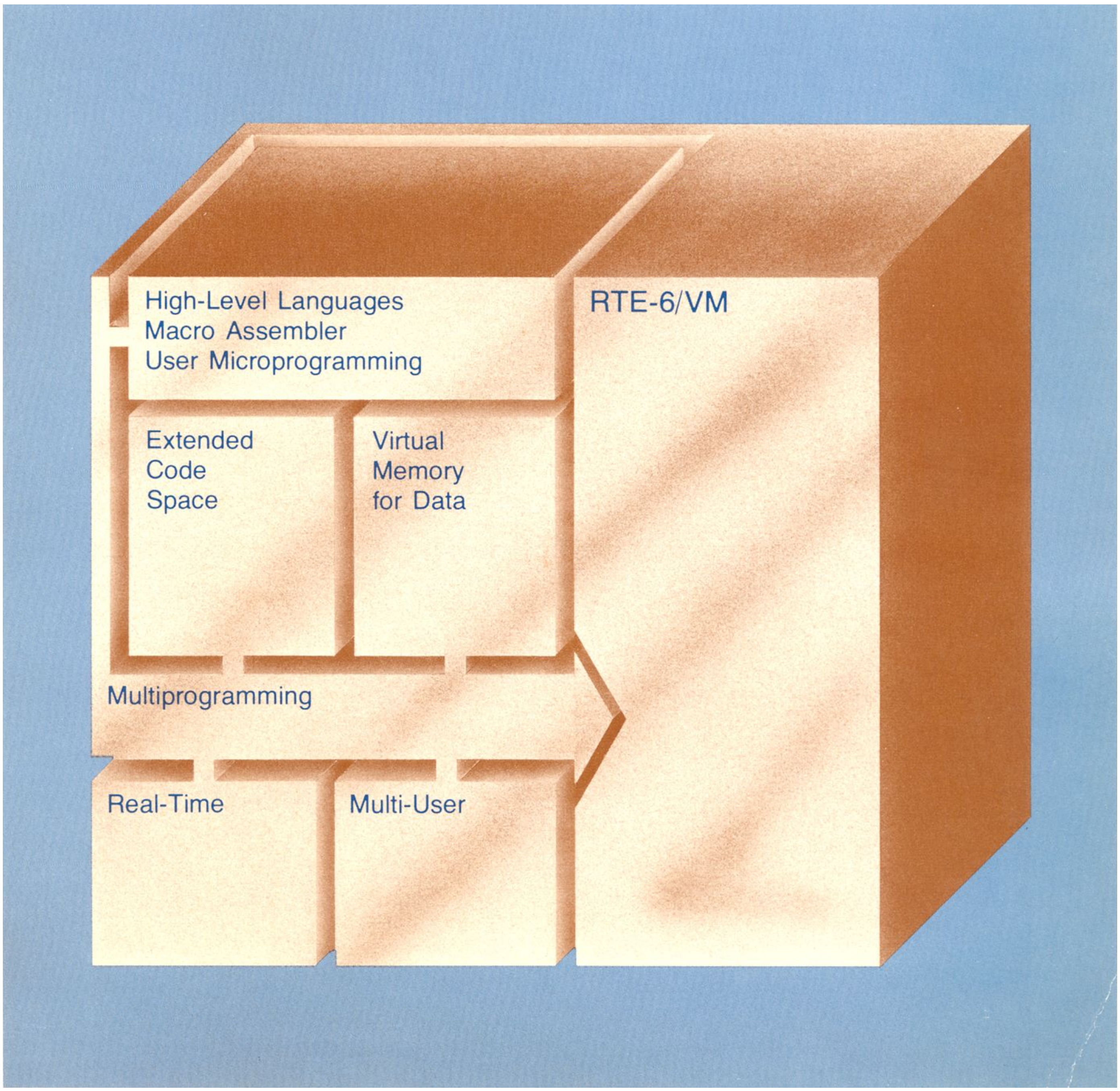
RTE-6/VM
Technical Specifications



RTE-6/VM
Technical Specifications

HEWLETT PACKARD

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014

MANUAL PART NO. 92084-90015
Printed in U.S.A. April 1983
E0483

PRINTING HISTORY

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, Update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past Updates, however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all Updates.

To determine what software manual edition and update is compatible with your current software revision code, refer to the appropriate Software Numbering Catalog, Software Product Catalog, or Diagnostic Configurator Manual.

First Edition .... Apr 1983

Table of Contents

RTE-6/VM DISPATCHER — CHAPTER 1

1.1 INTRODUCTION

The RTE-6/VM dispatcher is the central decision-making portion of the operating system. Every operator command and EXEC request in the system returns to the dispatcher entry point $XCQ. The dispatcher schedules the programs to execute, determines the area of execution, and controls CPU access. The dispatcher is made up of three modules: DISP6, DISPX, and OS6DP, that perform the following functions:

1. Program execution ordering
2. Program load and swap
3. Segment load
4. Multilevel Segmentation (MLS) node load
5. Program abortion
6. Memory management
7. Timeslice management
8. Critical program mapping

1.2 PARTITION USAGE

The dispatcher manages memory in user defined, fixed partitions. The total number of partitions in the system, specified by the generator, is contained in the word $MNP. Each partition is represented in memory by an entry in the Memory Allocation Table (MAT). Entry point $MATA points to the table, which extends from the entry point upward toward high memory. The format of each partition entry in the Memory Allocation Table is shown in Figure 1-1.

```
+--------------------------------------------------------------+
|                                                              |
|   15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0    Word    |
|  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+           |
|  | MAT Link Word                                 | 0  MLNK   |
|  |-----------------------------------------------|           |
|  | Partition Occupant Priority                   | 1  MPRIO  |
|  |-----------------------------------------------|           |
|  | ID Segment Address of Occupant                | 2  MID    |
|  |--|--|--|--------|-----------------------------|           |
|  | M|//| D|////////| Physical Start Page of      | 3  MADR   |
|  |  |//|  |////////| Partition                   |           |
|  |--|--|--|--|-----|-----------------------------|           |
|  | R| C| S| E|/////| Number pages in Partition   | 4  MLTH   |
|  |  |  |  |  |/////| (exclude Base Page)         |           |
|  |--|--|--|--|-----|--------------------|--------|           |
|  |RT|///////////////////////////////////| STATUS | 5  MRDFL  |
|  |  |///////////////////////////////////|        |           |
|  |--|-----------------------------------|--------|           |
|  | Subpartition Link Word (SLW)                  | 6  MSUBL  |
|  +-----------------------------------------------+           |
|                                                              |
+--------------------------------------------------------------+
```

Figure 1-1. Memory Allocation Table Entry Formats.

Where:

```
MAT
Link
Word = -1 if partition not defined during system
          generation or by parity error

     =  0 if end of list

M    =  1 if MAT entry is for a mother partition

D    =  1 if program dormant after save resource
          or serially reusable termination

R    =  1 if partition is reserved

C    =  1 if partition in use as part of chained
          partition

S    =  partition is also a shared EMA partition

E    =  partition is active in a shared EMA mode

RT   =  1 if MAT entry is for realtime partition
```

STATUS program dispatch status:

0 - program being loaded
1 - program is in memory
2 - segment being loaded or swapped out
3 - program is swapped out
4 - subpartition swap-out started for mother partition
5 - subpartition completed. Mother partition cleared.

Subpartition Link Word:

= 0 if MAT entry is not a subpartition or a mother partition

= next subpartition address if this is a subpartition

= mother partition MAT address if this is the last partition entry.

There are three different types of partitions:

1. Real Time (RT), headed by $RTFR at system startup.

2. Background (BG), headed by $BGFR at system startup.

3. Mother (MOM), headed by $CFR at system startup.

These three lists are established by the generator in order of increasing size. The only purpose for establishing RT partitions and BG partitions is to keep the two classes of programs, RT and BG, from contending for memory.The two classes of partitions are actually identical. Mother partitions are primarily for EMA, VMA and MLS programs.

Mother partitions are automatically defined during generation when you respond with a "YES" answer to the "SUBPARTITIONS?" prompt issued when a partition is larger than the maximum addressable space. Although Mother partitions are in a separate list, subpartitions may be linked into either a Mother partition or linked into a BG or RT (either free or allocated) list.

When the subpartitions are part of a Mother partition, the Mother partition MAT entry word 6, the Subpartition Link Word (SLW), will point to the Link Word (word 0) of the first subpartition. The SLW of this subpartition will point to the Link Word of the next subpartition, and so on throughout the subpartition chain. The SLW of the last subpartition will point to the Link Word of the Mother, if this is a circular list. This sequence is shown in Figure 1-2.

If no subpartitions are actually defined, but the response was "YES" to the

"SUBPARTITIONS?" prompt, the Mother partition is defined and its SLW points to the Link Word of the same MAT entry.

When a Mother partition is in use, the entire chain of subpartitions is considered to be in use, and and each subpartition's C bit is set. The chained partitions are thus treated as a single entity and may not be individually swapped. In this case, the whole Mother partition is swapped if needed. All partition status information (priority, ID segment address, Read Completion flag) is kept in the Mother partition MAT entry.

The Dispatcher checks for empty partition lists at start-up. If there are no Real Time partitions, the header of the RT partitions list points to the background used in the BG list. If there are no Mother partitions, BG partitions are used; if there are no BG partitions, the RT partitions are used.

The sizes of the largest non-reserved partition of each type are kept in three words:

$MRTP - RT partition.

$MBGP - BG partition.

$MCHN - Mother partition.

The sizes of the largest non-reserved, non-shared partitions (excluding EMA partitions) are kept in three words:

$NRTP - RT partition.

$NBEP - BG partition.

$NCHN - Mother partition.

The non-shared EMA partition sizes are used by LOADR and MLLDR to make sure that the shared EMA program can find a partition to run in.

```
      +------------------------------+
+---->| NEXT MOTHER PARTITIONS       |---->
|     |------------------------------|
|     | PRIORITY OF RESIDENT         |
|     |------------------------------|
|     | ID SEG ADDR OF OCCUPANT      |
|     |------------------------------|
|     | M=1            START PAGE (P)|
|     |------------------------------|
|     | C=1           # PAGES    (X) |
|     +------------------------------+
|     | SUBPARTITION POINTER (SLW)   |----+
|     +==============================+    |
|     |              0               |<---+
|     |------------------------------|
|     | SUBPARTITION 1               |
|     |------------------------------|
|     |              0               |
|     |------------------------------|
|     |                START PAGE (P)|
|     |------------------------------|
|     | C=1            # PAGES   (X1)|
|     +------------------------------+
|     | SUBPARTITION POINTER (SLW)   |----+
|     +==============================+    |
|     |              0               |<---+
|     |------------------------------|
|     | SUBPARTITION 2               |
|     |------------------------------|
|     |              0               |
|     |------------------------------|
|     |             START PAGE (P+X1)|
|     |------------------------------|
|     | C=1               # PAGE (X2)|
|     |------------------------------|
|     | SUBPARTITION POINTER (SLW)   |----+
|     +==============================+    |
|     |              0               |<---+
|     |------------------------------|
|     | SUBPARTITION N               |
|     |------------------------------|
|     |              0               |
|     |------------------------------|
|     |      START PAGE (P+X1+X2+..)|
|     |------------------------------|
|   C=1             # PAGES (Xn)|    |
|     +------------------------------+
+-----| SUBPARTITION POINTER (SLW)   |
     +------------------------------+
```

Figure 1-2. RTE-6/VM Mother Partitions

Each of the three types of partitions--BG, RT, and MOM--is kept in a list. The type of the list is determined by the state of the partition:

1. Free list. The partitions have no occupants. The list is ordered by size, from smallest to largest partition. This guarantees that a search of this list will find the smallest available partition first.

2. Dormant list. The partitions contain programs that cannot execute. The dormant programs include those that have been terminated serially reusable, or terminated saving resources, or user suspended. The list is ordered first by low-to-high priority programs and then by smallest-to-largest size. Thus a search of this list will yield the lowest priority programs in the smallest partition first.

3. Allocated list. This is really an extension of the dormant list, and contains those partitions whose programs are active and located in that partition's memory.

Figure 1-3 shows the partition linking scheme for RT partitions. The end of the list is indicated with a "0" entry box in the last partition.

```
+-------------------------------------------------------------------+
|                                                                   |
|           RT                                    RT                |
|       DORMANT LIST                        ALLOCATED LIST          |
|                                                                   |
| Head of                           Head of                         |
| Dormant                          Allocated                        |
|  List                               List                          |
|                                                                   |
| +-------+  if dormant list empty  +-------+                       |
| | RTDM  |- - - - - - - - - - - - ->| RTPR  |---+                  |
| +-------+                          +-------+   |                  |
|    |                                  ^        |                  |
|    |                                  |        |                  |
|    |                                  |        |                  |
|    |                                  |        v                  |
|    |   +------+    +------+    +------+ |  +------+    +----+--+ |
|    +-->| PTN  |--->| PTN  |--->| PTN  |-+  | PTN  |--->|PTN | 0| |
|        |  2   |    |  5   |    |  6   |    |  4   |    | 1  |  | |
|        +------+    +------+    +------+    +------+    +----+--+ |
|                                                                   |
+-------------------------------------------------------------------+
|                                                                   |
|                              RT                                   |
|                           FREE LIST                               |
|                                                                   |
|  Head of                                                          |
| Free List                                                         |
|                                                                   |
| +-------+                                                         |
| |$RTFR  |                                                         |
| +-------+                                                         |
|    |                                                              |
|    |                                                              |
|    |                                                              |
|    |                                                              |
|    |    +------+    +------+    +------+     +----+--+            |
|    +--->| PTN  |---->| PTN  |---->| PTN  |----->|PTN | 0|         |
|         |  7   |    |  9   |    |  8   |     | 3  |  |            |
|         +------+    +------+    +------+     +----+--+            |
+-------------------------------------------------------------------+
```

Figure 1-3. RT Partition Lists Linking

1.3 DISPATCHER CONTROL FLOW

Figures 1-4 through 1-6 illustrate the decision flow of the dispatcher.

1.3.1 Dispatch Task Director Routine $XCQ

Referring to Figure 1-4, the dispatcher first checks to see if a program terminated or aborted on last access to the operating system. If so, $XCQ immediately exits to the cleanup routine, ABORT, to free all resources. This is done because an otherwise dispatchable program could be waiting on a resource held by the aborted program. In the case of abnormal terminations, the program has been aborted for doing something very wrong and you must clear and reset the program resources (particularly peripherals) so that other programs are not affected.

Next the dispatcher checks for program state transitions by checking whether or not $LIST (the state transition routine) been called. (An example of a state transition is moving a program from scheduled to I/O suspended.) If a state transition has occurred, the scheduled list may have been affected by, for example, a high priority program that has just become unblocked.

If no state transitions have occurred ($LIST=0), the last interrupt or operating system service routine was non-significant and $XCQ exits to $IRT. $IRT will return to wherever the CPU was before the last interrupt, either the last user program executing or the idle loop.

If a state transition did occur, the dispatcher sets $LIST=0 and starts into what is potentially a very long algorithm to determine which program should execute next. It does this by driving itself with the scheduled list. The scheduled list is ordered by priority, with the highest priority programs at the head of the list. The dispatcher picks up the first program in the list and sees if it can be executed. If not, the dispatcher checks each following program on the list until an executable program is found (i.e., the program is in memory and in state 1).

Before beginning the program execution algorithm, the dispatcher calls $SIP, a microcoded optimization routine that checks for and services any pending system interrupts. This is done to assure a "clean slate", as far as interrupts are concerned, prior to dispatching a program.

When all pending interrupts have been serviced, the dispatcher begins at the head of the scheduled list (highest-priority programs) to select a program for execution. The program at the head of the list is not logically blocked. If a program was previously blocked, however, it is possible that the program is not in memory, and must be brought in from disc.

Continuing with Figure 1-4, the Dispatcher then checks to see if this was the last program to execute. If so, the time slice is checked and if the time slice has not been used, jumps to REENT. REENT resets some of the words in the base page communication area, sets up the memory protect fence, and then goes off to $IRT. $IRT will reload the program registers and do a UJP to the last point of suspension of the program.

If this program was not the last to execute, the Dispatcher checks to see if the program is still in memory or is a memory resident (type 1) program. The latter check is made at X0030 and, if it is, control is transferred to XOF40 which then goes to REENT to dispatch the program. Next a check is made to see if the program is assigned to a partition. If so, control is transferred to the routine associated with the partition type (RT, BG, or mother).

If the program is not type 1 and is not assigned, the Dispatcher checks the program type (BG or RT) and then checks the program size to determine if a mother partition will be required. If the program will fit into a non-mother partition then the program goes into a normal partition regardless of whether the program is RT, BG, EMA, or shared EMA. (In earlier operating systems, EMA programs always went into mother partitions no matter how small the EMA program was. To improve execution speed, RTE-6/VM avoids using mother partitions whenever possible.)

Once the the partition list to search has been determined, control is transferred to that routine. Note in the flow chart that the transfer of control is to the same place for both assigned and unassigned programs. In one case the user makes the decision, in the other case it is made programmatically.

At this point the flow of control diverges to one of:

1. Process background program (X0100)

2. Process real time program (X0200)

3. Process mother program (X0300)

These three routines are actually identical; control is split because each code area has a disc $XS10 call to load or swap a program. Since the $XS10 call is linked through the actual call it must complete before it can be reused. Separate calls were used for the various program types so that one type will not have to wait for loads and swaps of another.

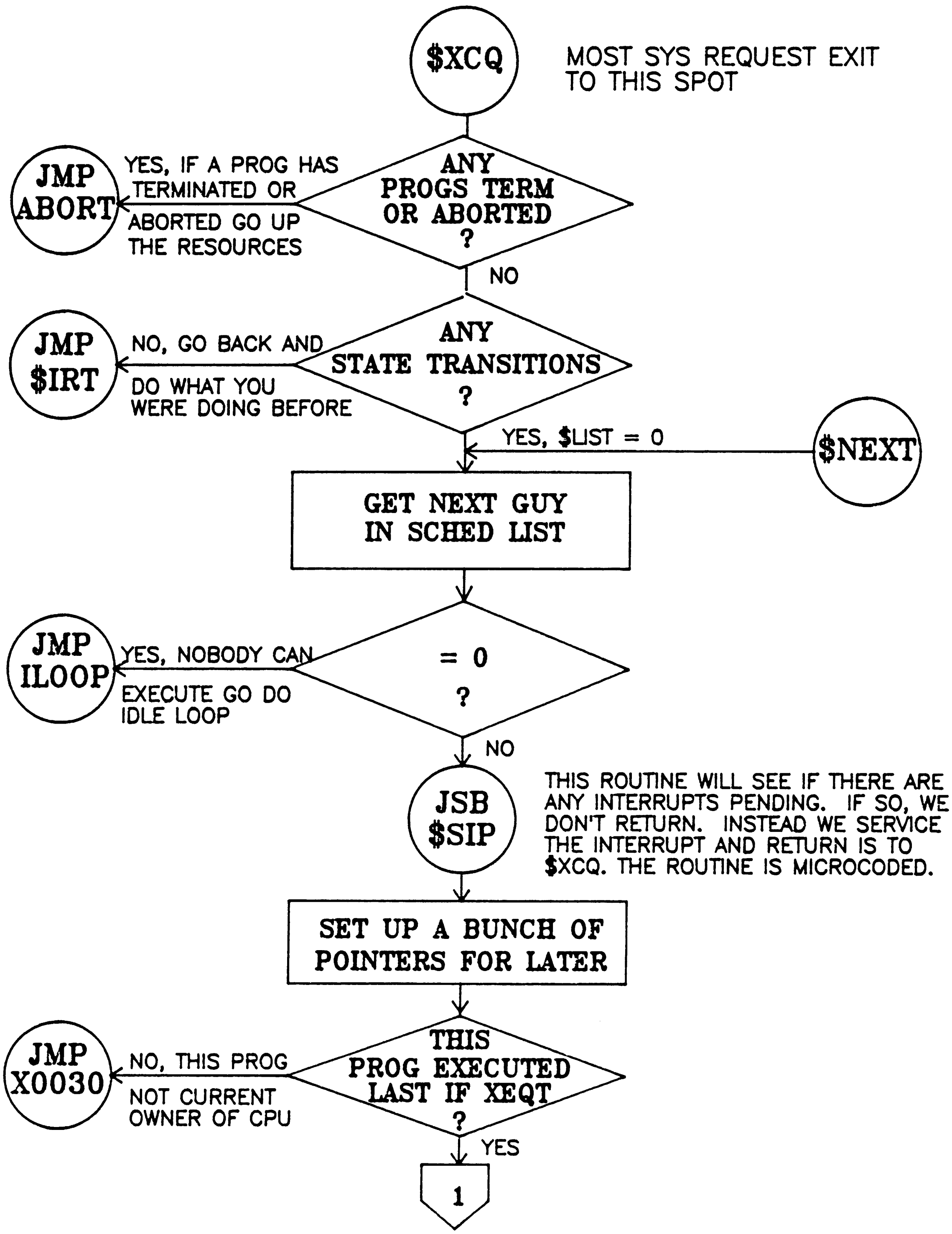


Figure 1-4. Task Director Routine $XCQ Flow Chart

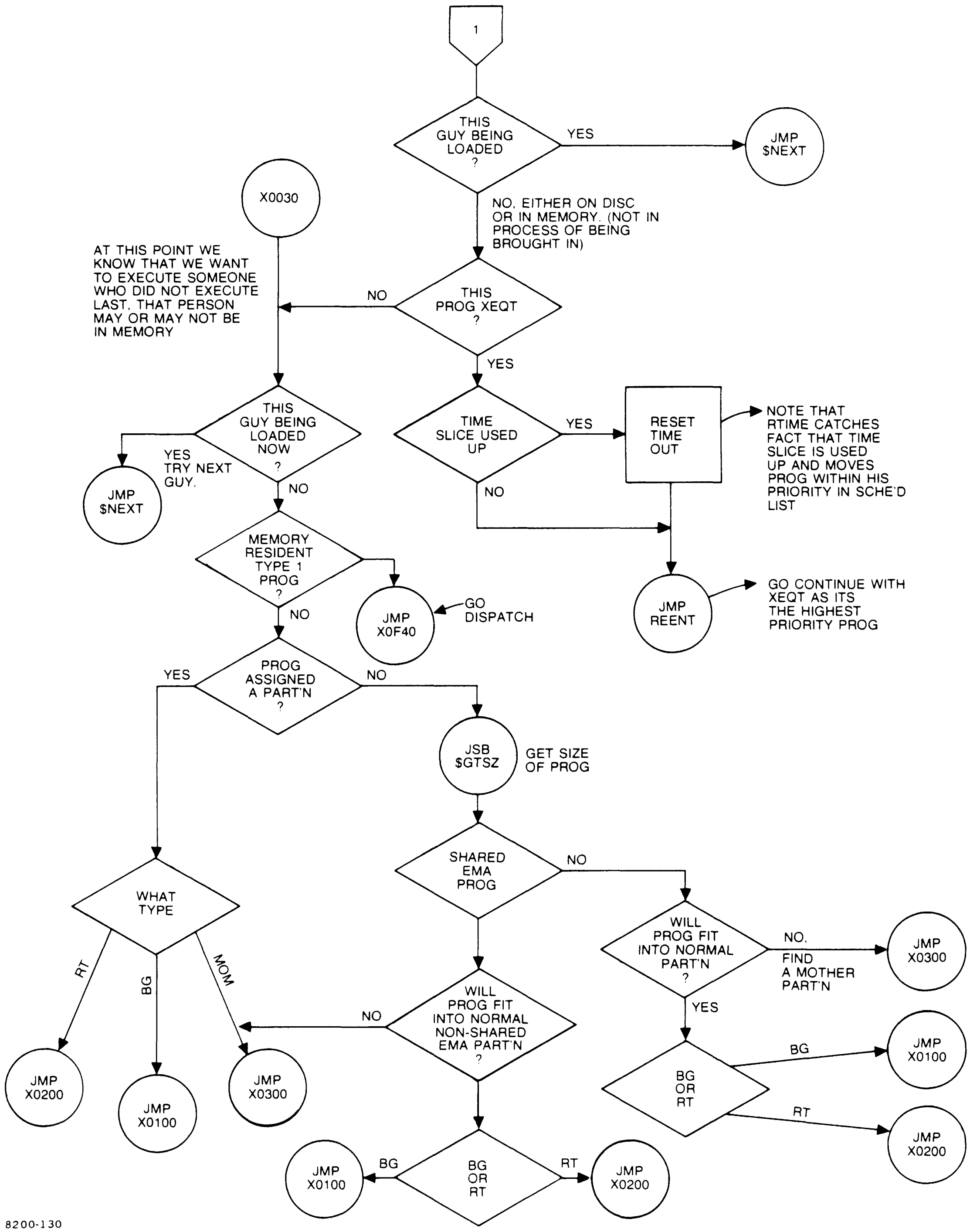


Figure 1-4. Task Director Routine $XCQ Flow Chart (Continued)

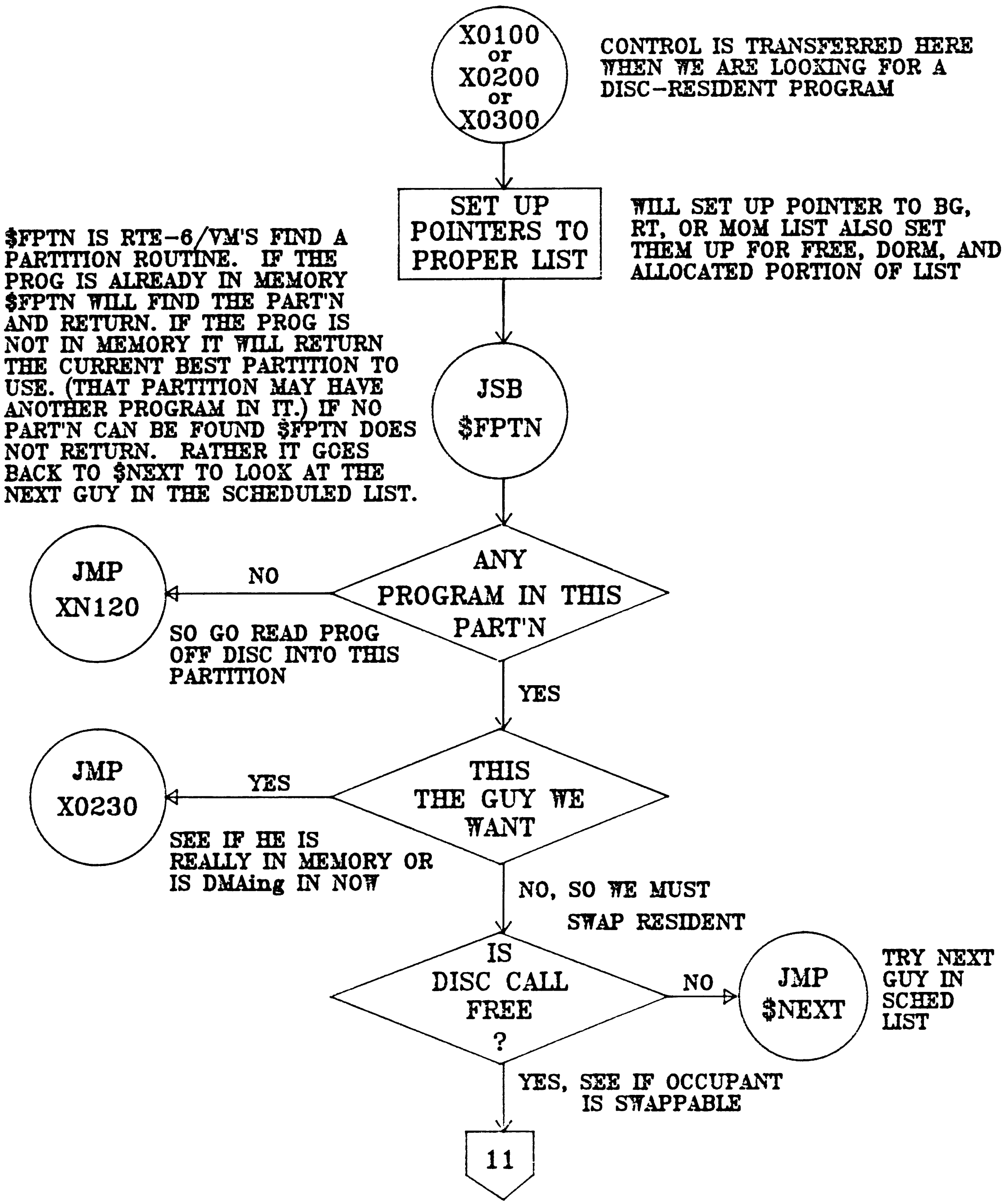


Figure 1-5. Task Director Routine X0xxx Flow Chart

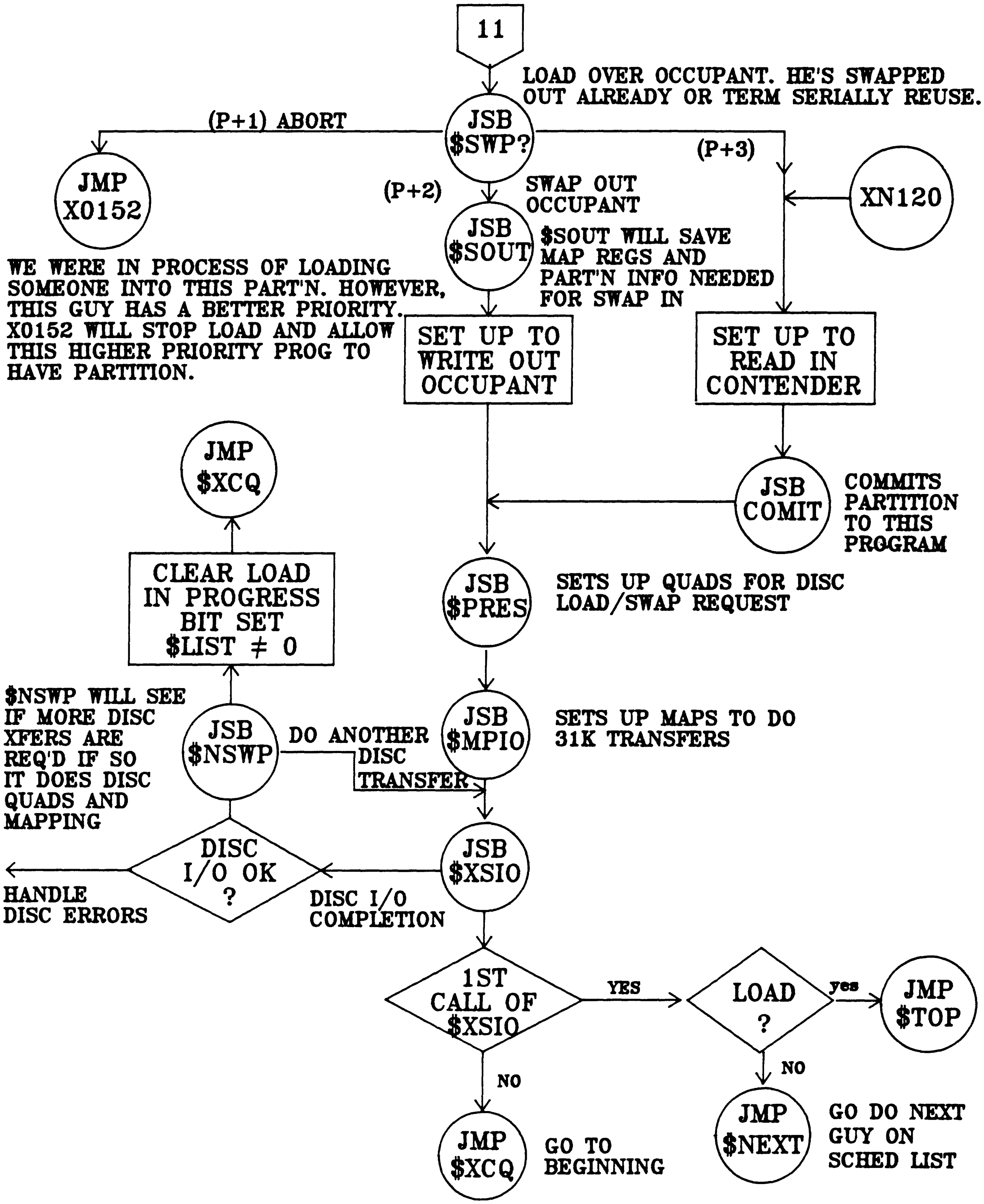


Figure 1-5. Task Director Routine XOxxx Flow Chart (Continued)

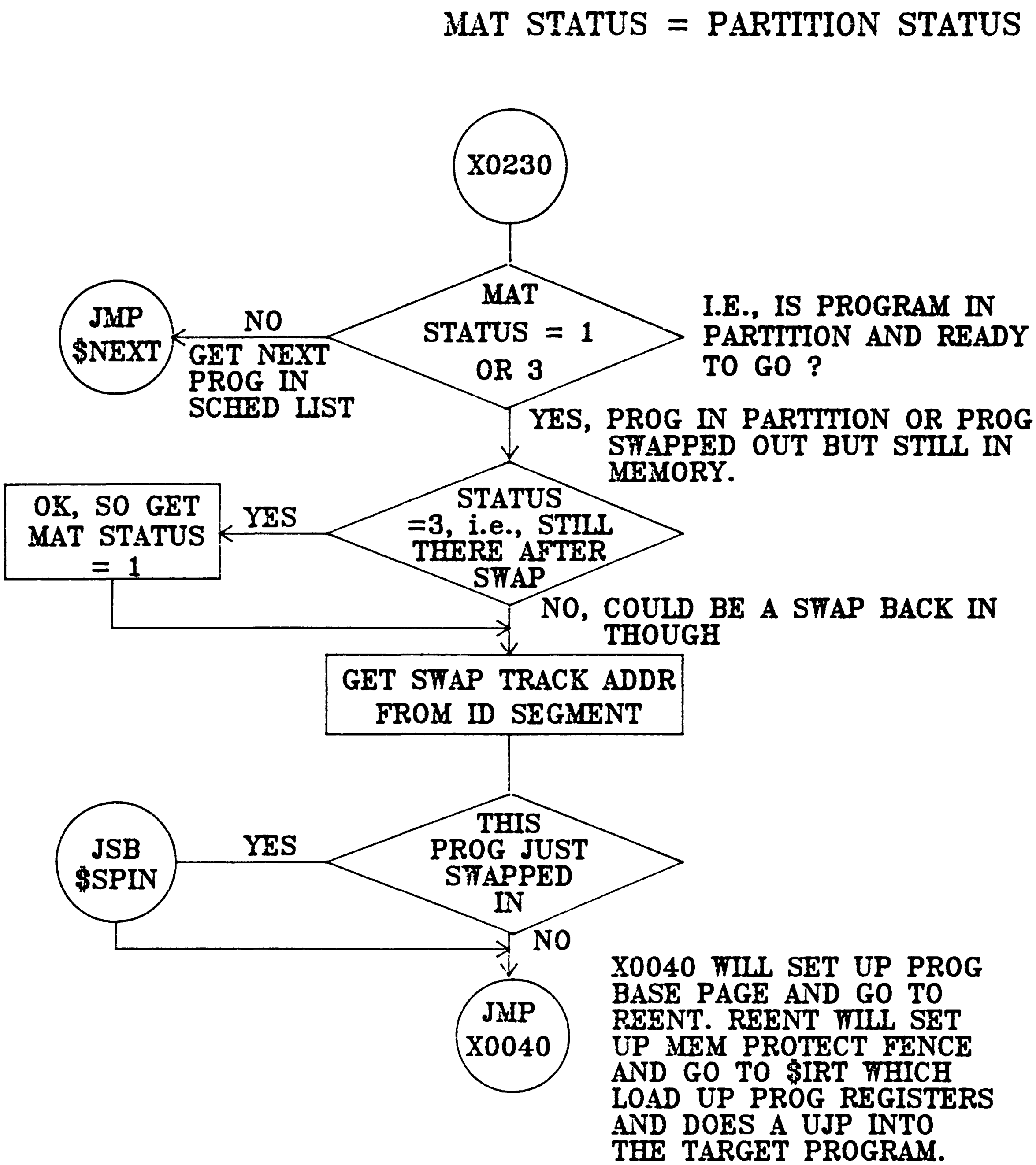


Figure 1-5. Task Director Routine X0xxx Flow Chart (Continued)

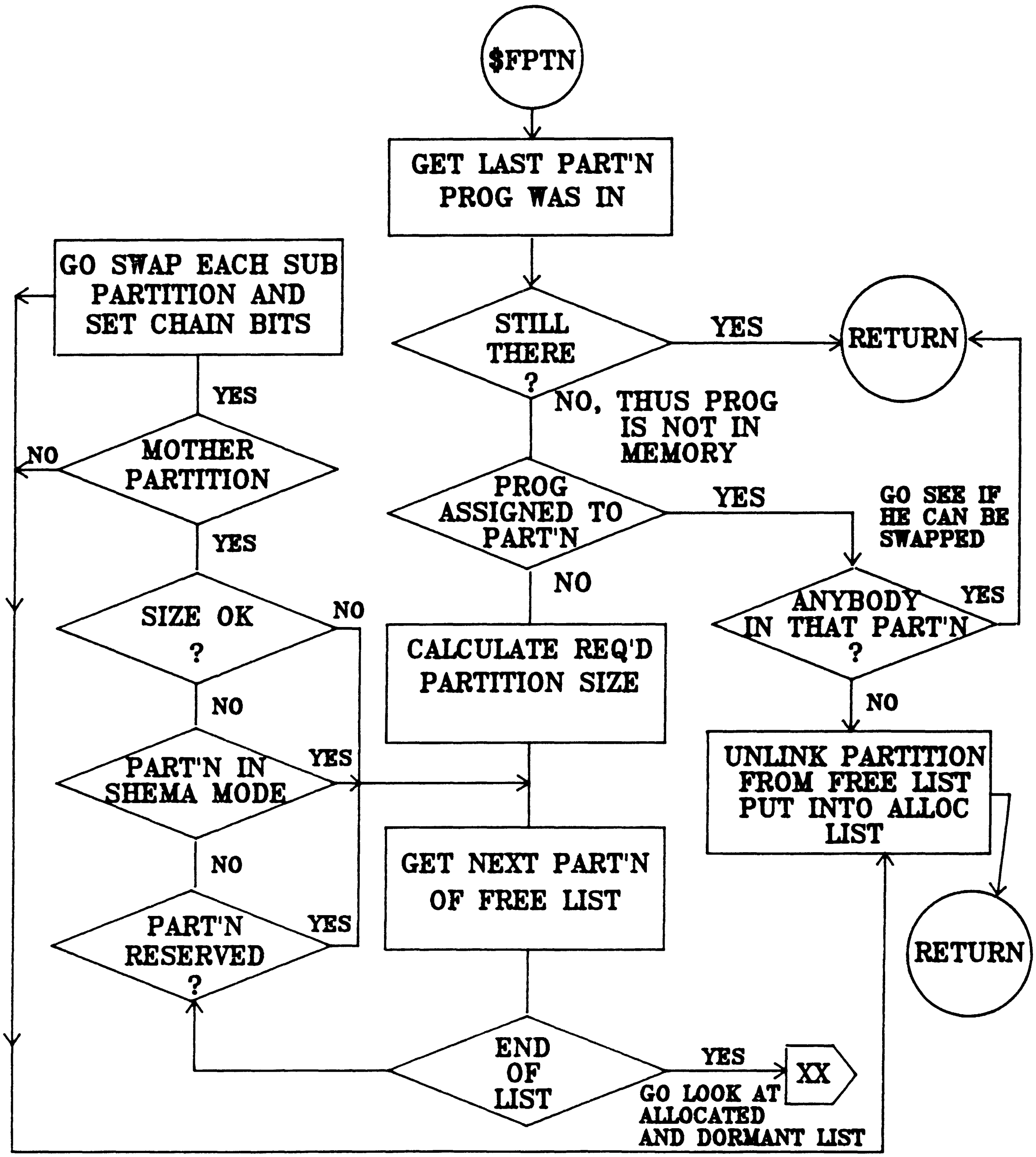


Figure 1-6. Task Director Routine $FPTN Flow Chart

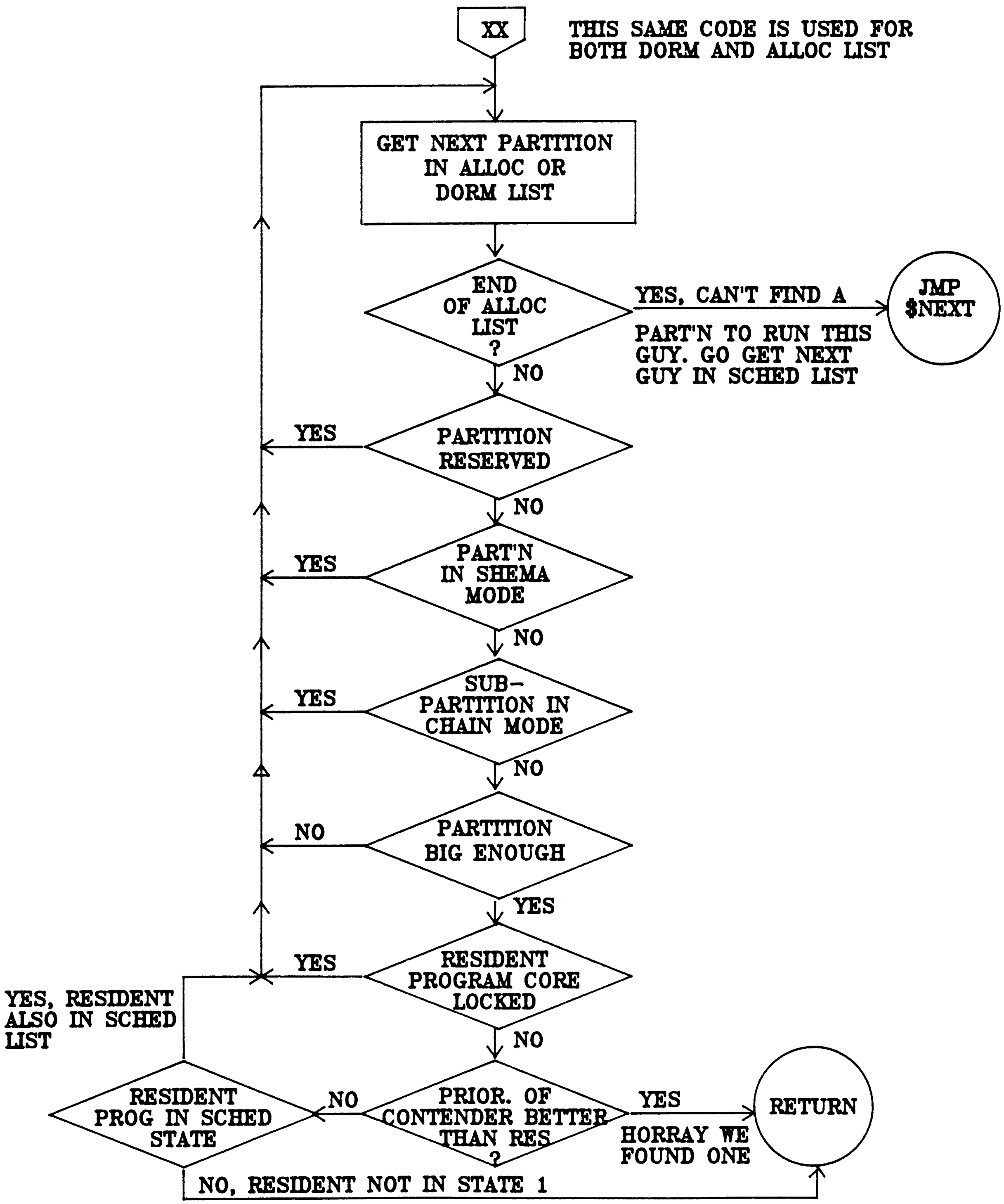


Figure 1-6. Task Director Routine $FPTN Flow Chart (Continued)

The routines are charted in Figure 1-5. Pointers to the appropriate lists are first set up: at X0100 pointers to the BG free, allocated and dormant lists are set up; at X0200 the RT pointers are set up; at X0300 the mother partition pointers are set up.

$FPTN is then called. $FPTN is the Dispatcher "find a partition for this program" routine. $FPTN will find a partition if possible and return the following MAT pointers to set up the appropriate partition:

MLNK	Points to word 1, link word.
MPRIO	Points to word 2, priority of occupant.
MID	Points to word 3, ID segment of resident or owner of the partition.
MADR	Points to word 4, starting physical page of partition.
MLTH	Points to word 5, the number of pages in the partition.
MRDFL	Points to word 6, partition status word.
MSUBL	Points to word 7, the next subpartition pointer word.

Figure 1-1 shows these pointers graphically. $FPTN will return pointers to the partition that is the best match for the program. (If the program is already resident in a partition, that partition is returned in the "M" pointer.)

Figure 1-6 shows the control flow for the $FPTN routine. (A detailed flow chart of $FPTN is included at the end of this chapter.) $FPTN first checks to see if the program is still in memory, by extracting the MAT table index of the last partition the program was in from ID segment word 21. If the MAT table MID entry has that ID segment, the program is still resident in the partition and $FPTN returns with the MAT pointers set up.

If the program is not still resident in its last partition, it is not in memory and a partition must be found for it. In this case $FPTN checks to see if the program is assigned to a partition. If so, that partition is inspected to see if there is an occupant. $FPTN returns if the partition is occupied; if the partition is free, it is removed from the free list and moved into the allocated list. $FPTN then returns.

If the program was not assigned a partition, the partition lists must be searched to see if a suitable partition can be found.

The free list is searched first. If there is an empty partition, the program will execute there as that memory is currently idle. A number of factors could preclude a partition from being used; each factor is checked, as shown in Figure 1-6, and $FPTN returns if any factor is true. When a free partition is found, it is removed from the free list, placed into the allocated list and $FPTN returns.

If no free partition can be found then a search is made of the allocated and free lists. Again, several things may prevent a partition from being used. If a suitable partition is found, $FPTN returns. No partition lists need to be modified as the target partition is not currently in the free list. If no suitable partition can be found, $FPTN jumps back to $NEXT to see if the next program in the scheduled state can be run.

Actually $FPTN will try a few other things to find a partition. As mentioned, a detailed flow chart of this process is included at the end of this document. It should prove interesting for those stout of heart and strong of mind.

When $FPTN returns, the X0100 code checks to see if that program is in a partition. If so, it goes off to X0230 which checks to see if the program was just swapped in. If so, $SPIN is called to adjust the saved map registers to account for the new partition. (Refer to the section on swapping for more information on $SPIN.)

On return from $SPIN, or if the program was not just swapped in, the X0230 routine goes to X0040 to set up base page and dispatch the program.

To summarize, X0100 called $FPTN to find the program a partition. Upon discovering that the program was still in memory, the dispatcher merely transferred control back to that program.

The next case is where $FPTN returns a partition that is empty. In this case, XN120 is executed. This routine is used to read a program in from disc or swap a program out to disc. In the case of a load into memory, parameters are set up in the $XS10 call to indicate that the call is busy now and that it is a read from disc. In addition, the MID word in the MAT table is set to the ID segment of the program. Setting the MID word commits the partition (by means of the COMIT routine). Next $PRES is called.

$PRES is used on load and swaps. In the case of a load, $PRES merely looks up the track and sector address of the program from the ID segment, calculates the number of words to transfer, and builds an $XS10 quad. Next $MPIO is called. $MPIO sets up information on the unused portion of the users base page that RTI06 looks at. It sets up the starting page of the transfer and the number of pages left in the program to be brought back into memory. RTI06 uses this information to set the DCPC port maps to load the program into memory in increments of up to 31 pages. (You cannot use 32 pages because DMA respects the base page fence.)

Now the $XS10 call is made. Recall the $XS10 has two returns. One is the return from the subroutine call, and the other is the return from RTI06 when the disc I/O is complete. When the subroutine returns, the Dispatcher goes to $NEXT to see if there is another program it can dispatch.

On completion of the disc $XS10 call, a return is made to the completion address specified in the call. At this point the dispatcher calls $NSWP to see if the program being brought back into memory was greater than 31 pages. If so, $NSWP will set up the next disc quad and call $MPIO to set up the next 31-page transfer. This loop continues until the entire program is brought into memory. When the entire program is in memory the $XS10 call is set free, the ID segment load in progress bit is cleared, and control transfers to $XCQ. $XCQ starts the whole process over again only this time the program will be in memory when $FPTN returns.

Referring back to Figure 1-5, the last path to consider is what happens when $FPTN returns a partition and that partition is occupied. In this case a call is made to the $SWP? routine to see if the resident program may be swapped by the contender. (The $SWP? routine is shown in Figure 1-8, located in the section Swapping.) If the resident program can be swapped, $SWP? returns; if not, $SWP? goes back to$FPTN to find another partition.

$SWP? will return one of three ways:

1. Return One loads over the current occupant (this is the same as if the partition was empty.

2. Return Two signifies that the resident program is in the process of being loaded into the partition but the contender has a better priority. (This return allows the caller to abort the residents load into memory if desired.)

3. Return Three signifies that it is all right to swap out the resident program. In this case the X0100 code takes on a different aspect. Instead of being called to load a program into memory it now swaps a program out. (The same $PRES, $MPIO, and $XS10 calls are thus used to do writes to disc as well as reads from disc.

One other routine, $SOUT, is called on the swap path in Figure 1-5. This routine saves the users maps in his base page so that when DMA writes the program on the disc the current map registers are saved on disc too.

1.4 FINAL DISPATCH

The final dispatch (redispatch) routines X0040 and REENT have been previously mentioned. At this point the program to execute next has been determined, and the Dispatcher now implements the decision. The first call is to $SUMP to save the map registers of the last program that executed and then set up the map registers of the next program to be executed.

The base page communication pointers are then set up. The logical address bounds of the program are set up in RTDRA, AVMEM, BKDRA and BKLWA, and the ID segment pointers are set up at XEQT. The X/Y registers save area address is also set up at XI. If the progam priority is higher than the time-slice limit, the time-slice set up is skipped. If ID word 30 is less than zero, it is used the time-slice value. Otherwise, the time-slice value is calculated by using the program's priority in the following equation:

Program Slice Value = Sys Slice * Z + SYS Slice

Where:

SYS
Slice = 1500 ms default. This value (1.5 sec) may be increased or decreased via the QU command.

Z = bits 8-11 (isolated and shifted right eight positions) of the programs priority.

Define $LICE to be equal to ;the address of ID 30. If ID 32 is a positive value, define $DCPU to address the CPU usage location of the session control block.

Now that the program is ready to execute, the address of where to begin or resume execution is determined. If the point of suspension address is zero, control is given to the program at the primary entry point. If the point of suspension is non-zero, control is returned to that address. The memory protect fence is set up according to the Memory Protect Fence Table index in the ID segment. Control is then turned over to the program by exiting through $IRT. This routine enables memory protect, and enables the interrupt system and user map. The memory protect fence table is shown in Figure 1-7.

```
                ADDRESS TO PLACE INTO MEMORY PROTECT FENCE

                        +----------------------------------------+
        $MPFT WORD 0    | DISC RESIDENT ADDRESS, NO COMMON       |
                        |----------------------------------------|
              WORD 1    | MEMORY RESIDENT ADDRESS, NO COMMON     |
                        |----------------------------------------|
              WORD 2    | ANY PROGRAM ADDRESS, COMMON 0          |
                        |----------------------------------------|
              WORD 3    | ANY PROGRAM ADDRESS, COMMON 1          |
                        |----------------------------------------|
              WORD 4    | ANY PROGRAM ADDRESS, SSGA              |
                        |----------------------------------------|
              WORD 5    | PRIVILEGED PROGRAM ADDRESS, NO COMMON  |
                        |----------------------------------------|
              WORD 6    | TYPE 6 PROGRAM                         |
                        +----------------------------------------+
```

Figure 1-7. Memory Protect Fence Table

1.5 SWAPPING

Whether a program may be swapped or not is determined by the $SWP? routine, shown in Figure 1-8. A program is swapped out of memory to make a partition available for another program to run. The first programs chosen to be swapped are those in the dormant list. These programs have terminated with either the save resources or serially reusable option, or are operator suspended and still in memory. Following these, programs with the lowest priority will be checked, in priority order, for swappability.

Referring to Figure 1-8, you can see that $SWP? not only checks the program for swappability but also considers the more general case of whether this partition can be used without a swap. $SWP? has four returns: P+1 through P+4.

The P+1 return is made when $SWP? finds that the contender has higher priority than the resident and the resident is not yet in memory. That is, $XS10 was called to read in the resident but the completion interrupt has not yet occurred. On return, the caller will have the opportunity to decide whether or not to abort that $XS10 request and give the partition to the contender.

