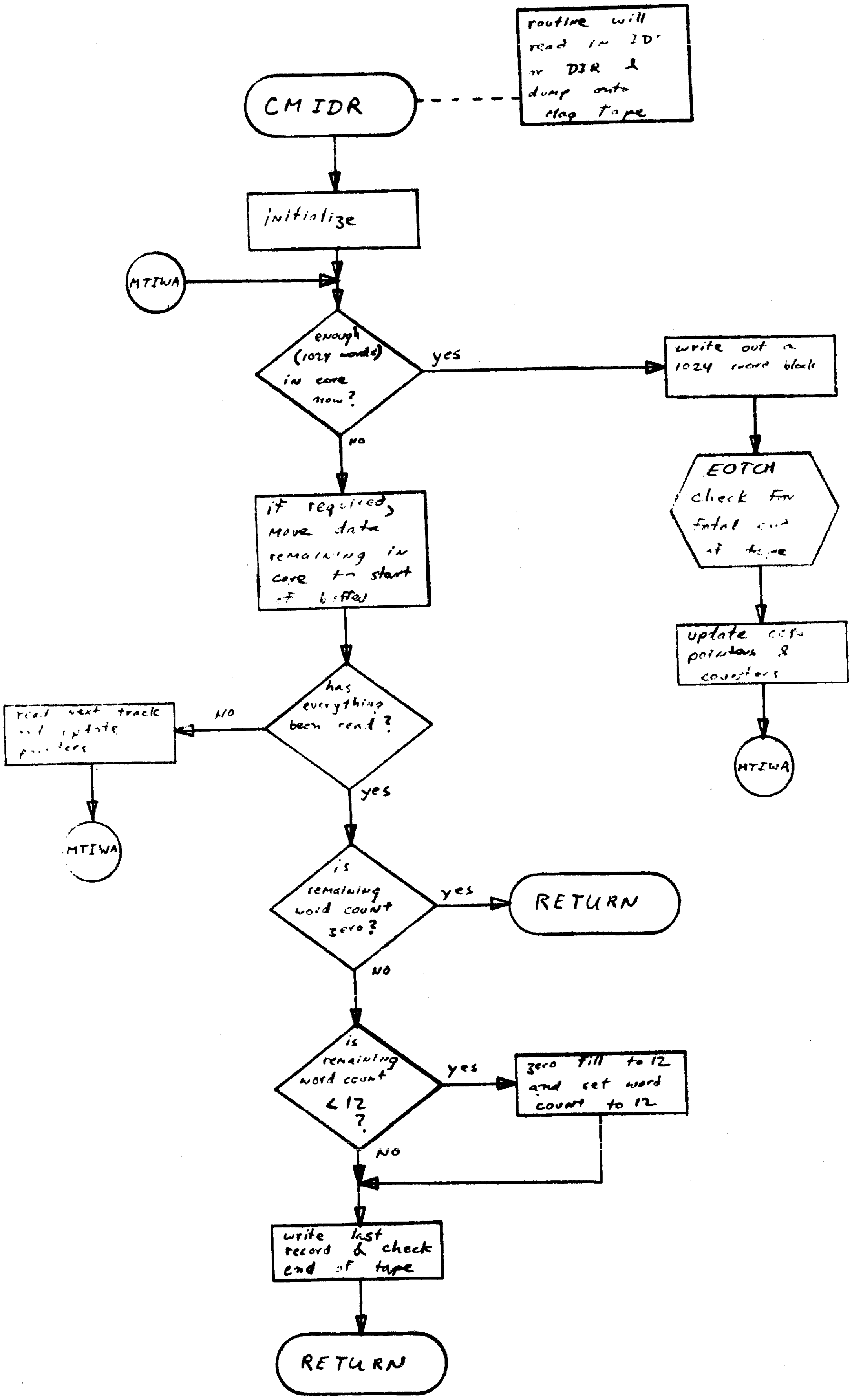

SDRCH
advance to next dir track. done?
NO
SDRCF
copy all of the IDT tracks to the disk
copy all of the disk ADT's to the disk.
RETURN