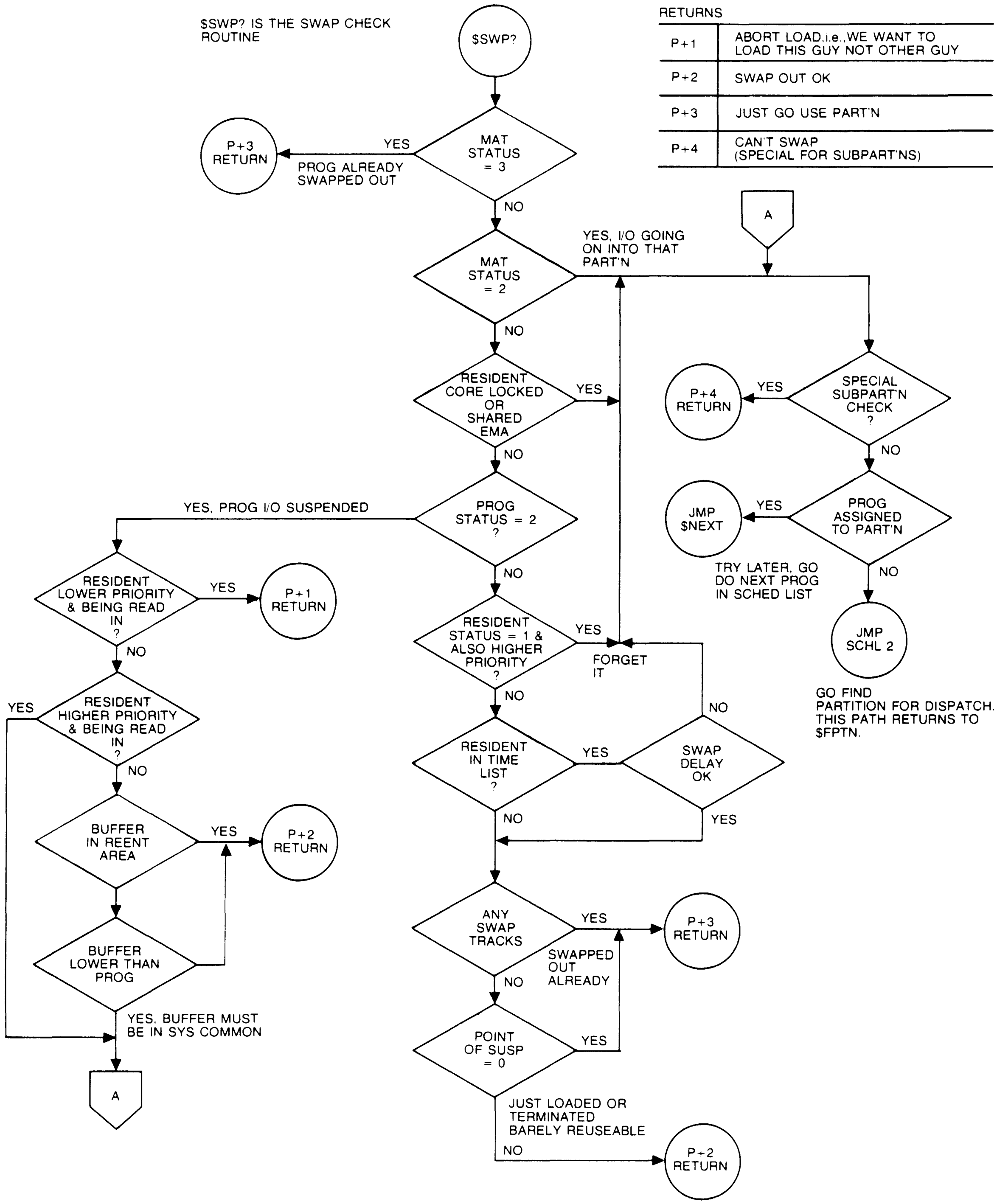


Figure 1-8. Program Swapping Routine $SWP? Flow Chart

The P+2 return signifies that the resident can be swapped out. This return is made when the resident is of lower priority and in a swappable state (regardless of program state), or when the resident is of higher priority but the program state is not 1. Thus it is possible for low priority programs to swap higher priority programs, provided the high priority program is in some blocked state (for example, waiting for a son program).

The P+3 return is made when there is a resident in the partition but it is not necessary to swap that resident out. That is, the resident can be overlaid. This return will occur if the resident has terminated serially reusable, if the resident is already completely swapped out, or if the resident was loaded into that partition but has not been executed (i.e., the swapped disc image is still good).

The P+4 return is a special return used for subpartition checks on a mother partition. It prevents $SWP?, when it finds that it cannot swap the resident, from jumping back into the $FPTN routine to look for another partition for the contender.

When $SWP? returns the P+2 condition, the following events occur to complete the swap:

1. Map registers are saved, if necessary

2. Disc space is reserved on the system disc, LU 2, and auxiliary disc, LU 3, for the swap.

3. Mapping information set up for RTI06 to perform swap in 31-page increments

4. The required $XS10 calls are made.

Referring back to Figure 1-5, you can see the progression on the return from $SWP?. The call to $SOUT saves any map registers on the unused portion of the user's base page.The call to $PRES gets the disc space required and calls $SETD to set up the quads for the $XS10 call. The call to $MPIO sets up the unused base page for $XS10 to do 31K I/O transfers. Figure 1-9 shows the structure of the user base page.

```
1777B +---------------------------+
      | MAP REGISTER SAVE AREA    |  User map save area
      |                           |  modified by $SOUT,
      |                           |  if necessary
1740B |---------------------------|
1737B | # OF PAGES TO SWAP OR LOAD | \
      |---------------------------|  > Set up by $MPIO
1736B | START PAGE OF TRANSFER    | /
      |---------------------------|
1735B | LAST PAGE OF OLD PARTITION | \
      |---------------------------|  > Set up by $SOUT
1734B | 1ST PAGE OF OLD PARTITION | /
      |---------------------------|
      |                           |
      |                           |
      |                           |
      |                           |
      |                           |
1500B |                           |
      |---------------------------|
      |                           |
      |                           |
      |                           |
      |     USER BP LINKS         |
      |                           |
      |                           |
      |                           |
      |                           |
    2 +---------------------------+
```

Figure 1-9. User Base Page

Later, the dispatcher will bring the swapped out program back into memory. RTIOC will call $SMAP to build a map identical to the initial load map for the swap back into memory. This map is not used to execute the program, rather, a call is made to $SPIN after the program is brought into memory. $SPIN rebuilds the user map and stores this map in words 1740B to 1777B of the user's base page (i.e., the first page of the partition). $SPIN does this by mapping the first page of the partition into the driver partition and comparing the map information of the old partition with the new partition. Recall that, before the swap, $SOUT saved the information on partition and maps. $SPIN will now use this information to set up the new maps. If the partition start page saved on swap-out is the same page number as the current partition (i.e., the program was swapped back into the same partition), the map information does not need modification. $SMAP will use this to set up the user map registers. If the program is being swapped into a different partition, the map registers must be modified. The algorithm used to do this is:

```
If: LOW$  =<   OP#   =<   HIGH$

    then       NP#   =    OP#-OSP# + NSP#

    else       NP#   =    OP#
```

Where:

LOW$ = Program start page number in the old partition

HIGH$ = Program last page number in the old partition

OP# = Old page number of the current map register

NP# = New page number to place in the map register

OSP# = Old partition starting page number

NSP# = New partition starting page number

This algorithm ensures that the map set up when the program executes reflects the program's state when the program was last swapped out. In addition, the algorithm also ensures that map registers pointing outside the current partition (to a shared EMA partition for example) are not modified by the swap. If an EMA or VMA program is swapped into a different partition, the start page of the EMA word in the ID extension is updated to the new physical page no.

Allocating disc space for the swap is done by $PRES via a call to $DREQ. $DREQ examines the track assignment table and allocates tracks from the top down (as opposed to user track allocation requests that are from the bottom up). When the tracks are returned, $PRES calls $SETD to set up the $XS10 quads.

If $DREQ returns with not enough tracks available for the swap, the diagnostic message

    XXXXX NO SWAP TRACKS

is printed, where XXXXX is the program it tried to swap. This message will only be issued 30 times in the life of a boot. If seen repeatedly, contact your local Hewlett-Packard representative.

The problem is that when this no disc space condition occurs the message could be repeatedly issued until an operator frees some disc space. Consider a situation where the highest priority program in the system cannot get into memory because the only partition it will run in is occupied by another program that cannot be swapped due to no disc space. In this case, when a low priority program makes a state change (i.e., unbuffered I/O request) the system returns to $XCQ. Recall that at $XCQ the dispatcher scans the scheduled list and will again try to dispatch the high priority program. It cannot due to no disc space and so again prints the XXXXX NO SWAP TRACKS message. Under this scenario the message would be printed forever or until some disc space is freed up.

Since swapping occurs on LU 2 and LU 3, and these discs may have up to 1600 tracks, this error should not occur if you reserve a minimum of 200 free tracks for swapping. To see if this is enough space for your needs, run LGTAT several times when the system is busy and see how many contiguous free tracks there are. If there are less 50 free contiguous tracks you have not allocated enough tracks to the system.

1.5.1 Partition Priority Aging

The new RTE-6/VM operator command AG allows partition priority aging (NOT program priority aging). This command, invoked by entering AC,x (where x = some number of tenths of a millisecond), causes the partition allocated list to age by the specified increment. This process occurs in the dispatchers $AGE routine. The routine RTIME counts down X, and when X=0, the $AGE routine is called. $AGE goes to the state 3 list that contains programs that are blocked for buffer limits, father son waits, resource lock waits, etc. For each program the state 3 list that is also resident in a partition, the partition priority is aged by 2 and the partition relinked into the partition list by its new priority. The effect is that, over time, programs that are blocked for one reason or another have their partition moved to the head of the allocated list where they are more likely to be swapped.

When the blocked program becomes rescheduled and redispatched, the partition priority is set back to the program priority and linked back to the proper place in the allocated list. This process is shown graphically in Figure 1-10.

```
ALLOCATED           SWAP SEARCH DIRECTION
LIST HEAD            ----------------->

+--------+   +--------+   +--------+   +--------+
| BASIC4 |-->| BASIC1 |-->| BASIC2 |-->| BASIC3 |-----+
+--------+   |   90   |   |   90   |   |   90   |     |
             +--------+   +--------+   +--------+     |
                                                      |
          +-------------------------------------------+
          |
          |    +-------+   +-------+   +-------+   +-------+
          +--->| FMGR1 |-->| FMGR2 |-->| FMGR3 |-->| FMGR4 |
               |  50   |   |  50   |   |  50   |   |  50   |
               +-------+   +-------+   +-------+   +-------+
```

Figure 1-10A. Allocated List State Without Aging

Figure 1-10A shows the state of the allocated list without aging. When the fourth BASIC is run, attempts will be made to swap out BASIC1, BASIC2, BASIC3, FMGR1, FMGR2, FMGR3, FMGR4, in that order. That is, swap attempts are in the order the program is in the list. Unfortunately the swap attempts on BASIC1, BASIC2 and BASIC3 are often successful and thus the four BASICs contend for three partitions while the four FMGR partitions go unused.

```
ALLOCATED
LIST HEAD

+--------+   +-------+   +-------+   +-------+
| BASIC4 |-->| FMGR1 |-->| FMGR2 |-->| FMGR3 |-----+
+--------+   | 32767 |   | 32767 |   | 32767 |     |
             +-------+   +-------+   +-------+     |
                                                   |
          +----------------------------------------+
          |
          |    +--------+   +--------+   +--------+   +-------+
          +--->| BASIC1 |-->| BASIC2 |-->| BASIC3 |-->| FMGR4 |
               |  90    |   |  90    |   |  90    |   |  50   |
               +--------+   +--------+   +--------+   +-------+
```

Figure 1-10B Allocated List State with Aging

Figure 1-10B shows what the allocated list would look like if aging were very fast when the fourth FMGR issued the EXEC request to schedule BASIC4. Note that the FMGRs (except FMGR4 which is executing) have marched to the head of the allocated list. Now when the fourth BASIC is run, a successful attempt to swap FMGR1 is attempted and all the BASICS will have their own partition.

1.6 PROGRAM TERMINATION/ABORTION

Program termination is orchestrated by the dispatchers ABORT routine. ABORT does not do the actual clean up, rather it calls the routines to do the clean up. It handles both normal and abnormal program terminations. ABORT does the following:

1. Sets the CPU ownership flag, $BOWN, to 0 if this program was the currently executing program.
2. Releases any program swap tracks.
3. Calls DS/1000 if it is installed in the system to do network clean up.
4. Advances the termination sequence counter in the ID segment. This is used by D.RTR to do clean up on files opened but not closed.
5. Clears out the session pointer.
6. Calls $ABRE in EXEC to release any reentrant memory.
7. Calls $RTST to release any string memory.
8. Reschedules any programs waiting on the aborted program. For example, the program's father.
9. Calls $TRRN to release any resource numbers owned or locked by this program.
10. Calls $EQCL to release any EQT locks.
11. If this is a shared EMA program, counts down the number of active users of the data area. If the count reaches 0, the $ECLR is called to release the shared EMA partition.
12. Calls $F.CL to clean up class I/O requests. Note that $F.CL handles normal and abnormal aborts in different ways.
13. If the program terminated serially reusable or saving resources, the partition is not returned to the system. On a normal or abnormal termination, the partition is moved from the allocated back to the free list.

At this point all program resources have been cleaned up. Return is again made to $XCQ. If more programs are to be aborted, $XCQ will again call ABORT to clean up. Note that from a resource allocation standpoint, doing the resource clean up first makes a lot of sense. Other programs in the system that wish to be dispatched may be blocked on the resources that are held by the terminating program.

1.7 OVERVIEW OF TIME-SLICING OPERATION

All programs competing for the central processor access it in an orderly manner, under the direction of RTE-6/VM. The system places programs into the scheduled list in order of their priority. When a program completes, terminates or is suspended, the RTE-6/VM Dispatcher searches the scheduled list for the next program of highest priority, and transfers control to it.

The scheduled list (see Figure 1-1) is divided into logical areas, each corresponding to a particular type of dispatching and priority level. Scheduling within each priority can be performed linear or a circular fashion.

The default priority range for linear scheduling is from 1 to 49. Programs of this type are given processor control until the program is either completed, terminated or suspended to await the availability of a required resource.

Circular scheduling is performed on all program priority levels lower (higher number) than the timeslice limit. Programs of this type are given processor control for an interval (Time Quantum) of maximum duration (or until completed, terminated or suspended). Control is then passed to the next program of the same priority (queue), continuing in a round-robin fashion until all programs of the specified priority have completed, terminated or suspended. The RTE-6/VM Dispatcher then searches the scheduled list, shown in Figure 1-11, for the next highest priority level that has programs prepared to execute.

```
+---------------------------------------------+
|                                             |
|  PRIORITY NUMBER                            |
|                                             |
|         1                                   |
|         2                                   |
|         3                                   |
|         4   linear scheduling               |
|         .                                   |
|         .                                   |
|         .                                   |
|        49                                   |
| ----------------------------                |
|        50                                   |
|        51                                   |
|        52   circular scheduling within      |
|             priority levels                 |
|                                             |
|         .                          Time-    |
|         .                          Slice    |
|         .                          Range    |
|       32767                                 |
|                                             |
+---------------------------------------------+
```

Figure 1-11. Scheduled List

Within the scheduled list, each priority level (in the timeslice range) may be thought of as a circular queue. The program at the head of the queue represents the next program of that priority to be executed. All programs in the scheduled list with a higher priority (lower number) have a chance to execute before a lower priority prog ram is entered. When a timeslice program is entered, a maximum execution slice is set up within the operating system. This program is then allowed to execute until one of the following occurs:

1. The program leaves the scheduled list (such as I/O suspend, memory suspend or dormant).

2. A higher priority program is ready to execute.

3. The program exceeds its timeslice quantum.

If a program leaves the scheduled list, its execution slice is assumed complete. Therefore, when the program is again ready to execute it is placed at the end of the queue within its priority, in the scheduled list. Also, when the program is again picked to execute, the original timeslice quantum is set up.

If a higher priority program causes a program to stop executing (but it is still scheduled), the remaining execution slice value is saved in the program ID segment. Then, when the program is again ready to be entered, the remaining count is set up as the timeslice quantum to be used.

When a program exceeds its execution slice, it is moved behind (in the scheduled list) all other programs of the same priority. The program remains scheduled but execution now passes to the new head of the scheduled list (also head of that priority's queue).

The System Manager can control the scheduled list (Timeslicing) in the following ways:

1. Modify the system (multiplier) time-slice quantum (QUantum command).

2. Modify the priority level at which program is time-sliced (QUantum command).

3. Modify a specific program time-slice level (PRiority command).

1.8 MAPPING USER PROGRAMS ($SMAP)

Once a partition is allocated for a program and the program is about to execute, the user map is set up for the program. If the program is being scheduled initially (progrm's first dispatch) the user map registers must be loaded by the Dispatcher and a copy saved in the user's protected portion of base page. If the program is being redispatched, to continue after being suspended or after being bumped by a higher priority program, the user map registers are set up by copying them from the saved copy in the protected portion of the user's base page.

A program's first dispatch is identified by the fact that the point of suspension word (XSUSP) is 0 in the program's ID segment. The base page register (logical page 0) is loaded with the first page number of the partition. (That value is in word 3 of the MAT entry.) The next registers are then loaded sequentially with numbers starting at one and incremented by one in each successive register. The number of registers set in this manner depends on the program type or whether or not the program uses COMMON.

If the program to be mapped is a type 6 this is the simplest program mapping. The user base page map register is loaded with the starting page of the partition and for the numbr of pages of the program or 31, whichever is less, the MAP registers are sequentially loaded with the next page.

If the program type is 2 or 3, the number of registers set sequentially is determined by one less than the value of $SDA added to $SDT2. Actually the number of registers mapped is one less than $SDA. The next registers mapped (number of registers is determined by $SDT2) have the write-protect bit set. This maps into the user map's Table Area I, the Driver Partition Area, COMMON (including SSGA), write-protected System Driver Area and Table Area II.

If the program is not type 3, the Memory Protect Fence Table Index (in the ID segment) is checked to see if the program uses any COMMON or SSGA. If COMMON or SSGA is used, the number of registers set up following the base page register is determined by one less than the value in $CMST. If COMMON or SSGA is needed, the value $SDA -1 is the number of registers to map in Table Area 1 the Driver Partition Area, and COMMON. The user program is mapped in the registers following these registers, pointing to the system areas.

The next registers are loaded with the next physical page numbers sequentially following the page used for the user base page. These are loaded into the map registers until the number of registers specified in word 21 of the ID segment have been set up.

The remaining registers in the user map will be read/write protected to ensure that a program cannot access memory outside of its partition. This mapping is all done in $SMAP, the only routine that loads the user map to describe a specific program. It is also called by RTIOC if maps need to be set up before entering a driver to do unbuffered I/O.

A copy of the user map is saved in the last 32 words of the user's physical base page. The system's map register for the driver partition ($DVPT) is used to map in the user's base page. This portion of the base page is not used during the program's execution since the system communication area is always mapped in on the top portion of the user base page.

With all of the above done, the program is ready to execute in the user map.

Note that $SMAP does not map memory resident programs. The memory resident program area is static and calculated by the generator. The map is 32 words and located at $MRMP. $SUMP sets up the memory resident map for type 1 programs.

1.9 PROGRAM SEGMENT LOAD

The dispatcher also handles program segment loads. Actually the EXEC 8 request is shared. The scheduler handles the initial portion of the request. It does the segment name look up, passes the optional parameters, does all the error checking, and then calls $BRED in the dispatcher to do the actual $XSIO request to bring the segment in off the disc.

$BRED is very similar to the X0100, X0200, and X0300 routines described earlier. $BRED checks to see if the $XSIO call is free if so it is marked busy and the code entered. Again $PRES is used to set up the quad for the disc I/O. Then $XSIO is called. On return, $LIST is called to set the program to the I/O suspend state.

When the completion interrupt occurs, the $XSIO call is set free and $LIST is called to put the program back into the scheduled list. Return is to $XCQ.

1.10 MLS DISC-RESIDENT NODE LOADS

An MLS disc resident node load occurs when an MLS program calls a subroutine that is in a disc node that is either not mapped in or is not in memory.

When loading an MLS program, MLLDR changes all JSBs to routines in a node down the path to JSB indirect. The JSB goes indirect through a table. If the routine to be called is in memory and mapped, the entry in the table just has a DEF to the appropriate entry point. If the routine is not mapped or not in memory, the entry points to the routine that calls the operating system to map or load the appropriate MLS disc node. The routine that calls the operating system looks like:

```
$DTHK  NOP
       JSB  $LOD$
.DTAB  OCT  XXXXX      START ADDR OF NODE
       OCT  XXXXX      LAST WORD + 1 OF NODE
       OCT  XXXXX      REL SEC FROM PROG START FOR NODE CODE
       OCT  XXXXX      REL SEC FROM PROG START FOR NODE BASE PAGE
.ORD   OCT  XXXXX      THIS NODE PATH #
.NOD#  OCT  XXXXX      THIS NODE ORDINAL #
        .
        .
        .
$LOD$  NOP
       JSB  EXEC       CALL EXEC TO DO NODE LOAD
       DEF  *+5
       DEF  =D8        MADE TO LOOK LIKE SEG LOAD
       DEF  EXEC+0     SECURITY CHECK
      DEF  $LOD$,I    PASS IN .DTAB ADDRESS
      DEF  $LOD$      PASS IN ADDR OF RETURN ADDR + 1
```

If the called routine is not in memory, $DTHK is called. $DTHK in turn calls LOD, which in turn calls the operating system like a segment load call.

The .DTAB table which has all the information about the node is filled in by MLLDR. Actually this information is the same as the information in the short ID segment. The only difference is that the information is buried in the program area. Just as in the segment load request, all the error checks (and a great deal of paranoid checks) take place in the scheduler. When the scheduler is satisfied it calls $NODL in the dispatcher to do the disc I/O and mapping.

$NODL first checks to see if the address from which the call to $DTHK was made was in the root. That is, did the root call a disc node or is this a disc node calling another disc node? If it is one disc node calling another, no special mapping is required. If it is the root calling, a quick check is made to see if the node requested is the one currently in memory. This is done by comparing the relative start sector of the node requested against word 5 of the program preamble where this is saved for every call from the root to a disc resident node. If they compare then no disc I/O is required. Note: this would be the case in a situation where the code might look like:

```
        DO 100 I = 1,1000
        CALL ABC (-,-,-)
        CALL XYZ (-,-,-)
   100  CONTINUE
```

Where ABC is in a memory resident node, XYZ is in a disc node, and the DO loop in the root.

With this optimization, no unnecessary disc I/O is performed. However, a call to $MNOD is made instead. $MNOD maps into the user map a disc resident node area. (The map registers formerly pointed to the memory resident node area.) At this point the new point of suspension is calculated (contents of $DTHK -1) and saved in the programs ID segment. $NODL then returns.

If the called routine was not in memory, disc I/O is required. If the root is the caller, $MNOD is called first to set up the correct map registers. In addition to setting up the correct map registers for the coming I/O request, $MNOD calls $SUMP to save the changed map registers on the progam's base page.

Once the map registers have been set up, $NODL proceeds like $BRED, the segment load call. The only difference is that instead of calling $PRES to set up the $XSIO quads, a call is made to $NSET to set up the quads. The reason for this is that $PRES extracts the necessary disc information from the short ID segment and in this case the node has no short ID segment. Instead $NSET extracts the necessary disc information from the .DTAB table and (like $PRES) calls $SETD to set up the quads for the $XSIO request.

When $NODL returns, the $XSIO call is made and (like $BRED) the program is set to the I/O suspend state via a call to $LIST. When the completion interrupt occurs, the program is placed back into the scheduled state and control is transferred to $XCQ.

1.11 SHARABLE EMA RESTRICTIONS

Sharable EMA programs require two partitions: one partition for the data and one for the program to execute in. Obviously both partitions must be available at the same time. The fact that the shareable EMA program requires two partitions requires the following restriction to prevent possible deadlock situations arising should a program attempt to compete for itself for partitions.

For a shared EMA program or any progeny of the shared EMA program, there must exist in the system a partition to run that program, and the partition must satisfy the following criteria.

1. That partition is not also a shared EMA partition or a subpartition of a shared EMA partition or the the mother partition of its shared EMA Partition. its shared EMA partition.

2. That partition is large enough to run the program.

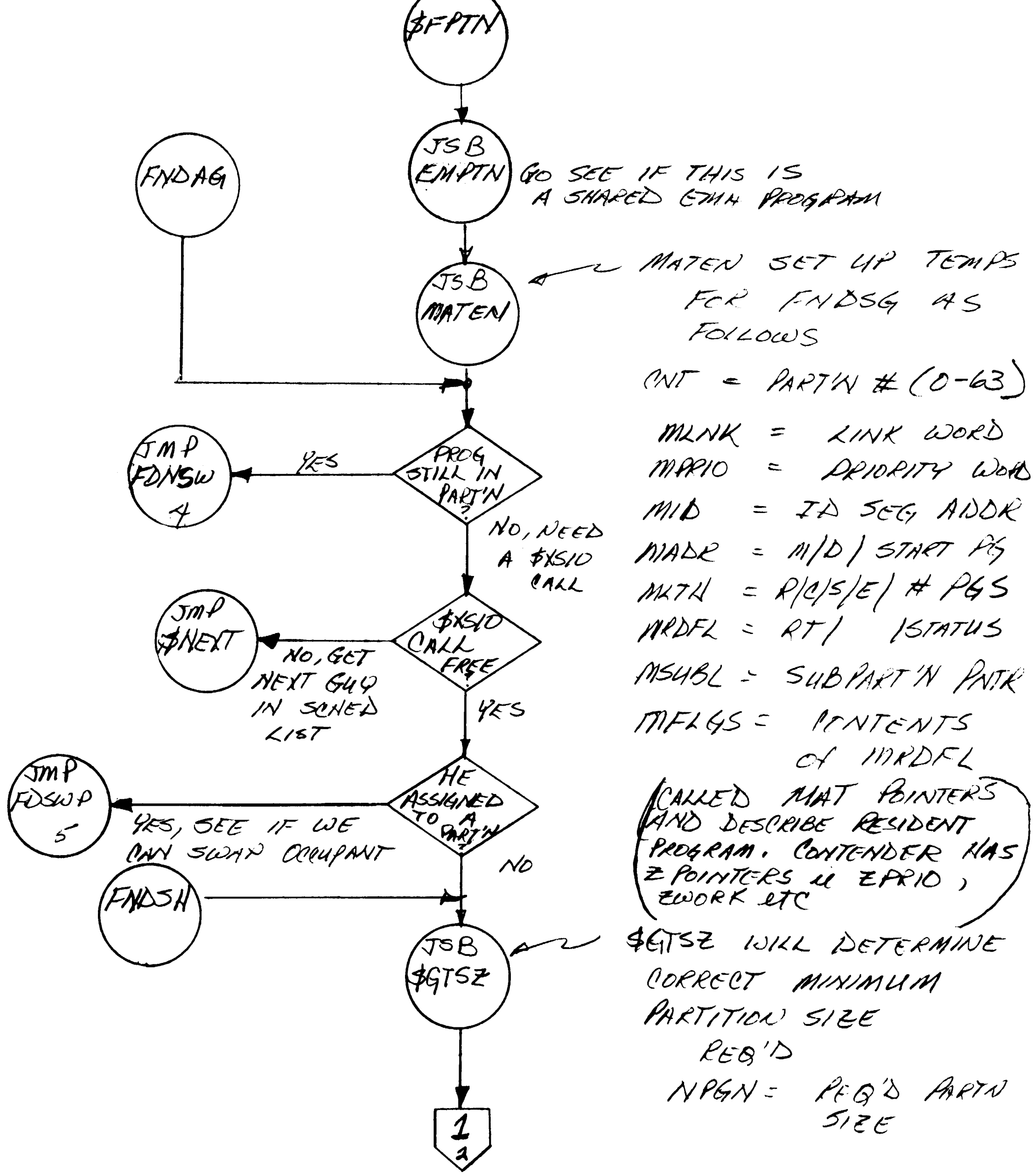
3. That partition is the correct type, i.e., mother, real time, or background.

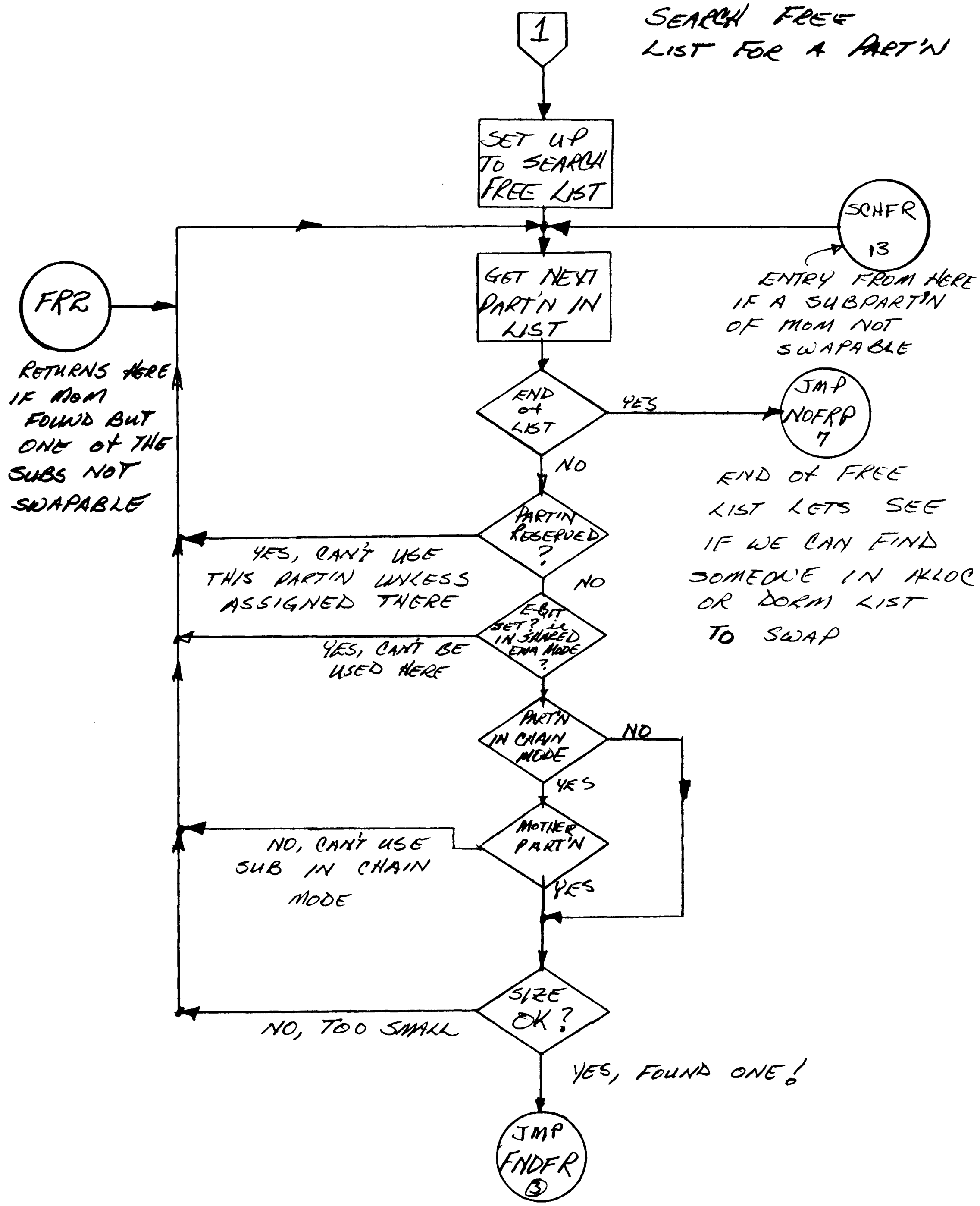
Note that if the partition is a mother partition, none of it may be a shared EMA partition.

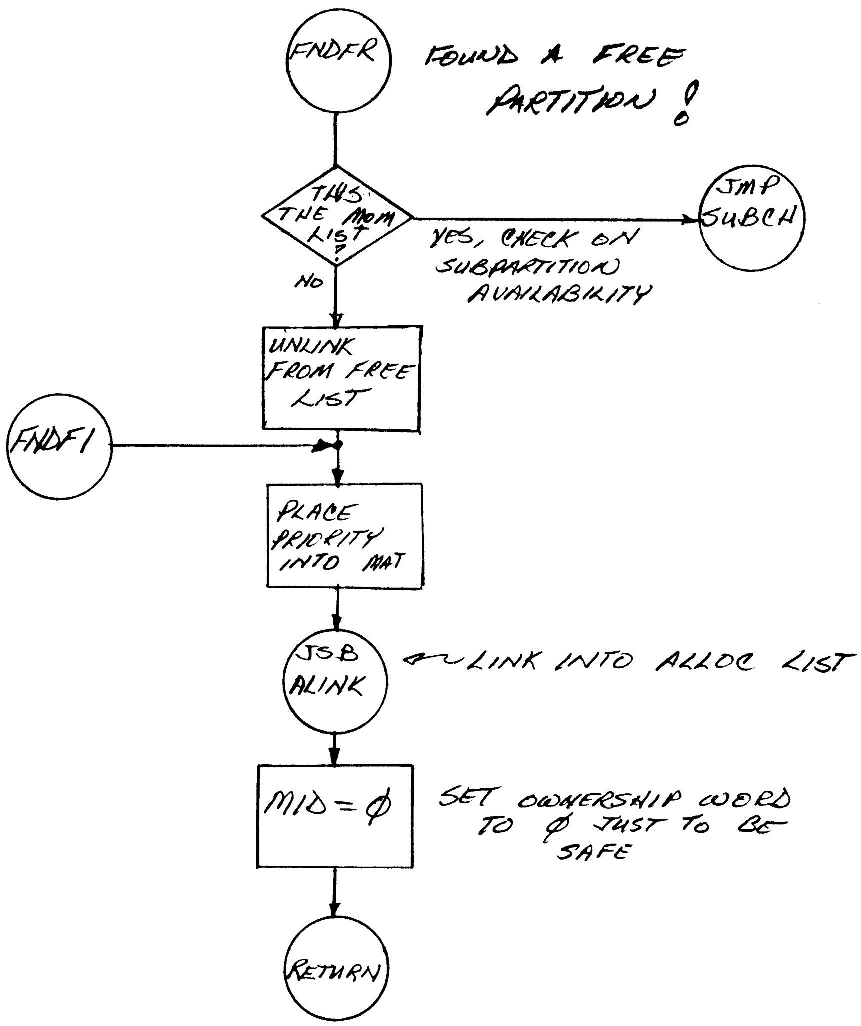
1.12 $FPTN FLOW CHART

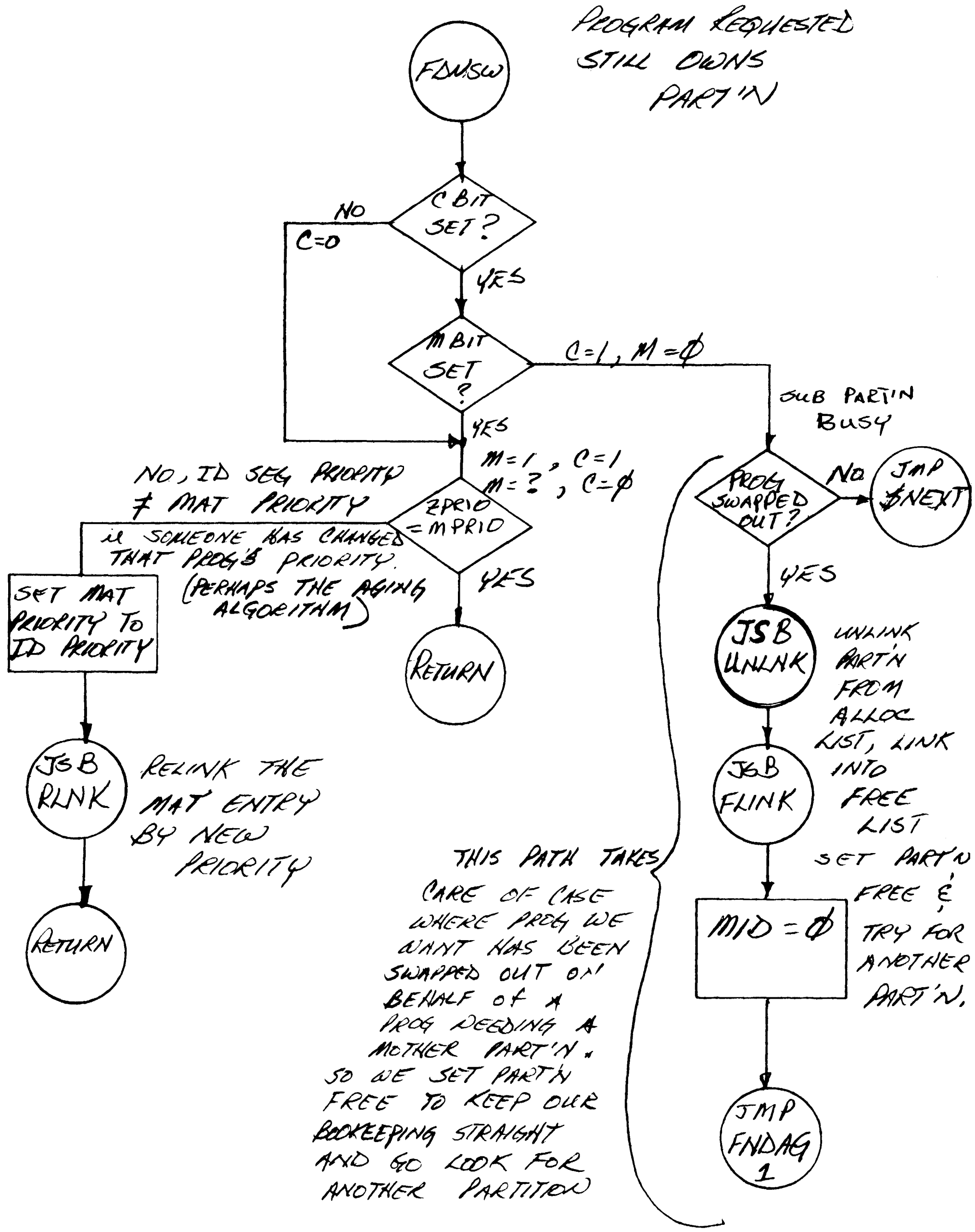
The following is a detailed flow chart set of the of the $FPTN routine.

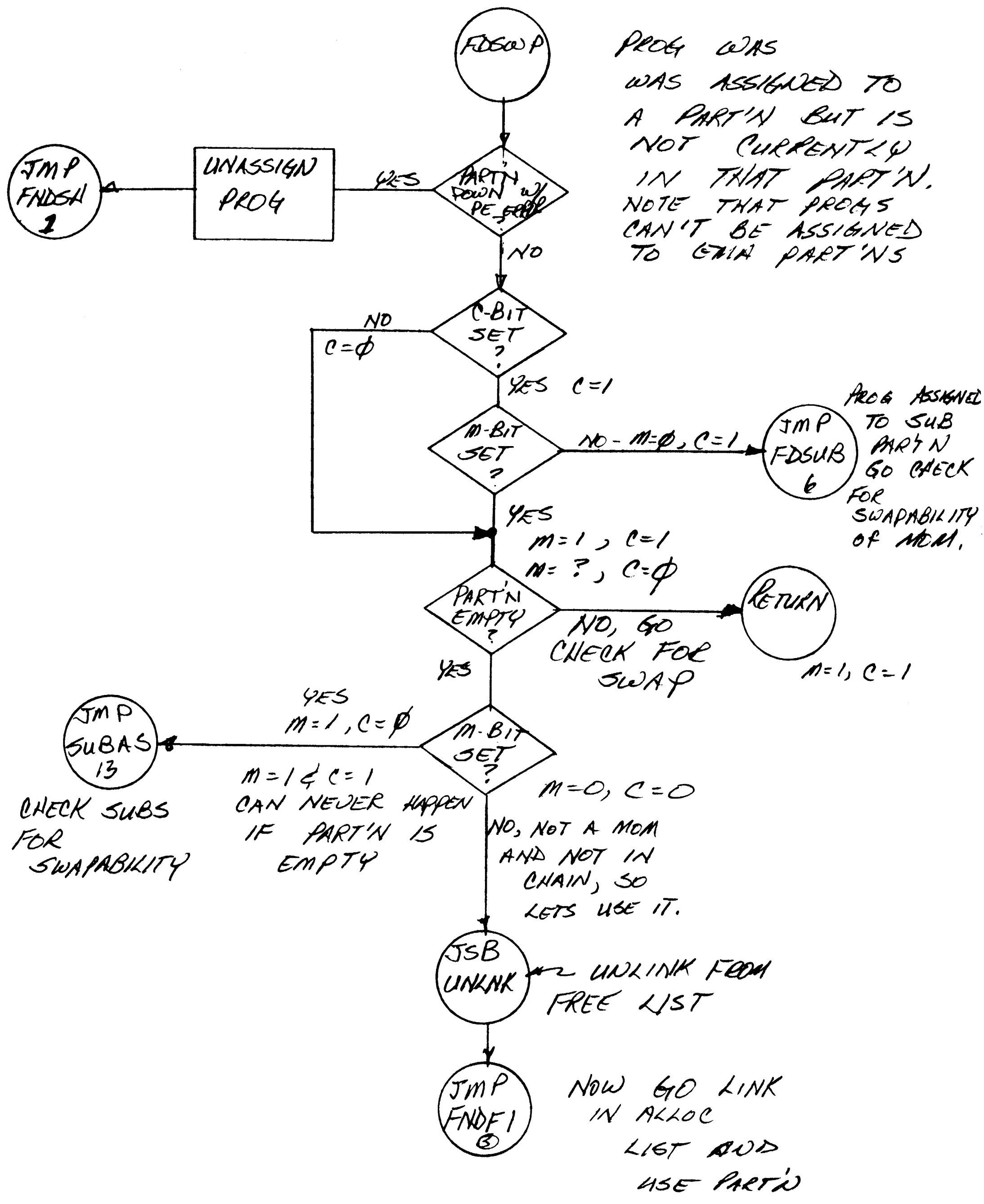
THE $FPTN ROUTINE IS THAT ROUTINE WHICH DECIDES WHICH PARTITION A PROGRAM WILL USE.

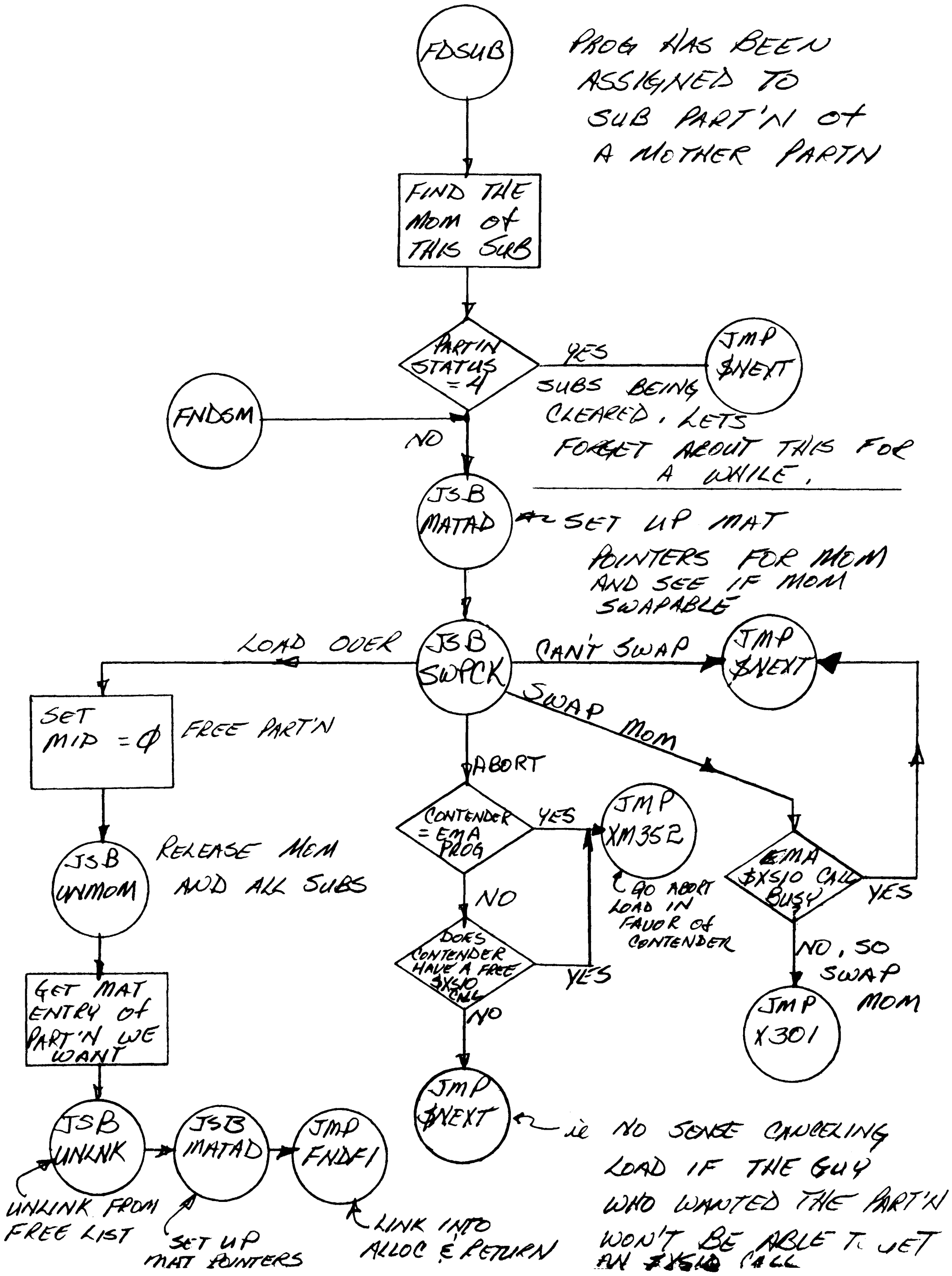



SEARCH FREE LIST FOR A PART'N
1
SET UP TO SEARCH FREE LIST
SCHFR 13
ENTRY FROM HERE IF A SUBPART'N OF MOM NOT SWAPABLE
GET NEXT PART'N IN LIST
FRR
RETURNS HERE IF MOM FOUND BUT ONE OF THE SUBS NOT SWAPABLE
END OF LIST
YES
JMP NOFRP 7
END OF FREE LIST LETS SEE IF WE CAN FIND SOMEONE IN ALLOC OR DORM LIST TO SWAP
NO
PART'N RESERVED ?
YES, CAN'T USE THIS PART'N UNLESS ASSIGNED THERE
NO
E-BIT SET? i.e. IN SHARED EMA MODE ?
YES, CAN'T BE USED HERE
PART'N IN CHAIN MODE
NO
YES
MOTHER PART'N
NO, CAN'T USE SUB IN CHAIN MODE
YES
SIZE OK?
NO, TOO SMALL
YES, FOUND ONE!
JMP FNDFR 3

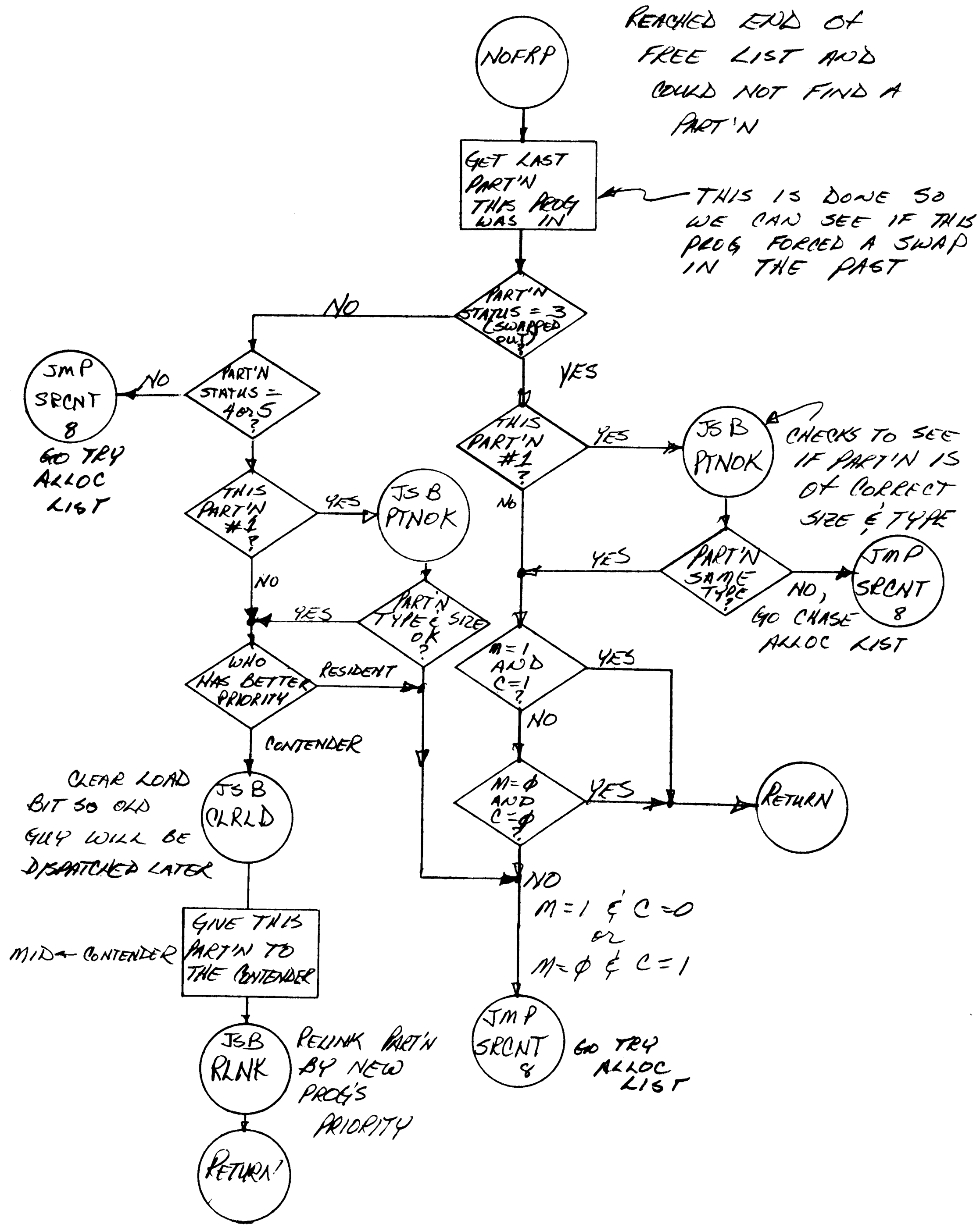

FNDFR
FOUND A FREE PARTITION !
THIS THE MOM LIST ?
JMP SUBCH
YES, CHECK ON SUBPARTITION AVAILABILITY
NO
UNLINK FROM FREE LIST
FNDF1
PLACE PRIORITY INTO MAT
JSB ALINK
LINK INTO ALLOC LIST
MID = Ø
SET OWNERSHIP WORD TO Ø JUST TO BE SAFE
RETURN

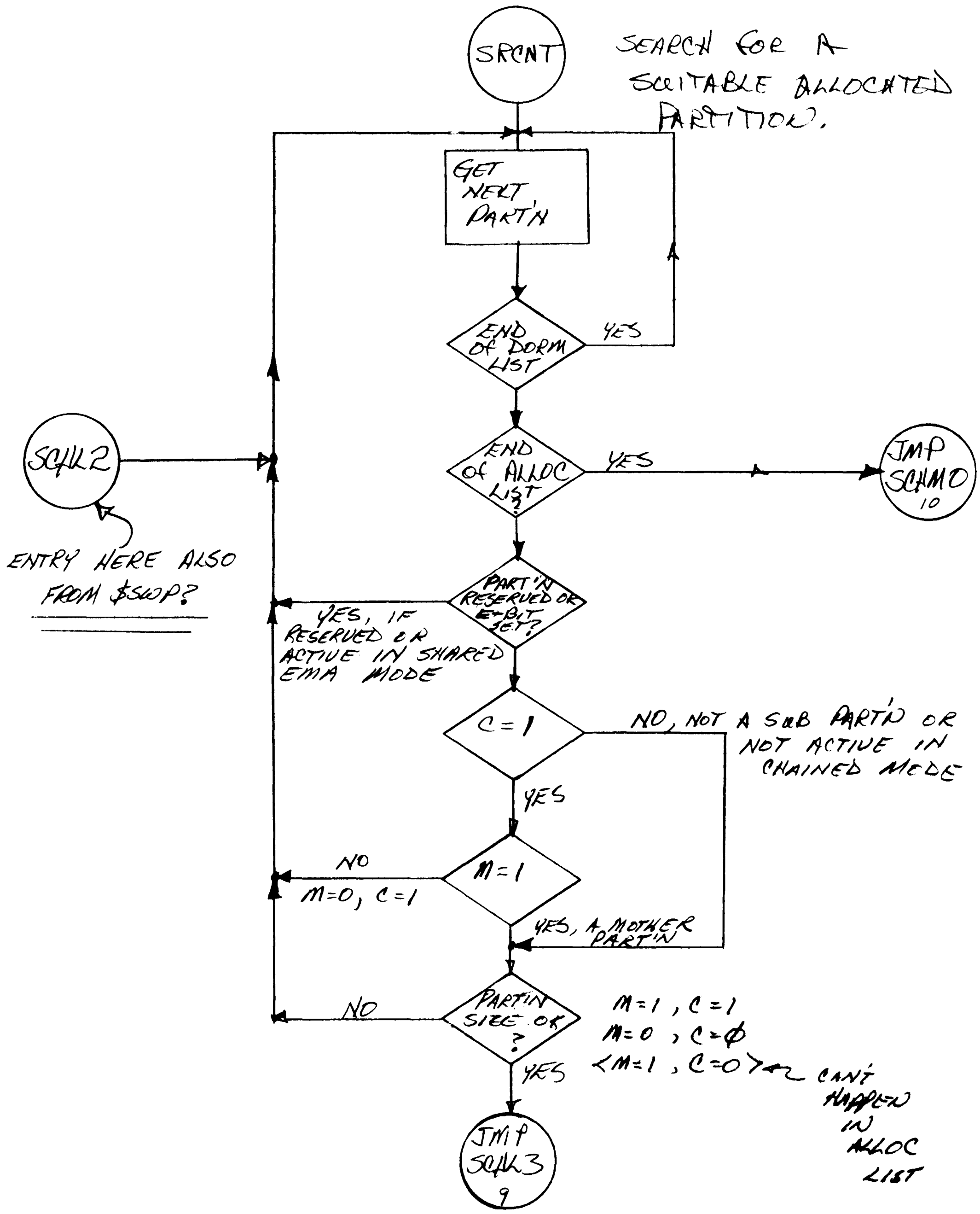

PROGRAM REQUESTED STILL OWNS PART'N
FDNSW
C BIT SET?
NO
C=0
YES
M BIT SET?
C=1, M=Ø
SUB PART'N BUSY
YES
M=1, C=1
M=?, C=Ø
NO, ID SEG PRIORITY ≠ MAT PRIORITY
ZPRIO = MPRIO
i.e. SOMEONE HAS CHANGED THAT PROG'S PRIORITY. (PERHAPS THE AGING ALGORITHM)
YES
SET MAT PRIORITY TO ID PRIORITY
RETURN
JSB RLNK
RELINK THE MAT ENTRY BY NEW PRIORITY
RETURN
PROG SWAPPED OUT?
NO
JMP $NEXT
YES
JSB UNLNK
UNLINK PART'N FROM ALLOC LIST, LINK INTO FREE LIST
JSB FLINK
SET PART'N FREE & TRY FOR ANOTHER PART'N.
MID = Ø
JMP FNDAG 1
THIS PATH TAKES CARE OF CASE WHERE PROG WE WANT HAS BEEN SWAPPED OUT ON BEHALF OF A PROG NEEDING A MOTHER PART'N. SO WE SET PART'N FREE TO KEEP OUR BOOKKEEPING STRAIGHT AND GO LOOK FOR ANOTHER PARTITION

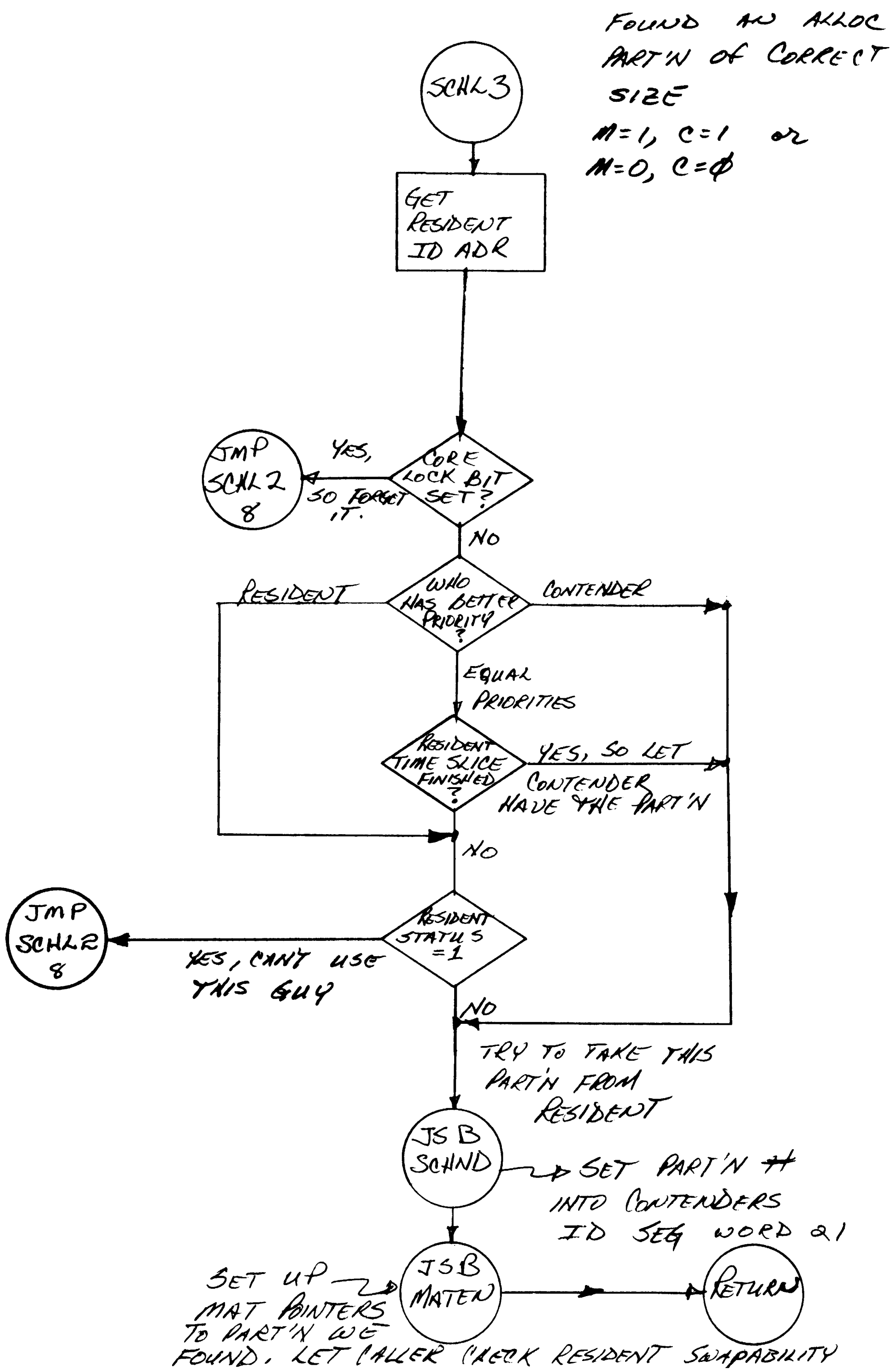

FDSWP
PROG WAS
WAS ASSIGNED TO
A PART'N BUT IS
NOT CURRENTLY
IN THAT PART'N.
NOTE THAT PROGS
CAN'T BE ASSIGNED
TO EMA PART'NS
JMP FNDSH 1
UNASSIGN PROG
YES
PART'N DOWN w/ PE ERR
NO
C-BIT SET ?
NO
C=Ø
YES C=1
M-BIT SET ?
NO - M=Ø, C=1
JMP FDSUB 6
PROG ASSIGNED TO SUB PART'N GO CHECK FOR SWAPABILITY OF MOM.
YES
M=1 , C=1
M= ? , C=Ø
PART'N EMPTY ?
NO, GO CHECK FOR SWAP
RETURN
M=1, C=1
YES
YES
M=1 , C=Ø
JMP SUBAS 13
M-BIT SET ?
M=0, C=0
CHECK SUBS FOR SWAPABILITY
M=1 & C=1 CAN NEVER HAPPEN IF PART'N IS EMPTY
NO, NOT A MOM AND NOT IN CHAIN, SO LETS USE IT.
JSB UNLNK
UNLINK FROM FREE LIST
JMP FNDF1 3
NOW GO LINK IN ALLOC LIST AND USE PART'N

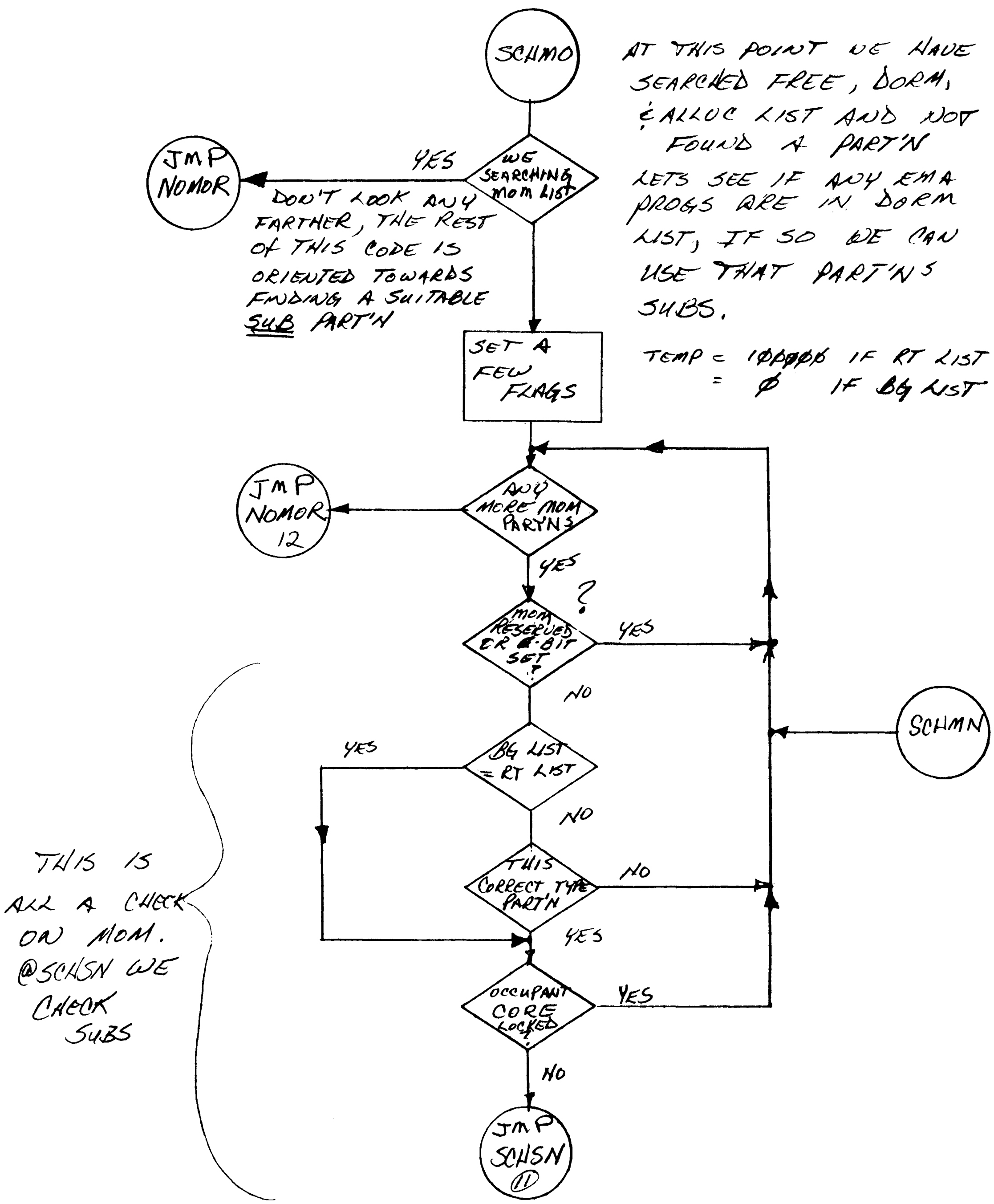


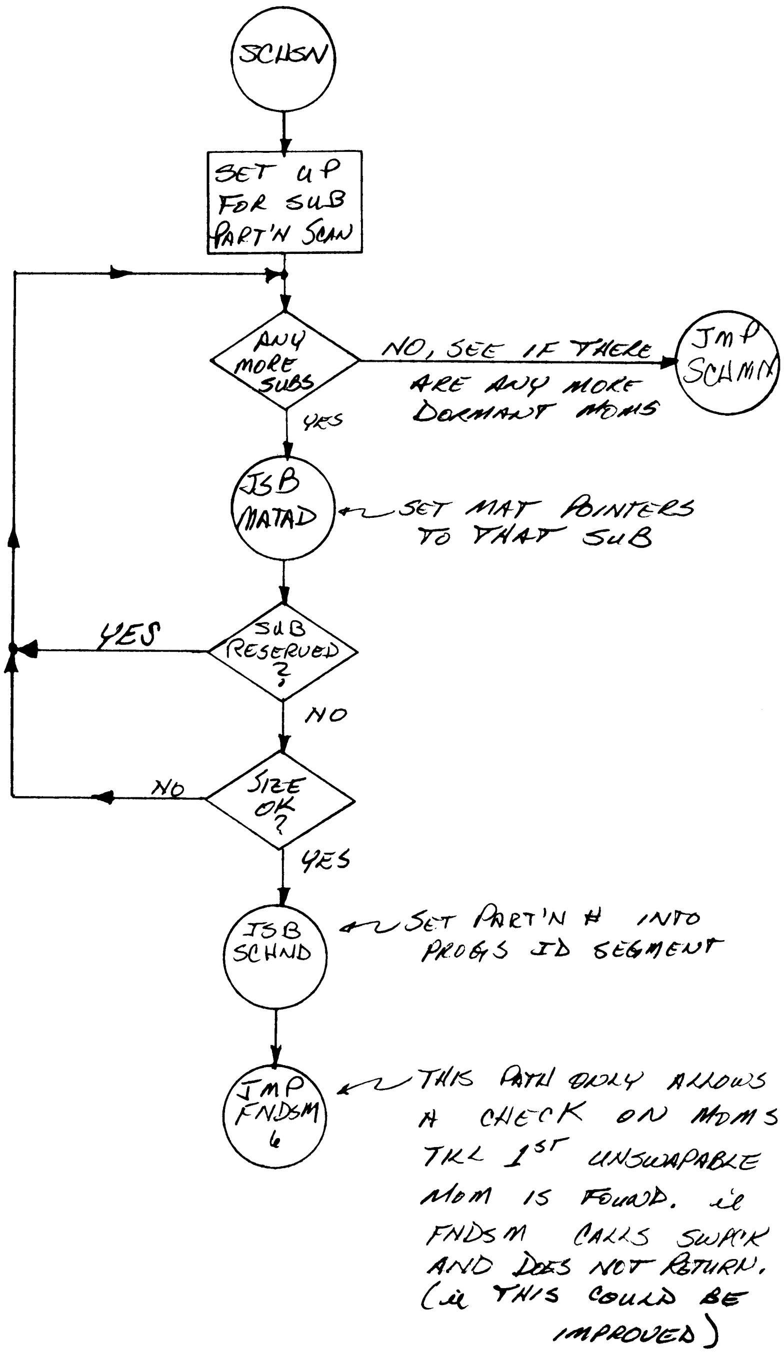
FDSUB
PROG HAS BEEN ASSIGNED TO SUB PART'N OF A MOTHER PARTN
FIND THE MOM OF THIS SUB
PART'N STATUS = 4
YES
JMP $NEXT
SUBS BEING CLEARED, LETS FORGET ABOUT THIS FOR A WHILE.
FNDSM
NO
JSB MATAD
SET UP MAT POINTERS FOR MOM AND SEE IF MOM SWAPABLE
JSB SWPCK
LOAD OVER
CAN'T SWAP
JMP $NEXT
SWAP MOM
ABORT
SET MIP = Ø
FREE PART'N
CONTENDER = EMA PROG
YES
JMP XM352
GO ABORT LOAD IN FAVOR OF CONTENDER
EMA $XS10 CALL BUSY
YES
JSB UNMOM
RELEASE MOM AND ALL SUBS
NO
DOES CONTENDER HAVE A FREE $XS10 CALL
YES
NO, SO SWAP MOM
GET MAT ENTRY OF PART'N WE WANT
NO
JMP X301
JMP $NEXT
ie NO SENSE CANCELING LOAD IF THE GUY WHO WANTED THE PART'N WON'T BE ABLE TO GET AN $XS10 CALL
JSB UNLNK
JSB MATAD
JMP FNDFI
UNLINK FROM FREE LIST
SET UP MAT POINTERS
LINK INTO ALLOC & RETURN

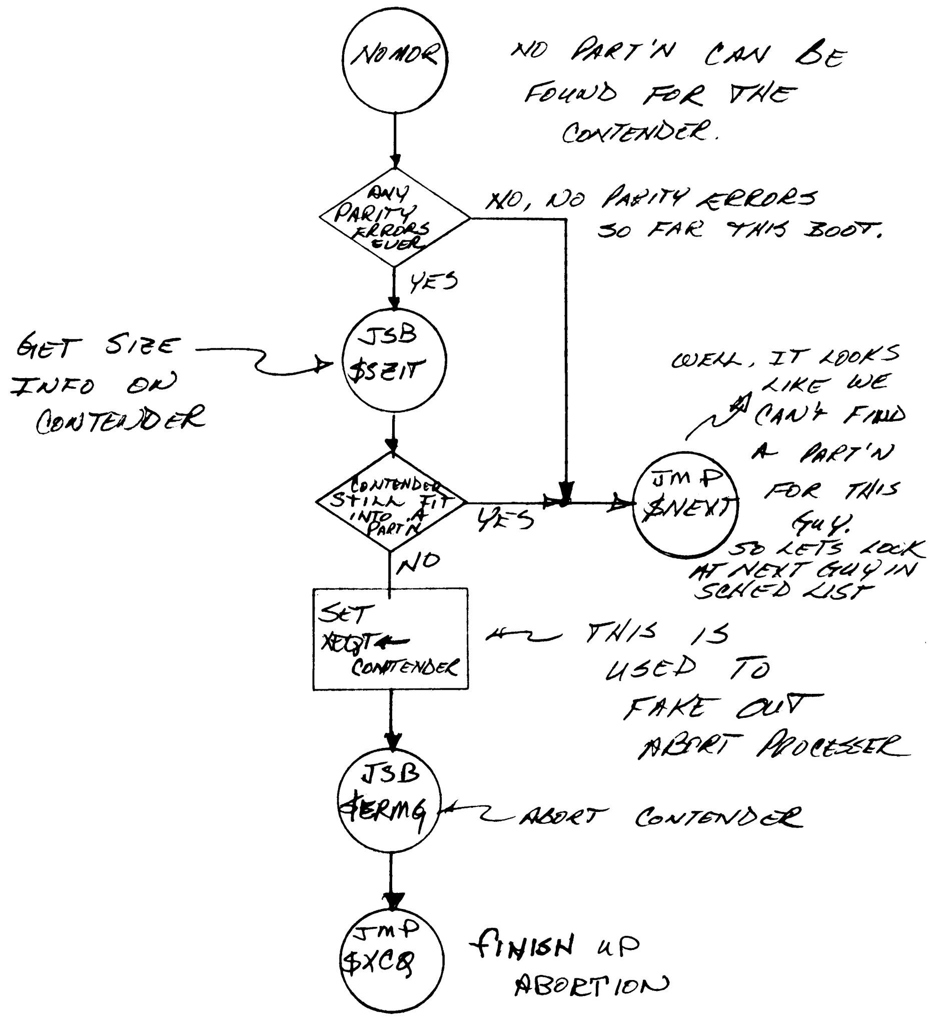

NOFRP
REACHED END OF FREE LIST AND COULD NOT FIND A PART'N
GET LAST PART'N THIS PROG WAS IN
THIS IS DONE SO WE CAN SEE IF THIS PROG FORCED A SWAP IN THE PAST
PART'N STATUS = 3 (SWAPPED OUT)?
NO
YES
PART'N STATUS = 4 or 5?
JMP SRCNT 8
GO TRY ALLOC LIST
THIS PART'N #1?
YES
NO
JSB PTNOK
CHECKS TO SEE IF PART'N IS OF CORRECT SIZE & TYPE
PART'N SAME TYPE?
NO, GO CHASE ALLOC LIST
PART'N TYPE & SIZE OK?
WHO HAS BETTER PRIORITY
RESIDENT
CONTENDER
M=1 AND C=1?
M=ϕ AND C=ϕ?
RETURN
M=1 & C=0 or M=ϕ & C=1
CLEAR LOAD BIT SO OLD GUY WILL BE DISPATCHED LATER
JSB CLRLD
GIVE THIS PART'N TO THE CONTENDER
MID← CONTENDER
JSB RLNK
RELINK PART'N BY NEW PROG'S PRIORITY
RETURN

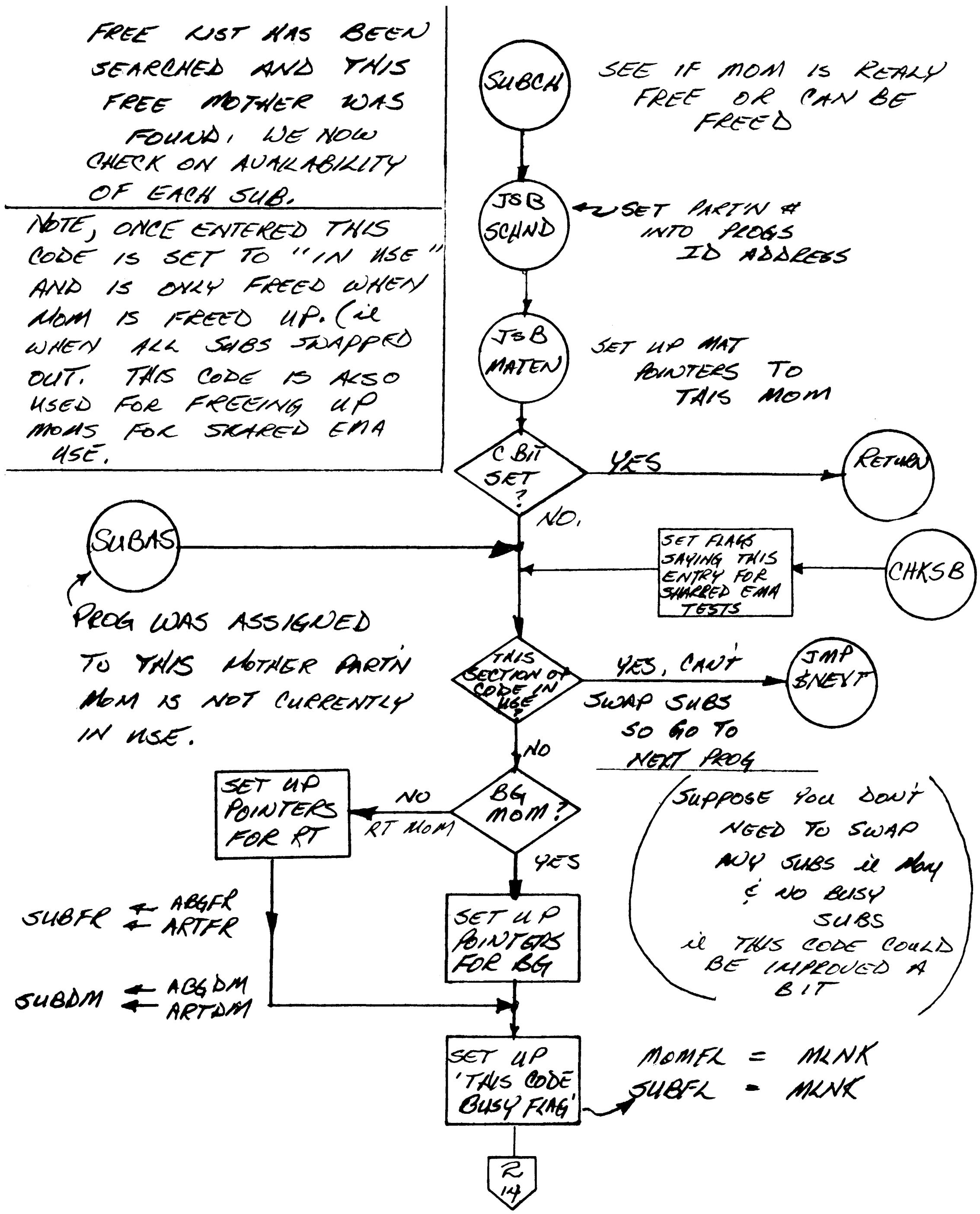

SRCNT
SEARCH FOR A SUITABLE ALLOCATED PARTITION.
GET NEXT PART'N
END OF DORM LIST
YES
END OF ALLOC LIST?
YES
JMP SCHMO 10
SCHL2
ENTRY HERE ALSO FROM $SWP?
PART'N RESERVED OR E-BIT SET?
YES, IF RESERVED OR ACTIVE IN SHARED EMA MODE
C=1
NO, NOT A SUB PART'N OR NOT ACTIVE IN CHAINED MODE
YES
M=1
NO
M=0, C=1
YES, A MOTHER PART'N
PART'N SIZE OK?
NO
YES
M=1, C=1
M=0, C=Ø
<M=1, C=0> CAN'T HAPPEN IN ALLOC LIST
JMP SCHL3 9

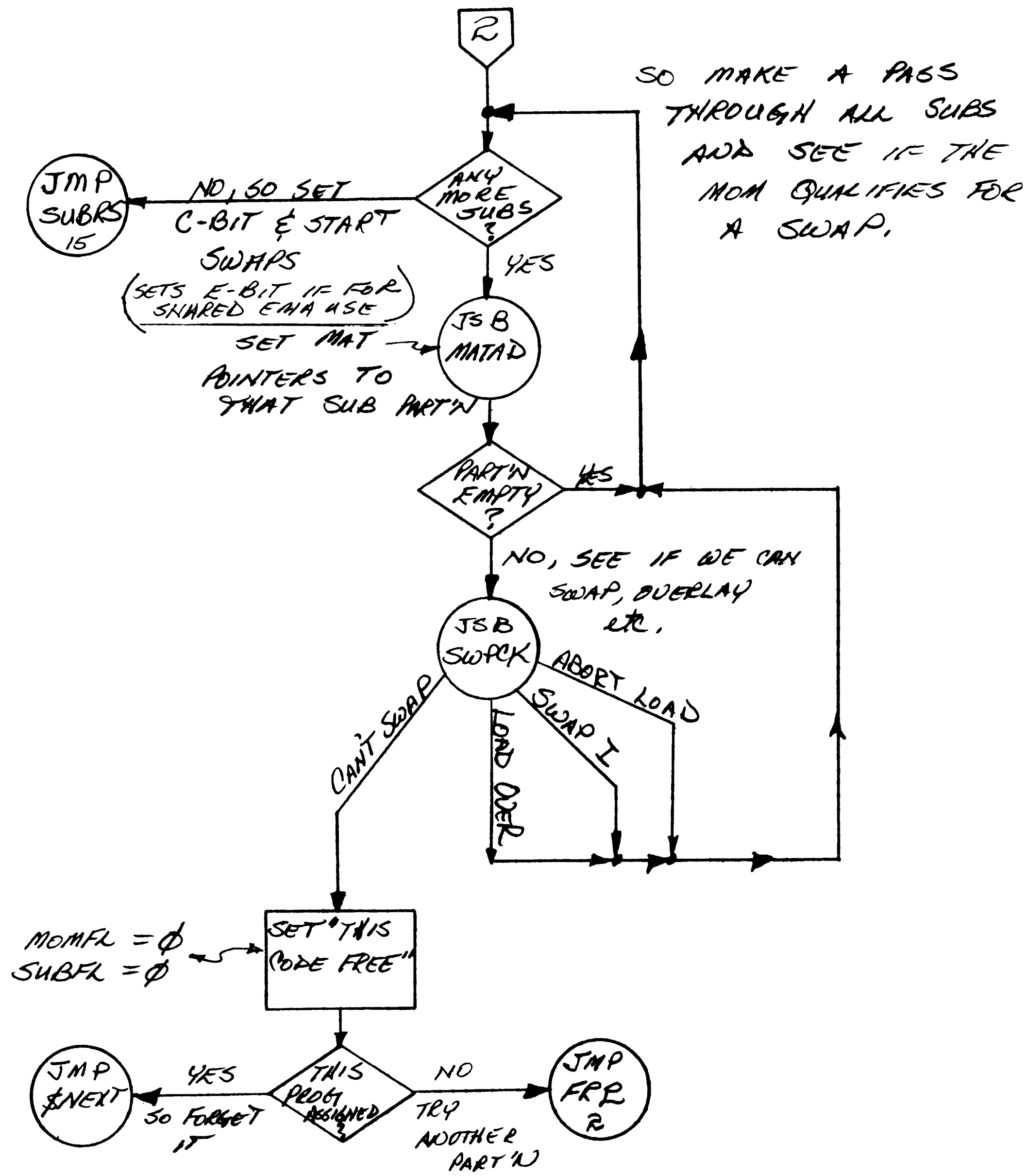

FOUND AN ALLOC PART'N OF CORRECT SIZE
M=1, C=1 or M=0, C=Ø
SCHL3
GET RESIDENT ID ADR
CORE LOCK BIT SET?
YES, SO FORGET IT.
JMP SCHL2 8
NO
WHO HAS BETTER PRIORITY?
RESIDENT
CONTENDER
EQUAL PRIORITIES
RESIDENT TIME SLICE FINISHED?
YES, SO LET CONTENDER HAVE THE PART'N
NO
RESIDENT STATUS =1
YES, CAN'T USE THIS GUY
JMP SCHL2 8
NO
TRY TO TAKE THIS PART'N FROM RESIDENT
JSB SCHND
SET PART'N # INTO CONTENDERS ID SEG WORD 21
SET UP MAT POINTERS TO PART'N WE FOUND. LET CALLER CHECK RESIDENT SWAPABILITY
JSB MATEN
RETURN


SCHMO
AT THIS POINT WE HAVE SEARCHED FREE, DORM, & ALLOC LIST AND NOT FOUND A PART'N
LETS SEE IF ANY EMA PROGS ARE IN DORM LIST, IF SO WE CAN USE THAT PART'N'S SUBS.
WE SEARCHING MOM LIST
YES
JMP NOMOR
DON'T LOOK ANY FARTHER, THE REST OF THIS CODE IS ORIENTED TOWARDS FINDING A SUITABLE SUB PART'N
SET A FEW FLAGS
TEMP = 100000 IF RT LIST
= 0 IF BG LIST
ANY MORE MOM PART'NS
JMP NOMOR 12
YES
MOM RESERVED OR E-BIT SET ?
YES
NO
SCHMN
BG LIST = RT LIST
YES
NO
THIS CORRECT TYPE PART'N
NO
YES
OCCUPANT CORE LOCKED ?
YES
NO
JMP SCHSN 11
THIS IS ALL A CHECK ON MOM. @SCHSN WE CHECK SUBS

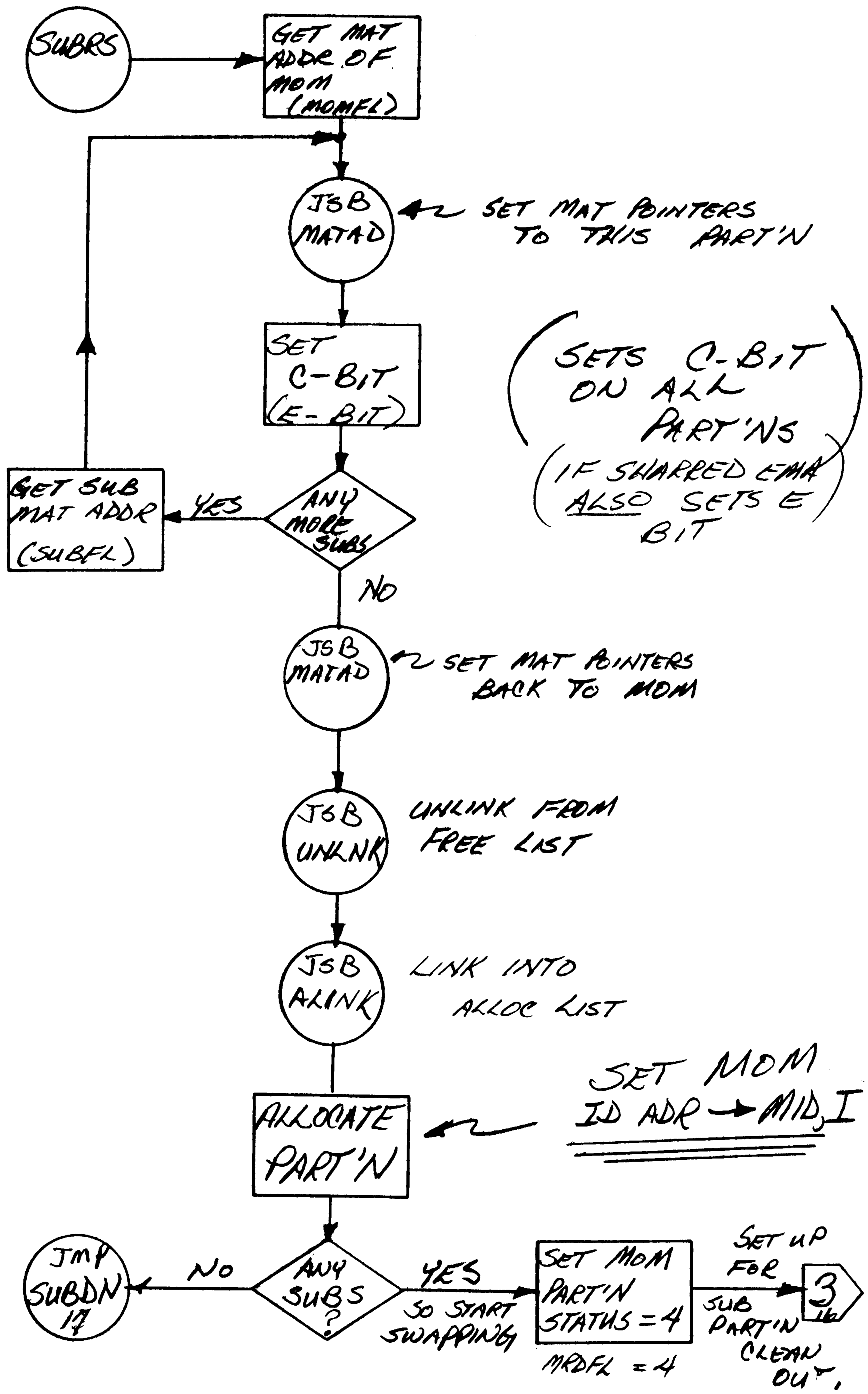

SCHSN
SET UP FOR SUB PART'N SCAN
ANY MORE SUBS
NO, SEE IF THERE ARE ANY MORE DORMANT MOMS
JMP SCHMN
YES
JSB MATAD
SET MAT POINTERS TO THAT SUB
SUB RESERVED ?
YES
NO
SIZE OK ?
NO
YES
JSB SCHND
SET PART'N # INTO PROGS ID SEGMENT
JMP FNDSM 6
THIS PATH ONLY ALLOWS A CHECK ON MOMS TILL 1ST UNSWAPABLE MOM IS FOUND. IE FNDSM CALLS SWPCK AND DOES NOT RETURN. (IE THIS COULD BE IMPROVED)

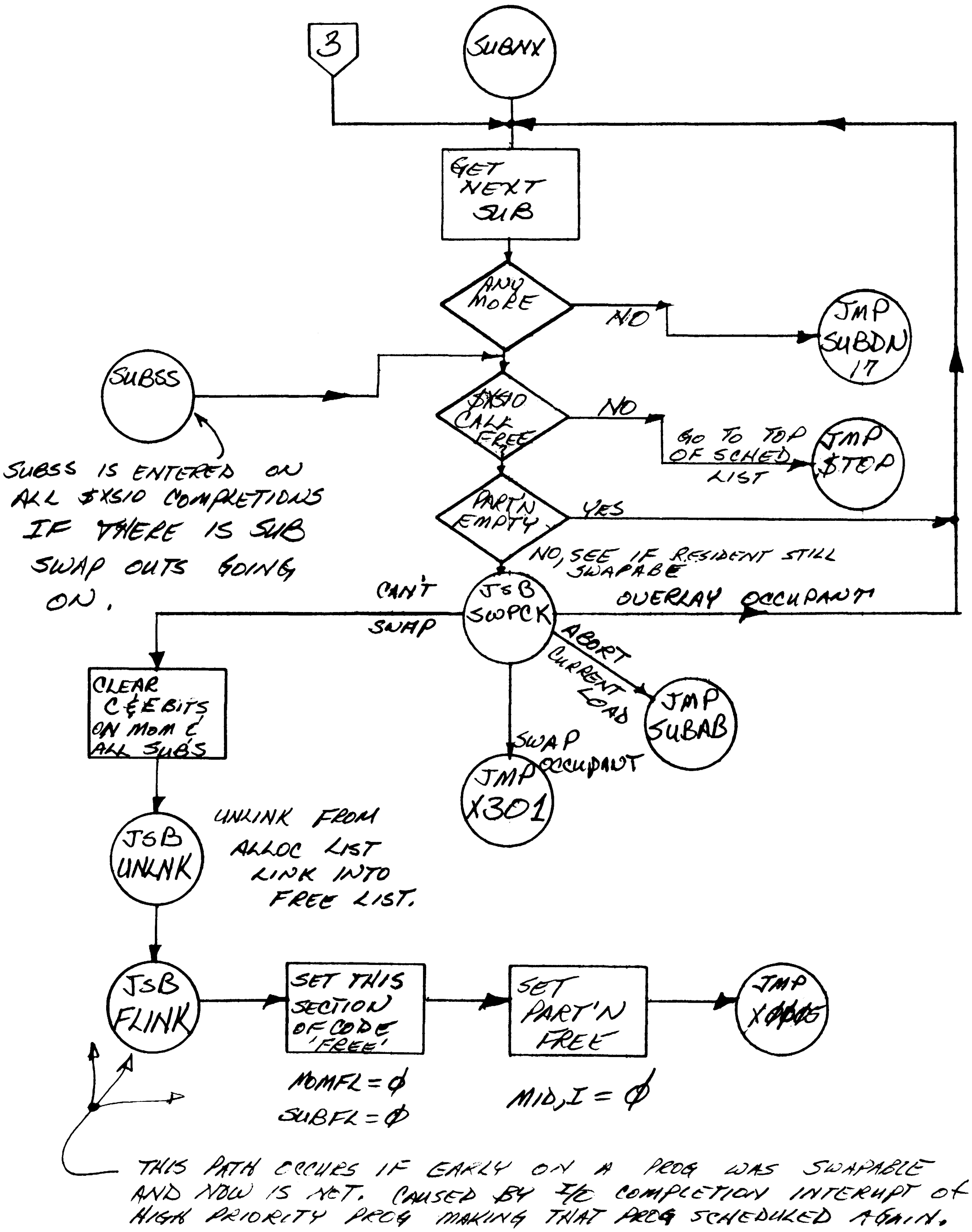

NOMOR
NO PART'N CAN BE FOUND FOR THE CONTENDER.
ANY PARITY ERRORS EVER
NO, NO PARITY ERRORS SO FAR THIS BOOT.
YES
GET SIZE INFO ON CONTENDER
JSB $SZIT
WELL, IT LOOKS LIKE WE CAN'T FIND A PART'N FOR THIS GUY. SO LETS LOOK AT NEXT GUY IN SCHED LIST
CONTENDER STILL FIT INTO A PART'N
YES
JMP $NEXT
NO
SET XEQT ← CONTENDER
THIS IS USED TO FAKE OUT ABORT PROCESSER
JSB $ERMG
ABORT CONTENDER
JMP $XCQ
FINISH UP ABORTION

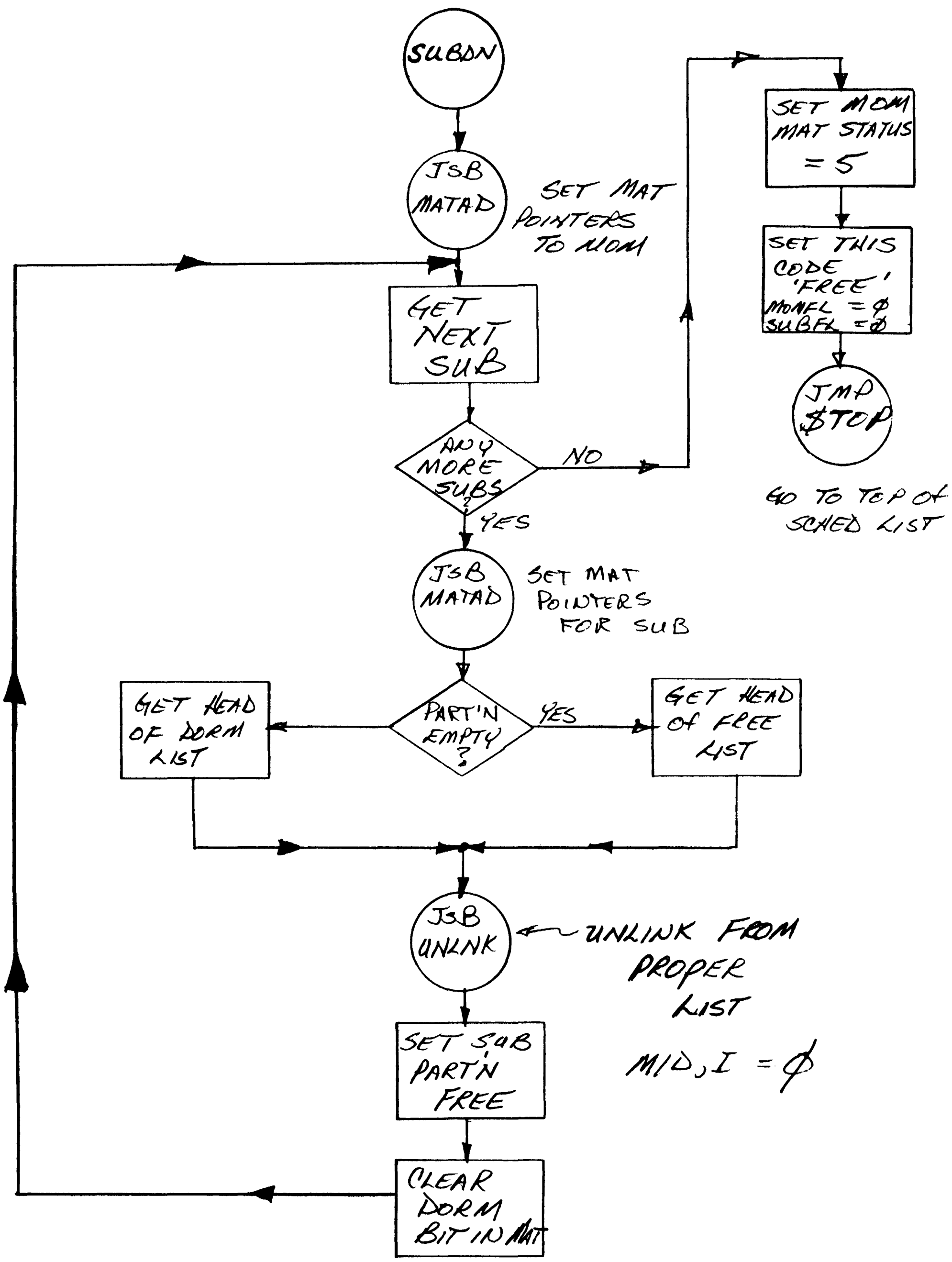

FREE LIST HAS BEEN SEARCHED AND THIS FREE MOTHER WAS FOUND, WE NOW CHECK ON AVAILABILITY OF EACH SUB.
NOTE, ONCE ENTERED THIS CODE IS SET TO "IN USE" AND IS ONLY FREED WHEN MOM IS FREED UP. (ie WHEN ALL SUBS SWAPPED OUT. THIS CODE IS ALSO USED FOR FREEING UP MOMS FOR SHARED EMA USE.
SUBCH
SEE IF MOM IS REALLY FREE OR CAN BE FREED
JSB SCHND
SET PART'N # INTO PROGS ID ADDRESS
JSB MATEN
SET UP MAT POINTERS TO THIS MOM
C BIT SET ?
YES
RETURN
NO.
SUBAS
PROG WAS ASSIGNED TO THIS MOTHER PART'N MOM IS NOT CURRENTLY IN USE.
SET FLAGS SAYING THIS ENTRY FOR SHARED EMA TESTS
CHKSB
THIS SECTION OF CODE IN USE?
YES, CAN'T SWAP SUBS SO GO TO NEXT PROG
JMP $NEXT
NO
BG MOM?
NO
RT MOM
YES
SET UP POINTERS FOR RT
SET UP POINTERS FOR BG
SUPPOSE YOU DON'T NEED TO SWAP ANY SUBS ie MOM & NO BUSY SUBS ie THIS CODE COULD BE IMPROVED A BIT
SUBFR ⇄ ABGFR ARTFR
SUBDM ⇄ ABGDM ARTDM
SET UP 'THIS CODE BUSY FLAG'
MOMFL = MLNK
SUBFL = MLNK
R 14



2
SO MAKE A PASS THROUGH ALL SUBS AND SEE IF THE MOM QUALIFIES FOR A SWAP.
ANY MORE SUBS ?
NO, SO SET C-BIT & START SWAPS
(SETS E-BIT IF FOR SHARED EMA USE)
JMP SUBRS 15
YES
JSB MATAD
SET MAT POINTERS TO THAT SUB PART'N
PART'N EMPTY ?
YES
NO, SEE IF WE CAN SWAP, OVERLAY etc.
JSB SWPCK
ABORT LOAD
SWAP I
LOAD OVER
CAN'T SWAP
SET "THIS CODE FREE"
MOMFL = Ø
SUBFL = Ø
THIS PROG ASSIGNED ?
YES
SO FORGET IT
JMP $NEXT
NO
TRY ANOTHER PART'N
JMP FRE 2


SUBRS
GET MAT ADDR OF MOM (MOMFL)
JSB MATAD
SET MAT POINTERS TO THIS PART'N
SET C-BIT (E-BIT)
SETS C-BIT ON ALL PART'NS
(IF SHARRED EMA ALSO SETS E BIT)
ANY MORE SUBS
YES
GET SUB MAT ADDR (SUBFL)
NO
JSB MATAD
SET MAT POINTERS BACK TO MOM
JSB UNLNK
UNLINK FROM FREE LIST
JSB ALINK
LINK INTO ALLOC LIST
ALLOCATE PART'N
SET MOM ID ADR → MID,I
ANY SUBS ?
NO
JMP SUBDN 17
YES SO START SWAPPING
SET MOM PART'N STATUS = 4
MRDFL = 4
SET UP FOR SUB PART'N CLEAN OUT.
3 16


3
SUBNX
GET NEXT SUB
ANY MORE
NO
JMP SUBDN 17
SUBSS
$XSIO CALL FREE
NO
GO TO TOP OF SCHED LIST
JMP $TOP
SUBSS IS ENTERED ON ALL $XSIO COMPLETIONS IF THERE IS SUB SWAP OUTS GOING ON.
PART'N EMPTY
YES
NO, SEE IF RESIDENT STILL SWAPABE
CAN'T SWAP
JSB SWPCK
OVERLAY OCCUPANT
ABORT CURRENT LOAD
JMP SUBAB
SWAP OCCUPANT
JMP X301
CLEAR C & E BITS ON MOM & ALL SUB'S
JSB UNLNK
UNLINK FROM ALLOC LIST LINK INTO FREE LIST.
JSB FLINK
SET THIS SECTION OF CODE 'FREE'
MOMFL = Ø
SUBFL = Ø
SET PART'N FREE
MID, I = Ø
JMP
THIS PATH OCCURS IF EARLY ON A PROG WAS SWAPABLE AND NOW IS NOT. CAUSED BY I/O COMPLETION INTERUPT OF HIGH PRIORITY PROG MAKING THAT PROG SCHEDULED AGAIN.


SUBDN
JSB MATAD
SET MAT POINTERS TO MOM
GET NEXT SUB
ANY MORE SUBS?
NO
YES
SET MOM MAT STATUS = 5
SET THIS CODE 'FREE' MONFL = Ø SUBFL = Ø
JMP $TOP
GO TO TOP OF SCHED LIST
JSB MATAD
SET MAT POINTERS FOR SUB
PART'N EMPTY?
YES
GET HEAD OF DORM LIST
GET HEAD OF FREE LIST
JSB UNLNK
UNLINK FROM PROPER LIST
MID, I = Ø
SET SUB PART'N FREE
CLEAR DORM BIT IN MAT

EMPTN

THE EMPTN ROUTINE IS THE DISPATCHERS SHARED EMA PARTITION SET UP ROUTINE. WHEN A SHARED EMA PROGRAM (SHEMA) IS ENCOUNTERED EMPTN MUST LOOK AT THE TARGET DATA PARTITION AND DECIDE WHAT TO DO. CERTAIN STATES ARE REFERED TO IN THE CODE AND HAVE TO DO WITH THE C-BIT (CHAIN BIT), M-BIT (MOTHER BIT), AND WHETHER THE TARGET PARTITION IS EMPTY OR NOT. THIS IS REFERED TO AS THE CME STATE.