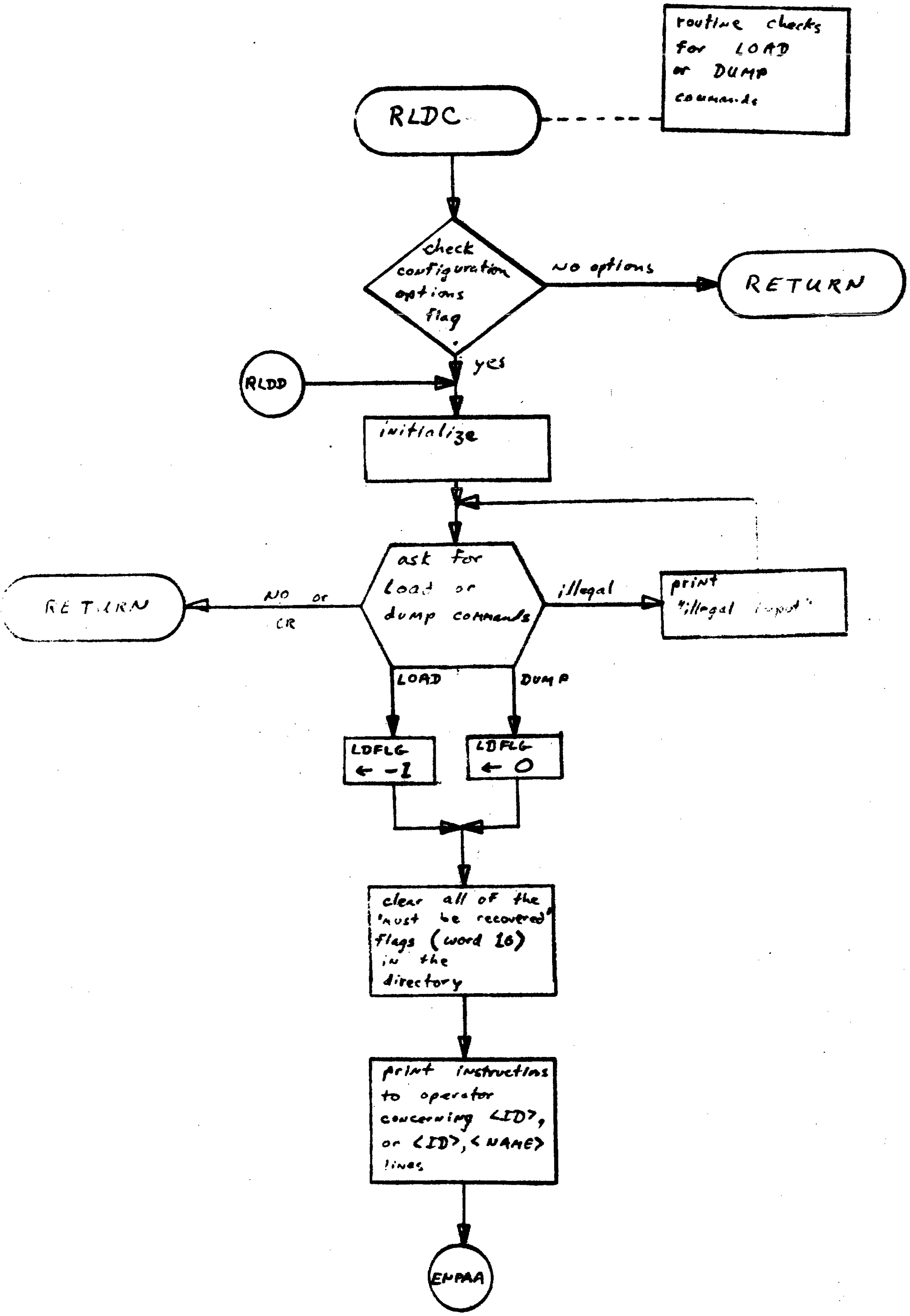

routine to handle end of tape while writing.
TENCH
is this a write or a verify pass?
write
TENW
verify
this determines if we are called from the same point at which the write pass encountered the end of tape
are the saved variables identical to SAVS?
NO
RETURN
yes
verify file mark, trailer label and following file mark
MTDMV

NOTE : RETURNS FROM THIS ROUTINE ARE ALWAYS TO THE ADDRESS PRESENTLY IN THE SAVE ZERO AREA. SINCE THIS SAVE AREA MAY BE CHANGED BY THIS ROUTINE, RETURNS ARE NOT ALWAYS TO THE IMMEDIATELY PRECEEDING CALL.