STATE TABLE

STATUS			
C-BIT	M-BIT	EMPTY	ACTION TO TAKE
Ø	Ø	Ø	PART'N NOT IN CHAIN MODE, NOT A MOTHER, AND IS EMPTY. SO TAKE PART'N AND USE IT
Ø	Ø	1	NOT IN CHAIN, NOT A MOM, HAS A RESIDENT PROG SO ATTEMPT TO SWAP RESIDENT
Ø	1	Ø	TAKE MOM AND CHECK SUBS FOR SWAPABILITY
Ø	1	1	IMPOSSIBLE STATE CAN'T HAVE C=Ø & MOM PART'N OCCUPIED.
1	Ø	Ø	FIND MOM AND ATTEMPT HER SWAP. THIS WILL FREE UP REQ'D SUB.
1	Ø	1	SOMEBODY IS IN PROCESS OF CLEARING OUT THIS MOM, MOM'S MAT STATUS MUST = 4. LEAVE THIS SITUATION ALONE IT WILL TAKE CARE OF ITSELF WHEN MAT STATUS = 5.
1	1	Ø	IMPOSSIBLE STATE CAN'T HAVE C=1 & PARTITION EMPTY
1	1	1	ATTEMPT TO SWAP MOM AND TAKE PART'N FOR SHEMA USE

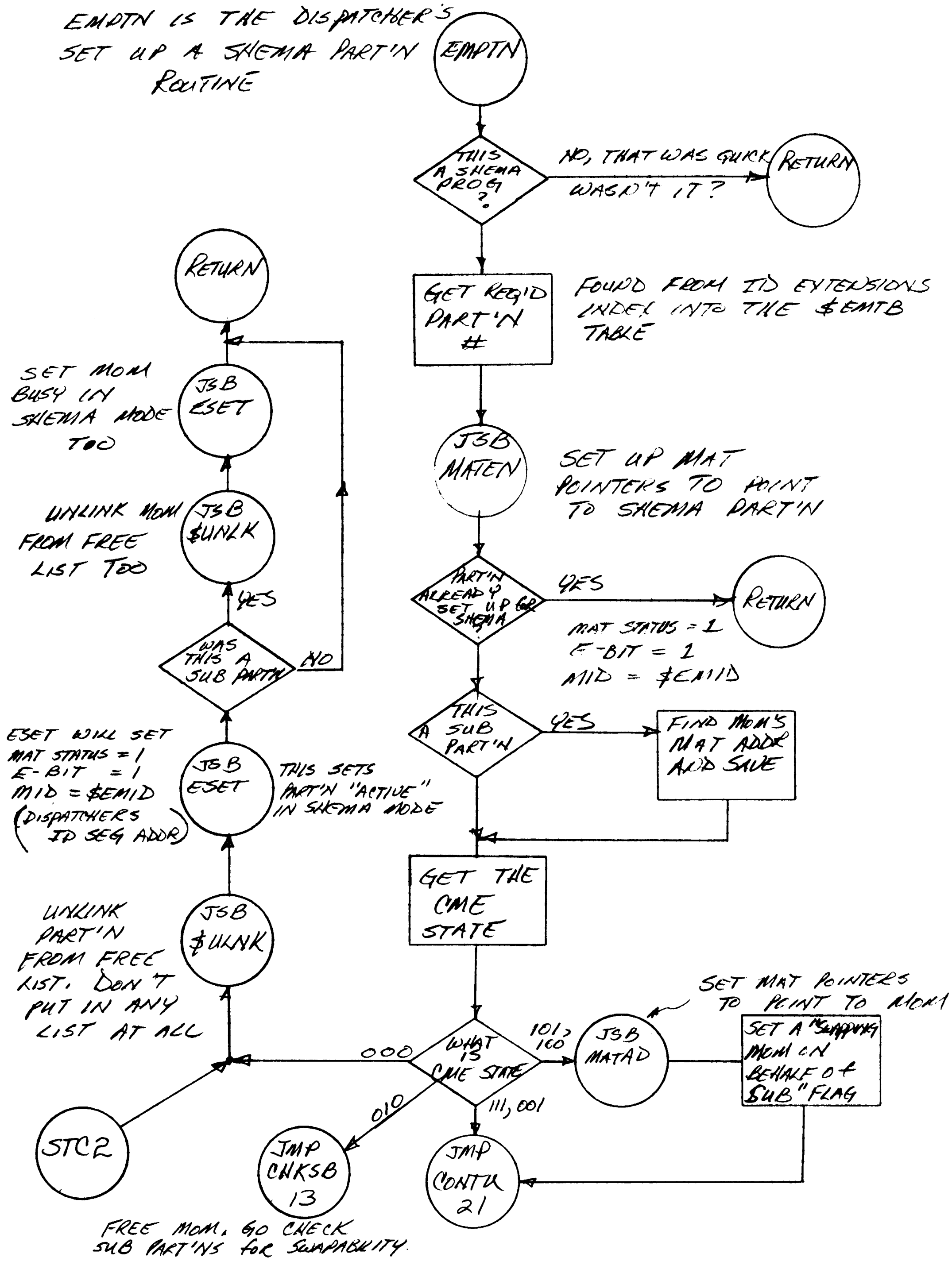
$EMTB TABLE

$EMTB

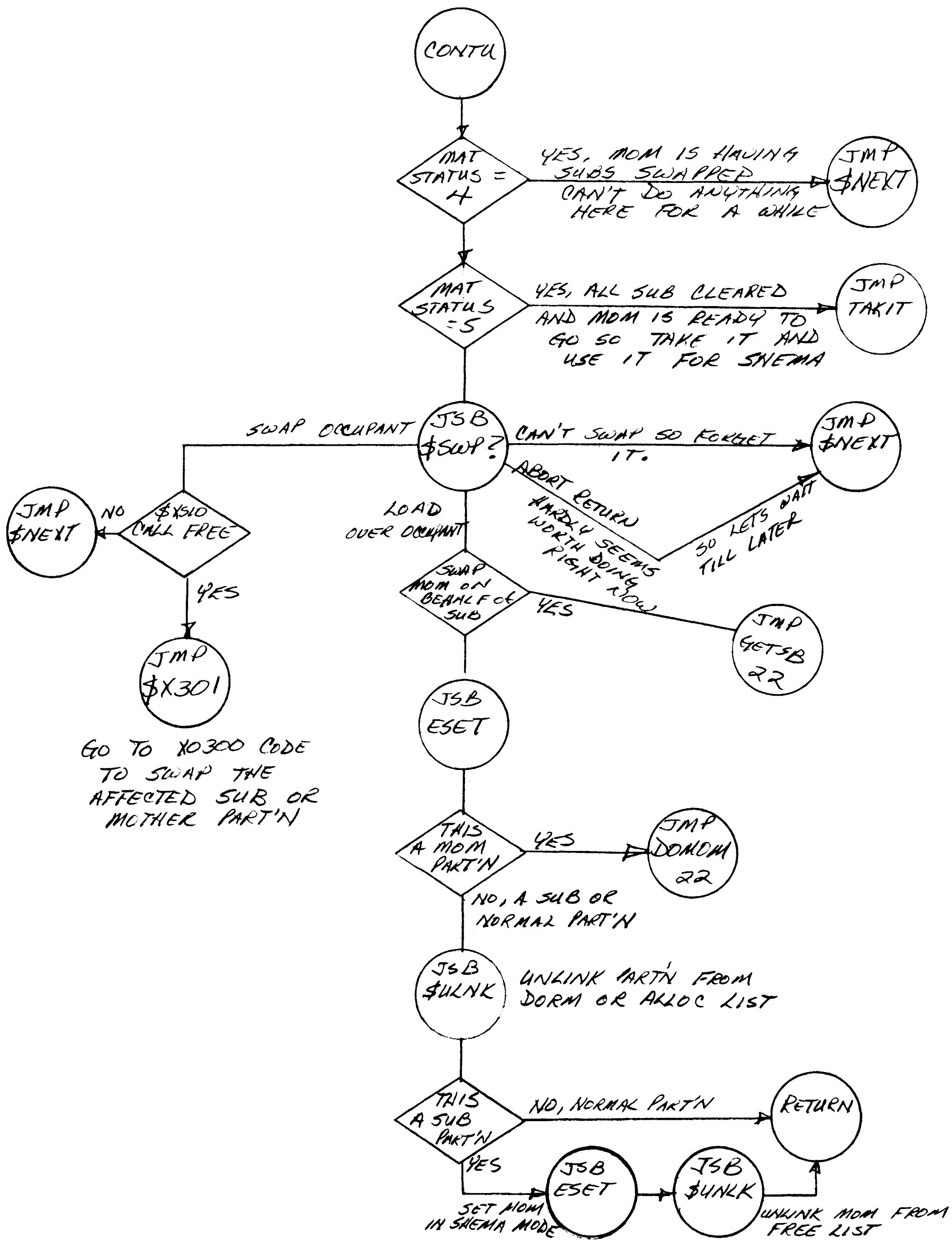
# OF ENTRIES IN TABLE	
L	A
B	E
L	MAT TABLE INDEX
# OF ACTIVE USERS	
RESERVED	
NEXT ENTRY	

5 WORD ENTRIES

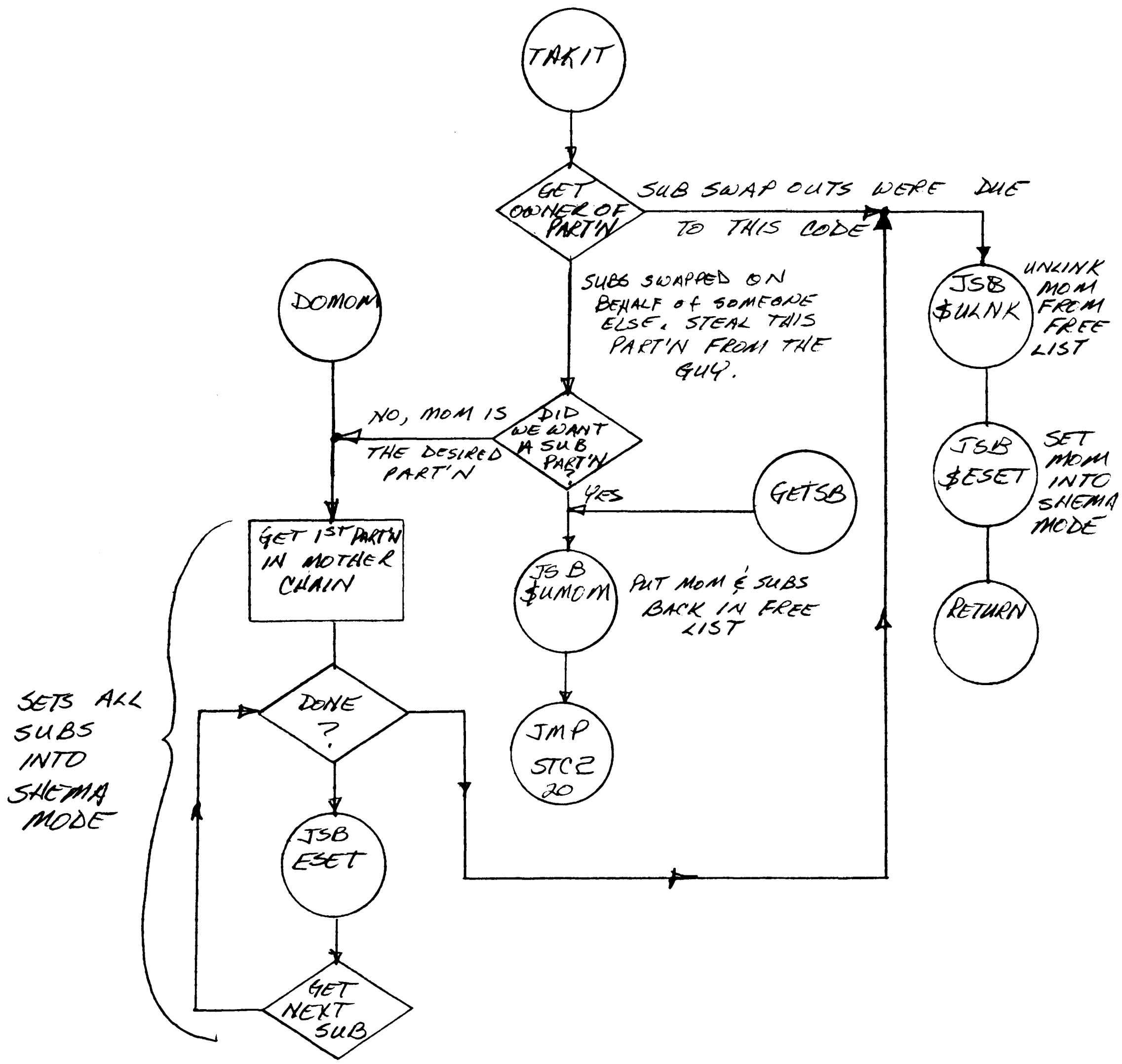
FOR SHEMA PROGS WORD 3 OF PROG'S ID EXTENSION POINTS TO ENTRY OF $EMTB WHICH CONTAINS POINTER TO THE DATA PART'N.


EMPTN IS THE DISPATCHER'S SET UP A SHEMA PART'N ROUTINE
EMPTN
THIS A SHEMA PROG ?
NO, THAT WAS QUICK WASN'T IT ?
RETURN
GET REQ'D PART'N #
FOUND FROM ID EXTENSIONS INDEX INTO THE $EMTB TABLE
JSB MATEN
SET UP MAT POINTERS TO POINT TO SHEMA PART'N
PART'N ALREADY SET UP OR SHEMA
YES
RETURN
MAT STATUS = 1
E-BIT = 1
MID = $EMID
THIS A SUB PART'N
YES
FIND MOM'S MAT ADDR AND SAVE
GET THE CME STATE
WHAT IS CME STATE
000
101, 100
010
111, 001
JSB MATAD
SET MAT POINTERS TO POINT TO MOM
SET A "SWAPPING MOM ON BEHALF OF SUB" FLAG
JMP CNKSB 13
JMP CONTU 21
FREE MOM. GO CHECK SUB PART'NS FOR SWAPABILITY.
STC2
JSB $ULNK
UNLINK PART'N FROM FREE LIST. DON'T PUT IN ANY LIST AT ALL
JSB ESET
THIS SETS PART'N "ACTIVE" IN SHEMA MODE
ESET WILL SET MAT STATUS = 1
E-BIT = 1
MID = $EMID (DISPATCHERS ID SEG ADDR)
WAS THIS A SUB PART'N
NO
YES
JSB $UNLK
UNLINK MOM FROM FREE LIST TOO
JSB ESET
SET MOM BUSY IN SHEMA MODE TOO
RETURN

Dispatcher


CONTU
MAT STATUS = 4
YES, MOM IS HAVING SUBS SWAPPED CAN'T DO ANYTHING HERE FOR A WHILE
JMP $NEXT
MAT STATUS =5
YES, ALL SUB CLEARED AND MOM IS READY TO GO SO TAKE IT AND USE IT FOR SHEMA
JMP TAKIT
SWAP OCCUPANT
JSB $SWP?
CAN'T SWAP SO FORGET IT.
JMP $NEXT
ABORT RETURN
HARDLY SEEMS WORTH DOING RIGHT NOW
SO LETS WAIT TILL LATER
JMP $NEXT
NO
$XSIO CALL FREE
YES
JMP $X301
GO TO X0300 CODE TO SWAP THE AFFECTED SUB OR MOTHER PART'N
LOAD OVER OCCUPANT
SWAP MOM ON BEHALF OF SUB
YES
JMP GETSB 22
JSB ESET
THIS A MOM PART'N
YES
JMP DOMOM 22
NO, A SUB OR NORMAL PART'N
JSB $ULNK
UNLINK PART'N FROM DORM OR ALLOC LIST
THIS A SUB PART'N
NO, NORMAL PART'N
RETURN
YES
JSB ESET
SET MOM IN SHEMA MODE
JSB $UNLK
UNLINK MOM FROM FREE LIST

AT THIS POINT WE KNOW THAT THE MAT STATUS = 5. THUS ALL SUBS HAVE BEEN CLEARED AND REMOVED FROM ALL LISTS.



I/O REQUESTS — CHAPTER 2

This Chapter consists of three sections:

1. I/O Request Types - I/O Request types will present three types of I/O requests and compare them.

2. I/O Overview - I/O Overview will present the information flow of an I/O call at a high level.

3. I/O Flow - I/O Flow will present a more detailed flow example for each type of I/O. The first request type will be presented in full. The remaining two types will have their differences presented. Please note: the flow charts presented cover the examples described, not the entire I/O system.

2.1 I/O REQUEST TYPES

There are three I/O request types:

User (Normal Operation) provides a straightforward I/O system call. A call is made. The program is I/O suspended while the I/O operation is performed. When the operation is completed the program is rescheduled.

User (Automatic Output Buffering) provides added features from the normal operation call. After the call is made, the buffer is transferred to SAM (System Available Memory). The program is rescheduled. The program does not get stuck in I/O suspend until the buffer limits are exceeded.

System (XSIO) calls provide I/O capability without all the overhead involved in an EXEC call (error checking) for modules of the OP system that need to perform I/O.

2.2 I/O OVERVIEW

In brief, the flow of control from an EXEC I/O call to the dispatcher is:

A. Process the interrupt. The EXEC call generates an MP violation. The interrupt processor saves the state of the machine.

B. Validate the EXEC call. The system determines if it is a valid I/O call, and establishes the type of call.

C. Validate and process the parameters. The call parameters are examined for validity, reformatted and saved. The caller is I/O suspended.

D. Set up for the driver. The various parameters are transferred to the EQT and control passed to the driver.

E. The driver initiates the data transfer and returns to the system, with information as to the result of the transfer (successful operation or error type).

F. The system cleans up after the drivers.

G. Control passes to the dispatcher to dispatch the next program (or redispatch the same program).

The driver handles the I/O. There are three returns from the driver continuation section: Completion return, Continuation return and Get/Give-up DCPC.

Completion Return. This return is taken when the driver has finished handling the I/O request; i.e., successful completion or I/O error.

Continuation Return. This return is taken when the driver has finished its current operation, but the entire request has not been completed.

Get/Give-up DCPC Return. This return is taken to get or to give up a Dual-Channel Port Controller (DCPC) channel.

2.3 I/O FLOW

To best describe the flow of an I/O operation, we will take the following sample program through the operating system path to process the I/O:

```
              JSB EXEC
              DEF RTN1
              DEF OUTPUT
              DEF DISC
              DEF BUFFER
              DEF BUFFERLENGTH
              DEF TRACK
              DEF SECTOR
RTN1          NOP
OUTPUT        DEC 2
DISC          DEC 2
BUFFER        BSS 128
BUFFERLENGTH  DEC 128
TRACK
SECTOR
```

The sample program is an unbuffered write to the disc. System conditions for the operation are assumed as:

1. Track 100 is allocated to the calling program.

2. No other requests are in progress on the LU.

3. The disc driver is DVM33, in select code 16B.

4. The system is not privileged.

5. The registers are:

   A = 1
   B = 37B
   X = 0
   Y = 10
   E = 0
   O = 1

The discussions that follows are referenced to Figures 2-1 through 2-8, flow charts of the process. Steps in the flow are coded on the charts, and are used as reference points to the discussion. The heading for each discussion contains labels to identify the location of the procedure in the source code. They define the label around which you can find the the program code that performs a particular operation or, in many cases, two labels that identify the beginning and end of the program code segment. The module in which the operation is performed is identified parenthetically as part of the heading for the segment description. For example, the heading

B2. $RQST - INDR (EXEC6)

identifies that portion of a flow chart coded with B2; the lines of code described in the discussion following are contained between labels $RQST and INDR in module EXEC6.

2.3.1 Process the Interrupt

(Refer to Figure 2-1.)

A1. $CIC (RTIOQ)

Test to see if the interrupt system is on or off. This is done with the SFS 0,C instruction. In either case, turn it off (the ,C does it). If it is off, bump $INT by one. Do this to indicate to the parity error routine (if it is a parity error interrupt) whether or not to reenable interrupts before returning from the parity error routine.

A2. $CIC - $DVC (RTIOQ)

The status of the Dynamic Mapping System is saved in $DMS. See the MEM status registers format in the HP 1000 Technical Reference Handbook (part no. 5955-0282).

A3. $CIC - $DVC (RTIOQ)

The interrupting select code is obtained for select code 4 (LIA 4, see the interrupt and I/O control summary in the HP 1000 F-Series Computer Technical Reference Handbook). The interrupting select code is saved in INTCD.

A4. $CIC - $DVC (RTIOQ)

If this was a violation on select code 5, you do not need to clear the flag. The flag will be used later. If the violation was not on SC 5, continue at step A6 to clear the device flag. EXEC calls generate an interrupt on SC 5.

A5. (RTIOQ)

Was the violation a parity error? If so, go to $PERR6. If not, continue at step A7. A parity error is indicated by bit 15 of the violation register being set to 1.

NOTE: The memory protect board (on select code 5) should not have its flag cleared. This would turn off the parity error interrupt capability and clear bit 15 of the violation register.

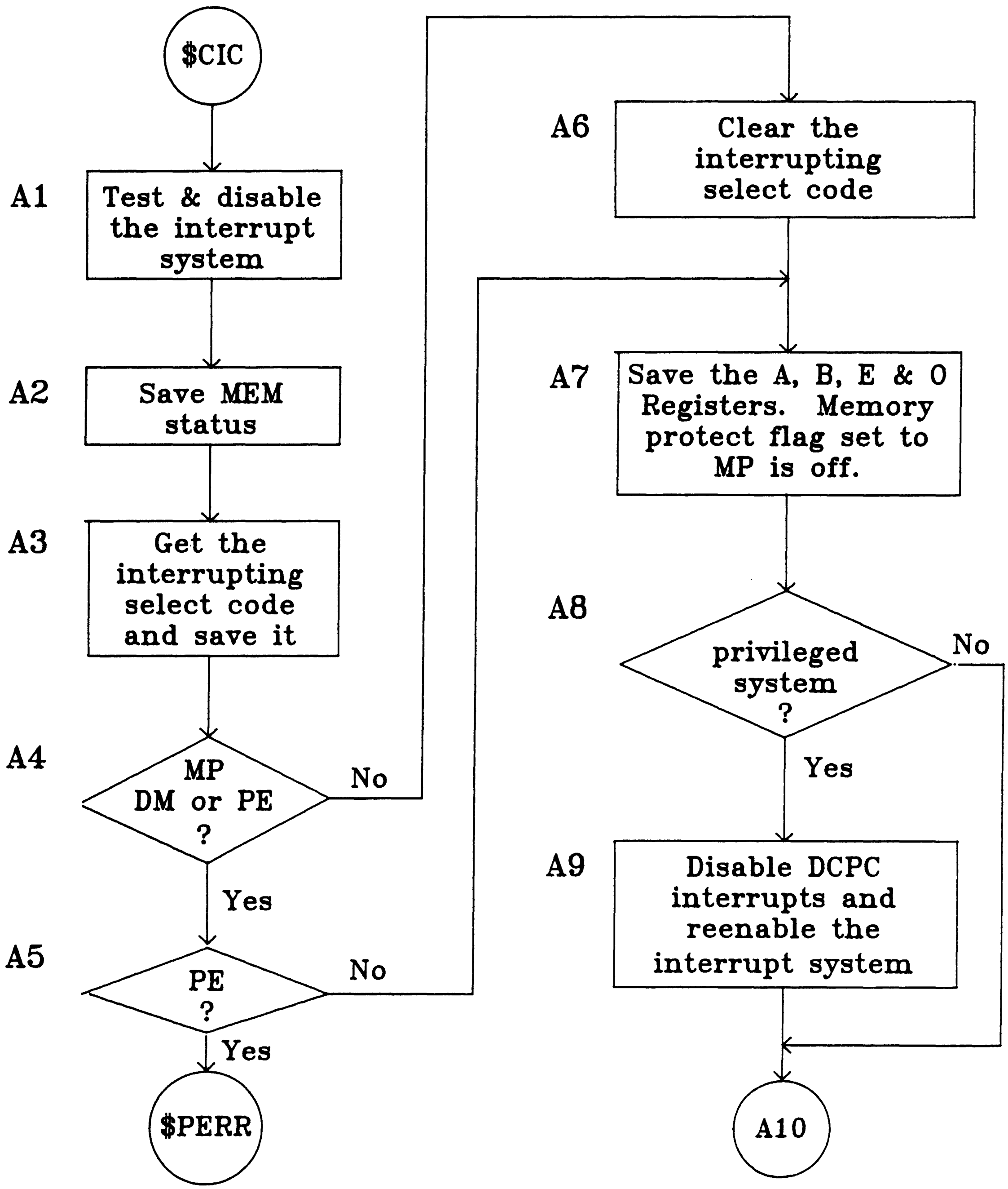


Figure 2-1. Processing the Interrupt

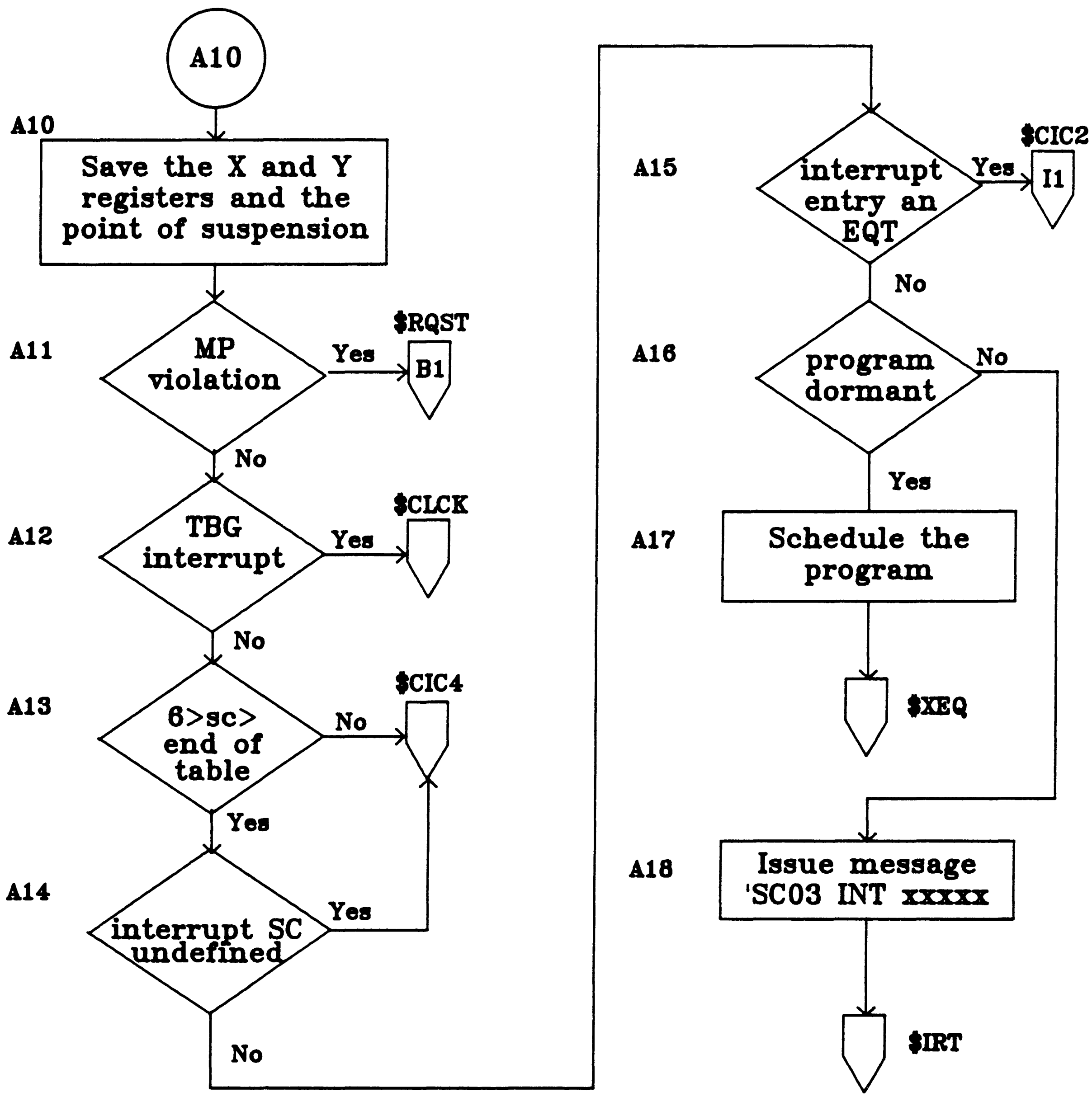


Figure 2-1. Processing the Interrupt (Continued)

The memory protect card will turn off its own flag when the interrupt system acknowledges the interrupt. There is a special flag that indicates a DMS violation. This flag can be checked with an SFS or SFC instruction. See the 12892A Memory Protect Theory of Operation in the HP 1000 Computers and Engineering and Reference Documentation (Part II, Section IV, part no. 92851-90001).

A6. $DVC (RTIOQ)

Build a CLF instruction to clear the flag on the interrupting device and execute it.

A7. CIC1 (RTIOQ)

Save the A, B, E, and O-Registers in the users ID segment (words 9, 10, 11, see the Program ID segment in the RTE-6/VM Programmers Reference Manual, part no. 92084-90005). Set the memory protect flag to 1. MPTFL to indicate that memory protect is now turned off.

A8. SW1 (RTIOQ)

Is this a privileged system? You can determine if it is privileged by checking DUMMY in the SYSCOM area. If DUMMY is zero, the system is not privileged. (Go to step A10, otherwise step A9.)

A9. SW1 - CIC.0 (RTCOM)

For a privileged system, you should set control on the privileged interrupt card. (The flag is already set from the last time.) Note that the first time through, the flag will not be set because there have not yet been any calls to $IRT. This does not matter because the first time is during loading of the second part of the operating system and at that time there should not be any active privileged operations.

A10. CIC.0 - $CJMP (RTCOM)

Save the X- and Y-Registers in the first two words of the program start page. Note that if the program starts in a page addressed as 42000B the program will be reloaded starting at 42012B.

A11. CIC.0 - $CJMP (RTIOQ)

Was this an MP violation? If so, go to RQST in EXEC6 to see if it is a valid EXEC call or not. Otherwise go to step A12.

2.3.2 Validate the EXEC Call

(Refer to Figure 2-2.)

B1. $RQST (EXEC6)

Get the address of the violation (LIB 5) and save it as the point of suspension in the user ID segment (word 8).

B2. $RQST - INDR (EXEC6)

Call $SNAP to count the number of interrupts on select code 5. MP, DM, EXEC, XLUEX, LIBR, LIBX, and calls to the memory-resident library are all counted.

B3. $RQST - INDR (EXEC6)

See if the call is a JSB EXEC. In this case it is, so go to R0 in EXEC6 (step B50) to continue processing. Compare the violating instruction with a JSB EXEC. If they match, it is an EXEC call.

B50. R0 (EXEC6)

The entry identifier indicates the call is an EXEC call ($CALL is positive) or an XLUEX call ($CALL is -1). In this case, it is an EXEC call.

B51. R0 - R1 (EXEC6)

The actual number of parameters is checked to see if it is less than 1 or greater than 8. The real # of parameters is the actual #+1. The extra parameter is the request code, not needed for performing the actual I/O transfer.

Return address - (address of JSB+1) -1 = Real # of parameters

B52. R1 (EXEC6)

Get the effective operand addresses. The addresses are stored in RQP1 through RQP9 in the system communication area on base page (1700B to 1710B). Be aware that if the A- or B-Register is specified as an address, it will be declared a request error.

B53. R101 - R3 (EXEC6)

See if the abort or no-suspend bits were set in the request code (bits 14 and 15). If they were not, continue at step B56. If they were, continue at step B54.

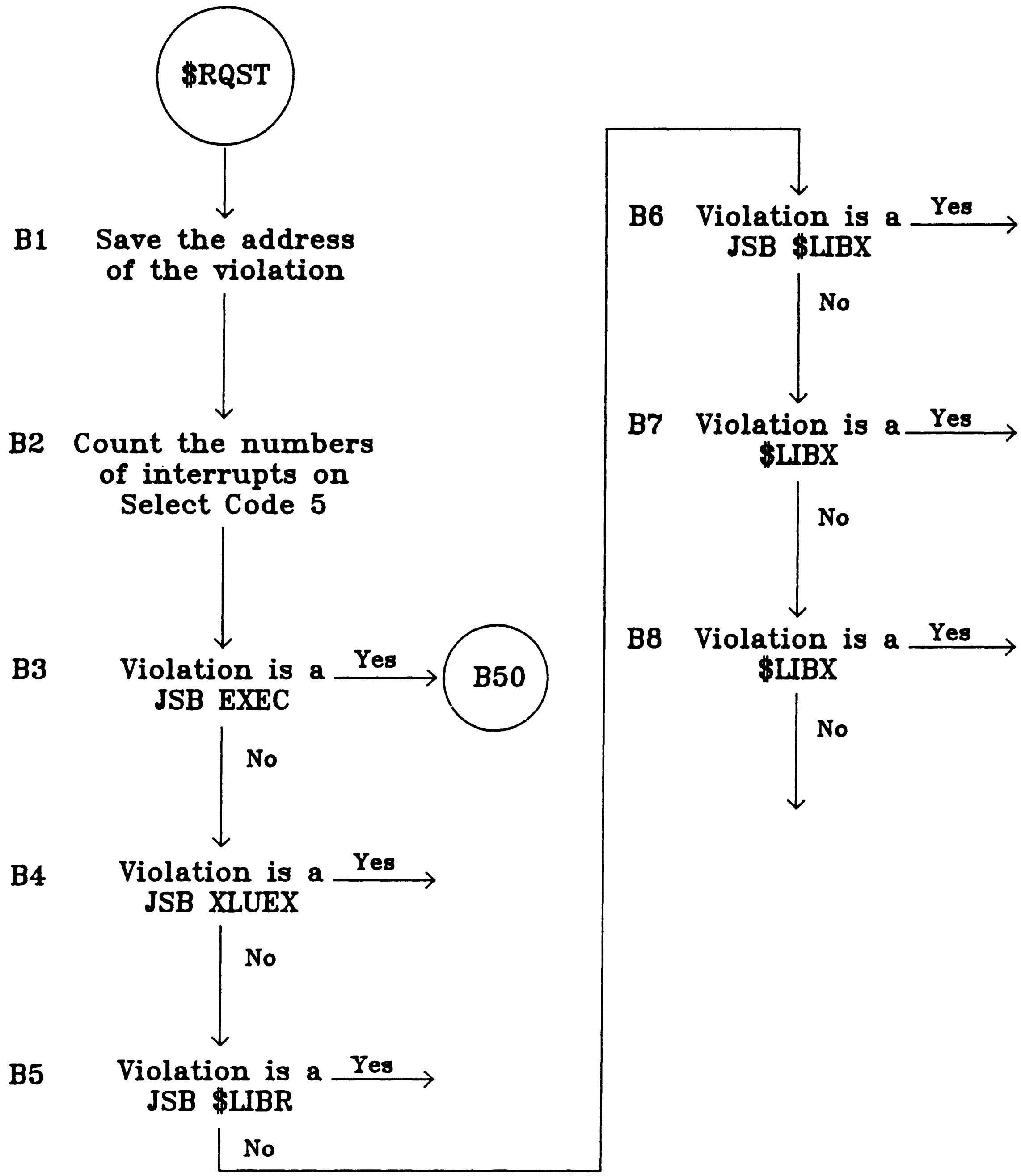


Figure 2-2. Validating the EXEC Call

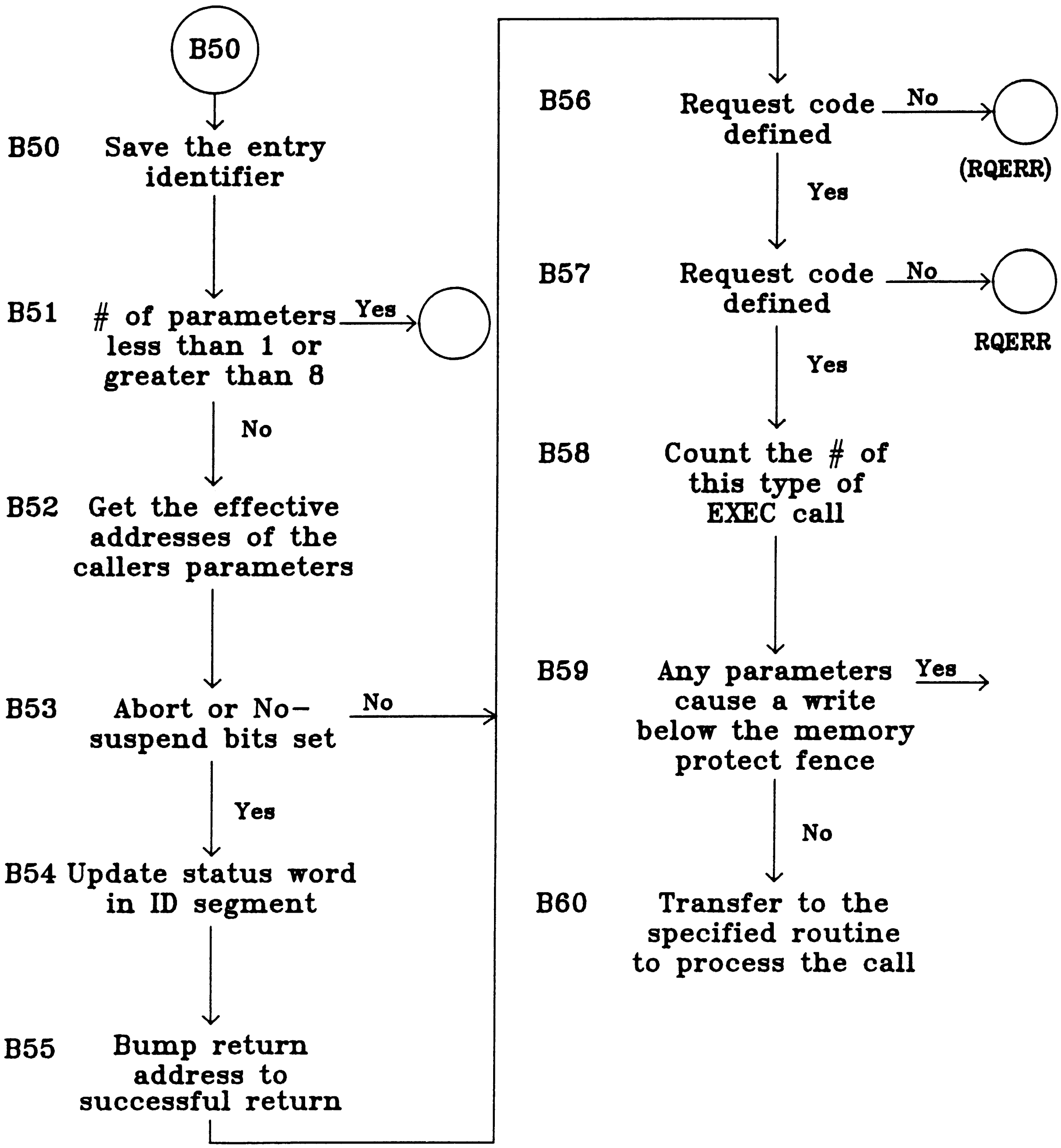


Figure 2-2. Validating the EXEC Call (Continued)

B56. R101 - R3 (EXEC6)

For the request code to be defined, it must not be less than 1 or greater than the # of entries in the table (TBL).

B57. R101 - R3 (EXEC6)

Because some of the request codes between 1 and the end of the table may not be legal, the address of the routine to process them is set to zero to indicate that it is illegal. If the request is legal (in the example it is), the address of the processing routine is saved in VECTR.

B58. R101 - R3 (EXEC6)

A call to $SNAP is made to count the number of this type of EXEC calls that have occurred.

B59. R3 (EXEC6)

A check is made of any parameters that would cause the system to write into a user buffer. The parameters are checked to be sure they do not point below the memory protect fence.

To determine which parameters to check, a table (NAMTB) contains parameter check bits, one for each possible parameter. If the bit is zero, the parameter is checked. If the bit is a one, the parameter is not checked. If any of the checked parameters fail to pass an RQ00 error is issued and we go to $XEQ to prepare for the next user.

B60. R4 - ERQ00 (EXEC6)

It is now time to transfer to the routine to handle the WRITE operation. The transfer address may be found in VECTR. (Transfer to $IORQ in RTIOQ.)

2.3.3 Validate and Process the Parameters

(Refer to Figure 2-3.)

C1. $IORQ (RTIOQ)

You must have at least one parameter (excluding the request code) and RQCNT must not be zero.

C2. GTPAR (RTIOQ)

The call parameters, an eight-word array labeled PARM2 - PARM9, are moved to a local buffer for ease of access.

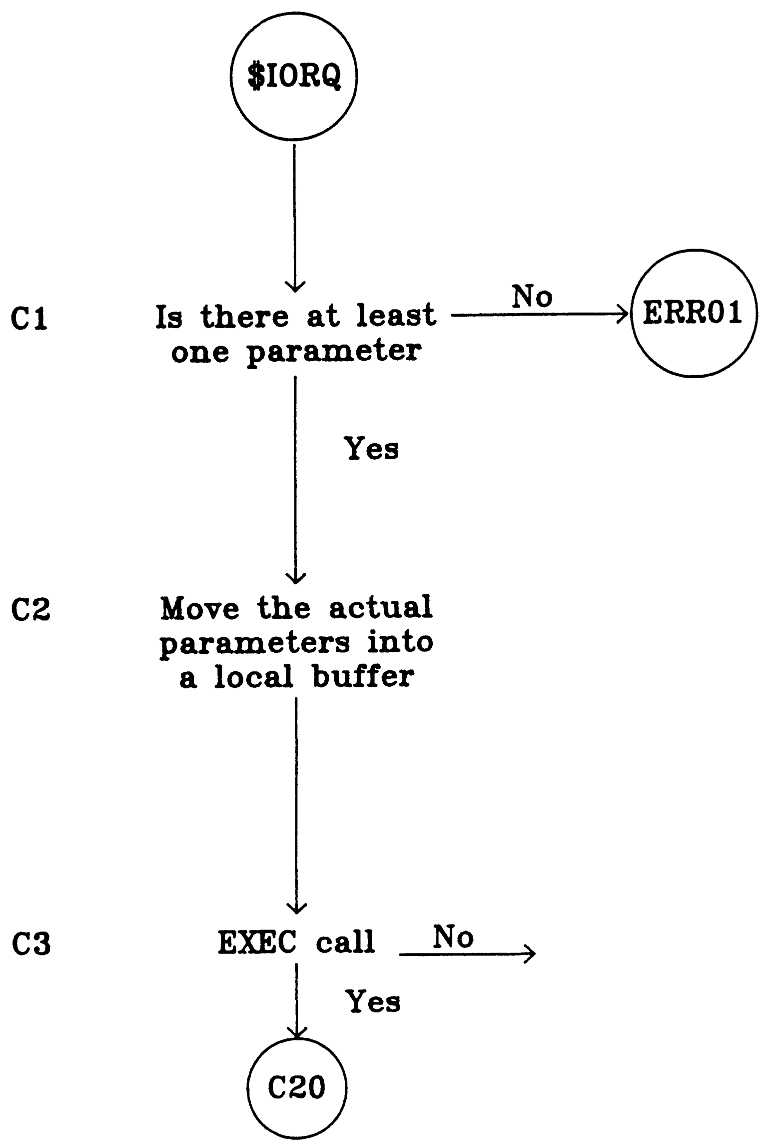


Figure 2-3. Validating and Processing the Parameters

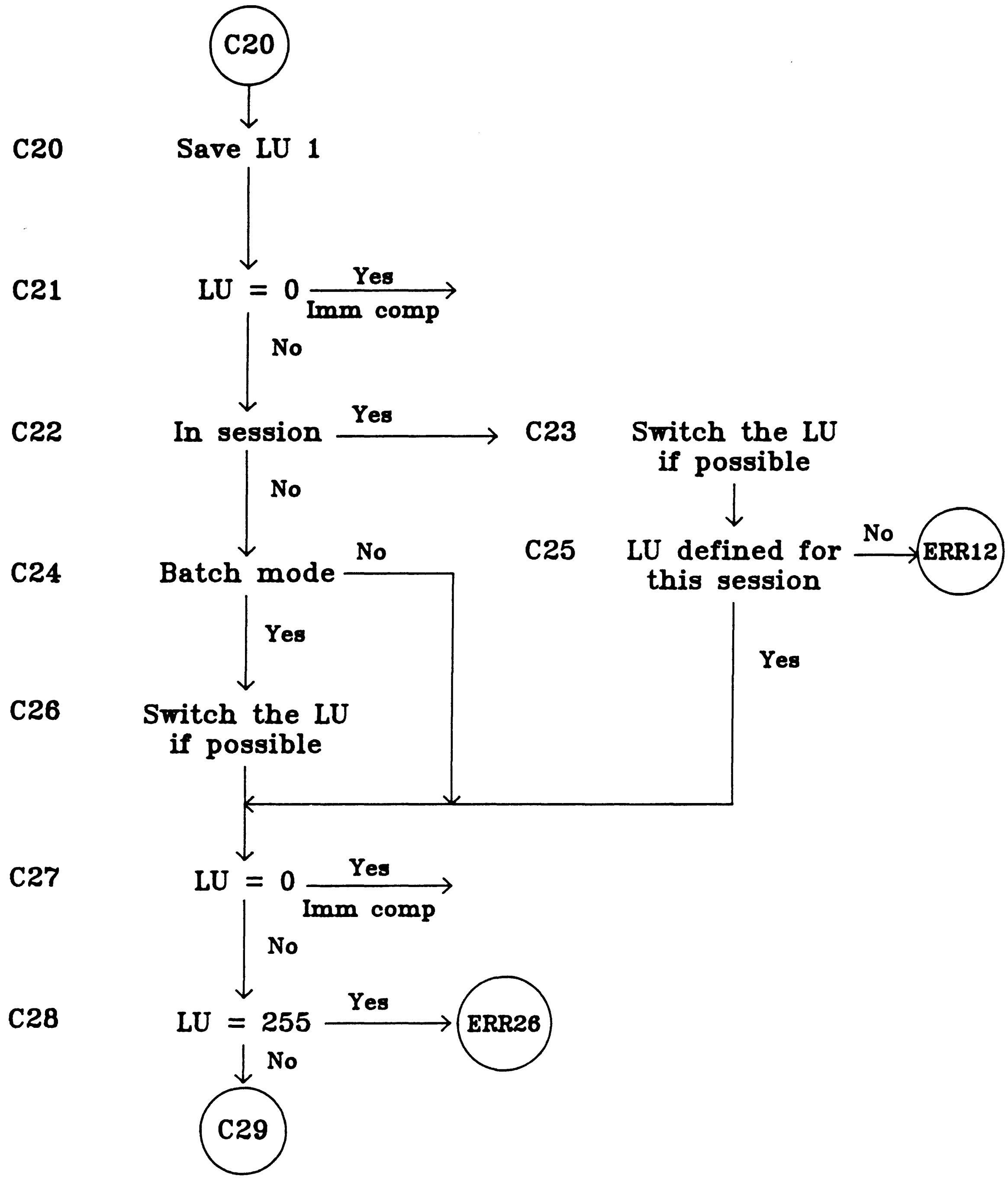


Figure 2-3. Validating and Processing the Parameters (Continued)

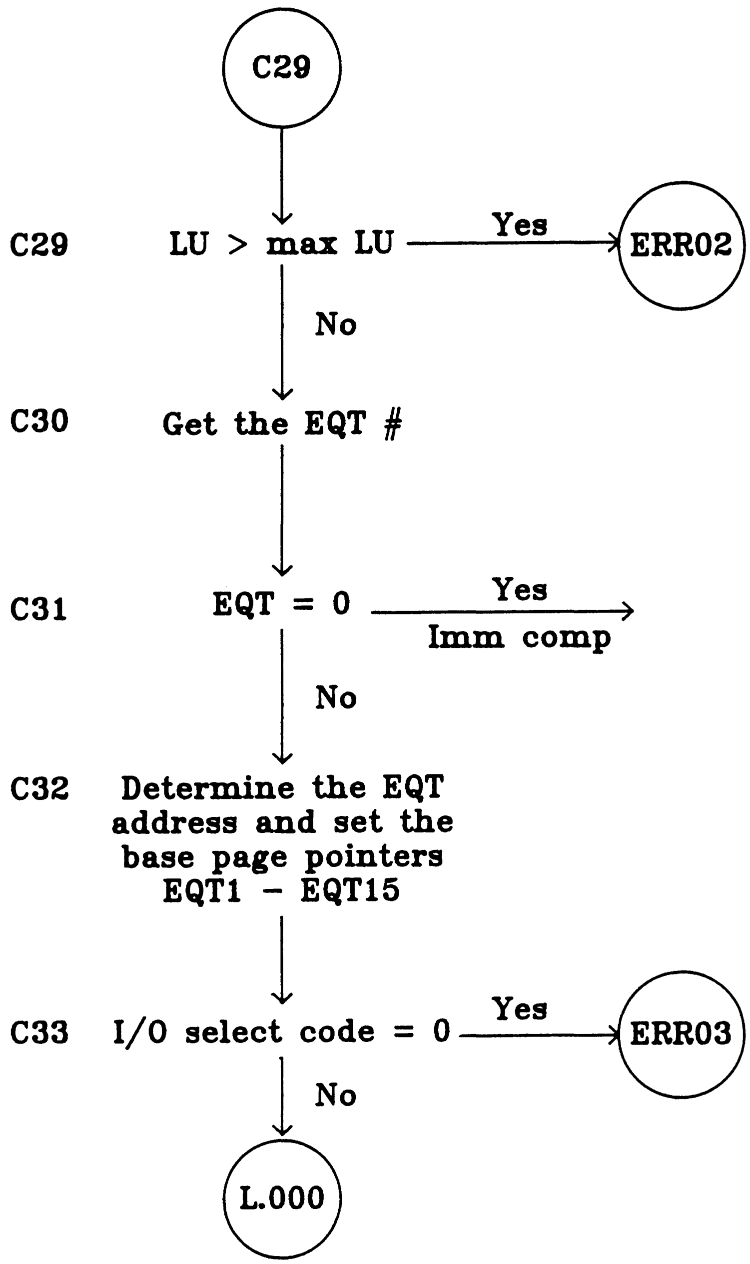


Figure 2-3. Validating and Processing the Parameters (Continued)

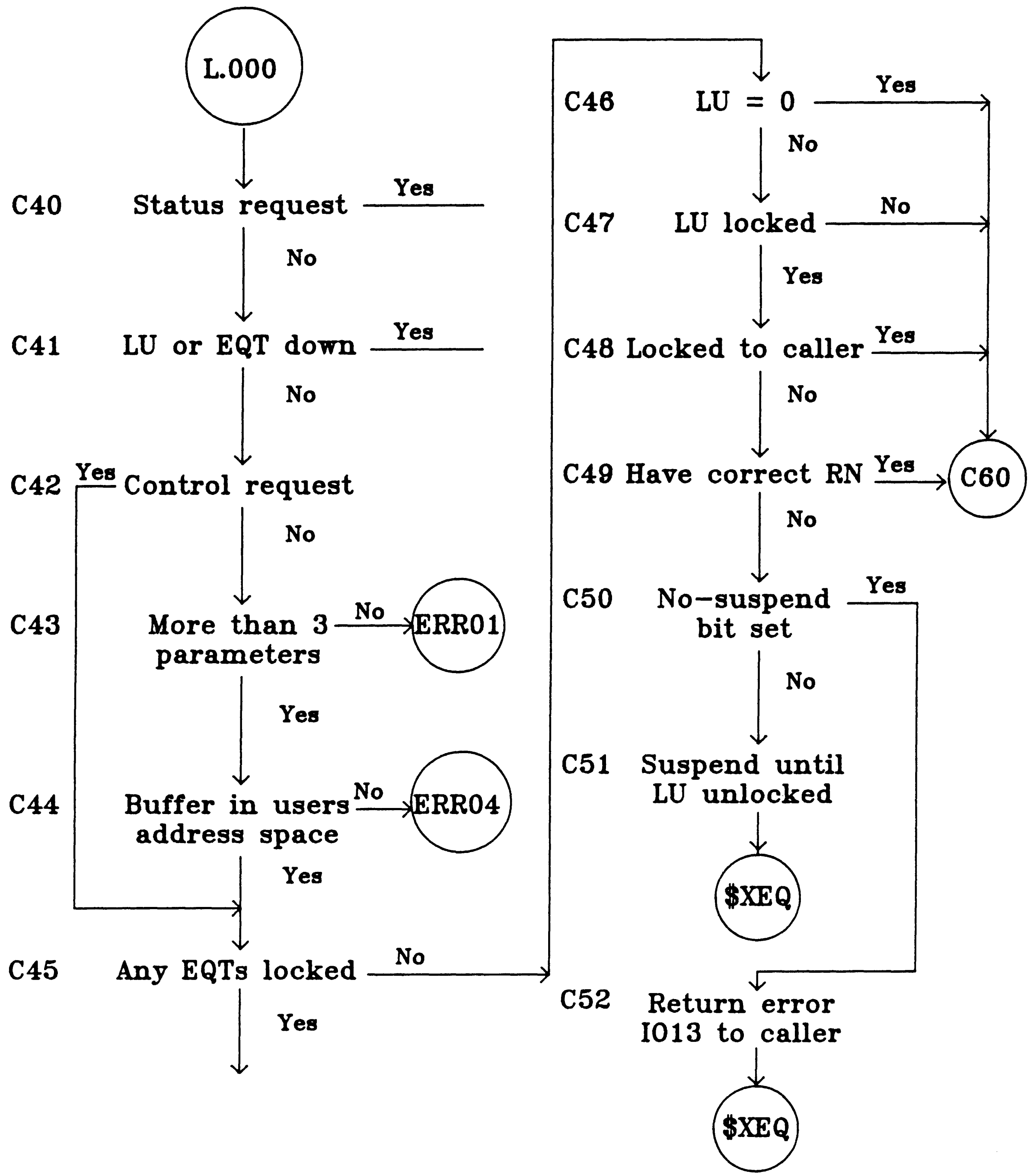


Figure 2-3. Validating and Processing the Parameters (Continued)

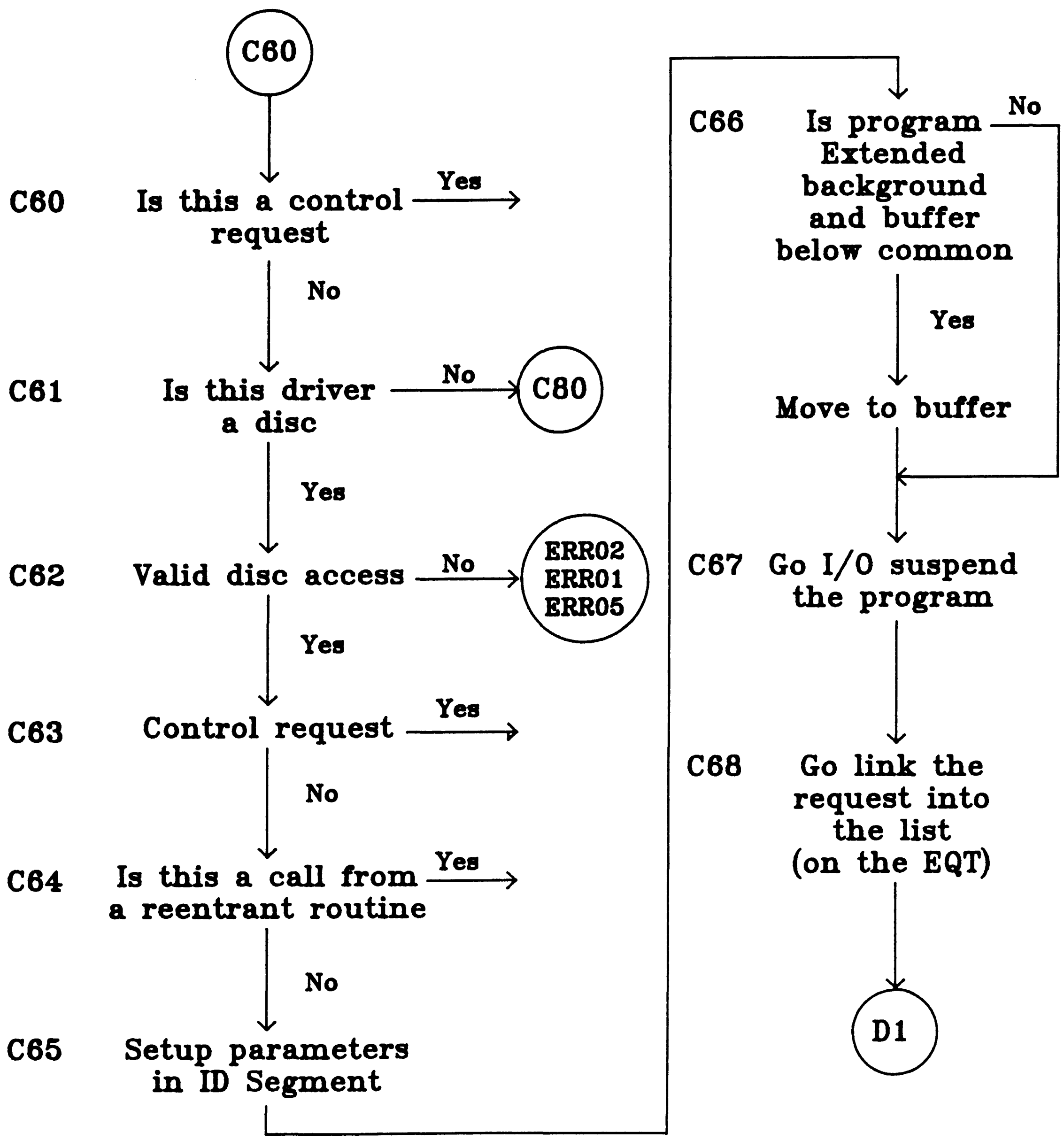


Figure 2-3. Validating and Processing the Parameters (Continued)

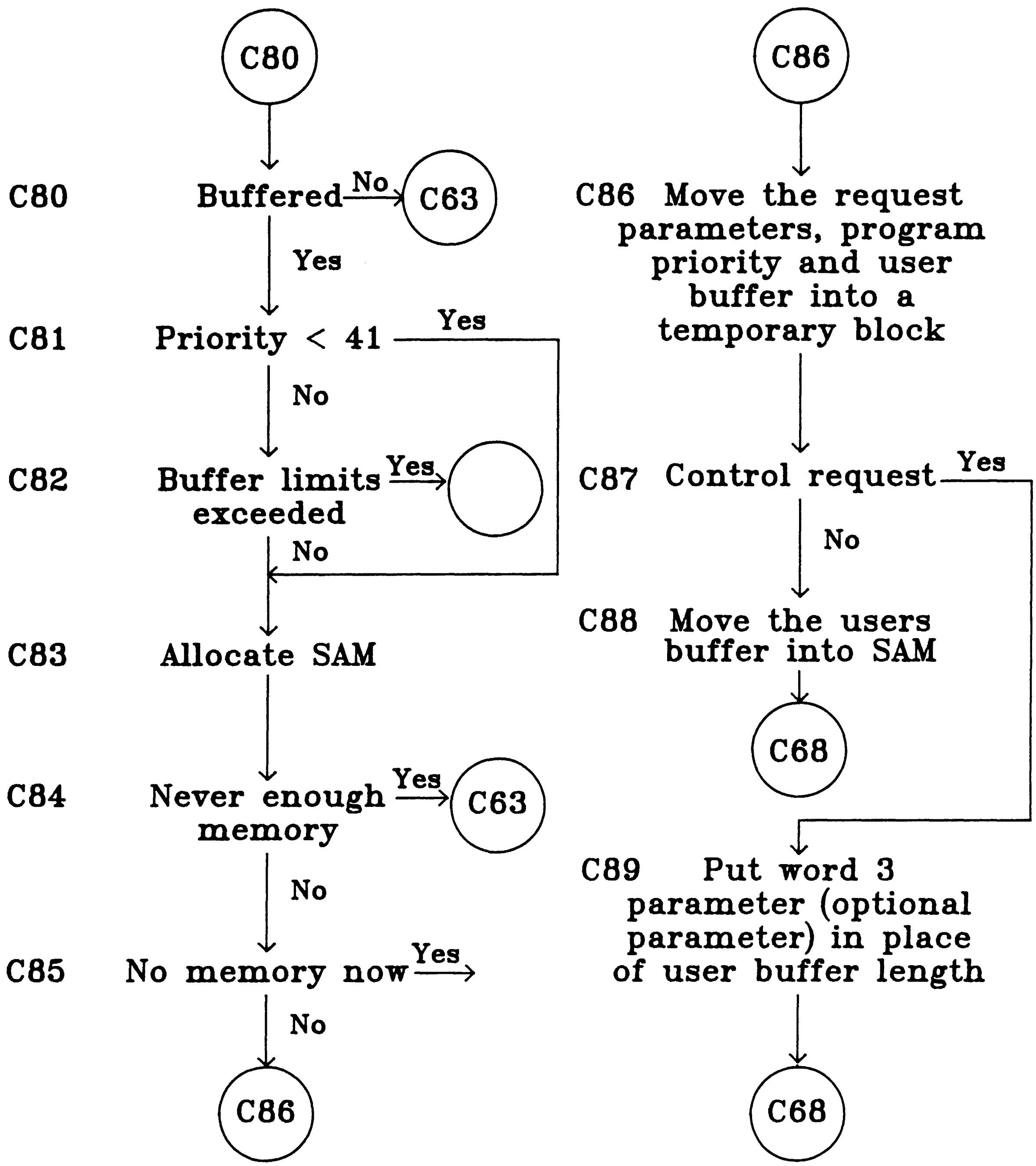


Figure 2-3. Validating and Processing the Parameters (Continued)

C3. GTPAR - OLD (RTIOQ)

See if this is an EXEC call or an XLUEX call. $CALL is negative for a XLUEX call and positive for an EXEC call.

C20. OLD (RTIOQ)

Save the LU 1 (session stores LU 1 in the SST) in REQLU for later use.

C21. OLD - NSESS (RTIOQ)

Is it zero? If it is (in the example it is not), perform an immediate completion.

C22. OLD - NSESS (RTIOQ)

Is the calling program in session? Check the session word in the ID segment (SCB or word 32). If positive and non-zero the program is in session.

C23. OLD - NSESS (RTIOQ)

Call $SWCK (RTIOQ) to convert the session LU to the system LU. The SST is searched by $SWCK.

C25. OLD - NSESS (RTIOQ)

If the session LU is not defined for this user (not in SST), go to ERR12 (RTIOQ) to issue an IO12 error.

C27. L.0.1 - L.000 (RTIOQ)

If the LU is 0, go to L.00x for an immediate completion.

C28. L.0.1 - L.000 (RTIOQ)

See if the LU number is 255. If it is, issue an IO26 error (I/O request made to a spool that has been terminated by the GASP MS command).

C29. L.0.1 - L.000 (RTIOQ)

Is the specified LU greater than the maximum LU? The maximum LU number may be found in the system communications area (1653B). If it is, go to ERR02 to issue an IO02 error.

C30. L.0.1 - L.000 (RTIOQ)

Get the EQT number by adding the starting address of the Device Reference Table (DRT) to LU-1. The lower 8 bits of this entry contain the EQT number. (See the DRT in the RTE-6/VM Programmers Reference Manual, part no. 92084-90005.)

C31. L.0.1 - L.000 (RTIOQ)

Go to L.00x for an immediate completion if the EQT number is 0. In the example it is not.

C32. L.0.1 - L.000 (RTIOQ)

Call $CVEQ in RTCOM to convert the EQT number to an address and then set the Base Page pointers (EQT1 - EQT15) to point to the EQT. If the first pointer is already set up, it is assumed the others are also.

C33. L.0.1 - L.000 (RTIOQ)

If the select code specified by the EQT is zero, go to ERR03 and issue an I001 error (illegal EQT referenced by LU in I/O call).

C40. L.000 (RTIOQ)

Determine if this is a status request (EXEC 13). Get the request type from RQP1 and check the low four bits. In the example, it is not (save it in RQPX for later).

C41. L.000 - L.01 (RTIOQ)

Determine if the LU or EQT is down (call $STDV in RTIOQ). To see if the EQT is down check bit 14 and 15 of word 3 (14 should be set and 15 should be clear). To see if the LU is down, check bit 15, word 2 of the DRT (The bit is set if the LU is down).

C42. L.000 - L.01 (RTIOQ)

If this is a control request, go to L.01 to handle it. In the example, it is not. A control request is a type 3 request, the example is a type 2 request.

C43. L.000 - L.01 (RTIOQ)

A check is made for at least three parameters. You must have at least three: LU, buffered address, and buffer length). Any thing less and you cannot perform a READ or WRITE (in our case a WRITE).

C44. L.000 - L.01 (RTIOQ)

A call is made to $BFCK in RTIOQ to see if the buffer is legal. To be a legal buffer it must not go past the end of the 32k address space (error). If the buffer is in common, then the whole buffer must be in common. If the buffer is not in common, the last page used by the buffer is checked for write protection. If the page is write protected, it means the memory is not available to the program. Go to ERR04 to issue an error message.

C45. L.01 (RTIOQ)

A check is made of the EQT locking table to see if there are any entries. In the example there are none, so we continue.

C46. L.019 - L.01A (RTIOQ)

Another check for LU 0. If it is LU 0, skip the LU-locked check. In the example, it is not LU 0.

C47. L.019 - L.01A (RTIOQ)

Determine if the LU is locked. To do this, pick up the lock byte of the LU in the third part of the DRT. If zero, the LU is not locked (as in the example).

C60. L.01A (RTIOQ)

If this is a control request, there is no need for further analysis of the call; go to the auto buffering check.

C61. L.01A - L.01B (RTIOQ)

If this is a disc driver, there is additional checking to be performed. Get EQT word 5 and mask it with a 36000B, then compare it to 14000B. If it matches, it is a disc (30, 31, 32, 33).

C62. L.01A - L.010

For a valid disc access, you must meet the following requirements:

1. If a class request - ERR02.

2. If less than five parameters - ERR01.

3. If LU 2 or 3:

   a. If starting sector less than 0 or greater than the track size - ERR05.

   b. Last track of user request >last track on LU - ERR05.

c. Input? User can access any track, so skip further tests (go to L.10).

d. Caller has legal access - owns the track (allocated to caller). Global access allowed

C63. L.10 (RTIOQ)

See if this is a control request. (Request code is three for a control request.) In the example, the request code is two - a write request.

C64. L.10 - L.102 (RTIOQ)

Check the RENT bit in the ID segment (word 20, bit 10) to see if the caller is re-entrant. (The example is not re-entrant.)

C65.

There are five temporary words in the ID segment. The call parameters are stored in the ID segment after they have been processed by the system. Label them XTEMP, XTEMP+1, XTEMP+2, XTEMP+3, XTEMP+4 (words 1-5).

The control word is built as follows:

```
XTEMP   +------------------------------------------------------+
        |  T   * S4* X * S5* S FUN  * SUB CHAN *  REQUEST CODE |
        | 15/14*13 *12 *11 * 10----6* 5------2 *  1/0          |
        +------------------------------------------------------+
```

XTEMP+1 Contains the buffer address of buffer 1 for a read or write operation or the optional parameter for a control operation (contains the parameter, not the address of the parameter).

XTEMP+2 Contains the buffer length of buffer 1.

XTEMP+3 Optional parameter 1 or buffer address if double-buffered call.

XTEMP+4 Optional parameter 2 or the buffer length of buffer 2.

If some of the parameters are not specified (i.e., the optional parameters), their contents are undefined.

C66. L.101 - L.13 (RTIOQ)

Call $EXB6 in RTEMA to process type 6 programs. If type 6 and the buffer is below the start of common (in Table Area I or in the driver partition) remap the buffer after common and update the buffer address in the user ID segment.

C67. L.101 - L.13 (RTIOQ)

Call $LIST to have the program I/O suspended.

C68. L.13 - L.135 (RTIOQ)

The request is linked with any existing requests. If priority is 0-40, it is linked by its priority and in a FIFO list. If the priority is 41-32767, it is appended to the end of the list. In the example, the ID segment is now linked off at EQT word 1.

2.3.4 Buffered I/O

(Refer to Figure 2-3.)

C80. L.027 - L.028 (RTIOQ)

For the request to be buffered it must not be an input and the EQT word 4 bit 14 (buffered bit) must be set. Further, the UB bit (Bit 14) in the control word must not be set and it must not be a dynamic status request.

C81. L.03 - L.031 (RTIOQ)

If the program`s priority is less than 41, do not perform the buffer check.

C82. L.03 - L.031 (RTIOQ)

If the buffer limits were exceeded, check to see if the user should be suspended.

C83. L.031 (RTIOQ)

Call $ALC (in $ALC) to allocate SAM to buffer the users data.

C84. L.04 - L.040 (RTIOQ)

If there will never be enough memory to buffer the users data, go to L.10 (C63) and proceed as an unbuffered call.

C85. L.04 - L.040 (RTIOQ)

If there is not enough memory now, suspend the user in a memory wait.

C86. L.06 - L.08

Now is the time to build the control information in the block of SAM and move the user`s data there.

The format of buffered request in SAM is:

```
WORD          CONTENTS

 1    < LINKAGE WORD           >
 2    <T, CONTROL INFO, CODE   >
 3    <PRIORITY OF REQUESTOR   > =0 IF SYSTEM
 4    <TOTAL BLOCK LENGTH WORDS>
 5    <USER BUFFER LENGTH      >
 6    <TRACK OPTION WORD       >
 7    <SECTOR OPTION WORD      >
 8    <WORD 1 OF USER BUFFER   >
 .       .        .        .
 .       .        .        .
 N+7  <WORD N OF USER BUFFER   >
```

C87. L.061 (RTIOQ)

If this is a control request put the optional buffer in place of the user buffer length. In our case it is not.

C88. L.065 (RTIOQ)

Move the users data to the SAM buffer.

2.3.5 Set Up for Driver

(Refer to Figure 2-4.)

D1. $DRVR (RTCOM)

Check the availability bits (bits 14, 15 of EQT 5) to see if the driver is waiting for DCPC. If it is, go to D16 to handle it. If its not (as in the example), continue.

D2. $DRVR - $DVR0 (RTCOM)

Check the availability bits to see if the device is down (bit 14 set) or busy (bit 15 set) If either case is true, go to the dispatcher ($XEQ) and dispatch the next program. In the example neither is true, go we continue on.

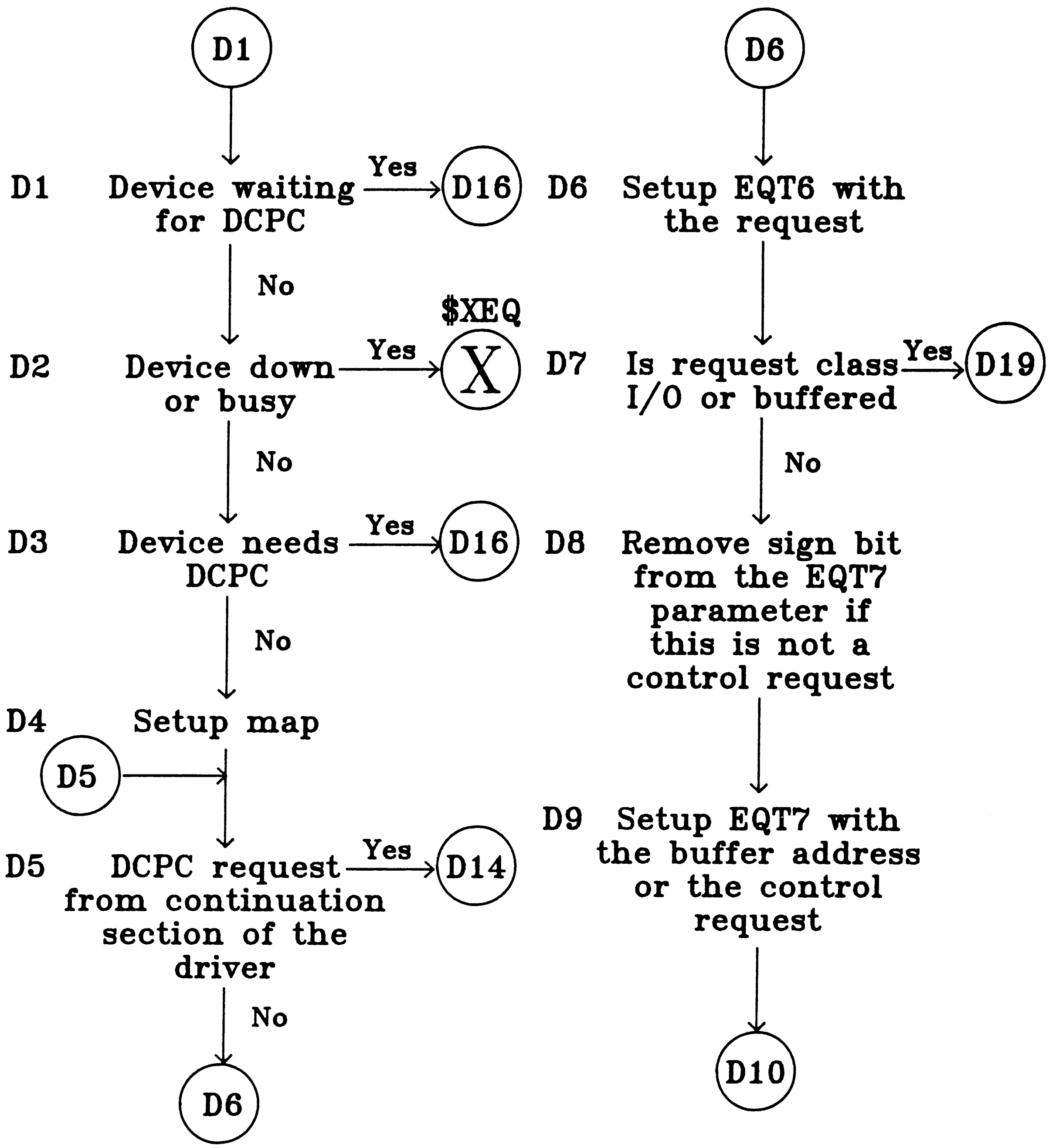


Figure 2-4. Driver Setup

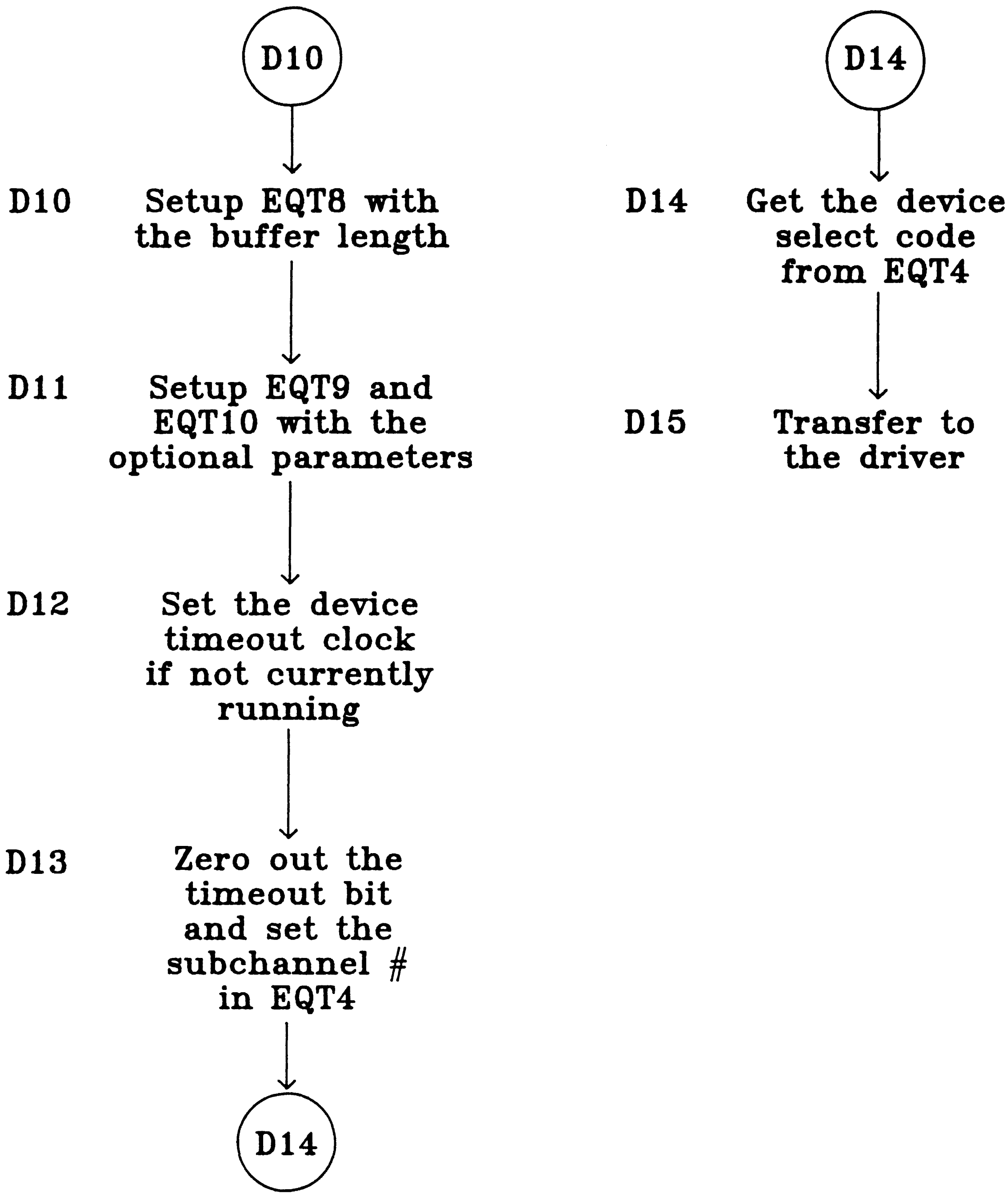


Figure 2-4. Driver Setup (Continued)

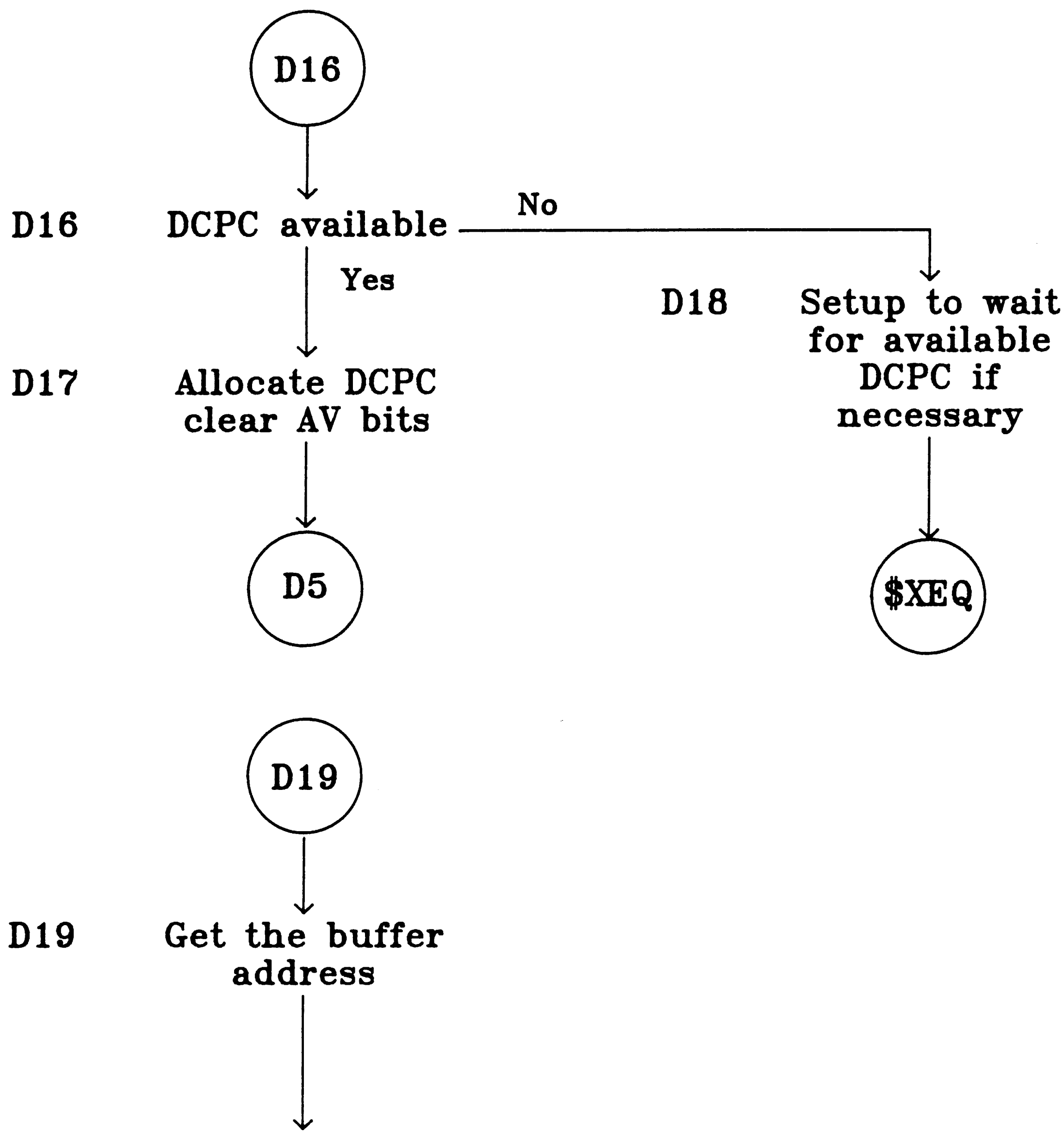


Figure 2-4. Driver Setup (Continued)

D3. $DRVR - $DVR0 (RTCOM)

Check the D bit (bit 15) in EQT 4 to see if this driver needs DCPC. In the example it does.

D16. $DVR0 (RTCOM)

The DCPC availability flag DMACF is checked to see if anyone is waiting for DCPC. If someone is waiting (DCACF <> 0), stack the request. In the example, a channel is available.

D17. DVR00 (RTCOM)

Check the interrupt table entries for select codes 6 and 7. If an entry is 0, the channel may be allocated. Set the system com area (CHAN (word 1673)) to the channel being allocated.

Set the EQT 1 address in the interrupt table. If the driver was waiting for DCPC (EQT 5 AV field bits 14 and 15 set), clear the bits and subtract 1 from the "Waiting for DCPC" flag (DMACF).

Call DRVMP (in RTCOM) to set up the map for the driver, then copy the map to the DCPC port.

D5. DV02C (RTCOM)

Check to see if the DCPC request was made from the continuation section of the driver (bit 15 set in EQT 3). If it was, the EQT is already set up. In the example, DCPC is being assigned for the initiation call.

D6. DV02C - DRV2 (RTCOM)

EQT word 6 is built and installed. The control word is built from the first temporary word in the ID segment.

D7. DRV2 (RTCOM)

Check the request type (bit 14 of EQT 6). If it is set, this is either a Class I/O request or a buffered request. If it is clear (as in the example), it is a standard call or a system call.

D8. DRV2 - DRV3 (RTCOM)

Remove the sign bit from the buffer address to be sure it is not treated as an indirect address.

D9. DRV3 (RTCOM)

The buffer address is placed in EQT 7. See the Equipment Table Entry Format in the RTE-6/VM Programmers Reference Manual. This is obtained from the second temporary word in the ID segment.

D10. DRV3 - DRV4 (RTCOM)

Set the buffer length in EQT 8. This is from temporary word 3 of the ID segment.

D11. DRV3 - DRV4 (RTCOM)

Fill EQT words 9 and 10 with information from ID segment temporary words 4 and 5. All five ID segment words are transferred to the EQT even if there is no valid data in them.

D12. DRV3 - DRV4 (RTCOM)

The device timeout clock is set if it is not now in operation (non-zero). To set it, EQT word 14 is copied to EQT word 15.

D13. DRV3 - DRV4 (RTCOM)

Clear the timeout bit (device has not timed out) and place the subchannel number (lower 5 bits) in EQT 4.

D14. DRV4 (RTCOM)

The select code to be used by the device is obtained from bits 0-5 of EQT 4 and placed in the A-Register.

D15. DRV4 - INUS (RTCOM)

Transfer control to the driver in the user map. Transfer control to the initiation address contained in EQT word 4.

2.3.6 EQT Words Set Up by RTE-6/VM

The following words are set up by the operating system before calling the driver:

```
WORD                            CONTENTS

      +---------------------------------------------------------------+
      | 15  14  13  12  11  10   9   8   7   6   5   4   3  2  1  0 |
      +---------------------------------------------------------------+
  1   |         I/O REQUEST LIST POINTER                              |
      |---------------------------------------------------------------|
  2   | R |     DRIVER INITIATION SECTION ADDRESS                     |
      |---------------------------------------------------------------|
  3   | R |     DRIVER INITIATION SECTION ADDRESS                     |
      |---------------------------------------------------------------|
  4   | D | B | P | S | T | SUBCH # (LOW 5 BITS)| I/O SELECT CODE    |
      |---------------------------------------------------------------|
  5   |   AV  |      EQUIP TYPE CODE      |          STATUS           |
      |---------------------------------------------------------------|
  6   |         CONWD (CURRENT I/O REQUEST WORD)                      |
      |---------------------------------------------------------------|
  7   | REQ. BUFFER ADDR. OR CONT. REQ. OPTIONAL PARAM     (IBUF)     |
      |---------------------------------------------------------------|
  8   |         REQUEST BUFFER LENGTH                      (IBUFL)    |
      |---------------------------------------------------------------|
  9   |         TEMPORARY STORAGE FOR OPTIONAL PARAM       (JBUF)     |
      |---------------------------------------------------------------|
 10   |         TEMPORARY STORAGE FOR OPTIONAL PARAM       (JBUFL)    |
      |---------------------------------------------------------------|
 11   |         TEMPORARY STORAGE FOR DRIVER                          |
      |---------------------------------------------------------------|
 12   |         TEMPORARY STORAGE FOR DRIVER (EQT EXT SIZE)           |
      |---------------------------------------------------------------|
 13   |         TEMPORARY STORAGE FOR DRIVER (EQT EXT START ADDR)     |
      |---------------------------------------------------------------|
 14   |         DEVICE TIMEOUT RESET VALUE                            |
      |---------------------------------------------------------------|
 15   |         DEVICE TIMEOUT CLOCK                                  |
      +---------------------------------------------------------------+
```

Bits 0 and 1 of CONWD, EQT word 6, specify the kind of call:

    01 = Read
    10 = Write
    11 = Control

Bit 2 is the most significant bit of the subchannel number contained in bits 2 through 5.

Bits 6 through 10 contain the function code.

Bits 14 and 15 specify the call type:

    00 = Standard
    01 = Buffered
    11 = Class

Note that the EQT pointers on the base page are set up to point to the program EQT and the A-Register contains the select code in bits 0-5.

2.3.7 Driver Rules for Initiation Return

Upon return from the driver initiation section, the status of the operation is returned in the A-Register:

    0   = Operation initiated; can dispatch next program.
    1   = Read or write illegal; program aborted (IO07)
    2   = Control request illegal; program aborted (IO07)
    3   = Equipment not ready or program I/O suspended (IONR)
    4   = Immediate completion; dispatch next program.
    5   = DCPC channel required; go to initiation section again.
    6   = DCPC channel assigned, driver is returning it;
           DCPC treated as 0.
    7-99 = Program making I/O request is aborted; I/O error number
           and message are displayed on console:

              7-59 - HP Reserved
             60-99 - User drivers

2.3.8 Clearing After the Device

(Refer to Figure 2-5.)

F1. DRVRT (RTCOM)

If the user map was changed, restore it. The flag DVMPS will be zero if the map was not changed. It must be reset if you were remapped (EB with buffer below common) or you are not the currently executing program.

F2. DRVRT (RTCOM)

In the example, we have a successful initiation. Continue at step F5.

F5. DRV00 (RTCOM)

Set the device busy by setting bit 15 of EQT 5 (the AV bits). Do not set the busy bit if no other requests are queued up.

F6. Exit from $DRVR back to RTTOQ.

2.3.9 Processing Interrupts

(Refer to Figure 2-6)

H1. $CIC (RTIOQ)

Test to see if the interrupt system is on or off. This is done with the SFS 0,C instruction. In either case, turn it off (the ,C does it). If it is off, bump $INT by one. Do this to indicate to the parity error routine (if it is a parity error interrupt) whether or not to reenable interrupts before returning from the parity error routine.

H2. $CIC - $DVC (RTIOQ)

The status of the Dynamic Mapping System is saved in $DMS. See the MEM status registers format in the HP 1000 Technical Reference Handbook (part no. 5955-0282).

H3. $CIC - $DVC (RTIOQ)

The interrupting select code is obtained for select code 4 (LIA 4, see the interrupt and I/O control summary in the HP 1000 F-Series Computer Technical Reference Handbook. The interrupting select code is saved in INTCD.

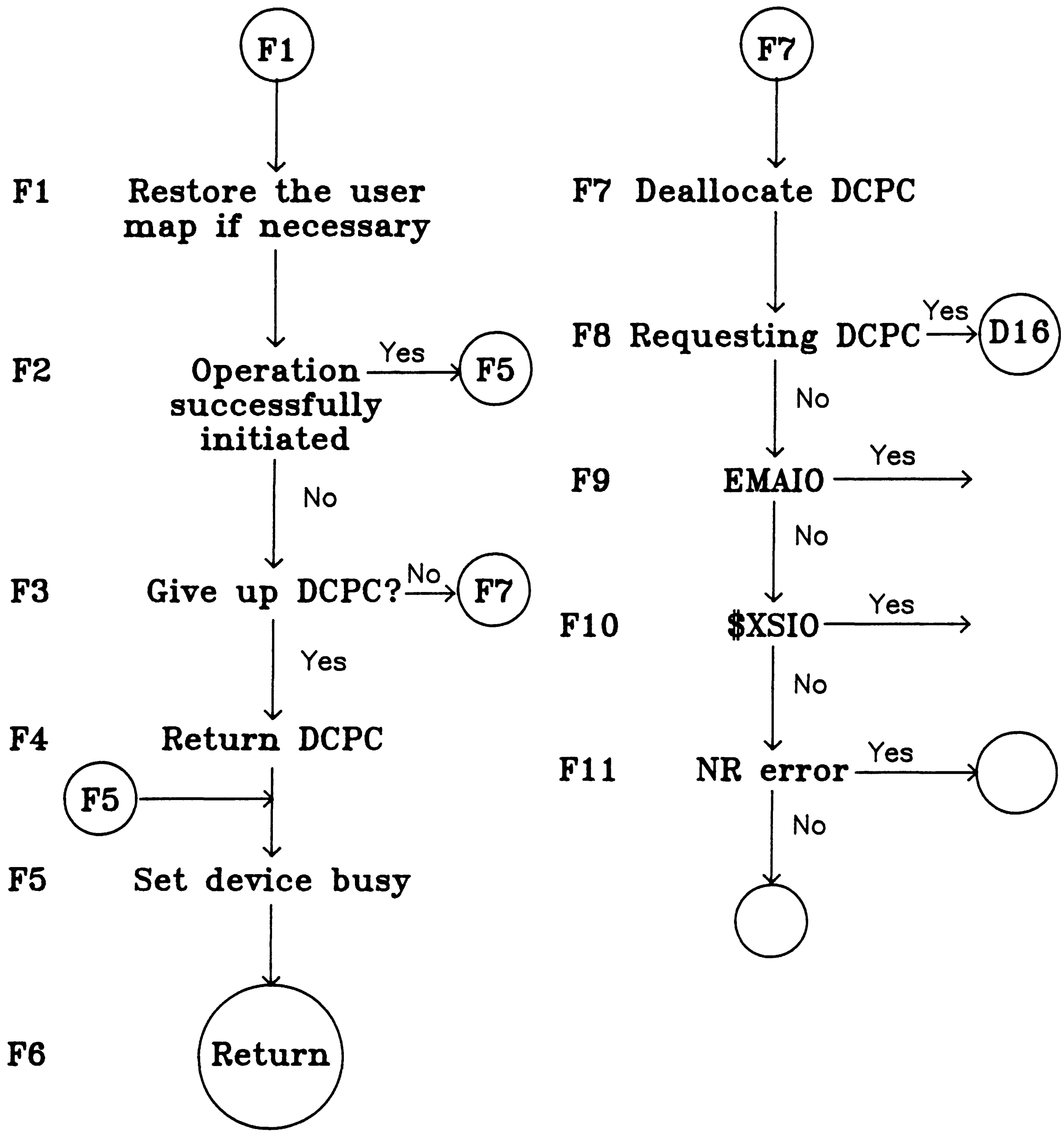


Figure 2-5. Clearing After the Driver

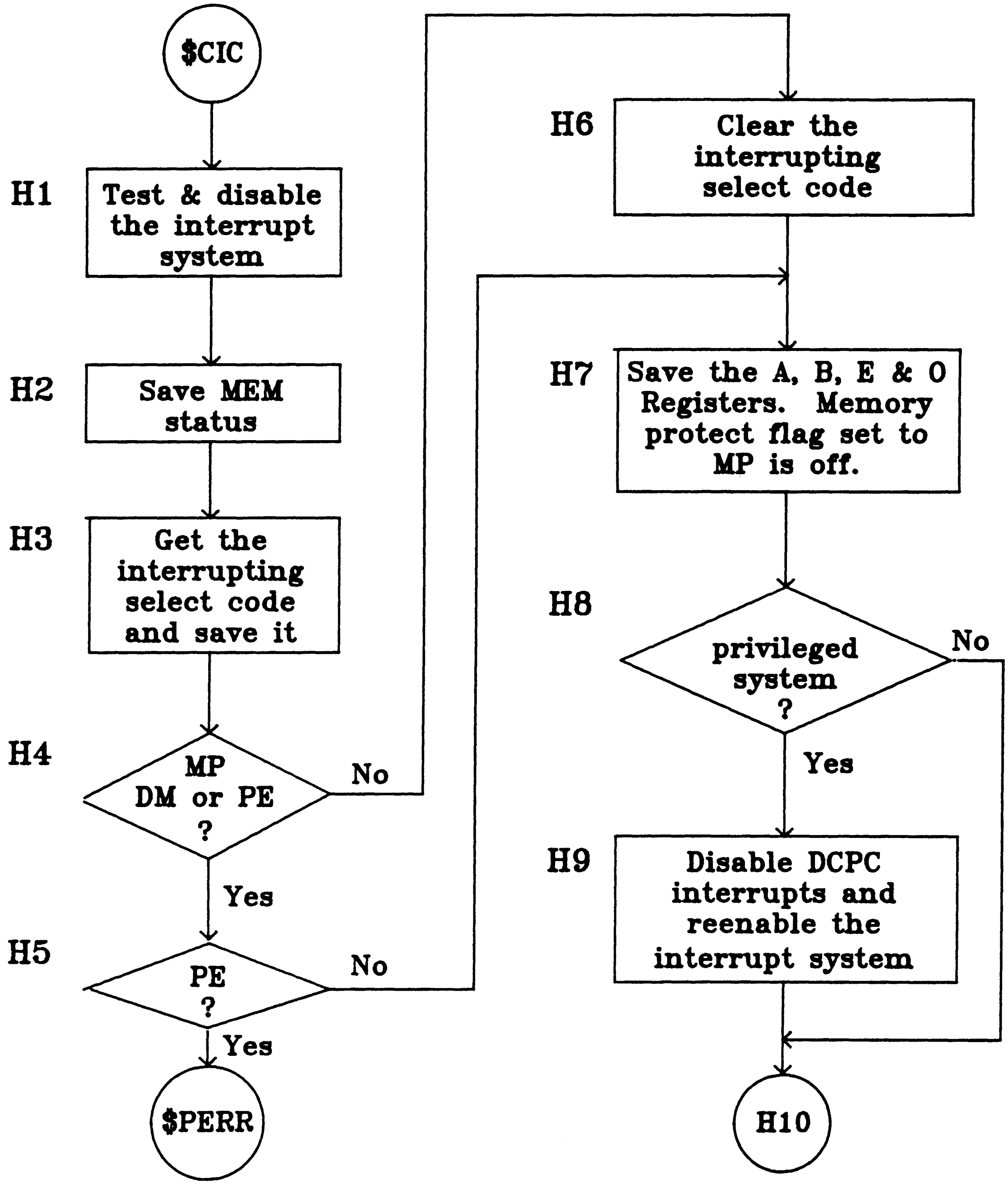


Figure 2-6. Processing the Interrupt

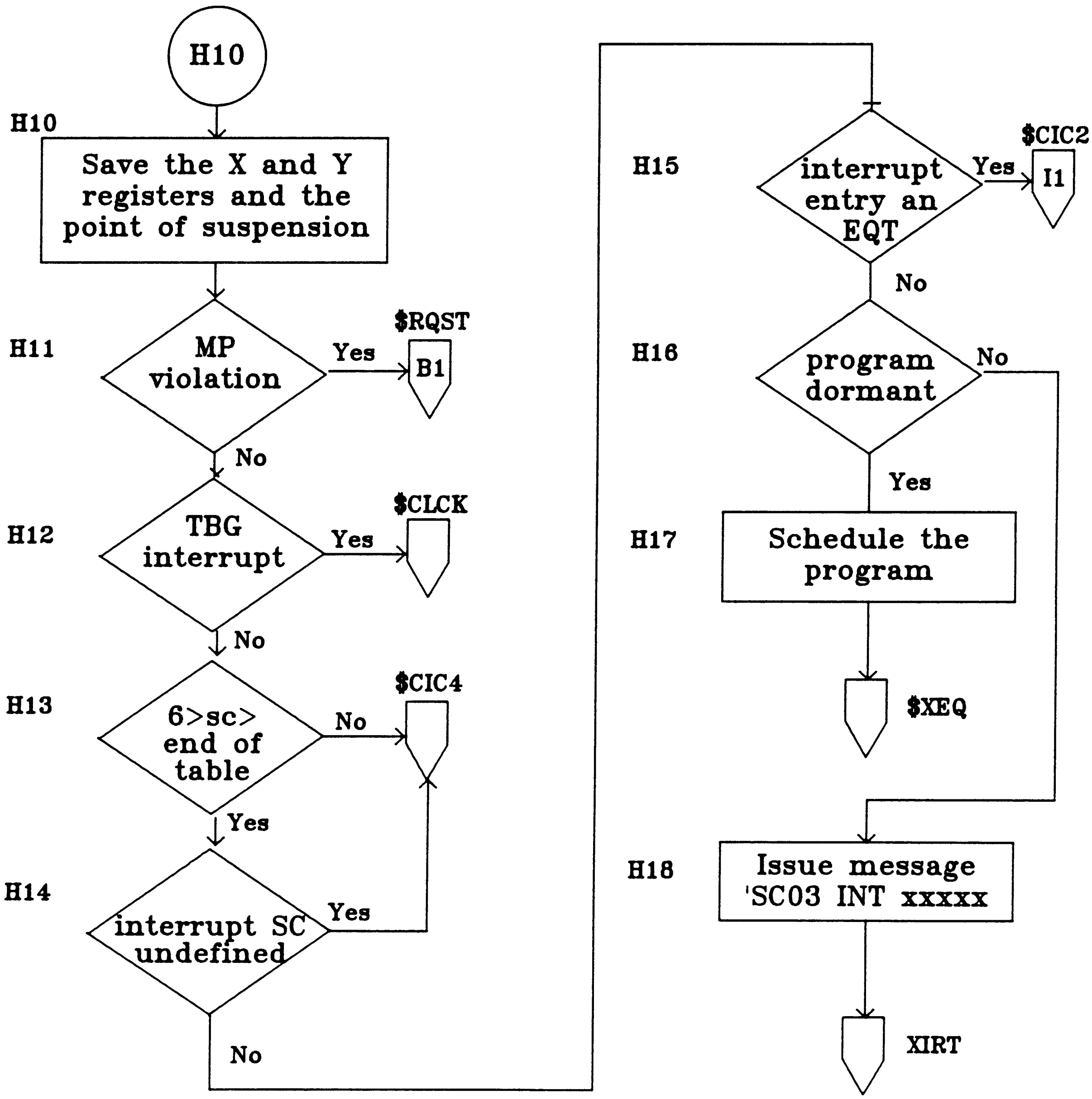


Figure 2-6. Processing the Interrupt (Continued)

H4. $CIC - $DVC (RTIOQ)

If this was a violation on select code 5 you do not need to clear the flag. The flag will be used later. If the violation was not on SC 5, continue at step H6 to clear the device flag. EXEC calls generate an interrupt on SC 5.

H5. (RTIOQ)

Was the violation a parity error? If so, go to $PERR6. If not, continue at step H7. A parity error is indicated by bit 15 of the violation register set to 1.

The memory protect board (on SC 5) should not have its flag cleared because this would turn off the parity error interrupt capability and clear bit 15 of the violation register.

The memory protect card will turn off its own flag when the interrupt system acknowledges the interrupt. There is a special flag that indicates a DMS violation. This flag can be checked with an SFS or SFC instruction. See the 12892A Memory Protect Theory of Operation in the HP 1000 Computers and Engineering and Reference Documentation (Part II, Section IV, part no. 92851-90001).

H6. $DVC (RTIOQ)

Build a CLF instruction to clear the flag on the interrupting device and execute it.

H7. CIC1 (RTIOQ)

Save the A, B, E, and O-Registers in the user ID segment, words 9, 10, and 11. (See the Program ID segment in the RTE-6/VM Programmers Reference Manual, part no. 92084-90005). Set the memory protect flag to 1. MPTFL to indicate that memory protect is now turned off.

H8. SW1 (RTIOQ)

Is this a privileged system? You can determine if it is privileged or not by checking DUMMY in the SYSCOM area. If it is zero it is not privileged. (Go to step H10, otherwise step H9.)

H9. SW1 - CIC.0 (RTCOM)

For a privileged system you should set control on the privileged interrupt card. (The flag is already set from the last time.) The first time through, the flag will not be set because there have not yet been any calls to $IRT. This does not matter because the first time through is during loading of part two of the operating system, and at that time there should not be any active privileged operations.

H10. CIC.0 - $CJMP (RTCOM)

Save the X- and Y-Registers in the first two words of the page the program starts on. Remember, if the program starts in a page addressed as 42000B the program will be reloaded starting at 42012B.

H11. CIC.0 - $CJMP (RTIOQ)

Was this an MP violation? If so, go to RQST in EXEC6 to see if it is a valid EXEC call. Otherwise go to step H12.

Test to see if the interrupt system is on or off. In either case, turn it off. If it is off, bump $INT by one. Do this to indicate to the parity error routine (if it is a parity error interrupt - for this example it is not) whether or not to re-enable interrupts before returning from the parity error routine.

H12. CIC.0 - $CJMP (RTIOQ)

Compare the interrupting select code with that in the system communication variable TBG. If it matches, this is a TBG interrupt.

H13. $CJMP - $SKED (RTIOQ)

Be sure the interrupting select code is contained within the interrupt table.

H14. $CJMP - $SKED (RTIOQ)

An undefined entry would be a zero. In our example it is not zero, so continue.

H15. $CJMP - $SKED (RTIOQ)

If the interrupt table entry is positive, it is an EQT entry. Go to $CIC2 to process it.

2.3.10 Set Up for Drivers

The EQT pointers need be set up only if the first one is not setup. Ensure that the correct map is set up, then call $SNAP to count the number of interrupts on this select code. Set the timeout only if a device timeout value is established. Now get the driver continuation section address from word 3 of the EQT.

2.3.11 Completion Return

(Refer to Figure 2-7.)

The following completion status is returned from the driver to the A-Register, and the associated error messages are delivered to the system console:

0 = Successful completion
1 = Device not ready (IONR)
2 = Unexpected end of transmission (IOET)
3 = Transmission parity error (IOPE)
4 = Device timeout (IOTO)

The B-Register will contain the amount of data transferred. If any errors occurred, additional status information will be found in bits 0-7 of EQT word 5.

K1. $CON1-$L.49 (RTCOM)

Call $RSM to restore the user map in case it was modified by an EXEC call from a type 6 program or it is not the current program in the user map.

K2. $CON1 - $L.49 (RTCOM)

If the DMA bit is set in EQT4 (Bit 15) or the driver returned with bit 15 of the A-Register set, release DCPC (call $CDMA). To release DCPC you clear the EQT ENTRY and do a CLC and STF on the DCPC channel.

K3. $L.49 (RTCOM)

If the I/O request list has no requests on it, then this is treated as an illegal interrupt.

K4. L.49B - L.50 (RTCOM)

If bit 15 of the I/O request (the T field) is set, then this is either a system request or a class I/O request. If it is, take care of it.

K5. L.49B - L.50 (RTCOM)

If this is a user normal request (the T field is 00) go take care of it. In the example program, it is.

K6. L.51 - L.51B (RTCOM)

If the request just completed was an input from an interactive device type 0 or type 5 or 7 subchannel 0, go to step K7 to schedule the program, else step K8.