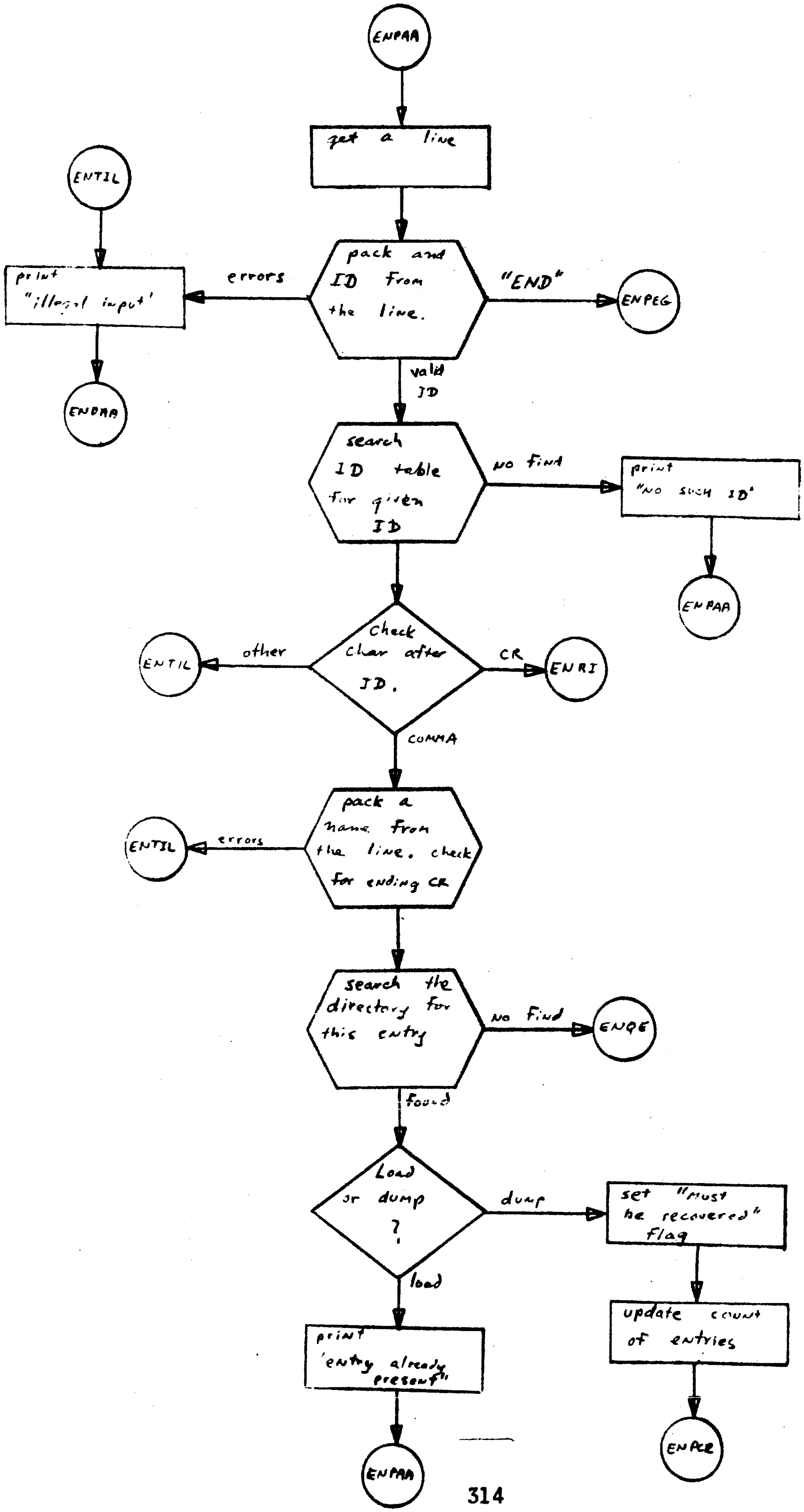

TENW
end of tape flag set? (by driver)
NO
RETURN
yes
copy current area to save area one
write end of File, trailer lable, and following file mark
rewind tape
ask "verify?" question
illegal
print "illegal input"
NO
space tape Forward (Kludge so 3030 will go to standby)
MTDHU
yes
set drivers to verify
TENBR
rewind tape
restore variables from save area two
RETURN