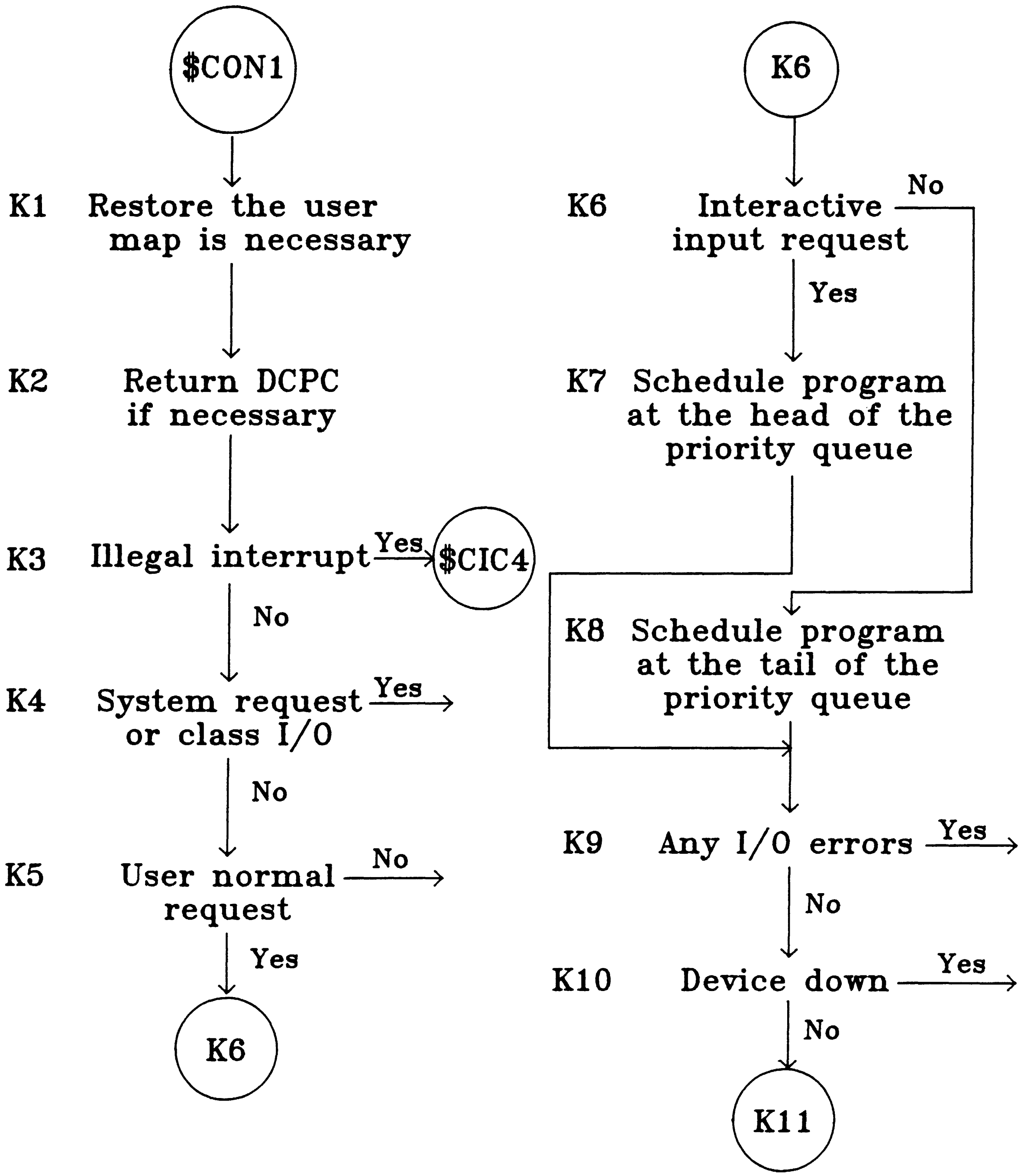


Figure 2-7. Completion Return

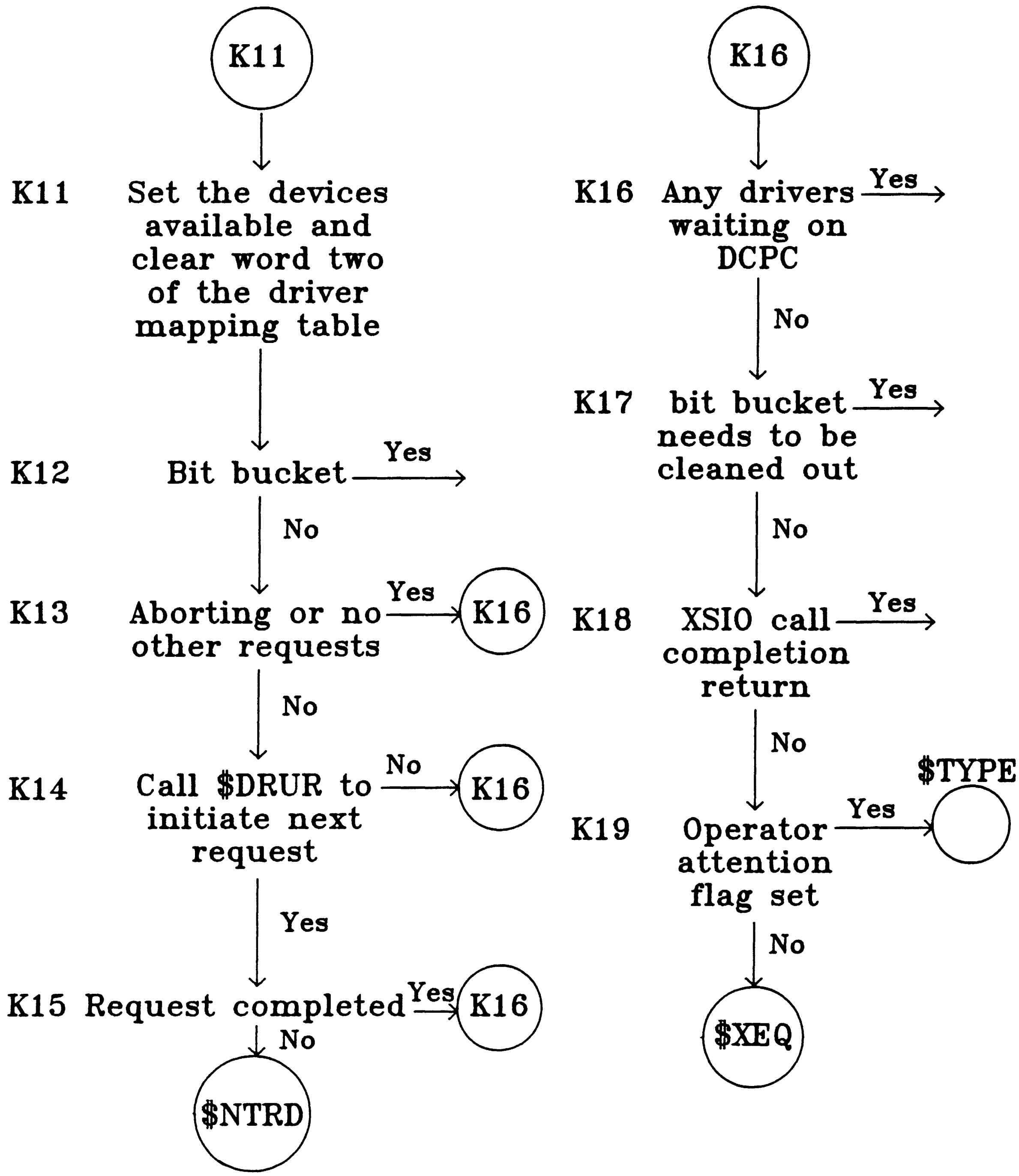


Figure 2-7. Completion Return (Continued)

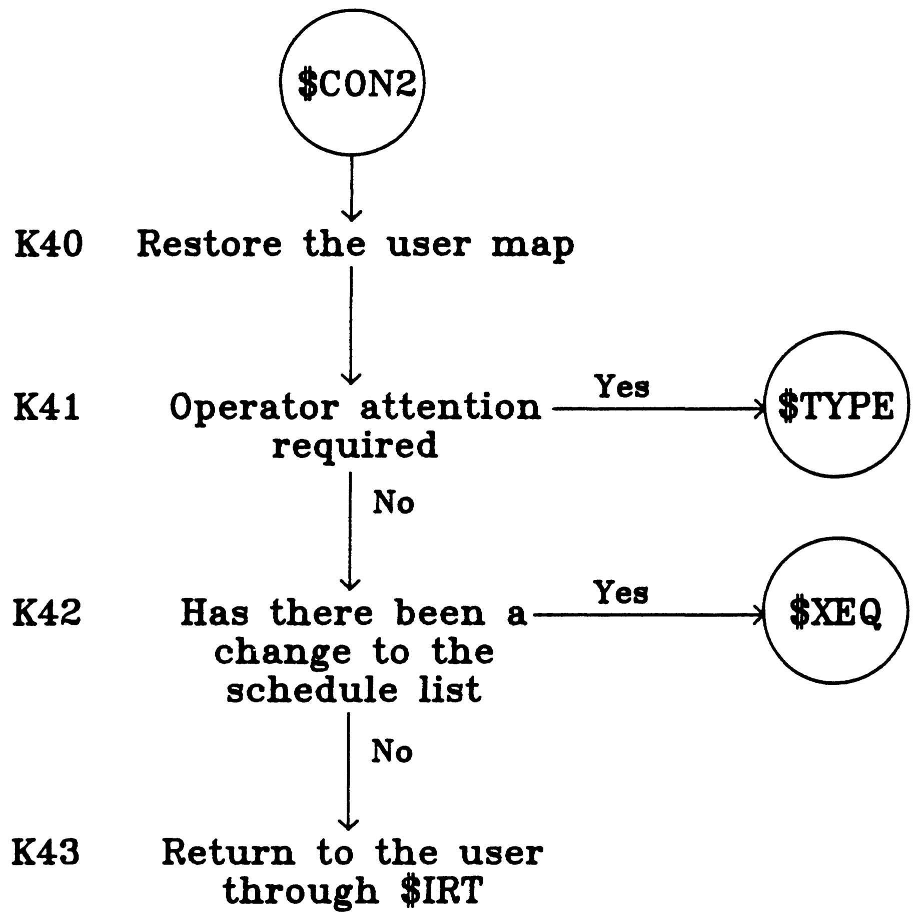


Figure 2-7. Continuation Return (Continued)

K8. L.51A (RTCOM)

Schedule the program at the bottom of its priority List.

K9. L.54U (RTCOM)

Any I/O errors? The status should have been returned in the A-Register from the driver. If it was non-zero it is an error.

K10. L.55 (RTCOM)

Check the AV bits of EQT5. If the device is down do not try to initiate the next transfer.

K11. L.6 (RTCOM)

Set the AV bits of EQT5 to 0 (device available) and clear driver map table word 2 so that the system map is referenced by the driver (default).

K12. L.68 (RTCOM)

If the EQT being referenced is the dummy EQT pointed to by $DMEQ (in RTIOQ) go get rid of it.

K13. L.68-$IOCX (RTCOM)

If the request is to be aborted (bit 15 of the EQT list pointer is set), or no other requests are pending (list is empty), exit to $IOCX.

K16. $IOCX (RTCOM)

Check DMACF to see if any requests for DCPC are pending.

K17. ICOX1 (RTCOM)

Check $BITB to see if there are any bit bucket requests pending.

K18. IOCX - XLOG (RTCOM)

If this is a $XSIO call and a completion return was specified go to it. A completion return is indicated by $CMPL containing a non zero address of where to return. If zero, it is either a $XSIO call without a completion return address or not a XSIO call.

K19. ICOX1 - XLOG (RTCOM)

Check the operator attention flag to see if someone at the system console has hit a key. If not, go to $XEQ.

K40. $CON2 (RTIOQ)

Call $RSM to restore the user map if necessary.

K41. IOCO1 - IOCO3 (RTIOQ)

Check the operator attention flag. If set (non zero), go to $TYPE.

K42.

Check the schedule list to see if there have been any changes (if zero then no changes). If there were no changes go to $IRT to return to the user that was interrupted, else go to $XEQ to dispatch someone new.

K43. XIRT - RTN (RTIOQ)

The return to the user is a multistep process:

1. Set up the return dump based on the suspend address in the ID segment (XSUSP).
2. Restore all the registers.
3. Set the memory protect flag on (0).
4. If this is a privileged system, set the flag on the privileged card and reenable any DCPC interrupts needed In a privileged system the sign bit should be set in the DCPC interrupt entries if the interrupt is needed.
5. Execute a UJP to return to the user map or an SJP to return to the IDLE loop.

K7. L.51B (RTCOM)

Put the program at the top of its priority list in the schedule queue. Programs that do a lot of I/O to an interactive device (for example, EDIT) typically do not use up their full timeslice in execution. In these cases, putting them at the top of the priority list will make them run faster without too much delay for execution of other programs. Continue at step K9.

2.3.12 System Calls

(Refer to Figure 2-8.)

X1. $XSIO (RTCOM)

Pick up the LU number from the caller. Note that the call contains the parameters after the JSB, not pointers to the parameters. Note also that the LU is saved as LU 1.

X2. $XSIO-XSI01 (RTCOM)

Use LU 1 as an Index into the Device Reference Table to get the EQT number.

X3. $XSIO - XSI01 (RTCOM)

With the EQT number, we can set up the pointers (in the system communication area) to the EQT (EQT1-EQT15).

X4. $XSIO - XSI01

If the EQT Lock Table is empty do not bother to check for locked EQTs.

X5. XSI03 - XSI04 (RTCOM)

If the user set bit 13 of the LU parameter the user will handle the I/O errors that occurs.

X6. XSI05 - $XIOE (RTCOM)

$XSIO will modify the caller's code to make it look like the data stored in the user ID segment. This allows the setup routine to work for system calls as well as unbuffered calls.

X7. XSI05 - $XIOE (RTCOM)

Call $LINK to put the request in the I/O list for the specified EQT.

X8. XSI05 - $XIOE (RTCOM)

Do not initiate the request if the device is Locked, busy, or down. Return to the caller in this case.

X9. XSI05 - $XIOE (RTCOM)

Call $DRVR to initiate the call.

X10. XSI05 - $XIOE

If the operation is accepted, return to the caller, else go to $NTRD.

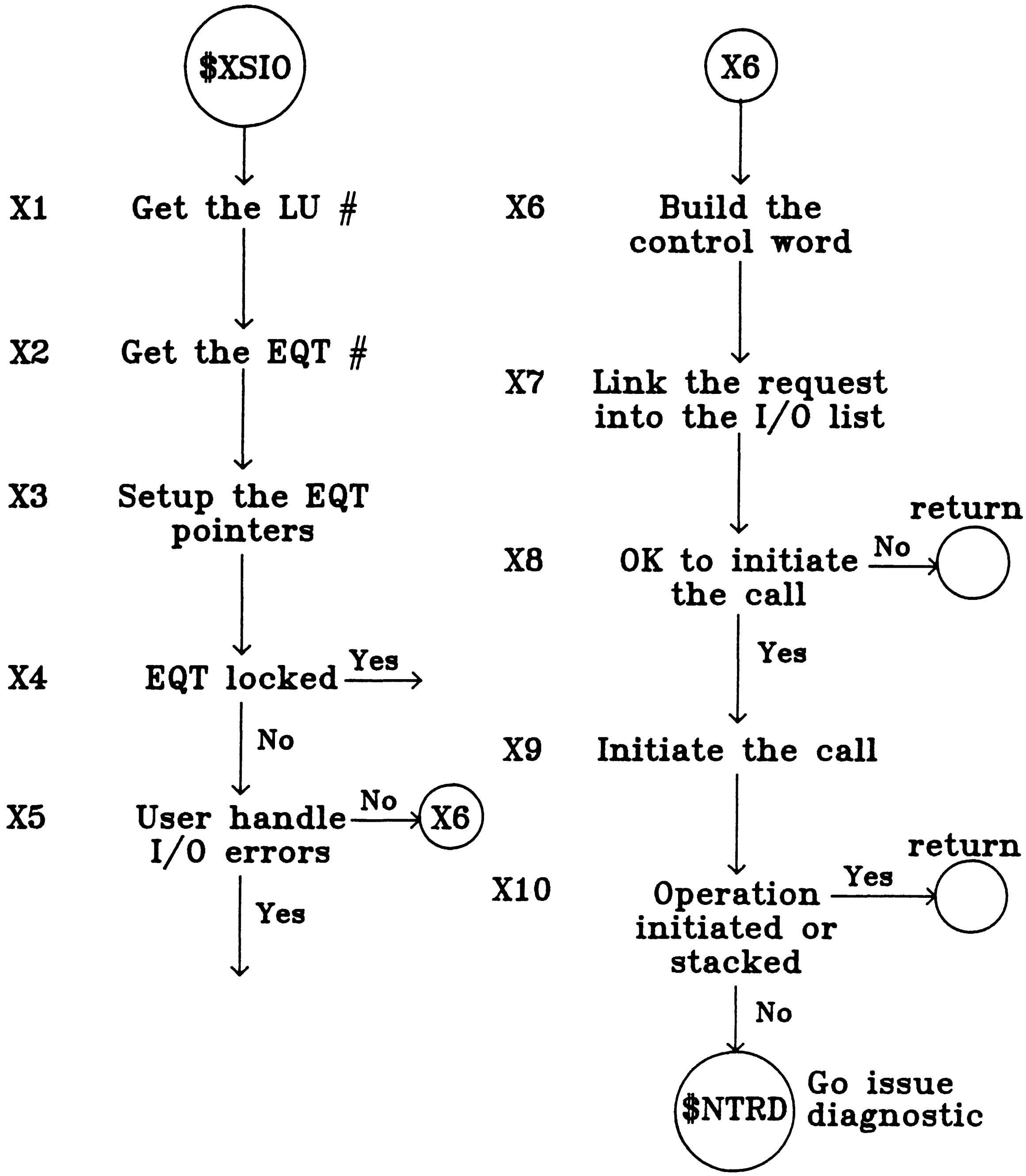


Figure 2-8. System Calls

EXEC AND $ALC	CHAPTER 3

3.1 INTRODUCTION

This chapter deals with the EXEC and system available memory portion of the RTE-6/VM Operating System. The EXEC is that portion of the operating system that checks for legality of all user EXEC requests, vectors legal requests to appropriate processors, vectors illegal requests to the abort processors, handles reentrant processing, and allows users to execute with the interrupt system off (privileged subroutines).

The $ALC portion of the system allocates System Available Memory (SAM) to system processors that request memory for buffer, tables, etc.

The MAPOS module in the system intercepts calls and jumps to the unmapped portions of the operating system, maps the appropriate module, and transfers control.

The EXEC modules contain five major sections:

1. System Request Analyzer (Memory Protect Violation Control)

2. Resident Library Execution Control (Dynamic Mapping Violation Control)

3. Privileged and Reentrant Subroutine Processors

4. Disc Track Allocation and Release Processors

5. General Error Message and Program Abort Processors

In order to understand how the system receives and handles an EXEC request, it is necessary to understand system memory protect and the rudiments of interrupt processing. The discussion below is a very brief description of interrupt processing with memory protect.

Suppose the user wishes to do output to the line printer from a high level language like FORTRAN. The FORTRAN statement would be as shown below:

```
      CALL EXEC (2,6,IBUFR,IBUFL)
```

where the 2 is a Write Request, the 6 is the LU, IBUFR is the buffer to write, and IBUFL is the buffer length.

The FORTRAN compiler would change this to something like:

```
      JSB EXEC
      DEF RETRN   Return address
      DEF IWRIT   Address of Request Code
      DEF LU      LU to write to
      DEF IBUFR   Buffer Address
      DEF IBUFL   Buffer Length
RETRN  :
```

When this code is executed the JSB EXEC will generate a memory protect. In fact any JMP, JSB, ISZ, STA, STB, DST, CBT, JLY, JPY, MVB, MVW, SAX, SAY, SBX, SBY, STX, or STY instruction which would either directly or indirectly affect a memory location below the MP fence will be inhibited and memory protect will force an interrupt to Location 5. The lower bound of protected memory is Location 2 the upper bound is set by the operating system with an OTA 5 (or OTB 5) where A is the address of the highest protected word.

Thus the JSB EXEC was never executed, rather the contents of trap cell 5 (the interrupting location) was executed. The contents of trap cell 5 is a JSB $CIC,I. This now allows us to enter the operating system into a module called RTIOC.

RTIOC is obliged to find out where the interrupt came from and what kind of interrupt it was. By executing a LIA 4 RTIOC will receive the interrupt code # of last interrupt. If the interrupt code corresponds to the Time Base Generator RTIOC jumps to $CLCK in the RTIME module. If the interrupt code is 5 (Dynamic Mapping, Memory Protect or Parity) RTIOC jumps to EXEC. If the interrupt code is anything else RTIOC uses the interrupt table to look up the appropriate processor.

If the interrupt was on interrupt code 5, then a LIA 5 (or LIB 5) will give the violation address; i.e., the address of the JSB EXEC.

Figure 3-1 shows a graphic representation of a JSB EXEC.

We now know how the system enters the EXEC.

```
+----------------------------------------------------+
|            ----------------------                  |
|            |                    |                  |
|            |                    |                  |
|      --------- JSB EXEC         |                  |
|      |     |                    |                  |
|      |     ----------------------                  |
|      |     |                    |                  |
|   ---X------------------------------- Memory       |
|      |     |                    |      Protect     |
|      v     |                    |      Fence       |
|            ----------------------                  |
|            |                    |                  |
|            |   RTIOC MODULE     |                  |
|      ----->|   $CIC  NOP        |                  |
|      |     |         CLF  0     |                  |
|      |     |          .         |                  |
|      |     |          .         |                  |
|      |     |          .         |                  |
|      |     |         JMP  $RQST -----              |
|      |     |                    |   |              |
|      |     ----------------------   |              |
|      |     |                    |   |              |
|      |     |   EXEC MODULE      |   |              |
|      |     |   $RQST LIB 5      |   |              |
|      |     |         LIA 4    <------              |
|      |     |          .         |                  |
|      |     |          .         |                  |
|      |     |          .         |                  |
|      |     |                    |                  |
|      |     ----------------------                  |
|      |     |                    |                  |
|      |     |                    |                  |
|      |     |                    |                  |
|      |     |                    |                  |
|      |     |                    |                  |
|      |     ----------------------                  |
|      |-----|5|       JSB $CIC,I |                  |
|            ----------------------                  |
|            |                    |                  |
|            ----------------------                  |
|            |                    |                  |
|            ----------------------                  |
|                                                    |
+----------------------------------------------------+
```

Figure 3-1. JSB EXEC

The user tries to execute a JSB EXEC, memory protect catches this and instead executes the contents of trap cell 5. This causes an entry into the module RTIOC. RTIOC turns off the interrupt system analyzes where the request is to go and turns control over to the appropriate processor.

3.2 EXEC CALL PROCESSOR

The primary function of this section is to provide for general checking and examination of EXEC CALL requests (EXEC requests) and to call the appropriate processing routine.

This section is called directly from the Central Interrupt Control (CIC) section (in RTIOC) when a memory protect (MP) or dynamic mapping violation (DM) is recognized. (All system requests from a user program cause a protect violation.) This section also determines non-legitimate protect violations in user programs such as executing halt or I/O instructions or attempting to write into a non mapped or protected area. It also recognizes user calls for resident library routines, reentrant, or privileged processing.

Upon entry from CIC, EXEC must decide whether the violation was a true memory protect, parity error, or mapping violation. The EXEC request analyzer examines all memory protect and Dynamic Mapping violations. If the violation is legal, the EXEC jumps to the appropriate processor.

A DM violation is distinguished from a MP violation by executing a SFS 05 instruction. A DM error will set the flag on channel 05, a MP error will clear the flag.

Since parity error and memory protect share the same interrupt locations it is necessary to distinguish which type of error is responsible for the interrupt. A parity error is indicated if, after the LIA (or LIB) 05 instruction is executed, bit 15 of the selected register is a logic 1; a memory protect violation is indicated if bit 15 is a logic 0. In either case, the remaining 15 bits of the selected register contains the address of the error location. Note, however, that parity errors are detected in RTIOC not EXEC.

Only one form of DMS violation is legal. This DMS violation will occur when a memory resident program tries to enter the memory resident library. The memory resident library is used only by memory resident programs. The physical address of the library will be above the memory protect fence if the program is using common; however, the pages containing the library are write protected. Thus any JSB, JMP, etc. to the library will cause a DMS violation. EXEC, after determining that it is a DMS violation, will check for three conditions. They are:

1. That the call is a JSB

2. That the destination is in the memory resident library

3. That the program is a memory resident program-Type 1

If any condition is not satisfied, EXEC aborts the offending program and issues a DM error message.

If all conditions are met, EXEC will jump to the routine if it is a privileged subroutine or jump to $RENT in the dispatcher if the routine is reentrant. $RENT does further processing for reentrant subroutines (such as resetting the memory protect fence).

If the violation was a true Memory Protect, EXEC looks at the destination. Only EXEC, XLUEX, $LIBR, $LIBX and $LOAD (called by the loader to map in a multilevel segment from memory) are legal destination ddresses. All other destinations are flagged as errors and the offending program will be aborted with an MP error. (The abortion is only partially done within the EXEC.)

If the memory protect destination was either EXEC, XLUEX, or $LOAD, then the EXEC further examines the request to see if the request is legitimate.

The EXEC checks to see if there are too many parameters, too few parameters, if the request itself is defined, or if the return address is illegal. EXEC also checks to see if any returned parameters would cause a store below the memory protect fence. This is done by using the NAMTB Table.

The NAMTB has one byte for each EXEC request. Each bit corresponds to a possible parameter. If the bit is set then the EXEC knows that the parameter is a possible store location and checks the address of the store. If the store address is below the memory protect fence, then the program will be aborted with a memory protect.

For example, consider the EXEC disc track allocation request:

```
 JSB EXEC
 DEF RETRN
 DEF ICODE    ICODE=4 or 15
 DEF #TRKS    # of tracks desired
 DEF STRAK    Returned start track #
DEF DISC     Returned Disc LU
DEF SECT#    Returned Sectors per track
```

There would be a bit set for STRAK, DISC and SECT# because these are returned values which must not overlay memory below the memory protect fence. No check would be made for #TRKS as this is not a possible store location.

This is also how the EXEC decides if a read operation should be aborted. That is, the store address (buffer location) would be below the memory protect fence.

The octal contents of the first 3 NAMTB table entries are shown below:

NAMTB Value	Request Code
000002	0/1 not used/Read
000000	2/3 write/control
007000	4/5 disc allocate/Release

Note the upper byte for word 3, the octal 7, it is there because the disc track allocate call returns the starting track number, disc LU, and sectors per track. The lower byte on word 1 has bit 1 set to indicate the buffer location is a storage location for a read request.

When EXEC decides that all the EXEC call parameters are okay, it jumps through the Request Code Table to the appropriate processor.

The request code table is a Table of EXEC request processor addresses. For processes external to EXEC the entry would be:

```
EXT $XXXX
DEF $XXXX+0
```

This gives the direct address of $XXXX for the JMP.

EXEC requests, entry points and the modules called are listed in Table 3-1.

Table 3-1. EXEC Requests and Entry Points

EXEC REQUEST	PURPOSE	ENTRY POINT	MODULE
1	I/O READ	$IORQ	RTIOC
2	I/O WRITE	$IORQ	RTIOC
3	I/O Control	$IORQ	RTIOC
4*	Local Disc Track Allocation	DISC1	EXECD
5*	Local Disc Track Release	DIS2	EXECD
6	Program Completion	$MPT1	SCMEDM
7	Operator Suspension	$MPT2	SCHEDM
8	Load Program Segment	$MPT3	SCHEDM
9	Program Schedule w/wait	$MPT4	SCHEDM
10	Program Schedule w/o wait	$MPT5	SCHEDM
11	System Time & Date	$MPT6	SCHEDM
12	Prog. Schedule after offset or Prog. Schedule at absolute time	$MPT7	SCHEDM
13	I/O Device Status	$IORQ	RTIOC
14	Get/Put String	$MPT9	SCHEDM
15*	Global Disc Track Allocation	DISCA	EXECD
16*	Global Disc Track Release	DISCD	EXECD
17	Class I/O READ	$IORQ	RTIOC
18	Class I/O WRITE	$IORQ	RTIOC
19	Class I/O Control	$IORQ	RTIOC
20	Class I/O Write/Read	$IORQ	RTIOC
21	Class I/O GET	$GTIO	RTIOC
22	Prog. Swapping Control	$MPT8	SCHEDM
23	Prog. Schedule w/WAIT & w/QUEUE	$MPT4	SCHEDM
24	Prog. Schedule w/o WAIT & w/QUEUE	$MPT5	SCHEDM
25**	Partition Status	$PTST	EXECD
26***	Memory Size Status	MEMST	EXECD

* The request is serviced in EXEC.
** This request has changed for RTE-6/VM and is serviced in EXEC.
*** This request is new for RTE-6/VM and is serviced in EXEC

Before transferring control to the appropriate processors, the EXEC places the address of all the request parameters in the base page as defined below:

BASE PAGE ADDRESS	LABEL	USE
1676	RQCNT	Request Count=# of EXEC call parameters -1
1677	RQRTN	RETURN address of EXEC call
1700	RQP1	The REQUEST CODE of CALL
1701	RQP2	2nd Request Parameter
1702	RQP3	3rd Request Parameter
1703	RQP4	4th Request Parameter
1704	RQP5	5th Request Parameter
1705	RQP6	6th Request Parameter
1706	RQP7	7th Request Parameter
1707	RQP8	8th Request Parameter
1710	RQP9	9th Request Parameter

These base page locations will always be visible regardless of map. The contents, however, refers to addresses in the user map. The EXEC executes under control of the system map.

3.3 ORGANIZATION OF EXEC

EXEC is divided into three chunks:

1. The first chunk resides in the system map at all times and contains:

    a. The code to check the validity of EXEC, $LIBR, and $LOAD calls.

    b. The "privileged" program format processing of the $LIBR call.

    c. The $LIBX call processing.

    d. The error message section (MPERR, $ABXY, $ERMG)

    e. The request code table.

2. The second chunk contains modules with "OSO" as the first three characters of each module name. Refer to the MAPOS section of this chapter for more details on OSx type modules. This chunk also contains:

    a. The system disc track allocation/release processors.

    b. The LG command processor.

    c. The Track Assignment Table.

3. The third chunk contains modules with "OS1" as the first three characters of each module name. This chunk also contains:

   a. The reentrant subroutine processors, partition size, and status request processor.

   b. The LU, EQ, TO command processors.

   c. The RN/LU lock clean-up routine.

   d. The parity error handler (message output portions of this routine reside in the system map at all times).

Extended EXEC*

The following set of extended EXEC (XLUEX) calls will provide HP subsystems access to logical units greater than 63 (decimal). The XLUEX calls will have similar calling sequences (with EXEC) and identical functions. The only difference is in the definition of the control word (RQP2). XLUEX will use two words to specify the logical unit and control information while EXEC uses one.

EXEC control word:

```
---------------------------------------
|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0|
---------------------------------------
|   reserved    | function  | logical  |
                    code       unit
```

XLUEX expands the control word into Logical unit and function code parameters:

XLUEX - Logical Unit word

```
---------------------------------------
|15|14 13 12|11 10 9|8 7 6|5 4 3|2 1 0|
---------------------------------------
| S|  reserved       | logical unit   |
```

XLUEX - Function Code word

```
---------------------------------------
|15|14 13 12|11 10 9|8 7 6|5 4 3|2 1 0|
---------------------------------------
|   reserved   | function |  reserved |
                   code
```

The S bit, if set, will inhibit the session or batch switch table mapping (i.e., the LU number supplied is the LU number to be used).

NOTE: This capability will not be documented for the user until all supported HP subsystems have been modified to enable access of the full range of logical unit numbers. Until that point in time, the user must access logical units > 63 via the session switch table.

CAUTION

This implementation may be a temporary solution as future projects may alter the external characteristics of these calls.

The following functions will be supported by XLUEX calls:

Read, write, control, status, class read, class write, class, write/read, class control.

Calling sequence: (refer to the RTE-6/VM Programmer's Reference Manual for Parameter details - other than those previously discussed.

READ/WRITE:

```
     EXT XLUEX

     JSB XLUEX
     DEF EXIT
     DEF RCODE          (READ=1, WRITE=2)
     DEF CONWD          Note:  New or changed (2 word parameter)
     DEF BUFFR
     DEF BUFFL
     DEF DTRAK          optional
     DEF DSECT          optional
EXIT  :
```

CONTROL:

```
     EXT XLUEX

     JSB XLUEX
     DEF RTN
     DEF RCODE          (control =3)
     DEF CONWD          Note:  New or changed (2 word parameter)
     DEF IPRAM          optional
RTN   :
```

STATUS:

```
     EXT XLUEX

     JSB XLUEX
     DEF RTN
     DEF RCODE          (Status=13)
     DEF LU             Note:  This word contains the logical unit only
     DEF ISTA1
     DEF ISTA2          optional
     DEF ISTA3          optional
RTN    :
```

CLASS READ OR WRITE OF WRITE/READ:

```
     EXT XLUEX

     JSB XLUEX
     DEF RTN
     DEF RCODE          (Read=17,write=18,write/read=20)
     DEF CONWD          Note:  New or changed (2 word parameter)
     DEF IBUFR
     DEF IBUFL
     DEF IPRM1          optional
     DEF IPRM2          optional
     DEF Class          optional
 RTN   :
```

CLASS I/O CONTROL:

```
     EXT XLUEX

     JSB XLUEX
     DEF RTN            (Class Control=19)
     DEF RCODE
     DEF CONWD          Note:  New or changed
     DEF IPRAM
     DEF ICLAS
 RTN   :
```

3.4 LIBRARY EXECUTION CONTROL

The Relocatable Library contains the set of subroutines required for floating point operations, intrinsic functions, FORTRAN run-time processors and general utility functions.

A program in this library is structured in one of the following formats:

1. Reentrant: (Type 6) During the execution of the routine, it may be suspended and entered again by a call from a higher priority program. Subroutines in this format may not modify in-line code (i.e., they are "read only" and all temporary variables must be grouped into a block within the program). This block is termed the "Temporary Data Block",TDB. The execution time of a reentrant routine is usually greater than 1 millisecond. In RTE-6/VM only those reentrant subroutines loaded into the memory resident library and called from Memory Resident programs or those subroutines loaded into SSGA can be reentered by different programs.

2. Privileged: (Type 6) A routine in this format is permitted to run with the interrupt system and memory protect disabled. A subroutine of this type should have an execution time of less than 1 millisecond. It also may not incorporate input/output calls, nor may it call a reentrant routine.

3. Utility: (Type 7) This classification is used for programs containing I/O functions or other features which do not allow reentrant or privileged structure. Examples of this type are the FORTRAN runtime routines PAUSE and STOP. There are no restrictions on internal program structure or features. The subroutine will always be appended to the end of the user's program.

3.5 RESIDENT LIBRARY SUBROUTINES

The resident library consists of those subroutines referenced by memory resident programs. Should that subroutine reference another subroutine, the second subroutine will also become part of the memory resident library. The memory resident library is shared by all memory resident programs. The sharing prevents commonly called subroutines from being appended to each memory resident program that calls it, thus affecting a conservation of memory for memory resident programs. Note that the resident library is created at generation time and that all routines which are loaded into the resident library are also put in the relocatable library for disc resident programs. Since the subroutine is shareable it should be written in a privileged or reentrant format.

3.6 UTILITY AND SINGLE-USER LIBRARY PROGRAMS

A utility subroutine can be called by only one user program. Therefore a copy of the utility program is appended to the absolute version of any user program which references it. All programs in reentrant or privileged format are reclassified as utility if they are not included in the Resident Library by RTGEN. A copy of each subroutine is appended to each disc-resident user program which references it. (Thus all type 6 routines put in at generation become type 7 after generation.)

All library type subroutines entered when the system is generated are reclassified as utilities and stored in packed relocatable format on the disc for use by the LOADR in loading programs on-line.

Users who wish to write subroutines which can be loaded into the memory resident library to be shared by memory resident programs should refer to Appendix C for the required format.

Reentrant and privileged subroutines require special pre and post processing. This processing is done by the routines $LIBR and $LIBX. The format is shown below. The code below the dotted line is needed for reentrant routines only. pre-,post proce

```
          EXT    $LIBR,$LIBX
ENTRY     NOP
          JSB    $LIBR
          DEF    TDB (or "NOP" if privileged)
          ---    First program instruction--
           -
           -
           -     body
           -     of
           -     program
           -
EXIT      JSB    $LIBX
          DEF    TDB (or DEF ENTRY if privileged)
------------------------------------------------------------------------
          DEC    N Return adjustment for reentrant
                   (Return=N + ENTRY)
TDB       NOP      Holds linkage to previous block
          DEC    K Total Length of TDB in words
          NOP      Holds return address of call
          -       -Blocks used
          -        for temporary
          -        storage of values
          -         generated by the program
```

The TDB (Temporary Data Block) and return adjustment is only for reentrant format. The return adjustment for reentrant format in the exit call is used to vary the return point to the calling program. The return address and return adjustment are added to determine the final return address.

The parameter following the JSB $LIBR (DEF TDB, or NOP) identifies the subroutine format to the system and the type of processing that is required. A NOP signifies a privileged subroutine.

Reentrant programs may call other reentrant and privileged programs. However, privileged programs may only call privileged programs.

The JSB $LIBR is intercepted by EXEC because it causes a memory protect.

3.7 PRIVILEGED AND REENTRANT PROCESSING

Privileged or reentrant processing starts whenever the initial memory protect or DMA violation for that service is detected. This can happen in two ways.

Consider the two cases below:

```
    CASE 1     ANY PROGRAM
          .
          .
          .
          JSB    SUB
          .
          .
          .
      SUB NOP
          JSB    $LIBR
          NOP


    CASE 2     MEMORY RESIDENT PROGRAM
          .
          .
          .
          JSB  SUB
          .
          .
          .

+------------------------------------------------------------+
!                                     MEMORY RESIDENT LIBRARY !
!                                                            !
!       SUB NOP                                              !
!           JSB $LIBR                                        !
!           NOP                                              !
!                                                            !
!                                                            !
+------------------------------------------------------------+
```

In Case 1 all code is within the users program. The JSB $LIBR causes the memory protect. As mentioned earlier $LIBR is a valid memory protect and thus the system starts the privileged or reentrant run.

In Case 2, however, a DM violation resulted due to the JSB SUB. This is because SUB resides in the memory resident library. Here the privileged or reentrant run started at the JSB SUB. EXEC places the return address (P+1 of JSB SUB) into SUB, that is, it simulates the JSB instruction and eventually returns control to three words past the SUB NOP (i.e., the target of the JSB). In this case the JSB $LIBR was never executed.

As can be seen from Case 2 all subroutines that are loaded into the memory resident library (type 6 subroutines) must be in the privileged or reentrant format.

EXEC examines the word (P+1) following the JSB $LIBR. If (P+1)=0 (NOP), the called subroutine is "privileged". $LIBR restores the registers, adds 1 to "$PVCN" (privileged subroutine nest count), leaves the interrupt system disabled, (which also means MP disabled) and transfers control to the word following the $LIBR call (i.e., P+2). The return address to the program (P+1) of the JSB SUB is stored in the entry point of the library subroutine if a protect violation occurred on the original call.

If the (P+1) of the JSB $LIBR is non-zero, the value is the address of the Temporary Data Block of the reentrant subroutine. The first word of the TDB is checked. If it is zero, then the subroutine is not being reentered.

The first word is then set up to point to the second word of a 4 word block of memory set up for each JSB $LIBR used in a reentrant run. This block is located in system available memory (SAM). The contents of this second word is the ID address of the program using the TDB. (More discussion on this reentrant list structure will be found in the following sections. Referencing to the list structure in Appendix B at this time should help in understanding the discussion below.)

If the link word is non-zero, the subroutine is being reentered (i.e., two memory resident programs want the same subroutine) and $ALC is called by EXEC MTDB routine to allocate a block in available memory equal to the length of the TDB (word 2). If $ALC rejects the allocation request, the main user program is suspended and linked into the memory suspend list.

If the block is allocated, the TDB is moved to the new block. If the new block is one word longer than requested (refer to discussion on $ALC), word 2 (word length of TDB) in the new block is set negative as a flag. The first word of the moved TDB in the system map is changed to point to the first word of the original TDB in the user map.

The address of the original program call is set in word 3 of the program as the return address. The reentrant program must not modify the first words of the TDB. EXEC then calls $RENT in the dispatcher who sets the memory protect fence to the beginning of the Resident Library area, removes DMS write protect, and restores the program registers. The interrupt system is enabled, memory protect turned on, and control transferred to the program.

For privileged subroutines, the system saves all registers going into the subroutine and restores them when the subroutine starts to execute. With nested privileged subroutines the system does not save the registers on 2,3,4, etc., call but neither does the system destroy the registers. That is, the A,B,Y,X,E and O registers may be used to pass parameters to and from privileged subroutines (and reentrant subroutines).

The return to the main program at the end of a reentrant or privileged subroutine is performed by a JSB $LIBX. The execution of this instruction is executed directly if a privileged program is executing; it causes a memory protect violation if a reentrant program is executing In the latter case, EXEC transfers control to $LIBX indirectly after the initial protect violation processing.

If the executing program is privileged (i.e., $PVCN>0), one is subtracted from $PVCN. If $PVCN is still non-zero, control is returned directly, with registers restored, to the return point in the calling privileged program. If now $PVCN=0, control is returned to the caller with the interrupt system enabled and the memory protect fence set to the beginning of the area of the original calling program.

If the executing program was reentrant, the return address is calculated adding the contents of the third word of the TDB which contains the P+1 original JSB SUB and the P+2 of the JSB $LIBX which may contain a return adjustment. This address is placed into the ID segments point of suspension. In addition, the necessary adjustments are made to the reentrant list and to system available memory. This structure is discussed below.

All $LIBR calls require an associated $LIBX call.

3.8 REENTRANT LIST STRUCTURE

Every reentrant call requires the creation of a 4-word table in system available memory called a reentrant table. All of these tables are connected through a list structure with its head in the EXEC (DHED) (the reentrant list). The list is a two dimensional list. The first dimensi dimension is a stack and is one entry per program. The second dimension is for programs that make nested reentrant calls and is a push down stack after the first entry (i.e., the one that got the program in the list in the first place.

The purpose, structure, and content of this reentrant ID list is graphically documented in Appendix B.

3.9 FORMAT OF REENTRANT SUBROUTINE LIST

The reentrant Table is a 4 word table in system available memory that is allocated every time a reentrant call is made (i.e., one for every reentrant JSB $LIBR).

Word	Purpose
1	Link to next 4 word block (0=End of List)
2*	ID address of user making this reentrant call
3**	Pointer to TDB buffer in reentrant subroutine
4	Used if one reentrant subroutine calls another. It points to next 4 word entry for this program.

* Sign Bit set if K+1 words of SAM allocated instead of K words asked for.

**Sign Bit of this word is set if TDB has been moved to system available memory. If sign bit set, pointer points to moved TDB in SA

The reentrant structure is also used to allow buffered input and output. The $REIO routine in EXEC is called by RTIOC (is never called by EXEC itself) anytime I/O is done in a reentrant subroutine. For example, the FORTRAN callable REIO routine; i.e., CALL REIO (I,LU, BUFR,BUFRL) does I/O from a reentrant subroutine and causes entry into $REIO.

Consider the case of normal (unbuffered) input. Since the input from the peripheral device is being placed within the program area itself, that program will be I/O suspended and unswappable. The program cannot be swapped because I/O is being done to a particular part of memory. If another program were placed there that area of the other program would be overlayed by the incoming data. Thus the unbuffered input has caused a lock of that partition meaning no other program can use it. The case of normal output is the same, an unusble partition for the length of the I/O.

This problem can be avoided by doing I/O from a reentrant subroutine where the I/O buffer is wholly within the TDB itself. $REIO is called from RTIOC anytime I/O is done from a reentrant subroutine. $REIO looks to see if the program has an ID Reentrant TAG (i.e., is it really reentrant and has done a JSB $LIBR) if so it then looks at the buffer address and length. If the entire buffer is within the TDB then $REIO has MTDB call $ALC for TDB space in system available memory, sets the $MVBF flag in RTIOC to the negative of the TDB address, and returns to RTIOC. RTIOC then knows that the buffer is not in the program area and this then makes the program swappable and frees the partition for other uses.

The process for output is essentially the same. The output buffer within the TDB is moved to S.A.M. and $REIO gives RTIOC the negative new address in $MUFB which will be outside the program area. The program may then continue to execute because all the data is outside the program area.

3.10 DISC TRACK ALLOCATION PROCESSORS AND REQUESTS

The system maintains complete control over the allocation and ownership the system (LU2) and auxiliary disc (LU3) tracks. User programs, through EXEC requests, can allocate tracks to themselves (local) or allocate tracks for general use by anyone (global). User programs can also release the tracks back to the available pool via EXEC requests.

A track, if allocated to a program, is such that only that program which requested it can write on it and/or release it. Any program can read from it.

A global track is such that any program can read from it, write on it, and/or release it.

Track control is maintained via the Track Assignment Table (TAT). Peripheral discs (NOT LU2 or LU3) are not managed through the track assignment table.

Figure 3-2 shows the structure of the system disc (LU2). The system disc has three distinct areas. The first area, from track 0 to approximately track 20 (this area will vary depending on the size of the system, 15 to 40 tracks is typical) is the system area of the disc. The virgin copy of the operating system, drivers and all user programs loaded at generation time are stored in this location.

The second area from approximately track 20 to track 100 is the track pool or scratch area of the disc. The upper boundary of this area is determined the first time a generated system is booted up. The boundary is set by the File Manager initialize command. (IN, master sec code, -LU, cartridge ref., label, start track, number of tracks).

The Track Pool is used by the system for swapping, text editing, loading permanent program additions, etc. There must be a minimum of 8 track pool tracks on LU2, however, a minimum of 70 track pool tracks is recommended.

If the extended memory feature of RTE-6/VM is being used more track pool area may be necessary to allow swapping of large arrays. The additional space needed can be gauged by recalling that one disc track contains space for 6144 words.

The third area of the system disc is for user files. The File Manager maintains this area. An auxiliary disc (LU3), Figure 3-3, can be used with RTE to extend the size of the track pool if desired.

```
+--------------------------------------------------------+
|                                                        |
|                 +-----------+                          |
|                 |           |                          |
|                 |    FMP    |                          |
|                 | FILE AREA |                          |
|                 |           |                          |
|                 |           |                          |
|                 +-----------+<--- TRACK 100            |
|                 |           |                          |
|                 |   TRACK   |                          |
|                 |   POOL    |                          |
|                 |           |                          |
|                 | (SCRATCH  |                          |
|                 |  TRACKS)  |                          |
|                 |           |                          |
|                 +-----------+<--- TRACK 15 TO 40       |
|                 |           |                          |
|                 |  SYSTEM   |                          |
|                 |  TRACKS   |                          |
|                 |           |                          |
|                 +-----------+<--- TRACK 0              |
|                                                        |
+--------------------------------------------------------+
```

Figure 3-2. LU 2

```
+--------------------------------------------------------+
|                                                        |
|                 +-----------+                          |
|                 |           |                          |
|                 |    FMP    |                          |
|                 | FILE AREA |                          |
|                 |           |                          |
|                 |           |                          |
|                 +-----------+<--- TRACK 50             |
|                 |           |                          |
|                 | EXTENDED  |                          |
|                 |  TRACK    |                          |
|                 |  POOL     |                          |
|                 |           |                          |
|                 +-----------+<--- TRACK 0              |
|                                                        |
+--------------------------------------------------------+
```

Figure 3-3. LU 3

3.11 TRACK ASSIGNMENT TABLE (TAT)

The TAT is a variable length table denoting the availability of each disc track on the system and auxiliary discs. It resides as a 1600 word buffer (maximum size for TAT) in an OS0 type module in the unmapped portion of memory. The word "TATSD" in the Base Page Communication Area contains the number of tracks on the system disc. "TATLG" contains the negative number of tracks on the system and auxiliary discs. The first TATSD words of TAT describe the system disc. The next ((-TATLG) - TATSD) words describe the auxiliary disc.

At boot-up time, EXEC assigns the first ($#TRK - 1) tracks in TAT to the operating system. The next track is assigned to D.RTR (if D.RTR exists).

The contents of a track assignment entry word may be one of the five values:

CONTENTS OF TRACK ASSIGNMENT TABLE

Contents	Meaning
0	Available
100000	Assigned to System (or protected)
077777	Assigned globally (anybody can write)
077776	Assigned to FMGR (FMP Package)
XXXXXX	ID segment address of owner

Base Page Words Used for Track Assignment

BP Word	Name	Purpose
1656	TAT	FWA of Track Assignment Table
1755	TATLG	NEGATIVE length of Track Assignment Table
1756	TATSD	# of Tracks of System Disc
1757	SECT2	# of Sectors/Track on System Disc (LU2)
1760	SECT3	# of Sectors/Track on Aux Disc (LU3)

Graphically, the TAT is searched as shown below:

```
+--------------------------------------------------------+
|                                                        |
|                 +-----------+TATLG - 1                 |
|               / |           |                         |
|              /  |    FMP    |                         |
|             /   |           |$DLU3                    |
|            /    +-----------+----------               |
|        LU3      |           |         |               |
|            \    |   TRACK   |         v               |
|             \   |   POOL    |         System Request  |
|              \  |           |TATSD                    |
|        ---------+-----------+                         |
|               / |           |         not included    |
|              /  |    FMP    |         in TAT search   |
|             /   |           |$DLU2                    |
|            /    +-----------+                         |
|           /     |           |                         |
|        LU2      |   TRACK   |         User Request    |
|          \      |   POOL    |         ^               |
|           \     |           |$#TAT    |               |
|            \    +-----------+----------               |
|             \   |           |                         |
|              \  | OPERATING |                         |
|               \ |  SYSTEM   |                         |
|                \|           |                         |
|                 +-----------+0                        |
|                                                        |
+--------------------------------------------------------+
```

Only the track pool entries are searched to process a disc track allocation/release request. The TAT is searched bottom-up for a user request and top-down for a system request. $DLU2 and $DLU3 entry points are set up as the first FMGR track on LU2 and LU3 by FMGR at boot-up or when initializing LU2 or LU3. If there are no FMGR tracks on LU2 or LU3, then $DLU2 or $DLU3 will equal -1.

The track pool area searched on LU2 is between (TAT + $#TRK) and (TAT + $DLU2). If $DLU2 = -1, then (TAT + TATSD). The track pool area on LU 3 is determined by (TAT + TATSD) and (TAT + TATSD + $DLU3). If $DLU3 = -1, then (TAT + (-TATLG)).

A subroutine TATMP in the system library and $MTAT in the MAPOS module in the operating system are used to map the TAT in the first driver partition area. When a program using TAT does I/O or EMA access, the mapping for TAT module is lost. In such cases, the program must call TATMP before accessing the Track Assignment Table again.

3.12 ERROR MESSAGE PROCESSOR

The EXEC will detect five classes of errors Memory Protect (MP), Dynamic Mapping (DM), Request Code (RQ), Reentrant Subroutine errors (RE), and Parity ERRORS (PE).

All of these errors will cause program abortion (even if the no abort bit is set). The error message and the error is discussed below:

3.12.1 MEMORY PROTECT

In RTE-6/VM the operating system is protected by a hardware memory protect. This means that any program that illegally tries to modify or jump to the operating system will cause a memory protect interrupt. The operating system intercepts the interrupt and determines it's legality. If the memory protect is illegal, then the program is aborted and the following message is reported to the system console:

```
MP INST = XXXXXX          XXXXXX = OFFENDING OCTAL INSTRUCTION CODE
ABE PPPPPP QQQQQQ R       CONTENTS  OF  A, B & E REGISTERS AT ABORT
XYO PPPPPP QQQQQQ R       CONTENTS  OF  X, Y & O REGISTERS AT ABORT
LEAF NODE = NN            NN = LEAF NODE OF CURRENTLY ENABLED PATH
MP YYYYY   ZZZZZ          YYYYY = PROGRAM NAME
                          ZZZZZ = VIOLATION ADDRESS
YYYYY ABORTED
```

The NN leaf node value above is displayed only for MLS programs.

3.12.2 DYNAMIC MAPPING VIOLATION

A dynamic mapping violation occurs when an illegal read or write occurs to a protected page of memory. This may happen when one user tries to write beyond his own address space to non existant memory or someone elses memory. In this case the program is aborted and the following message is printed:

```
 DM VIOL = WWWWW          WWWWW = CONTENTS  OF  DMS  VIOL  REGISTER
DM INST = XXXXX
ABE PPPPPP QQQQQQ R
XYO PPPPPP QQQQQQ R
LEAF NODE = NN
DM YYYYY  ZZZZZ
YYYYY ABORTED
```

3.12.3 EX ERRORS

It is possible to execute in the privileged mode (i.e. interrupt system off) in this case the user may not make EXEC requests because memory protect, which is the access vehicle to EXEC is off. An attempt to make an EXEC call with the interrupt system off will cause the calling program to be aborted and the following message printed:

```
LEAF NODE = NN
EX   YYYYY ZZZZZ
EX   ABORTED
```

This error is detected in $TB1. The error is detected by virtue of the fact that EXEC was entered directly instead of causing a Memory Protect.

3.12.4 UNEXPECTED DM AND MP ERRORS

The operating system handles all MP and DM violations. Certain of these violations are legal and others are not. In any case the operating system associates these violations with program activity. If a DM or MP error occurs and no program was active then, this is an unexpected MP or DM violation. Since no program is present, there is no program to abort in this case the following message will be printed:

```
DM VIOL = WWWWW
DM INST = XXXXX          OR          MP INST = XXXXX
ABE PPPPPP QQQQQQ R                  ABE PPPPPP QQQQQQ R
XYO PPPPPP QQQQQQ R                  XYO PPPPPP QQQQQQ R
DM <INT>     0                       MP <INT> = 0
```

```
*  WARNING  *  WARNING  *  WARNING  *  WARNING  *  WARNING  *
=================================================================
```

The above message which specifies <INT> as the program name is a signal to the user that an unexpected memory protect or dynamic mapping violation error has occurred. This is a serious violation of operating system integrity. Most times it means user written software (driver,privileged subroutine) has damaged the operating system integrity or inadequately performed required (driver) system housekeeping. It may also mean that the CPU has failed and that the operating system caught the failure in time to avoid a system crash.

If this error occurs it is suggested that users save whatever they were doing (i.e., finish up editing, etc.) and reboot the system. If only HP system modules are present in the operating system, CPU failure is highly suspected and CPU diagnostics should be run.

3.13 SYSTEM AVAILABLE MEMORY (SAM)

Reentrant subroutine ID tags, reentrant I/O, automatic buffering to I/O devices, and many other operating system features require blocks of memory to be made available at any time. In order to satisfy these temporary needs for memory an area of memory was set aside and called system available memory (SAM). Two routines manage SAM. The routine $ALC allocates memory to the requestor and the routine $RTN returns memory no longer needed back to SAM.

SAM is allocated in contiguous chunks of memory and is maintained via a list of available contiguous chunks. Over the course of time memory will be given away and returned many times. Memory that is returned is checked to see if it is contiguous from above or below to any existing free memory. If not it is linked to the currently existing free memory. The link structure uses the first two words of the chunk returned for the linkage. The first word is the number of words in that block and the second word contains the address of the first word of the next available free chunk of memory. If the returned memory is contiguous to an existing block then the returned memory is concatenated by just updating (or creating) the two word linkage at the beginning of the block to reflect the fact that the new block length is greater.