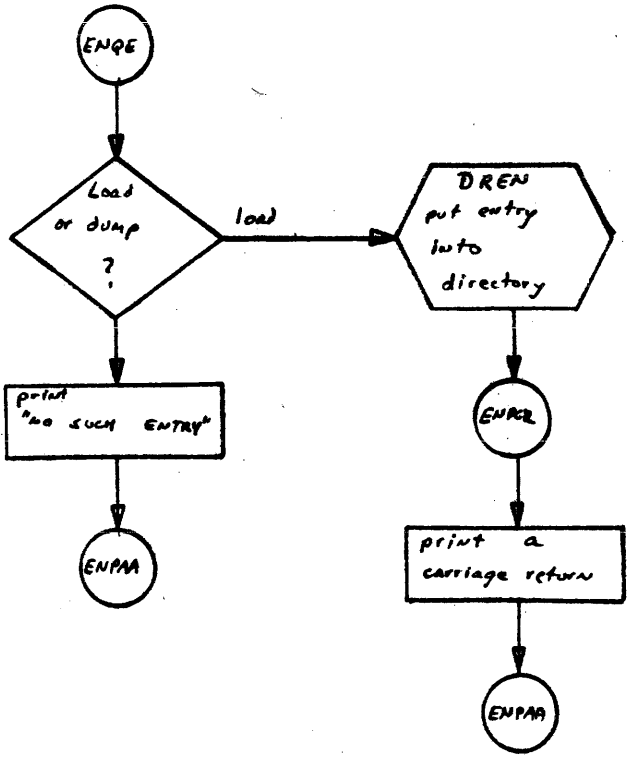

MTDMV
REWIND AND STANDBY
is this an end of set ?
yes
RETURN
NO
Advance reel number and print "MOUNT REEL NUMER XX, PRESS RUN"
HLT 77B
COPY CURRENT SAVE AREA TO SAVE AREA ONE AND TWO
set drivers for write and rewind the tape
write the new tape label
RETURN

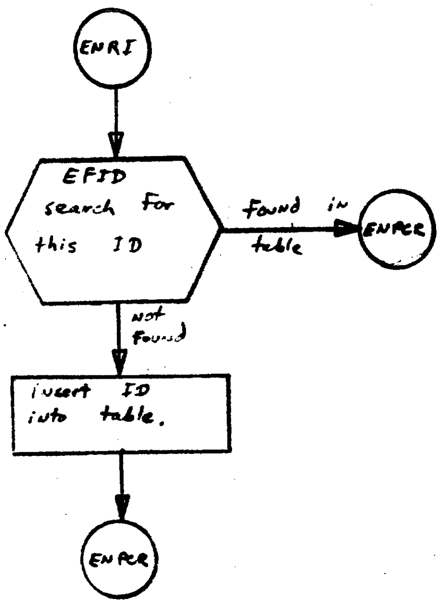

routine will read in ID' or DIR & dump onto Mag tape
CMIDR
initialize
MTIWA
enough (1024 words) in core now?
yes
write out a 1024 word block
EOTCH check for fatal end of tape
update core pointers & counters
MTIWA
NO
if required, move data remaining in core to start of buffer
has everything been read?
NO
read next track and update pointers
MTIWA
yes
is remaining word count zero?
yes
RETURN
NO
is remaining word count <12 ?
yes
zero fill to 12 and set word count to 12
NO
write last record & check end of tape
RETURN