$ALC allocates memory to the caller by giving that caller the amount of memory requested the first time it finds that much memory in a free block. No best fit algorithm is used as it has been found that best fit routines are too slow and wasteful of CPU time. Due to the way $ALC is linked, it can happen that the user will ask $ALC for N words and instead get N+1. This happens when a request for N words would only leave 1 word of system available memory left over in a queue block. Since $ALC requires 2 words for its link structure and only one word would be left, $ALC gives the other word to the user to force him to keep track of it. Appendix B also shows how this one extra word is carried along if the need arises. It is the users responsibility to detect this condition and return the extra word when $RTN is called. As mentioned memory is allocated in contiguous chunks; however, $ALC is written so that SAM need not be contiguous memory. The disconnected blocks of memory are linked through the first two words of each block. A drawing of the linkage for RTE is shown in Figure 3-4 so that the reader will understand how the routine will work in the general case.

If a block size request comes into $ALC and the size requested is larger than any currently contiguous free block, then $ALC returns a flag to the effect. The calling routine is obliged to check for this condition and may place the program, on whose behalf the request was made, into the memory suspend state (state 4) via a $LIST call. If a program does go into the memory suspend list, then the number of words requested must also be posted into the second word of the ID segment.

On all $RTN calls a check is made of the suspend list after the memory has been added to SAM. If enough contiguous memory has become available to satisfy the highest priority program in the list (i.e. the first one in the list), then $LIST is called for every program in the suspend state until the end of the list or until a request length is found that is greater than the currently existing largest block of SAM. For example, if programs A and B are in the suspend list with priorities of 10 and 20 respectively but with block requests of 1000 and 100 respectively B will never be rescheduled until enough memory has been collected for A. The philosophy here is that he who has the highest priority should get resources first. Note, however, that any future $ALC requests that come in will be honored if there is enough memory. This allows programs of lesser or greater priority to continue and hopefully give block memory at a later date.

Calling Sequences:

1.

```
$ALC     (Allocate section)
(p)      JSB $ALC
(P+1)    (# words needed)
(P+2)    -Return-
```

On return:

(A) = FWA of allocated block, or = 0 if reject
(B) = # words allocated (may be 1 greater than # requested)

If no block is large enough to alloctae the requested length,
(A) = 0 on return.

2.

```
$RTN     (Return block section)
(p)      JSB $RTN
(P+1)    (FWA of buffer)
(P+2)    (# words returned)
(P+3)    -Return:  Registers meaningless-
```

There are no error conditions detected by these sections.

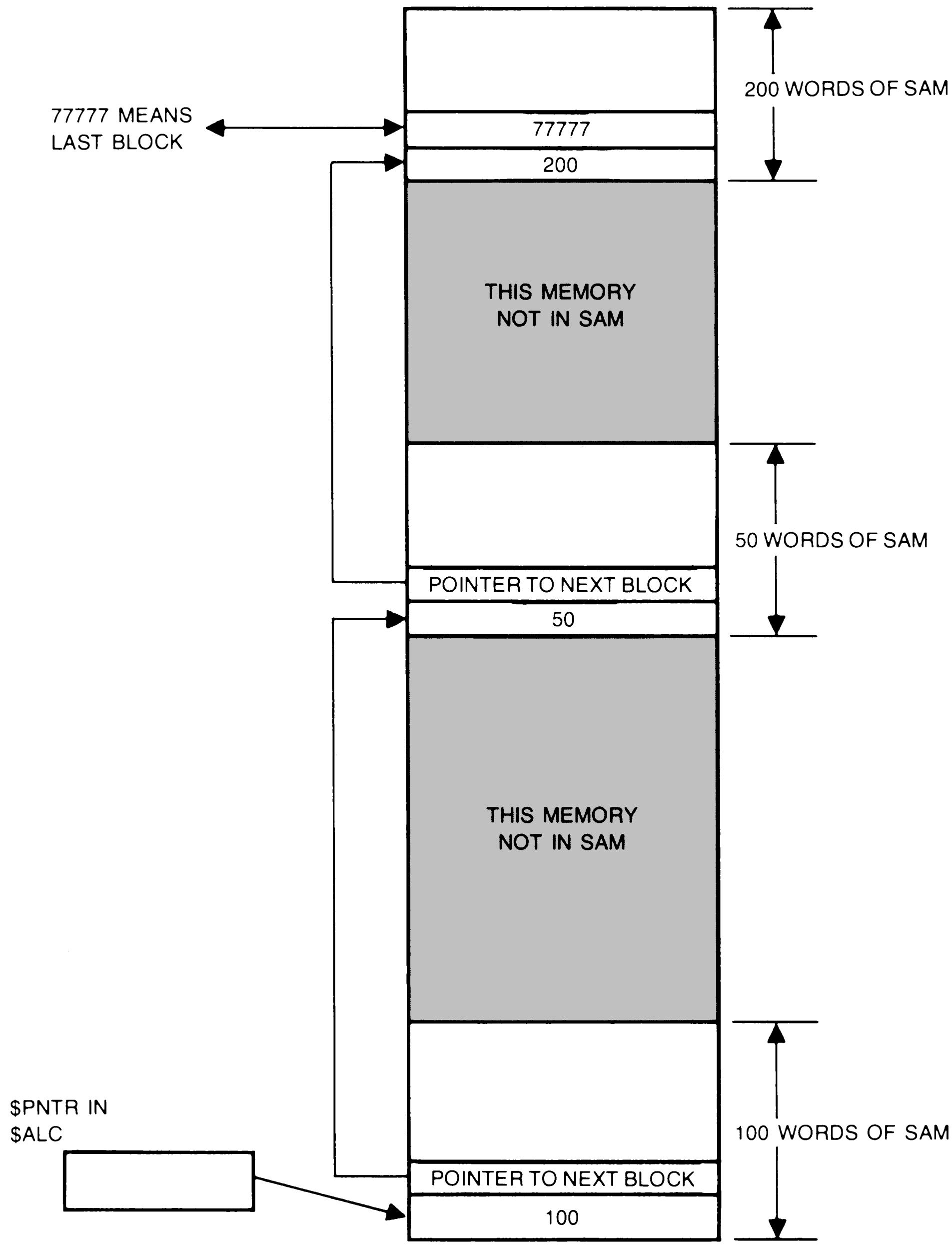


Figure 3-4. Example of SAM Linkage

Now suppose the user returns 35 words. See what SAM now looks like in Figure 3-5.

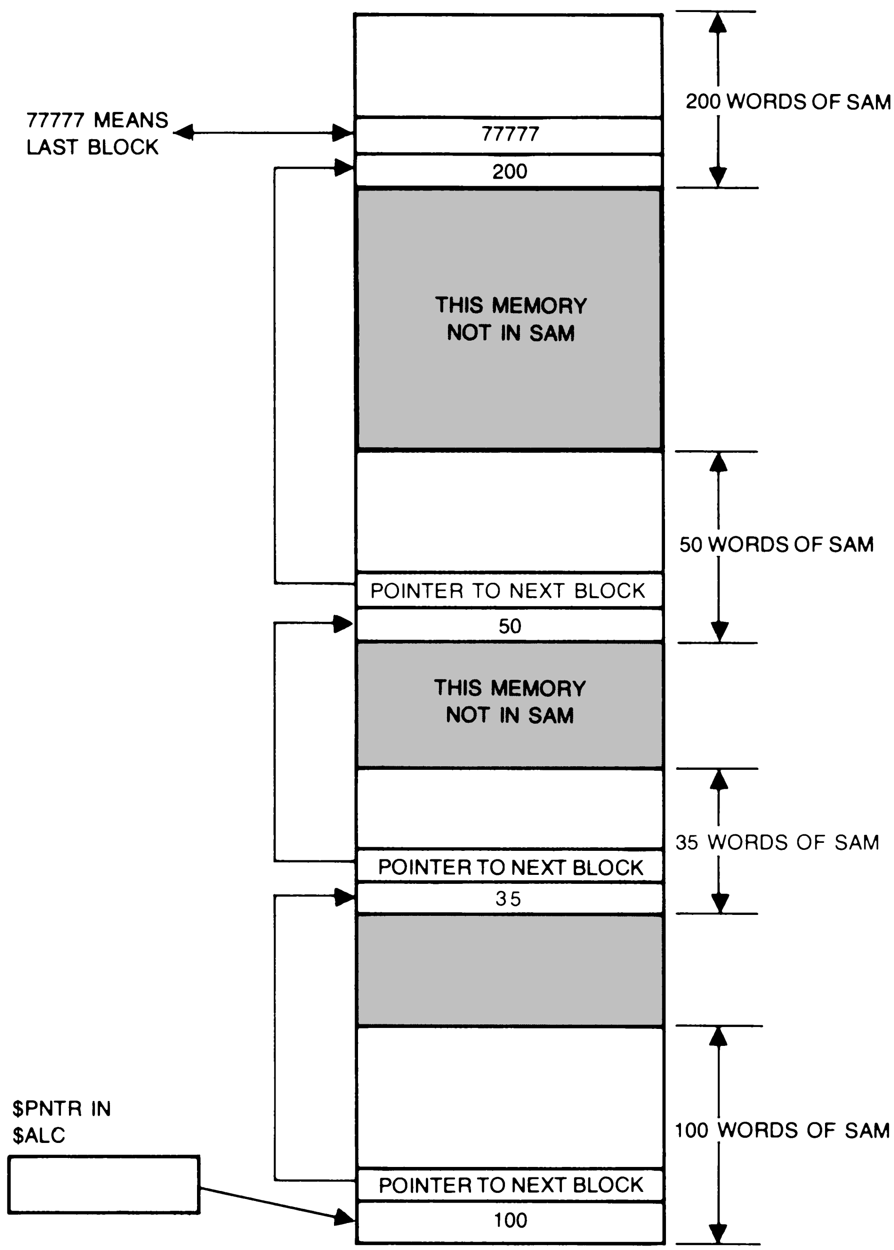


Figure 3-5. Example of SAM Linkage After Returning Memory

3.14 MAPOS

The operating system is divided up so that portions of it reside in mapped memory at all times. The remainder is divided into 2k chunks that reside in unmapped memory. All modules that are included in these chunks must have the first 3 characters of their name as "OSx", where x is a digit indicating which chunk number the module should be relocated. For example, modules with the names OS0XX, OS0YY, OS0ZZ will be relocated in the first chunk, chunk 0. The modules with this special naming convention are relocated as drivers in the driver partition area by the generator.

References to code within the OSx modules are intercepted by the MAPOS module. The $SYPT entry point contains the physical start page of the first module. Using this information, MAPOS maps the appropriate module in the driver partition and transfers control. By reducing the size of the system in logical memory, more space is made available for SAM.

To decide what portion of the operating system to include in an OSx type module, the following points must be considered:

1. Do not include code that is executed on every TBG tick.

2. Avoid including code that makes a call to a routine that does I/O requests. The driver partition mapping is changed when an I/O request is made. This implies the return from the subroutine call has to be trapped by MAPOS and cause a re-map of the OSx type module.

3. Avoid including code that makes a JMP or JSB to code included in another OSx type module that resides in a different 2k chunk. More than one map change while executing a given path in the operating system is very expensive with respect to time.

4. Do not include a very small piece of code that needs to be repetitively executed for a given operation.

5. The code and buffer area in each chunk should be a little less than 2k so that there is enough space for current page links and future modifications.

MAPOS contains a routine $MTAT which maps the Track Assignment Table in the driver partition area. This routine must be called if TAT needs to be examined directly (i.e., without going through the disc track allocation/release processor).

```
+--------------------------------------------------------------+-------------------+
|                                                              |                   |
|   SCHEDULER                                                  |   CHAPTER  4      |
|                                                              |                   |
+--------------------------------------------------------------+-------------------+
```

4.1 INTRODUCTION

The scheduler is the RTE-6/VM module which oversees program state transitions, responds to operator input commands, begins system start up at boot up, and satisfies or vectors to other processors eleven EXEC call requests (EXEC 6,7,8,9,10,11,12,14,22,23 and 24). All of this processing is done completely from within the system map.

Calls to the scheduler may come from either the user or other parts of the system itself and thus from either the user map or system map. For this reason a preamble to certain sections of the scheduler is found in the System Communication Areas on the base page in both maps. The entry points that start in the preamble are $LIST, $MESS, $IDNO, and $SCD3. In essence the purpose of this preamble is to get the current DMS status for return purposes, enable the system map, and jump to the appropriate processor. While this code is not specifically part of the scheduler, it is, so to speak, the front door.

The technical discussion on the scheduler which follows assumes that the reader is completely familiar with the 36 word RTE-6/VM ID segment and 5 word ID extension. For those who are not, Appendix A at the end of this manual contains a complete description of the ID segment.

4.2 ORGANIZATION OF THE SCHEDULER

The scheduler is divided into the following three parts:

1. The first part resides in the system map at all times and contains:

    a. This list processor.

    b. RT, OF, BR, LG message processors.

    c. Initial system start-up code.

    d. System console input/output section.

   e. Soft and hard abort routines.

   f. EXEC 8 processor for segment load and disc resident node load.

2. The second part of the scheduler resides in the second OS code partition (OS2SC module) and contains:

   a. ST, TI, TM, IT, LU, EQ, TO, DN, and BL command processors.

   b. EXEC request processors to obtain system real time and to put ID segment time values.

3. The third part resides in the third OS code partition (OS3SC module) and contains:

   a. Most of the initialization code.

   b. ON, SS, GO, PR, LS, AB, RU, SZ, AS, UR, UL, QU, EN, WS, VS, AG, SN, and CU message processors.

   c. Program completion, suspend, schedule, core-lock, and get/put string EXEC request processors.

4.3 LIST PROCESSOR

The list processor is a subroutine in the scheduler that is called to move a program from one state to another. In RTE-6/VM a program is always said to be in a state (sometimes a tizzy). The states are:

STATE NUMBER	STATE
0	DORMANT
1	SCHEDULED
2	I/O SUSPEND
3	GENERAL WAIT SUSPEND
4	MEMORY SUSPEND
5	DISC SUSPEND
6	OPERATOR SUSPEND

The state number is the number used in the status field (word 16) of the ID segment to indicate that a program is in a particular state. For each of these states, except the dormant state, a linearly linked list of all programs in that state is kept. The scheduler manages 5 of these lists. The lists and their heads are:

LOCATION	MAJOR STATE
1711	1 SCHEDULED LIST
1713	3 GENERAL WAIT LIST
1714	4 MEMORY SUSPEND LIST
1715	5 DISC TRACK WAIT SUSPEND
1716	6 OPERATOR SUSPEND

The I/O suspend state has a list headed at each EQT but these lists are managed by RTIOC not the scheduler.

Programs are moved in and out of these lists as their major state changes. The lists are maintained in priority order with the highest priority programs first. Programs of the same priority are added to the list behind the others of same priority. Each list is threaded through ID segment word 1 and is terminated with a zero.

Any number of things can cause a program to move from state to state. For example, suppose FMGR was executing, entering a *SS,FMGR on the system console would cause the system (list processor) to move FMGR from state 1 to state 6. Thus FMGR's status field would change from 1 to 6, word 1 of FMGR's ID segment would be taken out of the scheduled list and put into the operator suspend list.

There is no user interface to the list processor. All calls to the list processor come from other system modules. User requests are first processed in the EXEC or scheduler and then go to the list processor.

4.4 LIST PROCESSOR CALLING SEQUENCE

```
     JSB    $LIST
     OCT    (Address Code)(Function Code)
     DEF    (Address) <This word not always required>
ON RETURN
     If A = 0, then no message & B = PROG ID address
     If A not = 0, the A = ASCII error code address
        & B contains decimal error code

    Address codes of 0, 6, & 7 are reserved for drivers.
    The only function code allowed with these address
    codes is 1 (schedule)
    If successful A = 0 ELSE
                  B = 3 ILLEGAL STATUS
                  B = 5 NO SUCH PROG
```

For a driver that wants to convert a program name to an ID address:

```
JSB $LIST
OCT 217
DEF PNAME (Prog Name)
```

This performs a simple list move like changes to priority. (If the program is dormant it's a big NOP). Upon a successful return (A = 0) B will be the ID address of the program. If the program is scheduled many times, doing this removes the search time for the ID segment of the program.

Function Code

- 0 = Dormant Request
- 1 = Schedule Request
- 2 = I/O Suspend Request
- 3 = General Wait List Request
- 4 = Memory Available Request
- 5 = Disc Allocation Request
- 6 = Operator Suspend Request
- 16 = Special scheduler link program in front of priority list
- 17 = Relink Program Request
- 10 thru 16 are not assigned

Address Code

- 0 = ID segment address (5 parameters passed)
- 1 = ID segment address (as next octal value)
- 2 = ASCII program name address (a DEF)
- 3 = ID segment address in work (no DEF addr.)
- 4 = ID segment address in B-Reg(no DEF addr.)
- 5 = ID segment address in XEQ1 (no DEF addr.)
- 6 = ID segment address (Next parameter is value to put into B Reg at suspension
- 7 = ASCII program name (passes 5 parameters)

For example:

```
---0,7,&6 (Four Drivers)-------  ---1----  ---2----  ----3-----
-                             - -        -      - -         -

JSB $LIST    JSB $LIST    JSB $LIST  JSB $LIST JSB $LIST JSB $LIST
OCT 001      OCT 701      OCT 601    OCT 1XX   OCT 2XX   OCT 3XX
DEF RETRN    DEF RETRN    OCT IDADR  OCT IDADR DEF PNAME  ID ADR IN $W
OCT IDADR    DEF PNAME    OCT BVAL
DEF PRAM1*   DEF PRAM1*
DEF PRAM2    DEF PRAM2
DEF PRAM3    DEF PRAM3           *NO INDIRECT DEFS FOR ADDRESS
DEF PRAM4    DEF PRAM4            CODES 0 AND 7  IF THE DRIVER
DEF PRAM5    DEF PRAM5            IS IN THE USER MAP


     ---4-----        ------5--------
     -       -        -             -

     JSB $LIST        JSB $LIST
     OCT 4XX          OCT 5XX
      ID ADR IN B REG  ID ADR IN XEQT
```

The list processor breaks up the requests shown in the calling sequence into four general cases:

1. Dormant Request

2. Schedule Request

3. Operator suspend request

4. Non-operator suspend request

    a. I/O suspend

    b. Unavailable Memory suspend

    c. Unavailable disc space suspend

In general, before a call to the list processor is made other modules have done a considerable amount of error checking to see if the change is legitimate. These checks are of the nature "Does the program exist"? or "Were the parameters in the proper range"? etc. The list processor performs a "was-will be" check. That is, what was the last state; what will be the next state; are the two compatible? If the compatibility answer is yes, then the requested transition is made. If the answer is no, then the list processor decides on what the proper new state will be. In addition, one other answer can be made. The answer is "yes, but not now". In this case a bit is set to flag an action to be deferred. The R,D and O bits are deferred action bits in the ID segment.

4.5 DORMANT REQUEST

The transition processing by the list processor is done as follows:

A. If the abort bit is set then:

1. The 5 temporary ID segment words are cleared.

2. The program is placed into a push down stack, linked through word 9 of the ID segment, and headed at $ZZZZ in the dispatcher. (Refer to Appendix E for what the dispatcher does to this stack.)

3. XEQT is cleared (Base Page word 1717).

4. The entire status word is cleared and the CL bit.

5. If this is the currently executing program $PVCN, the privileged rest counter, is cleared.

6. Link processor is called to do the list move. (Link processor is discussed in the next section.

B. If the abort bit is not set and

1. Previous status is I/O suspend (state 2) or 0 bit set, then only set D bit and call link processor.

2. Save resource bit not set then go do A1 through A5 above.

3. If resource save bit set and 0 bit not set then CLEAR R&D bits, set status to zero; if this is not the currently executing program set the no parameters bit, and call link processor.

4.6 SCHEDULE REQUEST

The schedule request portion of the list processor checks actual program status information in the ID segment to see if the program is schedulable.

On a schedule attempt if the program's status is not 0, 2, or 6 then:

1. If dormant bit set jump to dormant request processor.

2. If the W bit is set, change the status field to 3 and call the link processor to put the program in the general wait list.

3. If not 1 or 2 above set entire status word = 1 this clears out all other bits; then

4. Call link processor to schedule program.

If the current status is 6 and ...

1. Dormant bit set too, then set status to 0, clear R&D bits, and call link processor to make dormant.

2. Wait bit set too, then change status to 3 (general wait) and call link processor to put program into general wait state.

3. Else call link processor to put program in scheduled list. That is done A1 through A4.

If the current status is I/O suspend, state 2,

1. If O bit set, and R or O bit set then change status field to a 6 and call link processor to make program operator suspended.

2. If D bit set jump to dormant request processor.

If the current status is 0, that is, first dispatch, then:

1. Perform C1 and C2 in case the program was in the time list and C on SS command set the O bit.

2. Check if the program is memory resident. If so, then go do A1 through A4.

3. The program is disc resident. Check if the current status is truly dormant and the program was not terminated saving resources (point of suspension is 0). If so, and if the program uses sharable EMA, increase the number of active programs entry in $EMTB for the program's sharable EMA partition by one.

4. If the program terminated saving resources or terminated serially reusable, or was operator suspended and the program is still in the partition (i.e., has not been swapped out or overlayed), then go to step 5 below, or go do A1 through A4 (if the above is not true).

5. If still in partition, then call the dispatcher routine $DMAL to set the partition up to be reused.

6. The final step is to force the programs timeslice word to a 1. This indicates that this is a re-dispatch or a new dispatch so the program is to receive a full timeslice. Then go do A1 through A4.

4.7 LIST CALLS BY DRIVERS

Certain $LIST calls have been set aside for use by drivers. These are list calls with function codes of 0, 6, and 7. The form of the call is:

```
JSB $LIST          JSB $LIST          JSB $LIST
OCT 001            OCT 701            OCT 601
DEF RETRN          DEF RETRN          OCT IDADR
OCT IDADR          DEF PNAME          OCT BVAL
DEF PRAM1          DEF PRAM1
DEF PRAM2          DEF PRAM2
DEF PRAM3          DEF PRAM3
DEF PRAM4          DEF PRAM4
DEF PRAM5          DEF PRAM5
```

For function codes of 0 and 7 up to 5 parameters may be passed. At least one parameter must be supplied. The five parameters are put into the XTEMP area of the ID segment and may be picked up by calling RMPAR.

The DEF RETRN must delimit the parameters and no indirect DEF's are allowed. For function code of 1, the ID address (IDADR) must be in the call. For function code of 7 PNAM points to a 3 word array containing the ASCII program name. For function code 6 BVAL is placed in word 11 of the ID segment, the B register at suspension.

Only schedule requests may be made. No other requests are allowed. Note that $LIST does almost no error checking for drivers and none for the op system. It is assumed that if you call $LIST you know what you are doing.

4.8 OPERATOR SUSPEND REQUEST

1. If the entire status word is 0 and the program is not in the time list or the status field = 6, then make an "Illegal Status" error return.

2. If current status field = 2, I/O suspend, then set 0 bit.

3. If status field = 0 (i.e. other bits =0) then set R&D bits, make status field = 6, and call link processor to make list move.

4. If not 1,2 or 3 above set status to 6 and call link processor.

4.9 NON-OPERATOR SUSPEND REQUEST

1. Put requested future status into status field of program's ID segment saving all the upper bits of the same word.

2. Call link processor to make list transition.

On return from $LIST

A = 0 means success
B = ID address of program referenced

else

A = ASCII error code address and
B = numeric error code
= 3 means illegal status (not dormant)
= 5 no such program

4.10 LINK PROCESSOR

The real-time executive Link Processor function is to remove a program from one list to add the program to another list.

When removing a program from a list, a check is made of the program status to see if it is in the I/O suspend list. NOTE: The I/O suspend list is not kept in SCHED, but is kept by I/O processor (RTIOC). Thus, if the program is in I/O suspend list, the program removal portion of the routine is bypassed. If program is not in the I/O suspend state, the removal request code value is used to compute the address of the "top of list" word for the particular list. If the program cannot be found in the list, or it is a null list, the program returns as if the action has been performed. This should be an impossible case. Assuming that the program is found in list, the action taken depends on where the program is in the list.

The removal of program from a list consists of:

1. If I/O list (code 2), then this is special case and does not require removal.

2. If NULL list, then error exit taken.

3. If first and only program in list, then list value set to zero.

4. If first program in list, but not the only program in list (linkage not

zero), then set list value to the linkage value.

5. If in middle of list, the linkage of the ID segment which points to the program to be removed is set to the linkage value of the program that is removed.

6. If last program in list, the linkage value of previous program in list is set to zero.

After the program has completed the removal portion of the routine, it can then be added to another list. The addition code value is examined to see if it is to be added to I/O suspend list, in which case return is made to calling program. Otherwise, the addition request code value is used to compute the address of the "top of list" word for the particular list. Programs are added to a list according to priority. The program is added to the list just prior to the program of lower priority. The program is added to the list in the following manner:

1. If I/O list (code 2), then this is special case and no addition made to list.

2. If NULL list, then list value set to point to id segment or program to be added and the linkage set to zero.

3. If not null list, the program is inserted into list according to priority level and linkages changed to reflect this insertion.

4. If a lower priority, than any program in list, then last linkage is set to point to the program to be added and the program linkage is cleared.

4.11 MESSAGE PROCESSOR

The operator input message processor, $MESS, accepts input commands programatically, generally through the system library routine MESSS or from the system console via the $TYPE routine.

The $TYPE routine is entered by an interrupt created by the operator striking any key on the system teletype. Upon entry, the system teletype ready flag is checked for busy. If the flag is busy, then control is given to $XEQ. If the flag is zero, then check the session mode flag.

If not in session ($ENBL=0), an asterisk (*) is output to the system teletype via $XSIO and a request for teletype input is made via $XSIO with the completion address TYPIO. The system teletype flag is set and control given to $XEQ. When the operator has input his request (signified by LF), the operator message processor routine ($MESS) is called. Upon return from $MESS, the A-register is checked for zero or non-zero. If non-zero, then a message is to be output from $MESS on the system teletype. The A-register contains the address of the buffer which contains the message. The first word of this buffer contains the number of characters to be output and the ASCII message begins at the next word. This message is output via $XSIO and teletype busy flag is cleared and control given to $XEQ. If the A-register is zero upon return from $MESS, the teletype flag is cleared and control given to $XEQ.

If in session ($ENBL not 0 and invoked by the "EN" command), then we look for a session control block defined for LU1. This is done by checking word 3 (session identified) of the first SCB in the list headed by $SHED. If this word is not a "1", we issue the LOGON prompt and start the read of the response ($XSIO with a completion address of SESIN). When the read has completed, the user response is sent to a communication program, $YCOM, in a string buffer. $SYCOM then transmits the request to the "LOGON" program to perform the actual log on. The session bit map (!BITM) is updated to indicate that a log-on is in progress for LU1, the system console busy flag is cleared and exit is to $XEQ.

If a session already exists for LU1, the break mode prompt ("S=1 command") is issued. The read of the command is then issued ($XSIO with a completion address of BRKIN), the system console busy flag is set, and exit is to $XEQ. When the input is complete, a check is performed to see if the command entered was an "OP" command. If not an "OPerator" command the command is sent to $YCOM who then sends the request onto RPN for processing. If the command was an "OP" control is transferred to the command processor who processes the command as if it was entered from the system console while not in session mode.

Why, you might ask yourself, do you go to so much trouble in the processing of the system console. The answer is simple -- you should never be locked out of the system console. For example, if the standard PRMPT and RPN processing were to replace $TYPE while the system console was enabled for session use, what would happen if you could not dispatch a program (Disc down, or priority 1 program in a tight loop)? Answer -- nothing! In this example the only course of action would be to reboot the system. With the OP command you simply enter a command and the problem is corrected.

NOTE: The following prompt is issued whenever standard processing cannot be performed (no memory for string, log-on started but not complete, etc.).

    S = ??COMMAND?OP,

When this prompt appears, the only command permitted is the "OP" command (note that the prompt contains the first part of the "OP" command as a reminder).

If the log-on or log-off process cannot complete (possibly no programs will run) commands may still be entered via this version of the "OP" command.

The entry point $MESS is in Table Area 1. It is a front end to the actual processing itself. It contains:

```
$MESS NOP
      SSM $MEU
      SJP $MSG
```

The entry point $MEU will then contain the DMS status of the system when the $MESS call was made. This status will be restored when $MESS returns.

$MESS is not a closed subroutine. For example, the OF command will cause a program to be aborted and the associated clean up code to be executed. The return is to the dispatcher not to the caller of $MESS.

The following things are done for calls to $MESS:

1. The command's existence is verified.

2. The command is parsed.

3. The command is dispatched.

The first of these operations is done by checking the transmission log. If zero characters were received, $MESS just exits.

If, upon entry to $MESS, character count is non-zero then the internal parsing routine is called and parses the entire operator input. The output of the parse routine is a 33 word internal buffer. The calling sequence and two examples are shown below:

The Parsing routine scans the ASCII input buffer and stores the data into parameter tables. Commas are used to flag separation of parameters. The character count from teletype driver is assumed to be in the B register upon entry.

A parameter may be up to six ASCII characters in length. There may be up to seven parameters and one operation code input with a maximum of eighty characters. As the input is scanned, a count of parameters and count of characters for each parameter is kept. Characters are stored left justified in the buffer. Word PARAM contains the parameter count and OP,P1,...,P7 contains the ASCII parameter values. The character count for each parameter is kept in word just prior to buffers. PARAM is kept as positive integer and character counts are negative integers.

4.12 SYSTEM PARSE ROUTINE

Calling sequence:

```
JSB $PARS
DEF PBUFR     33 word buffer for parsed output

A-REG = input buffer address
B-REG = positive character count
```

The parse routine will accept up to 8 parameters delimited by commas. Each parameter is parsed into 4 words where the first word describes the type of parameter. The format is shown below:

WORD	CONTENTS
1 (TYPE)	0 if null, 1 if numeric, 2 if ASCII
2	binary # if type = 1, 1st two ASCII chars if type = 2
3	used for ASCII only = 2nd two ASCII characters
4	used for ASCII only = 3rd two ASCII characters

Example:

PQ, P Q RST,55,,10B,556377X,ABCDEFGHIJ

Notes:

1. All blanks are ignored.

2. Any ASCII characters past the first 6 are ignored.

3. To enter ASCII 77 enter ,77 X, where X is any ASCII character.

After the command is parsed its existance must be verified. This is done by a table look up. The Table is at LDOPC and is just a simple list of ASCII opcodes. If the opcode is valid, then a jump is made through table LDJMP. Each entry in LDOPC has a corresponding entry on LDJMP. LDJMP contains the address of the various processors. Note how easy this makes adding new commands. One merely places the ASCII opcode into LDOPC and the address of the processor into LDJMP.

Commands not in the table are dispatched to a routine which returns the proper error.

Errors are returned to the caller of $MESS to be printed in the proper place (or not at all). Recall that $MESS can be called from a program via MESSS (see the library section of your manual).

4.13 SYSTEM STARTUP

When the user pushes the run button the final time on system boot up a jump is made to the $STRT routine in the scheduler. $STRT's job is to get the system going. This section of code is executed once and is later overlayed.

The first thing that the start up routine does is to set up the system map.

To begin with the first 32K of physical memory will be the system map none of which will be write protected. A JSB is then made to $CNFG, the slow boot routine. This will allow the user to reconfigure system available memory, I/O, and partitions. After this the slow boot returns to $STRT so that set up of the system map can be finished. This mapping routine uses the following information about system available memory.

```
                      1st PHYSICAL "CHUNK" of SAM
                      ---------------------------

            15            10   9                 0
            ----------------------------------------
$MPSA       | # of PAGES     | PHYSICAL START PAGE |
            ----------------------------------------
BP 1660       LOGICAL START ADDRESS
BP 1661       NUMBER OF WORDS


                      2nd PHYSICAL "CHUNK" OF SAM
                      ---------------------------

                          10   9
           -----------------------------------------
$MPS2      | # OF PAGES     |  PHYSICAL START PAGE  |
           -----------------------------------------

BP 1662    LOGICAL START ADDRESS
BP 1663    NUMBER OF WORDS
BP 1664    LOGICAL START ADDRESS
BP 1665    NUMBER OF WORDS
BP 1666    LOGICAL START ADDRESS
BP 1667    NUMBER OF WORDS
BP 1670    LOGICAL START ADDRESS
BP 1671    NUMBER OF WORDS
```

The first area of SAM, which is a minimum of 2 pages, is set up by the generator and does not change. Physically it is located directly behind the operating system. The second area is set up at generation time but is changable via $CNFG at boot up. It physically resides after the memory resident program area (i.e., before the first program partition). Note that the second area is divided into four pieces. This allows the user (with the slow boot) to work his way around any bad pages of memory that may exist within SAM.

While the two areas are not physically contigious, they will be made logically contigious. This is done by taking the physical page numbers of both areas of SAM and placing these numbers contigiously into the DMS registers corresponding to their logical address in the system map.

The number of words at the end of Table Area I up to the next page boundary are set up as the third area of SAM by the generator. This area is neither physically not logically contiguous with the first two areas. $RTN, the System Available Memory return routine, is then called to return SAM. Typically, it would look as shown in Figure 4-1.

The $STRT routine also initializes the contents of a few system entry points for later use by other system modules. The following entry points are set.

```
              |
              |
              |
              V
    -----------------------+----------------------+ 31K (Logical)
     Z = 32-(X+Y)          |                      |
       = Unused portion    |                      |
         of System Map     |                      |
    ---------------------  |  --------------------| Logical Stop
                           |                      |
              |            |    Second Chunk      |
              |            |       of SAM         |
      Y Pages of SAM       |                      |
              |            |                      |
              |            |                      |
    +--------------------  |  --------------------|*  -------
    |                      |     First Chunk      |     | 2+
    |                      |       of SAM         |     | Pages
    |                      |  --------------------|   -------^
    |                      |                      |          |
    |                      |      Operating       |          |
1st X pages                |       System         |  Logical|
of physical        ^       |                      |  Start
  memory           |       |----------------------|  --------------
    |              |       |    TABLE AREA 2      |          |
    |              |       |----------------------|          |$SDT2
    |              |       | SYSTEM DRIVER AREA   |          |
    |   Write Protected----|----------------------|*<-- $SDD ------
    |     from here up     |     COMMON/SSGA      |          |$COML
    |                      |----------------------|*<-- $CMST------
    |                      | DRIVER PARTITION #1  |
    |                      |----------------------|*
    |     from here up     |     Third Chunk      |
    |                      |       of SAM         |
    |                      |----------------------|*
    |                      |    TABLE AREA 1      |
    |                      |----------------------|*
    |                      |  SYSTEM BASE PAGE    |
    +----------------------+----------------------+  OK (Logical)
```

Figure 4-1. System Map for Startup

$CMST	Starting page of common. Logical and physical pages are the same for $CMST. $CMST = bits 14-10 of $DLP shifted down $DLP = disc resident program load point set up by generation
$COML	Number of pages of common $COML - bits 14-10 shifted down of [MPFI (3) + BGCOM-$DLP] Where: BGCOM = base page 1753 length of background common MPFI(3) = fourth entry in memory protect fence table. Start of background common.
$SDA	Starting page of system driver area. $SDA = $CMST + $COML Note that logical and physical pages are the same for $SDA.
$SDT2	Number of pages occupied by the system driver area and table area II. $SDT2 = Bits 14-10 shifted down of $PLD-$SDA Where: $PLD is the privileged program load point set up by the generator.
$RLB	Logical starting page of the memory resident library. $RLB = bits 14-10 shifted down of LBORG (Base Page location 1745) LBORG is the address of the library set by the generator.
$RLN	Number of pages in memory resident library. $RLN = bits 14-10 shifted down of [MPFI(1) - LBORG]

$STRT checks all pages of partitioned memory for uninstalled memory. If there are more pages in partitioned memory than the actual installed memory, $STRT halts with HLT 20B and the last page number installed is displayed in the A and B register.

$STRT calls the $ZZZZ routine in the dispatcher. At this time XEQT is cleared; the interrupt system is cleared; the memory protect fence register is set to 0, swap delay is set up; a check is made to see if there are background, real time, and chained partitions and if not the partition list headers are reset, and lastly FMGR is scheduled. This section of code is only executed once and is later overlayed. A return is made to $STRT.

Next, $STRT picks up the ID address of FMGR, D.RTR, and SMP. These addresses are used later by the system for various types of error checking.

The scheduled list of programs is checked for use of sharable EMA. If any of these programs uses sharable EMA, the number of active programs entry in $EMTB is incremented. $STRT then jumps to the EXEC to finish the system start up.

EXEC also saves the addresses of D.RTR and EDIT for error checking so that their disc tracks are not released improperly by the user. Tracks 0 to ($#TRK - 1) are set up in the Track Assignment Table, $TAT, as belonging to the operating system. The next track is assigned to D.RTR.

A last jump is made then to $SCLK in RTIME to start up the real time clock.

The $SCLK routine starts the time base generator, uses the RTIOC routine $SYMG to print out 'SET TIME' and lastly jumps to $XEQ in the dispatcher. The system is now ready to go.

4.14 EXEC REQUEST HANDLERS

Currently there are eleven EXEC REQUESTS involved in the scheduler. They are:

EXEC REQUEST #	PURPOSE	ENTRY POINT
6	Program Completion	$MPT1
7	Program Suspend	$MPT2
8	Load Background Program Segment	$MPT3
9	Schedule w/wait	$MPT4
10	Schedule w/o wait	$MPT5
11	System Time Request	$MPT6
12	Schedule at absolute time or with time offset	$MPT7
14	GET or put string	$MPT9
22	Program Swap Control	$MPT8
23	Schedule w/wait and w/queue	$MPT4
24	Schedule w/o wait and w/queue	$MPT5

Control is transferred to the entry points shown above from the EXEC. Briefly, the EXEC call creates a memory protect interrupt which goes to the $CIC routine in the RTIOC module. $CIC transfers control to EXEC after finding that the interrupt was due to memory protect. EXEC checks the parameters for various error conditions and if all is well transfers control to the appropriate entry point.

As can be seen from the table above many of the requests ultimately deal with the list processor. In general, the processors pull in the request parameters locally, check them for validity, and if the parameters are valid, a call to the list processor is made.

Four of these requests are briefly discussed here. The other requests are discussed in conjunction with other scheduler functions.

4.15 PROGRAM SUSPEND REQUEST

This is an EXEC 7 Request. The processor first checks the program's batch bit. If set, an SC00 error is generated and the program aborted. This is because programs under batch may not be suspended. If clear, $ALDM, which is a dispatcher subroutine that will move the partition out of the allocated list and into the dormant list, is called. Lastly, $LIST is called to operator suspend the program.

4.16 SEGMENT LOAD REQUEST

This is an EXEC 8 request. The processor first checks if it is a disc resident node load call. If so, the $NODL routine in the dispatcher is called to do the node load. If it is a segment load request and it is issued from an MLS program, then an SC12 error is issued. The processor then looks at the request count.

If bad, an SC01 error is generated. If OK, the system subroutine TNAME is called to get the ID address of the segment. If it is not found, an SC05 error is generated. The entry point address of the segment is then fetched and made the return address of the segment load EXEC call. $BRED in the dispatcher is called to do the actual load. Any parameters that are to be passed are placed in the temporary words of the ID segment. Control is then transferred to $XEQ.

4.17 SYSTEM TIME REQUEST

This is an EXEC 11 request. It returns the current system time. The time is kept in two words. ($TIME and $TIME+1) in Table Area II. Each bit corresponds to 10 MSEC with the most significant bits in the upper byte of the second word.

The scheduler checks input parameters for errors, picks up the time words and turns the rest of the processing over to the $TIMV routine. $TIMV formats the words into hours, days, minutes and 10ths of MSECs.

4.18 TIME SCHEDULE REQUEST

Only the request count and resolution codes are checked in the scheduler. GETID is called to get the programs ID address. All other processing is turned over to the $TIMR routine.

4.19 PROGRAM TERMINATION

In RTE-6/VM there are nine ways a user may terminate a program. In addition, the system may abort programs too. The user has three variations of the OF command, five variations of the EXEC 6 request, and the EXEC 12 REQUEST. Some of these may be grouped, however, in terms of what the system does.

1. TYPE 1 SOFT ABORT

    a. OF,PROG

    b. CALL EXEC (6,0,2)

2. TYPE 2 HARD ABORT

    a. OF,PROG,1

    b. CALL EXEC (6,0,3)

    c. SYSTEM ABORT

3. TYPE 3 REMOVE PROGRAM FROM SYSTEM

    a. OF,PROG,8

4. TYPE 4 TERMINATE SAVING RESOURCES

    a. CALL EXEC (6,0,1)

    b. CALL EXEC (12,...)

5. TYPE 5 TERMINATE SERIALLY REUSABLE

    a. Call EXEC (6,0,-1)

6. TYPE 6 NORMAL PROGRAMMATIC COMPLETION

    a. CALL EXEC (6,0,0)

The type 6, normal completion request, requires the least processing and is by far the most common of program terminations. First, SCHED checks if this is a deferred termination request. Deferred termination is used by the MLS loader when the profile action is requested. These checks are ignored and normal termination is performed if this is a father program terminating the son.

If the DE bit in word 22 of the ID segment is set and the NA or NS bit in word 16 is clear, then the program is not terminated. The location at program's load point + 7 contains the transfer address of the profile routine, .STAR. This address is placed into the program's point of suspension and control is transferred. If the NA and NS bits are set in word 16, then a normal termination is performed. The rest of the procession for a type 6 request is mostly done in the scheduler TERM routine.

Next, the TERM routine first calls the list processor to put the program dormant. If the father's waiting bit (FW) is set for this program, then the system finds the father and clears his W bit which was set, and if he is in state 3, the list processor is called to schedule him. It is possible that the father is waiting but is not in state 3. This would indicate that he is possibly dormant because his father made him dormant or that he is in another state with the W bit set. For this reason he is rescheduled only if he is in state 3. For other cases the list processor picks up the fact that he should be scheduled by the indication that was left by clearing the W bit. The TERM routine then clears all but the RM and RE bits.

PW and RN bits in word 21 of the program being put dormant, and returns. The RN bit of the ID segment indicates that the program has resource numbers. The RM flag indicates that it has re-entrant memory that has been moved. These resources will be released by DISPA when it finds the program linked into the abort list at "$ZZZZ" (refer to Appendix E for a description of this process).

Next, any optional parameters supplied in the termination request are placed in the 5 word temporary word of the ID segment. This allows the original scheduling parameters (or any others) to be picked up with the system subroutine RMPAR.

The SH bit in word 32 of the program's ID segment indicates that this program or its ancestor is using sharable EMA. If the SH bit is set, it is cleared.

Finally, the TERM routine checks the save resources (R) bit. If the R bit is clear and the point of suspension is not zero (i.e., the program's current state is not dormant or if the dormant, then it was terminated saving resources and an OF,prog,x command is issued), then set bit 2 of word 21 of the ID segment. This bit indicates to the dispatcher that if the program is using sharable EMA, the number of active programs entry in $EMTB must be decreased by one.

Next, bit 1 is set in word 21 of the ID segment to indicate normal termination.

This flag is checked when the ABORT processor in the Dispatcher calls $EQCL to handle EQT's locked to the program. Refer to the Equipment Locking Capability section in Chapter 2 for details.

This is the minimum processing for program completion.

The Type 1, soft abort termination, requires a little more processing. The soft abort starts with a call to the SABRT subroutine in the scheduler.

The first thing SABRT does is to clear the 'R' and D bit in the status word. This will force the list processor ($LIST) to truly put the program dormant. The system then calls $TREM (in OS2SC), which will remove the program from the time list. This clears the ID segments T bit.

The W bit is checked next, if set then this program is a father waiting for a son. (Recall that son's ID address is in word 2 of fathers ID segment.) In this case the sons FW bit is cleared. This insures proper processing when the son terminates.

The TERM subroutine, described earlier is next called.

Lastly the SABRT routine checks to see if this program is the son of another program. If so then a 100000B is placed into word 2 of the fathers ID segment and the address of word 2 is placed into word 11, the B register at suspension word. This allows the father to do a RMPAR call and to get back a word (the first of 5) that indicates that the son program was aborted. This is how FMGR, for example, knows to generate the "ABEND XXXXX ABORTED" message.

Next in order of processing is type 2, the hard abort. The hard abort is performed in the $ABRT subroutine. However, before calling this routine a check is made of the programs current status. If the status is I/O suspend (state 2) a jump is made to the RTIOC routine $IOCL.

Briefly, $IOCL CLEARS out any 'hang up' conditions caused by program input or output. It scans all the EQT's I/O linked lists looking to see if the program is in the list. (Linked through first word in ID segment). If any I/O is found the program is delinked and the I/O cleared. $IOCL then calls $ABRT to finish the abort.

$ABRT sets the abort ("A") bit in the programs status word (recall that we discussed this bit in the $LIST discussion). The "A" bit being set indicates a hard abort to $LIST and forces it to set the program dormant. $ABRT then calls SABRT which we just discussed. $ABRT then calls $SDRL in EXEC which releases any disc tracks the program owns, and, if any are released, calls $LIST to schedule all programs waiting for disc tracks. The exception here is that $SDRL will not release tracks belonging to D.RTR. After $SDRL returns, $ABRT sets up the program abort message and sends it to

$SYMG in RTIOC which will send it to the system console.

Next in order of processing is the power abort, type 3. Normally this is not done programmatically (call to $MESS), it is done with the OF command. The power abort calls $ABRT to do the hard abort first. The TM bit is next checked if the TM bit is set, it indicates that the program was loaded temporarily online, and there is no copy of its ID-segment on the disc. Only in this case can the OF processor clear the ID segment. The rest of the OF code computes the number and location of the tracks holding the program (words 23-27 and 36 of the ID segment) calls $DREL in EXEC to release the tracks. The OF request assumes an ID segment owns a track only if it references sector 0 on that track. This convention prevents double release of tracks in cases where background segments start in the middle of a track. Furthermore, $DREL will only release the tracks if they are owned by the system (i.e., it will not free FMP tracks). $DREL also reschedules any programs waiting for disc tracks by calling $LIST.

When $DREL returns, the OF routine clears the 3 name words (except for the SS bit, which indicates a short ID-segment, and the track assignment words), it releases any EMA ID extension, and then goes to $X

The type 4, save resources termination is a special case of the normal termination. In this case the dispatcher subroutine $ALDM is called. This routine unlinks the partition the program executed in from the allocated list and puts the partition into the dormant list. The $MATA entry D bit is also set. Next the R bit in the ID segment is set. This is done so that the list processor will not put the program in the clean up stack headed at $ZZZZ. (Refer to Appendix E) ($LIST will clear the R bit).

Now if this case is a father terminating his son then all that is left to do is a $LIST call to place the program dormant. The more general case, however, is the program terminating itself.

In this case the $WATR routine is called. All $WATR does is check the PW bit to see if any other program wants to schedule this program that is doing the save resources termination. If the bit is set then a search of the general wait list is made to see who is waiting. (Recall word 2 of the waiting program will have the prospective son's ID address. The prospective son is now doing the save resources termination). If the prospective father can be found a $LIST call is made to reschedule him. This allows the schedule request to be reissued. The rest of the processing is done exactly like the normal program termination.

Lastly, the serially reusable completion, type 5. A check is made to make sure a father is not terminating a son as serially reusable. If this is detected, normal termination results. If the program is terminating itself, the TERM subroutine is called. Next the least significant bit of the father ID number word is set as a flag to the dispatcher clean up routine that the programs partition is not to be put in the free list. $ALDM is then called to take care of the partition. Any optional parameters supplied are placed in the ID segment temporary area.

4.20 PROGRAM SCHEDULING

There are four ways to schedule a program in RTE-6/VM. The program can be scheduled by time, event, operator command, or another program.

NOTE: If a session program is in the time list, the following access restrictions are enforced:

- EXEC schedule or program time value requests referencing a program in the time list may only be issued by another program of the same session. An attempt to reference a session's time scheduled program by another session or a non-session program will result in an SC11 error.

- The session operator commands IT, RU and ON have the same access restrictions as described above. Attempts to reference a time scheduled program belonging to another sssion will result in an "ILLEGAL STATUS" error.

NOTE: The above restrictions apply to session programs and operator commands only.

To schedule a program by time the program must have been in the time list already. (This would require the operator ON request earlier). Every time the time base generator interrupts control is transferred to the $CLCK routine in the RTIME module. Here every program in the time list (threaded through ID word 17) is checked to see if it is time to execute. If words 19 & 20 of the ID segment equal the system time stored at $TIME & $TIME+1 and if the program is dormant, a call is made to the list processor to schedule the program. Regardless of program state, the next start time is calculated and stored back into the ID segment. (The new time is not computed if the multiple value is 0. This means the program is to be removed from the time list.)

Scheduling by event is typically done by drivers. DVR00 and DVR05 for example, schedule the program PRMPT due to an event, that is, an interrupt. This scheduling is done by a $LIST call.

The ON and RU commands are another way to schedule a program. These two commands differ in that the RU command will schedule a program now regardless of the time list parameters. The ON command is capable of putting a program in the time list and/or scheduling the program immediately. In both cases a call is made to $LIST to do the scheduling.

Before the $LIST call is made the program is checked to see if it is dormant. If not an "illegal status" message is returned. If the 'IH' was not entered in the schedule command and parameters are allowed on schedule (i.e. NP bit Clear), then any parameters supplied with the command are put into a string block in system available memory. The first five of the parameters are placed into the temporary words of the ID segment. (String processing is discussed in the next section.) In the case of the RU command the $LIST call is made next and that's the end of the RU processing.

The ON processor looks at the programs ID segment resolution code to determine the next process. If the resolution code is 0, only a $LIST call is made. If the resolution code is not 0 then the $ONTM processor finishes the processing. Basically $ONTM checks for the NO (NOW) in the command. If present then the program is put into the time list and executes at the current system time and 10 milleseconds. If the NO is absent $ONTM places the program into the time list. The program then executes at the time specified in words 19 and 20 of it's ID segment.

The last way to schedule a program is programmatically (EXEC 9, 10, 23 and 24 requests). The processing here is somewhat more involved than the ON or RU commands because a father son relationship is involved. Most of the processing is done in the IDCKK subroutine. The routine does the following:

1. Makes sure the program exists, else generates an SC05 error.

2. Makes sure the name specified is not a segment name, else generates an SC05 error.

3. If a program is scheduled with wait and the father has the SH bit set in word 32 of its ID segment, then set the SH bit in the program's ID segment. This bit indicates that the program itself or one of its ancestors is using sharable EMA, and it must be able to run in a non-sharable EMA partition to prevent dead locks.

4. Makes sure the program will find a partition large enough to execute in, else generates an SC09 or SC08 error.

5. Places perspective son's NP bit and bits 0-3 of status field into the perspective father's A-Register at suspension word.

6. Calls the string passing routines if necessary. (i.e., if RQP9 = 0 no string passing.)

7. Makes sure that the first five optional scheduling parameters are put into the sons ID temporary words.

For Exec 9, 10, 23, and 24 requests, the RU, ON, SZ and AS commands, the $SZIT subroutine is called to see if a partition exists that is large enough to execute the program. Thus insuring that a program scheduled is dispatchable. For memory resident programs the check is ignored.

The program size is determined using the procedure described for the AS and SZ commands in the Scheduler Interface With Dispatcher section of this chapter. The program size is checked as follows:

pages =< $MBGP/$MRTP for a non-MLS, non-EMA, background/realtime program.

pages =< $NBGP/$NRTP for a non-MLS, background/realtime program using sharable EMA or a descendant of a sharable EMA program.

pages =< greater of $MBGP/$MRTP or $MCHN for a non-EMA, MLS, background/realtime program or a program using local EMA.

pages =< greater of $NBGP/$NRTP or $NCHN for an MLS, background/realtime program using sharable EMA or a descendant of a sharable EMA program.

If the program is using sharable EMA, the EMA size is checked against the sharable EMA partition. If this partition is down, the number of active programs using it is zero and the lock bit is not set. The scheduled program is then aborted with an SC09 error.

Alternatively, if the program is assigned to a partition (RP bit in ID segment set) then the partition # field is used as an index into the $MATA table to see if the destination partition is large enough for the program and if the partition is still defined. (Note programs already in memory with an allocated partition may not have their sizes changed. The SZ operator request error check routine guards against this.)

It may also happen that the maximum applicable partition size is larger than a 32k address space. In this case the check # of pages =< MAX ADDRESS SPACE is used.

If the check fails, an SC09 or SC08 error (SIZE ERROR) will result. However, if the DE bit (EMA default) is set then the EMA size is reset and the check is performed again. If the check now passes all is well and the EMA size of 1 will be used by the dispatcher as a flag to give the program the largest possible EMA size.