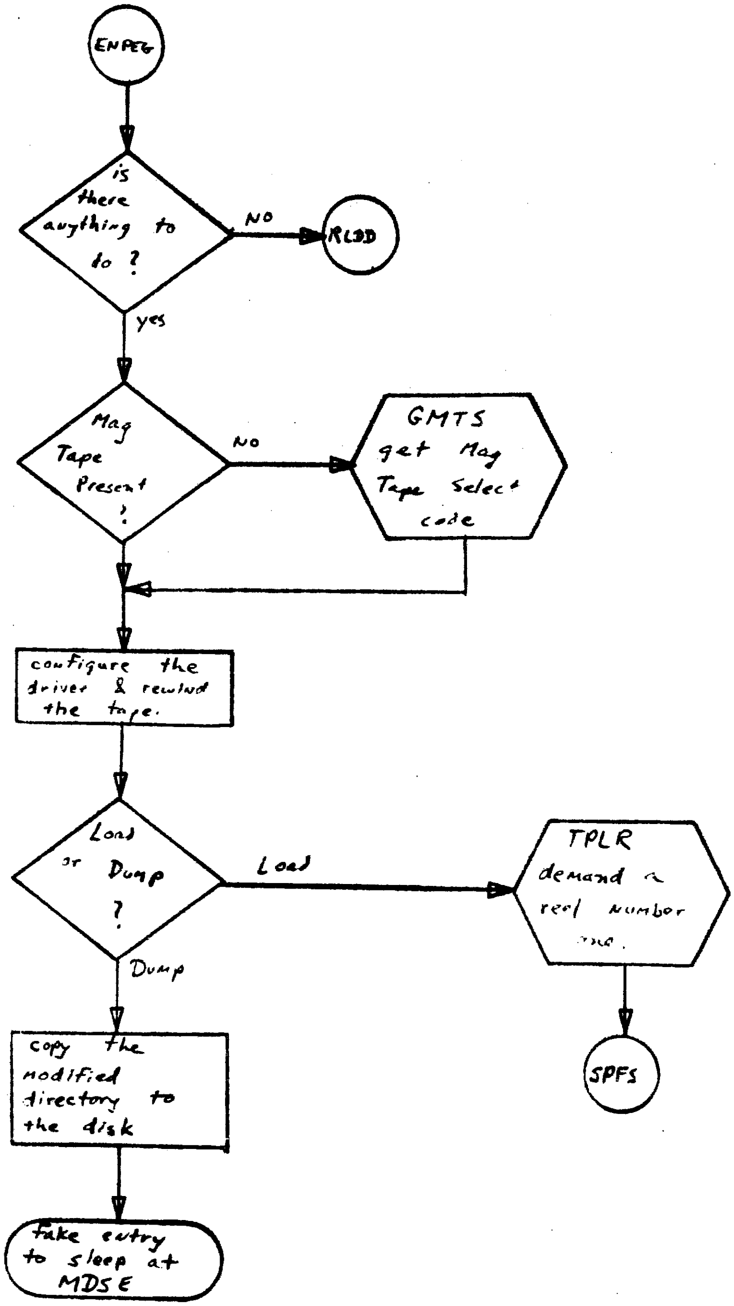

routine checks for LOAD or DUMP commands
RLDC
check configuration options flag
no options
RETURN
yes
RLDD
initialize
ask for Load or dump commands
NO or CR
RETURN
illegal
print "illegal input"
LOAD
DUMP
LDFLG ← -1
LDFLG ← 0
clear all of the 'must be recovered' flags (word 16) in the directory
print instructions to operator concerning <ID>, or <ID>,<NAME> lines
ENPAA

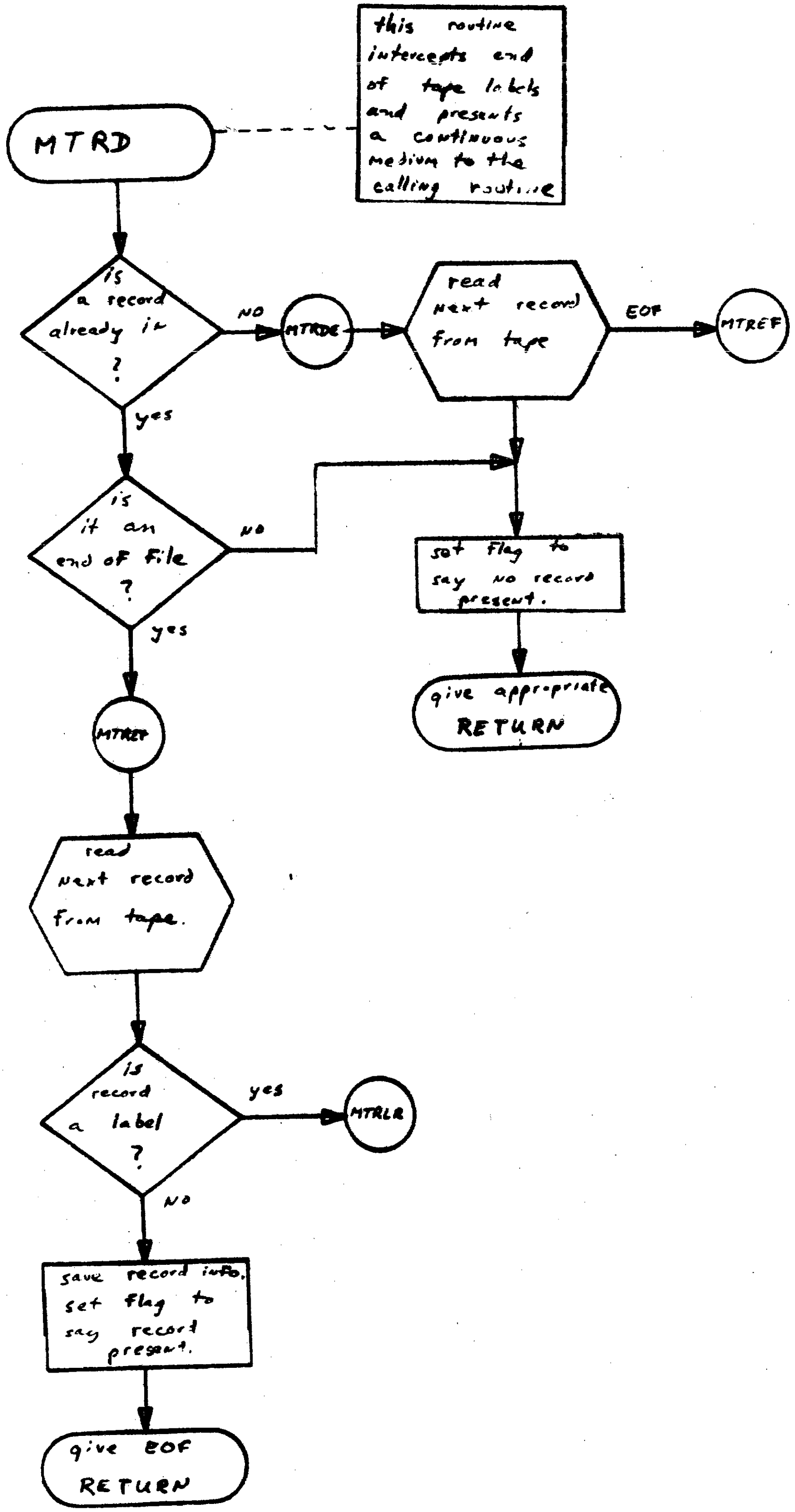
LOAD & DUMP page 2

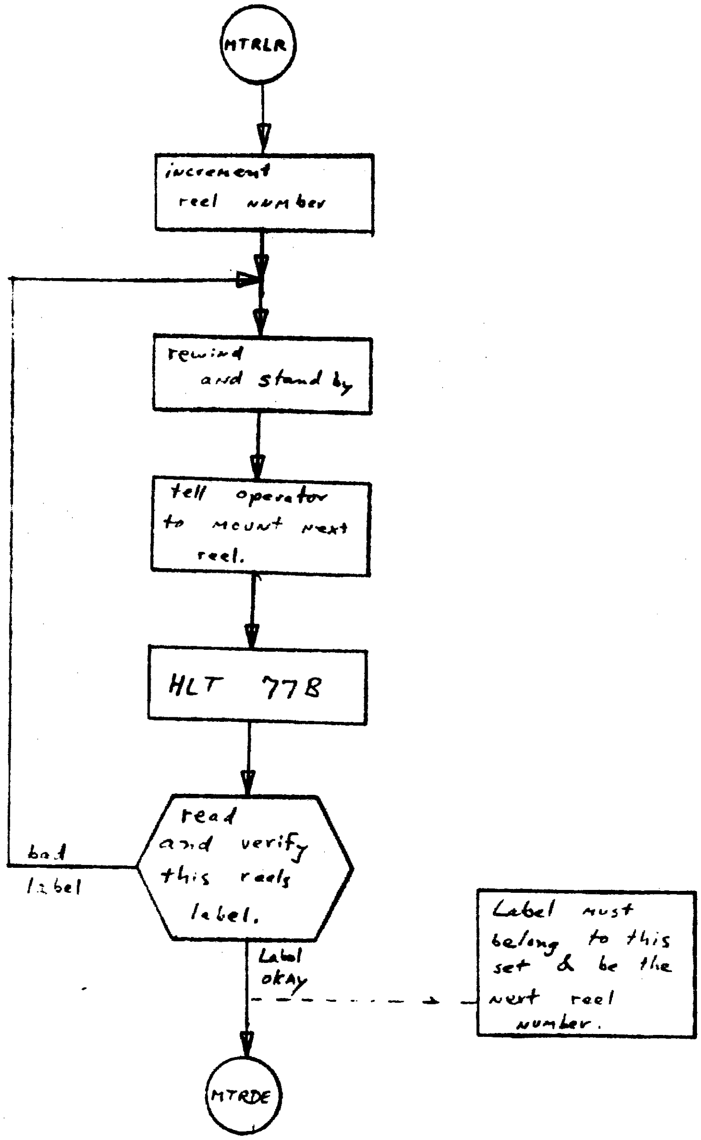


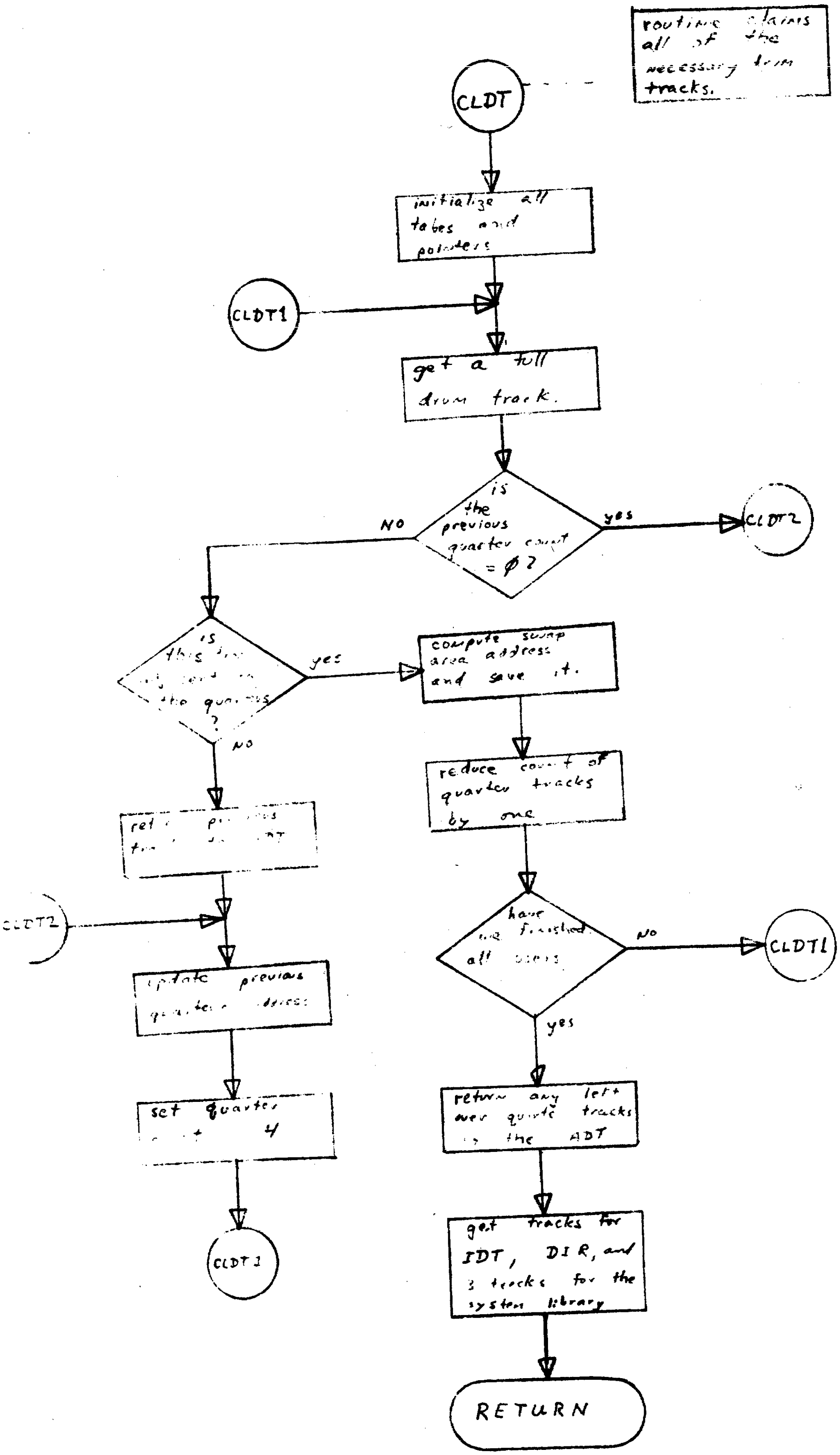

ENQE
Load or dump ?
load
DREN put entry into directory
print "no such ENTRY"
ENPAA
ENPCR
print a carriage return
ENPAA


ENRI
EFID search for this ID
found in table
ENPCR
not found
insert ID into table.
ENPCR


ENPEG
is there anything to do ?
NO
RLDD
yes
Mag Tape Present ?
NO
GMTS get Mag Tape Select code
configure the driver & rewind the tape.
Load or Dump ?
Load
TPLR demand a reel number one.
SPFS
Dump
copy the modified directory to the disk
fake entry to sleep at MDSE


MTRD
this routine intercepts end of tape labels and presents a continuous medium to the calling routine
is a record already in ?
NO
MTRDE
read next record from tape
EOF
MTREF
yes
is it an end of File ?
NO
set flag to say no record present.
give appropriate RETURN
yes
MTREF
read next record from tape.
is record a label ?
yes
MTRLR
NO
save record info, set flag to say record present.
give EOF RETURN


MTRLR
increment reel number
rewind and stand by
tell operator to mount next reel.
HLT 77B
read and verify this reels label.
bad label
Label okay
Label must belong to this set & be the next reel number.
MTRDE



routine claims all of the necessary drum tracks.
CLDT
initialize all tabes and pointers
CLDT1
get a full drum track.
is the previous quarter count = ø ?
NO
yes
CLDT2
is this [illegible] the quarters ?
yes
NO
compute swap area address and save it.
ret [illegible] previous [illegible] count
reduce count of quarter tracks by one
CLDT2
update previous quarter address
have we finished all users
NO
CLDT1
yes
set Quarter count 4
return any left over quarter tracks to the ADT
CLDT1
get tracks for IDT, DIR, and 3 tracks for the system library
RETURN

Bootstrap procedure

start BBDL at 77760B

BBDL   reads in drum sector Ø

Sector Ø   reads in drum sector 1

Sector 1   reads in disc blocks 1 & 2
NOTE: this loader is configured to read from unit Ø of the disc controller in select code 17B.

Disc block 1 & 2   this block scans all disc drives and builds a table of drives vs. logical unit number. This bootstrap contains the system segment table and uses it to read in the system.

Sleep Tape Format

The following lists the records on a sleep tape in order starting from the load point marker.

Tape Label

The tape label is a seven word record and appears at the beginning and end of each sleep tape. The words are:

0	ASC 1,LB
1	ASC 1,TSB
2	(unused)
3	(reel number)
4	(year e.g. 71)
5	(hour of year)
6	(tenth of seconds)

EQT

∿ 120B words

DIREC

∿ 560 words

IDT

1024 word blocks, the IDT and Directory are packed into 1024 word blocks as they are read in, without regard to track or entry boundaries.

Directory

1024 word blocks

see IDT format

System Segment Table

System

The system is written out according to the information in the System Segment Table.

System Library

Each system library program is dumped in a separate record.

EOF

This end of file ends the system information. It must be on the first reel of tape.

USER Library

Programs and Files are written out in 1024 word records (4 blocks. Files remain interleaved. i.e. the first tape record of a 2η block long File contains records 1, η + 1, 2, η + 2). The First tape record contains the 12 word directory entry at its beginning and may be up to 1036 words long. A File mark is written after the last tape record of each program or File.

EOF

An EOF in place of an initial record of a program or File indicates the end of the sleep set.

NOTES:

1. When an end of tape marker is detected, the mag tape dump will write an EOF, a label record, and another EOF and ask for another reel. During reading, the sequence of an EOF followed by a label (7 word) record will indicate the end of a reel. No other test for end of reel is possible.