If the reader has already read the sections on the AS and SZ commands, the question may come up, "Why check for size, this is already done in the LOADR and for on line commands?", the reason is that the FMGR 'SP' and 'RP' commands allow the user to save programs whose size or assignment may not match the currently defined partitions. The error checking prevents a mismatch of program and partition from causing system problems.

NOTE that every time a program is scheduled, $MCHN, $MBGP, &MRTP, $NCHN, $NBGP, or $NRTP (or the destination partition size) is used as a check to see if the program can fit into a partition. If $MCHN, $MRTP, $MBGP, $NCHN, $NBGP, or $NRTP = 0, then no partitions of that type is available and the program is not dispatchable.

This may happen if a parity error causes a partition or partitions to become undefined. Should the scheduler detect this condition, the program will not be scheduled and an SC08, SC09, or 'SIZE ERROR' will be reported to the system console.

4.21 STRING PASSING

Upon scheduling a program with the RU, ON or GO commands, a section of system-available-memory (SAM) will be allocated for storage of any command string and entered in a push down stack linked through the first word of each block (see Figure 4-2). The head of the stack will have the name $STRG and reside in the SCHED module. A command string is defined as everything following the prompt in a scheduling call.

If the program is scheduled by a RUIH,ONIH, or GOIH, then the string storage portion of the command will be inhibited. The first word of each block of memory will contain a pointer to the next memory block. The last block of memory in the stack will contain 0 in its link word. The second word of each block of memory will contain the ID address of the scheduled program. The sign bit, when set, will indicate that the memory block has an additional word (see system description of the memory allocation routine, ($ALC)).

The third word of each block will contain the character count of the command string. The fourth through (N+1+3)/2 words will contain the N characters in the command string.

Upon scheduling a program with the RU, ON or GO command, the following steps will occur at parameter storage time:

1. If there is no parameter string, continue at Step 5.

2. Store parsed parameters into ID segment words 2 to 6 as before.

3. If the command is RUIH, ONIH or GOIH, then do not store paramter string and continue at 5.

4. Deallocate any string block(s) associated with the scheduled program.

   Allocate a block from SAM, store the entire command string into the block and enter it into the stack. If SAM is not available, then the request is ignored, the following error message is issued to the operator's terminal:

   CMD IGNORED - NO MEM

and control is returned to the system at $XEQ.

5. Schedule the program for execution.

The user can retrieve the string by using the EXEC 14 request or the system library routine GETST. Both routines release the string memory back to the system. Alternately, programs can still recover the first five parameters (treated as one computer word each) by using the RMPAR call as the first call in the program.

Any time a program goes dormant, normally or abnormally, any command string block assigned to the program will be returned to SAM. This is accomplished in the ABORT routine of the dispatcher.

4.22 SCHEDULER INTERFACE WITH DISPATCHER

Several portions of the scheduler interface to the dispatcher. The list processor portion of the scheduler interfaces on program scheduling. The list processor also interfaces with the dispatcher on program completion as described in Appendix B. In addition, the UR, AS and SZ operator commands affect the dispatchability of a program. The error checking for these commands is discussed below.

```
   $STRG
 +-------+     +---------------+    +------------+    +------------+
 |   *---|---->|         *-----|--> |      *-----|--> |    0       |
 +-------+     +---------------+    +------------+    +------------+
     +-------->| | ID address  |    | |          |    | |          |
     |         +---------------+    +------------+    +------------+
     |         | N characters  |3   |            |    |            |
     |         +---------------+    +------------+    +------------+
 Extra word bit | char  | char |4   |            |    |            |
 set if $ALC   +---------------+    |            |    |            |
 returns N+1   |               |    |------------|    |------------|
 words and only|               |    |            |    |            |
 N requested   +---------------+
               | char N        | (N+1)/2 + 3
               +---------------+
               | extra word    |
               +---------------+
```

Figure 4-2. Stacking of Memory Blocks

The AS and SZ both require the program referenced to be dormant and not memory resident. Moreover, the program must not still own the last partition in which it executed. (Recall that a serial reusable, save re termination, operator suspension does not release the partition.) The partition # field of word 22 is used as on index into the $MATA tabl and the $MATA residency word is checked to make sure the referenced prog no longer owns the partition. If any of these conditions are not met the "ILLEGAL STATUS MESSAGE" is output.

The AS command does not allow a program to be assigned to a sharable EMA partition or a part of one. Therefore, if a mother partition is a sharable EMA partition, a program may not be assigned to any of its subpartitions. If a subpartition is a sharable EMA partition, a program may not be assigned to its mother partition.

Other error checking is performed for the AS command. The partition must exist. Next, the size of the program is checked against the size of the referenced partition as follows:

1. For non-MLS, non-EMA programs,
   # pages (word 22) <= partition size.

2. For MLS, non-EMA programs:

   a. No memory or disc resident nodes:
      #pages (word 22) <= partition size

   b. Disc resident nodes only:
      #pages (word 22) <= partition size

   c. Memory resident nodes only:
      [#pages in memory resident nodes and root (word 34)
      + #pages in dynamic buffer area (word 35)]
      <= partition size

   d. Memory and disc resident nodes:
      [#pages in memory resdient nodes and root (word 34)
      + #pages in longest path of disc resident nodes (word 34)
      + #pages in the dynamic buffer (word 35)]
      <= partition size

3. Programs using local EMA:

   a. For non-MLS, MLS programs without nodes, or
      MLS programs with disc resident nodes only:
      [# pages (word 22) - MSEG + EMA size (word 29)]
      <= partition size

   b. For programs described in 2.c and 2.d above:
      [P1 + EMA size (word 29)] <= partition size

      where P1 is the number of pages obtained using
      formulas in 2.c and 2.d above.

4. Programs using sharable EMA:

   a. For non-MLS programs, MLS programs without nodes, or MLS programs with disc resident nodes only:
      [# pages (word 22) - MSEG] <= partition size

   b. For programs described in 2.c and 2.d above:
      P1 <= partition size

      where P1 is the number of pages obtained using formulas in 2.c and 2.d above.

If at the end of all the error checking, the AS command is determined to be valid, then the RP bit is set and the partition # is set into partition # field.

(Partitions count from 0. That is; AS,PROGX,7 will result in a 6 being placed into the partition # field.)

The SZ command processor performs the program partition and size checks mentioned earlier plus a few more. Word 30 of the program ID segment for segmented programs and MLS programs with disc resident nodes only or word 24 for non-segmented programs is used as the lower limit of the error check. For MLS programs with memory resident nodes only or memory and disc resident nodes, the lower limit is:

```
        # pages in memory resident nodes + root (word 34)
      +
        # pages in longest disc resident node path (word 34)
```

The upper limit is defined by the program type as described in the Program Scheduling section of this chapter.

If the program is assigned to a partition, then

new SIZE-1 <= ASSIGNED PARTITION SIZE

If the size is found to be valid then the # of pages field is updated to reflect the new size. (Note that the # of pages field does not include base page.)

NOTE also that $MRTP, $MBGP, $MCHN, $NRTP, $NBGP, $NCHN, or the partition size is not used if the MAX address space is smaller than these values. That is, a program plus the associated system tables may not exceed a 32k address space.

EMA programs have a special form of the SZ command (i.e., SZ,PROG,P1, P2). As mentioned earlier checks for partition and program status are made. Other checks are also made. The DE bit, word 1 of the ID extension must be set to change EMA size or the command is invalid.

In this case P1 is the new EMA size and P2 is the new MSEG size. P1 is checked as:

> P1 + PROG CODE SIZE <= $MCHN or assigned partition size if the program uses local EMA.
>
> or
>
> P1 <= sharable EMA partition size if the program is using sharable EMA.

P2 is checked as:

> P2 + PROG CODE SIZE <= PROG Address space

If both of the above are satisfied P1, the new EMA size is placed into the EMA size field of word 29 of the ID segment and P2 is placed into the MSEG field of the 1st word of the ID extension.

The last operator command that affects partitions is the UR command. This command clears the R bit in the referenced partition's $MATA table entry.

This command may affect the system entry points $MCHN, $MBGP, $MRTP, $NCHN, $NBGP, or $NRTP. These entry points contain the size of the largest unreserved partition of that type (i.e. Mother, background and real time).

If a partition is being unreserved and it would then be the largest unreserved partition of its type then $MAXP will be called to do the appropriate updating.

4.23 SHARABLE EMA

The sharable EMA partition entries are kept in the $EMTB table:

```
15            8 7           0
-----------------------------
¦      -  # entries         ¦ 0
-----------+-----------------
¦     L       ¦     A       ¦ 1
--------------¦--------------
¦     B       ¦     E       ¦ 2
--------------¦--------------
¦     L       ¦partition  # ¦ 3
--------------¦--------------
¦L ¦          ¦current  # of¦ 4
¦  ¦          ¦active users ¦
--------------+--------------
¦          NOT USED         ¦ 5
-----------------------------
```

$EMTB has 5 words per entry where:

LABEL	is the sharable EMA partition name.
partition #	is the sharable EMA partion number.
L	is the lock bit. The sharable EMA partition is locked if the L bit is set.

The sharable EMA partitions are defined at generation or configuration time. Programs may not be assigned to a sharable EMA partition or a part of one (i.e., if a mother partition is a sharable EMA partition, a program may not be assigned to any of its subpartitions. If a subpartition is a sharable EMA partition, then a program may not be assigned to its mother partition). The relation between the program and its sharable EMA partition is set up by the index # field in the program's ID segment extension. It is the index number into $EMTB for the sharable EMA partition entry.

A count for the number of active programs using a sharable EMA partition is kept by the schedule request list processor. If the program being scheduled is truly dormant and was not terminated saving resources, then increment the number of active programs count by one. This count is decremented by the dispatcher abort processor when the program using the sharable EMA partition terminates. The $TERM routine in the scheduler sets bit 2 of the program's ID segment word 16 if the program is not terminating saving resources and an already dormant program is not being terminated. (Bit 0 of word 16 is set if the program is terminating serially reusable. Bit 1 is set upon normal termination. This bit is checked when handling EQT's locked to the program.) The dispatcher abort processor decrements the number of active programs count for the sharable EMA partition if the terminating program has bit 2 set in word 16 of the ID segment and the program uses sharable EMA. If this count drops to 0 and the sharable EMA partition is not locked (L bit is clear in the $EMTB entry), the dispatcher releases the partition to be entered to the free list.

If a parity error occurs in a partition that is currently being used for sharable EMA, the partition is downed and the program that accessed the parity error causing location is aborted. The rest of the programs using this partition are allowed to run and new programs using this sharable EMA partition may also be scheduled. When the number of active programs using this sharable EMA goes to 0 and the partition is not locked, the sharable EMA partition is released and taken out of the free list. Future programs that are scheduled using this sharable EMA are now aborted with an SC09 error.

RTE-6/VM PARITY ERROR MODULE	CHAPTER 5

5.1 PARITY MODULE OVERVIEW

The Parity Error module's main task is to report parity errors detected by the hardware and to continue operation of the RTE-6/VM system if possible. PERR6 also tries to reproduce parity errors to identify and warn system users of soft parity errors: errors which may be intermittent or may be generated erroneously.

5.2 EXTERNAL COMMUNICATION

The Parity Error module communicates with the rest of the operating system through the system tables, base page communication area, and subroutine calls to other modules in the system.

5.3 SYSTEM TABLES REFERENCED

The System tables used by PERR6 are:

a. ID Segment entry for accessing program status.

b. $MATA table for accessing partition configuration information.

c. INT table for determining PORT map status.

5.4 SYSTEM BASE PAGE COMMUNICATION

XMATA	1646	Address of current MAP entry
INTBA	1654	Address of interrupt table
EQT1	1660	Address of current EQT entry
XEQT	1717	Address of current program ID Segment entry.

5.5 EXTERNAL SUBROUTINES CALLED

$CNV1 - convert number to ASCII (one word)
$CNV3 - convert number to ASCII (three words)
$ERMG - used by PERR6 to print "PE" error message and abort user program
$MAXP - reestablish maximum size words of unreserved partitions
$YMG - print message on system console
$UNPE - unlink a partition entry from the proper list and undefine the partition.

5.6 OTHER EXTERNAL REFERENCES

$CIC - entry point to Central Interrupt Control routine (contains address of last point of interrupt).

$DMS - two word save area.
word 1 - DMS status at last interrupt
word 2 - Interrupt status at last interrupt
0 if ON, 1 if OFF.

$XCQ - entry point of Dispatcher. This is used instead of the return point at $CIC when a program is aborted.

$IDLE - entry point to idle loop. This is used to check if the parity error occurred during a DMA transfer while in the idle loop.

5.7 DETAILED TECHNICAL ASPECTS OF OPERATIONS

This portion of the Technical Specifications is a detailed description of the major portions of the Parity Error module, PERR6. It is assumed the reader is familiar with the detailed operations of the Dispatcher (DISP6) and the I/O module (RTIOQ).

5.8 PARITY ERROR DETECTION

Because parity error interrupts can occur even when the interrupt system is off, the code at $CIC must be able to save the complete system status. The major hole in being able to save the complete state is in saving the interrupt system state. In order to do this in both the 21MX and the 21XE the instruction 103300 was used to both test the interrupt system and turn it off.

Parity error interrupts may be generated at almost anytime because DCPC transfers may be stealing memory access cycles. If it occurs while the system is in the idle loop, $CIC can not save the registers in XA, XB, etc. because all of these are actually one location. It was necessary for $CIC to identify the source of interrupt before saving all the registers. Only the A-register needs to be saved temporarily so that LIA 4 and a LIA 5 can be done. PERR6 is entered only when LIA 4 = 5 and LIA 5 = 1xxxxx. This is also detected by the operating system firmware that transfers control to the PERR6 module through $VCTR.

PERR6 saves all registers in local locations. It requires that 2 words be set up at entry point $DMS by $CIC. The first word being the DMS status register contents containing the memory protect status and mapping information. The second word indicates the status of the interrupt system at the last interrupt (the parity error interrupt). The logical parity error address from the violation register is saved. The contents of location 5 are saved and replaced by a JSB indirect through a base page location to a PERR6 routine.

5.9 PARITY ERROR VERIFICATION

The routine TRYPE is called to test if the parity error is in the system map. [The DMS status word cannot be used to determine the map under which the parity error occurred because certain DMS instructions change maps in the course of their execution and do not change the DMS status register.]

TRYPE saves the map indicator value and then re-enables the parity error system. If the system map is needed, a regular load is done from the logical address of the parity error. The next instruction is executed if there is no parity error at tested location. If the user map is needed, a cross-map load instruction is used to read from the logical address of the parity error. The next instruction is executed if no parity error is detected. CLF 5 is used to turn off parity error until another verification attempt is made. A NOP is needed between the XLA LOGPE,I and the CLF 5 because of timing delays required by the HP 1000 M, E, and F Series computers.

If a parity error cannot be reproduced in the system map an attempt is made in the port maps. The user map is saved before the port Maps are checked. The interrupt table is checked to see DCPC channel 1 is busy. If it is, the Port A map registers are copied into the user map. The TRYPE routine is called to try and reproduce the parity error. If no error is found the next DCPC channel is tested in the same manner.

After both DCPC channels have been tried without success, the user map registers are restored and TRYPE is called once again. The user map is tried last to avoid an erroneous report in the case where a swap out was taking place in one of the port maps. The user map may still contain a copy of the same user (left over from the set up for the port map by RTIOQ).

5.10 PARITY ERROR RECOVERY PHILOSOPHY

While it is possible to always detect the occurrence of a parity error, it is not always possible to effect a complete recovery from a parity error. There are a number of reasons why 100% recovery is not possible; these will be explained below. The overriding philosophy is to maintain system Operation whenever possible and eliminate, if feasible, the possibility of future parity errors.

5.11 WHO DUNNIT?

When a parity error is detected, the violation register records the logical address of the word containing bad parity. The P-register saved in the interrupt handler's entry point may or may not point to the instruction which caused the bad location to be referenced. This is especially difficult to trace back when the instruction was a multiple word instruction such as XLA, MVW, or DLD. So while we may verify that a location in the system contains bad parity, we cannot determine that a user program caused the reference to the bad location via use of a XLA instruction.

5.12 THE SUDDEN BLOW

A parity error detected during a DCPC transfer while the system map was enabled means the operating system was executing and it is a privileged system. Since the system may still be in RTIOC following a DCPC initiation, in the DISPATCHER in the EXEC abort routine, or in the system console driver; these routines would have to be reentered to print parity error messages or abort a program. So these are not recoverable.

5.13 IT'S AN INSIDE JOB

A parity error detected within the operating system itself may cause erroneous execution of the system. For example, if a parity error was in a JMP instruction, it is possible the P-register may not get set correctly. This type of error is also not recoverable.

5.14 SOFT PARITY ERROR

If a parity error cannot be reproduced (by reading a word at the logical parity error address in the system map, Port A map, Port B map, and user map) then it is considered to be a soft parity error. This type of error usually indicates an equipment problem: There may be intermittent memory parity errors, it may be a memory controller/ backplane problem, or even a firmware error.

Soft parity errors cause a message to be printed which gives the logical parity error address and the DMS status register contents at the time of the interrupt. These messages should help indicate where intermittent failures may be located, especially if these soft parity error messages become more frequently reported.

5.15 SYSTEM PARITY ERROR

Parity errors in memory locations in the system itself cannot be recovered as described in Parity Error Recovery Philosophy section. The system is halted (102005) with the A-register containing the physical page number and the B-register containing the logical parity error address. The table areas and system COMMON areas are also considered to be part of the system.

5.16 USER PROGRAM PARITY ERROR

Parity errors within the memory resident area will cause the program to be aborted. The physical page number and the logical parity error addresses are printed on the system console in addition to the program abort message. The system then continues operating.

Parity errors within disc-resident programs are retried with alternating ones and zeros written in three different patterns. If the parity error cannot be reproduced, the page number is checked against the entries in a table of pages where soft parity errors have already occurred. If found in this table, the soft parity error is treated as a reproducible (or hard) error. If not found in the table, the page number is entered in the table and the program is aborted with the parity error messages, as is the case for memory-resident programs.

Reproducible (hard) parity errors within a disc resident program require the partition or partitions affected to be undefined. The program's MATA (Memory Allocation Table) entry is examined to see if it is in a regular partition, a subpartition, or a mother partition.

If the parity error is detected in a program in a regular partition or a subpartition, an attempt is made to check if the physical page number of the parity error is actually within the partition's physical page definition. If the page is not in the partition, the error is treated as if it were in the system area and halts (102005). If the page is in fact part of the partition, the partition MATA entry address is saved. The partition is then unlinked from any partition lists and is undefined by a call to $UNPE. If there is a Mother partition, $UNPE is also called to undefine that partition.

If the parity error is in a program which occupies a Mother partiton. The partition MATA entry address is saved. Then a search is made through all of its subpartitions to see which subpartition is also affected. That subpartition's MATA address is then saved and the subpartition is removed from the system by $UNPE. $UNPE also releases all the other subpartitions back into the appropriate regular partition free list.

Finally the partition number or numbers are printed out as being downed. Then the program is aborted along with the parity error messages as in the case for memory resident programs.

5.17 DCPC PARITY ERRORS

If a parity error is verified to have occurred under a DCPC transfer, the DMS status register is checked (this is almost the only time when DMS status register can reliably indicate the correct map which was enabled at the time of the parity error interrupt). If the system was enabled at the time of the interrupt, a halt (103005) is necessary because the operating system must not be reentered. If a user or the idle loop was interrupted, the I/O request currently queued on the EQT which had the DCPC channel is examined. If the request was a system or buffered request, a halt (102005) is done. If it was a user request, the parity error is treated as in the case of a user program parity error (see the System Parity Error Section).

SYSTEM LIBRARY ROUTINES — CHAPTER 6

his chapter describes the RTE-6/VM system library routines. The routines are grouped functionally as:

I/O Requests and Related Routines

Program Status Routines

Conversion Routines

Session Related Routines

System Calls and Entry Points

A last group contains those routines that do not clearly fall into one of the functional areas. Within each group, the routines are presented in alphabetical order.

6.1 I/O REQUESTS AND RELATED ROUTINES

The I/O routines consist of the following:

```
$SUBC      ISSR
%SSW       ISSW
%WRIS      JFDVR
%WRIT      LDTYP
.STIO      LOGLU
.TAPE      LUTRU
BINRY      MAGTP
CHEL       PTAPE
DSCPR      REIOD
EQLU       SREAD
EQTRQ      SYCON
FNDLU      TRMLU
IFDVR      XREIO
IFTTY
```

6.1.1 $SUBC Function

Entry point: $SUBC

$SUBC extracts the subchannel from an EQT for the benefit of the calling routine. This is meant to be a standard interface to minimize changes when the subchannel field changes. Currently, the subchannel is:

EQT 4 - Bits 0-4 of subchannel - in bits 6-10

EQT 6 - Bit 5 of subchannel - in bit 2

Calling parameters:

On entry: none

On exit : A Register contains the subchannel
B Register is not changed
E Register is not affected

6.1.2 %SSW Function

Entry point(s): %SSW

External references: ISSW

FORTRAN callable. Passes the address of an argument to ISSW. Returns with the setting of the switch register in A rotated to put the image of the switch to be tested into the sign position - A(15)

6.1.3 %WRIS Subroutine

Entry point(s): %WRIS,%WRIN,%WEOF

External references: EXEC

This routine will write source data on an RTE disc in LS format. %WRIS is used by compilers, editors, assemblers to write source data onto a disc such that it can be reread for another pass of the source. The tracks are owned by the calling program, which should release the tracks when they are no longer required.

Calling sequences:

```
      JSB %WRIN      INITIALIZES
    <ERROR RETURN>   NO DISC SPACE: A-REGISTER=-1
      <RETURN>       A-REG = !15  DISCLU  8!7  TRACK#  0!
```

If buffer length = 0, embedded file mark is written
If buffer length > 0, true end-of-file mark is written
If buffer length < 0, -(BUFLN-1)/2 words are written

```
      JSB %WRIS      WRITES RECORD ON DISC
      DEF *+4        GOOD RETURN
      DEF BUFFR      POINTER TO 1ST WORD OF BUFFER
      DEF BUFLN      NEG. NUMBER OF CHARS IN BUFFER
    <ERROR RETURN>   NO DISC SPACE: A-REGISTER=-1
      <RETURN>       A-REG. = LAST WRITTEN LU/TRACK

      JSB %WEOF      WRITES OUT AN END OF FILE MARK
      <RETURN>       A-REG  = LAST WRITTEN LU/TRACK
```

Return:

A-Register: Disc LU in bits 7-8 (LU = 2 or 3);
Track number in bits 0-6 (track = 0-255), or -1 if no track available

The %WRIN entry point is in this routine primarily to re-initialize a new file write to the disc. The %WEOF entry point is to write a file mark and post the In-Memory buffer. A file mark write with %WRIS will write a file mark, but will not post the possible In-Memory buffer. Be aware that you should always specify an even character count or pad an odd character count with a trailing spece when writing a record. This routine will write ASCII records on program-owned tracks of an RTE system in LS format. The base page LS pointer, however, is not set.

Errors: The track return from %WRIS is not recoverable, therefore any tracks previously written should be returned to the system.

6.1.4 %WRIT Subroutine

Entry points: %WRIT,%WRIF,%WBUF

External references: EXEC,$OPSY

This routine writes relocatable records on disc. %WRIT is used by compilers to write the relocatable records it produces in the RTE LG area. The format on disc is the same as the paper-tape format.

Calling sequence:

```
JSB %WRIT  (ALL INITIALIZATION DONE BY SYSTEM)
DEF *+3
DEF BUFFR  FIRST WORD ADDRESS OF WRITE BUFFER
DEF RLEN   ADDRESS OF NUMBER OF WORDS TO WRITE
<RETURN>   P+4

JSB %WRIF  POST ANY PARTIAL RECORD IN MEMORY
<RETURN>   P+1
```

The system will abort the calling program with an I006 error if the LG area was not defined, or an I009 error if the LG area overflows. NAM relocatable records must always start on a sector boundry. TRherefore, when an END relocatable record is written, the entry point %WRIF must be called to post to disc any partial record still in memory.

6.1.5 .STIO Subroutine

Entry point: .STIO

Calling sequence:

```
  LDA SC       A=SELECT CODE (LOW 6 BITS)
  JSB .STIO    INVOKE SUBROUTINE
  DEF RTN
  DEF IO.1[,I] POINTERS TO LOCATIONS TO CONFIGURE
  DEF IO.2[,I]
     - - -
  DEF IO.N[,I]
  RTN < --- >  RETURN POINT
UPON RETURN, A IS UNCHANGED, B IS ??
```

On return, the A-Register is unchanged, the B-Register content is unknown. This routine is used to configure a driver for the select code currently in use.

6.1.6 .TAPE Subroutine

Entry point: .TAPE

External references: EXEC

This routine is used to initiate tape operations for FORTRAN compiled programs. When .TAPE is invoked, the A-Register contains 030XYY, where

```
X=4 --- REWIND
X=2 --- BACKSPACE
X=1 --- ENDFILE
YY ---- LOGICAL UNIT NUMBER
```

6.1.7 BINRY Subroutine

Entry points: BREAD,BWRIT

External references: EXEC,$OPSY

The BINRY subroutine is called to transfer information to or from a disc device.

Binary Read/Write routines: BREAD/BWRIT

Calling sequence:

```
JSB BREAD(BWRIT)
DEF *+7
DEF A      = FWA OF BUFFER
DEF N      (NO. OF WORDS)
DEF LUN    (LOG.UNIT NO.)
DEF TRACK
DEF SECTR
DEF OFSET (OFFSET IN SECTR)
(RETURN)
```

FORTRAN call: CALL BREAD(A,N,LUN,ITRAK,ISECT,IOFST)

6.1.8 CHEL Subroutine

Entry point: CHEL

External references: $ELTB

The check equipment lock routine allows a driver or other program to determine whether a given equipment is locked or not. It is up to the calling routine to take appropriate action based on that knowledge. This routine is appended to the caller's memory space.

Calling sequence:

```
 LDB   EQT# Whose lockedness (state of lockedupivity) is sought
 JSB   CHEL
(RETURN)
```

On return:

(A) = 0, If specified EQT. is not locked, else
(A) = locker's ID segment address.

Sequence of events:

1. Check which map is currently enabled (used to examine the EQT locking table via direct loads or cross loads).
2. Examine the EQT locking table ($ELTB) for locked EQT's.
3. If table empty, return.
4. Check entry for requested EQT.
5. If found get ID-segment address from table and return.

6.1.9 DSCPR Subroutine

Entry point: DSCPR

External references: .ENTR,.GOTO,ABREG,LDTYP,XLUEX

Purpose: Get disc parameters for a given lu

Calling sequence:

```
CALL DSCPR(LU,PARM,ISTAT)
```

Where:

LU   Disc LU to get the parameters for. LU need not be in user SST.

PARM   Ten-word word integer array to receive disc parameters (see below for format of PARM).

ISTAT   Status returned here: -1 ==> XLUEX error or LU is not a disc 0 ==> Disc parameters returned in PARM.

Layout for PARM array:

word	meaning	disc type:	9885A	7900	MAC	ICD	CS80	PAIRED DISC LU
1	addr		x	x	x	x	x	n/a
2	unit		n/a	n/a	n/a	x	x	n/a
3	volume		n/a	n/a	n/a	n/a	x	n/a
4	cyl	\ starting	n/a	x	x	x	x	n/a
5	head	> block	n/a	x	x	x	x	n/a
6	# surfaces/ for CS80		n/a	1	x	x	x	n/a
7	# tracks		n/a	x	x	x	x	x
8	# spares		n/a	0	x	x	0	n/a
9	# sectors/track (*)		60	x	x	x	x	x
10	Reserved							

(*) 64 word sectors

Fields marked 'n/a' are not applicable for that particular kind of disc and should be ignored. These unused fields are set to zero.

LINUS disc parameters will be returned, but the interpretation of the various paramaters is different in some cases. See the appropriate driver manual.

6.1.10 EQLU Function

Entry point: EQLU

External references: .ZPRV

EQLU is used to find the logical unit number of a device given the address of word 4 of its equipment table.

Calling sequence:

```
LDB EQT4        (Passed from DVR00/DVR65)

JSB EQLU    -or-    JSB EQLU   -or-   CALL EQLU (LUSDI)
DEF *+2             DEF *+1
DEF LUSDI

A-REG. = 0   if not found -or-
A-REG. = the logical unit number if found
LUSDI  = returned same as A-register
B-REG. = ASCII 00 or LU in ASCII
```

Sequence of events:

1. Set up loop for DRT scan.
2. Get next DRT entry and extract EQT from DRT
3. Determine EQT4 address.
4. Match current request?
5. Yes, return LU number in A-register and optional parameter
6. No, go to step 2.

6.1.11 EQTRQ Subroutine

Entry point: EQTRQ

External references: $LIBR,$ERAB,$XEQ,$LIST,$PVCN
LUTRU,$CVT3,MESSS,$ELTB,$SCD3,$DRNT

The equipment lock feature allows a program to exclusively lock the equipment (controller) associated with a given LU. Any other program is put in the wait list when it either requests a lock on effectively the same equipment, requests a lock when there is (temporarily) no place to dock the lock (i. e., the locking table is full), or attempts I/O through an equipment that is locked (to someone else). when the equipment is eventually unlocked, attempt will be made to schedule the waiting program. When a program terminates normally (i. e., through an EXEC 6 call with the not saving resources option), all equipment locked to it will be released. If the termination is saving resources, equipment locked stay locked to the program's ID segment address. If a program terminates abnormally (defined as the complement of the meaning of "normally" above), and had specified a lock on abort option in the locking request, the equipment stays locked--though not to this program per se. Another program (father? clone??) can then relock it and proceed to use it. Unlocking of an equipment is also done with this same call.

Calling sequence:

```
     LU    DEC LU # whose associated eqt. is to be (un)locked
     IOPT  DEC OPTIONWORD (described below)


     CALL  EQTRQ (IOPT, LU)     *with IOPT bit 14 clear
     Return point----

                    -or-

     CALL  EQTRQ (IOPT, LU)     *with IOPT bit 14 set
     GO TO Error routine
     Return point----
```

Bit assignments in the option word are as follows:

```
    Bit   0    1/0   lock/unlock

                     No abort on call      /
         14    1/0   Error, return ASCII  /  Abort on error
                     Code in (A) & (B)   /
```

In addition, for the lock request:

```
    Bit  13    1/0   keep EQT    /  release
                     locked on  /  on
                     abortion  /  abortion

         15    1/0   without wait  /  with wait
```

(1) The abort errors for this call are:

Mnemonic	Meaning
EQ00	Illegal LU specified (maps into system console EQT)
EQ01	non-existent LU specified (LU specified > lumax)

(2) On return from lock without wait:

(A) = 0 If successful, or EQT already locked to this program (locking a bit bucket is always successful, and results in a big fat NOP) also, (B) = locked EQT #

-1 If equipment lock table full,

1 If EQT. associated with specified LU locked to another program.

2 If EQT. associated with specified LU has one or more associated LU's locked to another program.

(3) On return from the lock with wait request:

(A) = 0 and (B) = locked EQT #.

If the equipment specified is locked to another program, or the equipment lock table is full, the calling program is put in state 3 (general wait) until the request can be fulfilled.

(4) On return from unlock:

(A) = 0 If successful,

-1 If EQT associated with specified LU was not locked to begin with,

1 If EQT associated with specified LU locked to another program,

2 If EQT is busy with locker's I/O.

Sequence of events:

1. Go privileged
2. Check for valid system LU issue error message through $ERAB in EXEC6.
3. Decode option word and perform table ($ELTB) update.
4. Return through $XEQ

6.1.12 FNDLU Subroutine

Entry point: FNDLU

External references: $DRNT

Routine to find the logical unit number of a device given the address of word 4 of its equipment table

Calling sequence:

```
LDB EQT4       (Passed from DVR00/DVR65)

JSB FNDLU    -or- JSB FNDLU  -or-  CALL FNDLU (LUSDI)
DEF *+2          DEF *+1
DEF LUSDI

A-Reg. = 0   if not found -or-
       = the logical unit number if found
E-Reg. = 0 if device is up -or-
       = 1 if device is down (all other regs invalid)
X-Reg. = possible RN# bypass word
Y-Reg. = device type (isolated)
LUSDI  = returned same as A-reg.
B-Reg. = ASCII "00" or logical unit in ASCII (i.e. "16")
```

Sequence of events:

1. Set up loop for DRT scan.
2. Get next DRT entry and extract EQT from DRT. If done return.
3. Determine EQT4 address.
4. Match current request? No, go to step 2.
5. Yes, check if type '00' or '05' and subchannel 0.(save in Y)
6. Extract lock word from DRT part III.
7. Check status (up/down) from EQT5 and set E-reg.
8. If device up, determine RN lock word and set in X-reg.
9. Return LU number in A-reg. and optional parameter

6.1.13 IFDVR Subroutine

Entry point: IFDVR

External references: .ENTR,XLUEX

This routine examines an LU and determines if it is a DVA32 or DVR32 LU, by knowing that only DVA32 processes its own timeouts. The first time IFDVR is entered, it makes an EXEC 1 (Get track map) request to ensure that the driver has been entered. This gives the driver an opportunity to set the timeout-processing bit. This routine alwo works in a non-session environment.

Calling sequence:

```
CALL IFDVR(LU)

LU:  the LU whose EQT needs to be examined
```

```
Return: -1 ==> DVA32  0 ==> DVR32
```

6.1.14 IFTTY Function

Entry points: IFTTY,.TTY,XFTTY

External references: XLUEX

The routine IFTTY is used to determine if the specified LU is interactive. XFTTY may called for extended LU's. .TTY is an alternate entry for IFTTY.

Calling sequence:

```
IFLAG = IFTTY(LU)          JSB IFTTY  (or)  JSB XFTTY
                           DEF *+2
                           DEF LU
```

Returns:

```
IFLAG = A Reg = -1       If the LU is interactive
              =  0       If the LU is non-interactive
              =  1       If LU is not in SST

        B Reg = upper byte = device type
                lower byte = subchannel number
```

Sequence of events:

1. Get LU in question?
2. Make no abort EXEC 13 call to get status.
3. If abort return set A-reg to 1 and return.
4. Check for driver type.
5. If type 00 return as interactive.
6. If type 05 or 07 and subchannel is 0, return as interactive.
7. Else return non-interactive.

6.1.15 ISSR Subroutine

Entry point: ISSR

Sets the S-register

Calling sequence:

```
 CALL ISSR(IVAL)     or     JSB ISSR
                            DEF *+2
                            DEF IVAL
                           <RETURN>
```

Where: IVAL is the value to set in the S-register

6.1.16 ISSW Function

Entry point(s): ISSW

Returns with the setting of the switch register in A rotated to put the image of the switch to be tested into the sign position - A(15)

6.1.17 JFDVR Subroutine

Entry point: JFDVR

External references: .ENTR

This subroutine inspects an LU and decides if it is a DVM33 or DVR33 by checking the distance (in octal) between the initiation address and the continuation address. If this distance is less than 1000B words, the driver is DVR33 (the actual number is 311B). If the distance is 1000B or greater, the driver is DVM33. These values allow room for expansion (or shrinksion) in both drivers.

Calling sequence:

```
CALL JFDVR(LU)
```

Where: LU = the LU whose EQT needs to be examined

Return:

```
 0 ==>  DVR33
-1 ==>  DVM33
```

6.1.18 LDTYP Function

Entry point: LDTYP

External references: .ENTP,.GOTO,ABREG,IFDVR,JFDVR,XLUEX

Calling Sequence:

```
TYPE=LDTYP(LU[,NTYP[,IA,IB]])
```

Where:

LU — Logical unit to be identified. Must be in the users SST. To pass a system LU, complement the sign bit. If sign bit is not set, and LU is not in SST, TYPE is -1000B (see below) and IA and IB will contain ASCII I012.

TYPE — Logical device type returned here. If positive, it is a 2 character mnemonic. If it is negative, it is error code (-1000B) or the negative device type returned by the exec(13) call (-1 to -77B).

NTYP — Optional parameter. This is a numeric device type that gives a further breakdown of the device type than TYPE.

IA,IB — Optional parameter. If specified, A and B registers are returned here after the exec(13) call.

Device types:

ASCII	Numeric	Meaning
BI	0	Bit bucket
TT	100	Interactive device
TP	120	Paper tape punch
TR	140	Paper tape reader
TA	200	Cartridge tape unit
TA	220	9-track mag tape
LI	240	CTD

PR	300	Printer
PR	320	26XX terminal accessory printer
DI	600	9885A floppy
DI	610	7900 disc
DI	620	ICD disc
DI	630	MAC disc
DI	640	TOCS disc
DI	650	Mirrored disc drive
PM	700	PROM

6.1.19 LOGLU Function

Entry point: LOGLU

This routine finds the LU number from which the program originated.

Calling sequence:

```
LU = LOGLU(SYSLU)            JSB LOGLU
                             DEF *+2
                             DEF SYSLU


LU = A REG = Number of LU at which RU or ON was entered -or-
             if scheduled by a father, LU at which father was
             scheduled.
   = 1       If program scheduled by interrupt or time list.
   = B REG = ASCII LU# (session terminal LU)
   = SYSLU = System LU of session terminal if in session -or-
             -LU if not in session
```

6.1.20 LUTRU Subroutine

External references: $DLUS,.ENTP,.ZPRV

Entry point: LUTRU

The routine LUTRU translates a session or batch LU into a true system logical unit.

Calling sequence:

```
CALL LUTRU(LUTST,ISYS,ISCB) or I=LUTRU(LUTST)
```

```
Where:  LUTST= The logical unit to be tested
        ISYS = Location for return of result
        ISCB = If supplied, test specified LU against this SCB.
```

```
Returns:  ISYS and/or (A) = True system LU  -or-
                            -1 if LUTST not defined for this session
                      (B) = 0
```

Sequence of events:

1. Isolate LU and check for <= 0.
2. Extract SST length word from id-segment
3. Call SWTCK to get system LU from users SST.
4. Return

6.1.21 MAGTP Functions

Entry points: IEOF,IERR,IEOT,IWRDS,LOCAL,ISOT,RWSTB

External references: .ENTR,EXEC

Performs utility functions on the Mag Tape. See the RTE-6VM Relocatable Library Manual for calling sequences and purpose of each routine.

6.1.22 PTAPE Subroutine

Entry point: PTAPE

External references: EXEC,.ENTR

This routine positions the mag tape. A backspace file leaves the tape at the beginning of the file. An End-of-Tape condition causes an immediate return

Calling sequence:

```
          CALL PTAPE(UNIT,#FILES,#RECORDS)

Where:    UNIT     = EQT ordinal for EXEC call
          #FILES   =  >0 for forward  -or-
                      <0 for forward
          #RECORDS =  >0 for forward  -or-
                      <0 for reverse
```

Note: A file mark encountered during record spacing is counted as one record.

6.1.23 REIO Subroutine

Entry point: REIO

External references: .DFER,$LIBR,$LIBX,EXEC,.ENTR,$OPSY

This routine performs reentrant I/O if the user buffer is 13 or more words above the program load point. This restriction is enforced because the user buffer is used as a TDB for the reentrant processor, and thus three words (plus 2 for the save X and Y register words and 8 for the user program preamble) are required above it. The three words are saved locally and the TDB is set up. After the I/O has completed, the words are restored.

If the buffer is too close to the load point, the I/O is performed in the standard manner. This is also true if the buffer is more than 129 words long (this is to conserve system memory).

NOTE: For memory resident programs, the buffer must be five or more words above the progam load point.

The calling sequence is the same as the EXEX I/O call without the track/sector words.

6.1.24 SREAD Subroutine

Entry points: %READ,%JFIL,%RDSC

External references: $OPSY,EXEC

This routine reades the source device or disc if LU=2. SREAD is used by compilers, editors and assemblers to read source from devices or from the RTE source disc file area, LS.

Calling sequence:

```
JSB %JFIL  Initialize for :JF or *LS pointer
<RETURN>


LDA LUTRK  Initialize for given disc LU/track
JSB %RDSC
<RETURN>


JSB %READ  Default :JF,*LS if %JFIL,%RDSC not called
DEF *+5
DEF LUN    LU of input device
```

```
          DEF BUFFR  Pointer to first word of buffer
          DEF RLEN   -(# char in buffer)
        <EOF RETURN> End of file return (disc only)
          <RETURN>   A-REG = !15 DISCLU 8!7 TRACK# 0! LAST READ
                     B-REG = Character transmission log (pos.)
```

The B Register will return zero if the End-of-Tape is read, or an embedded disc file mark is read. An even character count is always returned when reading the disc. The %JFIL and %RDSC entry points may be used to re-initialize (rewind) a read from disc.

6.1.25 SYCON Subroutine

Entry point: SYCON

External references: .ENTR,XLUEX

The SYCON routine writes a message to the system console (LU 1)

Calling sequence:

```
JSB SYCON
DEF *+3
DEF IBUF      Buffer to be written
DEF IBUFL     Buffer length
```

6.1.26 TRMLU Subroutine

Entry point: TRMLU

TRMLU is called to find the logical unit number of a device given the address of word 4 of its equipment table.

Calling sequence:

```
 LDB EQT4       (Passed from DVR00/DVR65)

JSB TRMLU    -OR- JSB TRMLU -OR- CALL TRMLU (LUSDI)
DEF *+2           DEF *+1
DEF LUSDI

A-REG = 0   If not found -or-
A-REG = The logical unit number if found
LUSDI = Returned same as A-reg.
B-REG = ASCII 00 -or- LU in ASCII
```

Sequence of events:

1. Set up loop for DRT scan.
2. Get next DRT entry and extract EQT from DRT
3. Determine EQT4 address.
4. Match current request?
5. Yes, check if type 00 or 05 and subchannel 0.
6. Yes, return LU number in A-reg. and optional parameter
7. No, go to step 2.

6.1.27 XREIO Subroutine

Entry point: XREIO

External references: .DFER,$LIBR,$LIBX,XLUEX,.ENTR,$OPSY

This routine performs reentrant I/O if the user buffer is 13 or more words above the program load point. This restriction is enforced because the user buffer is used as a TBD for the reentrant processor and thus three words (plus 2 for save X and Y register words and 8 for the user program preamble) are required above it.

These three words are saved locally and the TDB is set up. After the I/O has completed, the words are restored. If the buffer is too close to the load point, I/O is performed in the standard manner. This is also true if the buffer is more than 129 words long. (This is done to conserve system memory.)

For memory resident programs, the buffer must be five or more words above the program load point.

The calling sequence is the same as the XLUEX I/O call, without track/sector words.

6.2 Program Status Routines

The program status routines consist of the following:

BNGDB	IFBRK
COR.A	LIMEM
COR.B	PNAME
GETST	PRTN
GTID#	RMPAR
IDGET	

6.2.1 BNGDB Function

Entry point: BNGDG

This routine is used to determine if the current program is being debuged by the symbolic debugger.

```
BNGDB: <0 (ie. true ) if being debugged.
BNGDB: >=0 (ie. false) if not being debugged.
```

6.2.2 COR.A Subroutine

Entry point: COR.A

External references: .ZPRV

This routine is used to find the address of the first word of available memory for a given ID segment.

Calling sequence:

```
LDA IDSEG     Get ID segment address to A
JSB COR.A     Call this routine
```

Return:

A = First word of available memory (MEM2 from ID)

6.2.3 COR.B Subroutine

Entry point: COR.B

External references: .ZPRV

The COR.B routine returns the first word address of free memory for a main program, this address is high main + 1 for a non-segmented program, and high largest segment + 1 for a segmented program.

Calling sequence:

```
A reg = id segment address of main program

JSB COR.B
```

Returns:

```
A reg = 0 if normal return  -or-
        -1 if error return (B reg is meaningless)
B reg = FWA of free memory for main program
```

COR.B makes an error return if the id segment address passed is that of a short id segment

Sequence of events:

1. Check for short id-segment --> error
2. Get value of high main + 1 from id-segment
3. Get value of high largest segment+1 word from id-segment
4. If value from step 3 is non zero return that otherwise return value from step 2
5. Return

6.2.4 GETST Subroutine

Entry point: GETST

External references: EXEC,.ENTP,.ZPRV

GETST is a FORTRAN callable subroutine that can be used to retrieve any parameter string from a command string that follows the second comma (third if the second parameter is NO and NOW). Only the first 80 characters of the command string are checked.

Calling sequence:

```
      JSB GETST
      DEF RTN
      DEF IBUFR
      DEF IBUFL
      DEF ILOG
RTN   ...
      .
IBUFR BSS N          Buffer to store string in.
IBUFL DEC N(-2N)     Words(+) or chars(-) to transfer.
ILOG  BSS 1          Transmission log.
```

Return:

```
<B>=ILOG = Positive number or words(chars)transferred.
         = 0 implies no buffer found.
```

Sequence of events:

1. Make EXEC 14 call to retrieve the runstring.
2. Set source and destination byte addresses.
3. Call GETCH to scan for first character after two commas.

4. Check for NO or NOW, if found, throw away.
5. Save current source buffer address and set up loop to start transfer of either transmission log or user count, whichever is less.
6. Set transmission length and return.

6.2.5 GTID# Subroutine

Entry point: GTID#

The routine GTID# computes the ID segment number of a program. Note that no error-checking is performed.

Calling sequence:

```
LDB ID-SEGMENT ADDRESS
JSB GTID#
```

Return ID number in B register

Sequence of events:

1. Look through key word block until requested ID segment is found.
2. Return difference between start of key word block and found ID segment as the ID number.

6.2.6 IDGET Function

Entry points: IDGET,ID.A,IDSGA

External references: .ZPRV

This routine obtains the address of the ID segment of the name given. If the name is null, IDGET finds the blank IDSEG address.

Calling sequence:

```
IDSEG = IDGET(NAME)
```

Where:

NAME = Three-word ASCII (5 chars) buffer with program name
IDSEG = ID segment address of name.

Return:

A-REG = ID segment address of name if found, 0 if not found

E-REG = 0 if name found, 1 if name not found.
B-REG = 0

6.2.7 IFBRK Function

Entry point: IFBRK

External references: $LIBR,$LIBX

This routine tests then clears the break flag.

Calling sequence:

```
  IF(IFBRK(IDMY)) 10,20
```

WHERE: 10 = Branch will be taken if set, and will clear it
20 = Branch will be taken if not set

```
   JSB IFBRK
   DEF *+1
  <RETURN>    A-Reg = -1 if set, else = 0
              (Break bit always cleared if set)
```

6.2.8 LIMEM Subroutine

Entry point: LIMEM

External references: EXEC,.ENTP,.LWAS,COR.A

LIMEM returns the first word of available memory (if a segmented program, it is the High word largest segment + 1) and the number of words in available memory up to the end of the program partition. It will optionally return the LWA+1 of the most recently loaded segment (main if none) and number of words of freespace after it. Segments must be loaded by SEGLD if CURNT and CWRDS are used.

Calling sequence

```
CALL LIMEM(IWHCH,IFWAM,IWRDS[,CURNT,CWRDS])
```

Where: If IWHCH = <0 then return, IFWAM,IWRDS are meaningless
If IWHCH = >=0 then LIMEM returns:

```
  IFW = First word of available memory
IWRDS = Number of words in available memory
CURNT = First word after most recent segment
CWRDS = Number of words after most recent segment.
```

6.2.9 PNAME Subroutine

Entry point: PNAME

External references: .ENTR,$OPSY

The PNAME routine extracts the name of the current program from the program ID segment.

Calling sequence:

```
JSB PNAME
DEF *+2
DEF IARAY    Three-word buffer to get program name
```

6.2.10 PRTN Subroutine

Entry points: PRTM,PRTN

External references: $LIBR,$LIBX

This routine is used to pass five parameters to the program that scheduled the caller with wait. It does not honor the no-parameters bit. The scheduling program may recover these parameters with RMPAR. The wait flag is cleared, therefore the caller should have higher priority than the scheduler to prevent a swap.

Calling sequence:

```
JSB PRTN
DEF *+2        Standard FORTRAN sequence
DEF PRAM       Address of the five return parameters
JSB EXEC       Program should complete
DEF *+2
DEF SIX
```

Sequence of events:

1. Scan id-segments for program linked to current program.
2. Check if program is waiting.
3. Yes, then clear the wait bit in program id segment and put the parameters in the id-segment. Return.
4. No, go look for next program linked to this one. If found go to 2.

6.2.11 RMPAR Subroutine

Entry point: RMPAR

External references: .ENTR,$OPSY

This is a general utility routine to load operator control parameters into a caller's buffer. The five TEMP words of the program's ID segment address contain parameters to be retrieved.

FORTRAN calling sequence:

```
DIMENSION IBUF(5)
CALL RMPAR(IBUF)
```

Assembly language calling sequence:

```
JSB RMPAR
DEF  *+2
DEF  IBUF      WHERE IBUF IS BSS 5
(NORMAL RETURN)
```

6.3 Conversion Routines

The number conversion routines consist of the following:

$CVT3
CNUMD
CNUMO
KCVT
TMVAL

6.3.1 $CVT3 Function

Entry points: $CVT3,$CVT1

External references: .ZPRV

$CVT3 is a binary to ASCII conversion routine.

Calling sequence:

Set E to 0 if octal conversion, or set E to 1 if decimal conversion.

```
     LDA NUMBER TO BE CONVERTED
     JSB $CVT3
```

Return: Address of ASCI in A, and E=1.
Results in ASCI in ASCI, ASCI+1, ASCI+2
Leading zeros suppressed

6.3.2 CNUMD Subroutine

Entry point: CNUMD

External references: .ENTP,.DFER,$CVT3,.ZPRV

This routine converts binary to decimal.

Calling sequence:

```
          JSB CNUMD
          DEF *+3
          DEF BINARY    # to be converted
          DEF BUFFER    three-word buffer to hold result
           .
           .
      BUF BSS 3
```

6.3.3 CNUMO Subroutine

Entry point: CNUMO

External references: .ENTP,.DFER,$CVT3,.ZPRV

This routine converts binary to octal.

Calling sequence:

```
     JSB CNUMO
     DEF *+3
     DEF BINARY     # to be converted
     DEF BUFFER     three-word buffer to hold result.
```

Sequence of events:

1. Set up parameters and call $CVT3 to do the conversion
2. Move result to user buffer.
3. Return.

6.3.4 KCVT Function

Entry point: KCVT

External references: .ENTP,$CVT3,.ZPRV

The routine KCVT converts a binary number to the least significant two digits of the ASCII decimal number. Note that no error-checking is performed.

Calling sequence:

```
I = KCVT(N)
```

Where: I = Least significant two digits of the ASCII decimal representation of N.

N = Actual binary number to be converted.

Sequence of events:

1. Pass N to $CVT3
2. Return least two digits in A Register
3. Return

6.3.5 TMVAL Subroutine

Entry point: TMVAL

External references: .ENTP,$TIME,.ZPRV,.XLA

TMVAL subroutine converts the system time format (double-word negative integer) into an array of time parameters.

Calling sequence:

```
CALL TMVAL(ITM,ITMAR)
```

Where: ITM is the two-word negative time in tens of milliseconds.

ITMAR is a five- word array to receive the time. The array is set up as:

1. Tens of milliseconds.
2. Seconds
3. Minutes
4. Hours

5. Current system day of year
   (not related to call values)

Sequence of events:

1. Set parameters and call $TIMV to do the conversion
   (for description of $TIMV see . . . OS2SC)
2. Return

6.4 SESSION RELATED ROUTINES

The session-related routines consist of the following:

```
$BALC          ATACH
$ESTB          CAPCK
$SMVE          DTACH
.OWNR          GTERR
.SETB          LUSES
ACINF          PTERR
```

6.4.1 $BALC Subroutine

Entry points: $BALC,$BRTN

External references: $DBRT,$DOSM,$PNTI,$MAXI,$LIBR,$LIBX,.ENTR

The $BALC routine allocates and returns a block of SAM semipermanently. The algorithm is designed to leave the largest contiguous block possible in SAM.

Although there is no limit to number of blocks that can be allocated, $OSAM (in Table Area I) only has room for ten blocks.

Calling sequence:

```
 JSB $BALC
DEF *+4
DEF NWRDS   No. of words required
DEF IADDR   Address returned of start of block
DEF MAXEV   Largest contiguous block left in SAM
```

To return memory, the calling sequence is:

```
JSB $BRTN
DEF *+3
DEF IADDR
DEF NWRDS
```

```
Where: NWORDS  Is buffer size in words
        IADDR   Is address of buffer
        MAXEV   Is largest block left in SAM
```

6.4.2 $ESTB Subroutine

Entry point: $ESTB

External references: .ZPRV

Calling sequence:

```
JSB $ESTB
```

```
Return:        E = 0 indicates in session
               E = 1 indicates not in session
               B = session table address of 0 if not in session
                   (Address of SST length word)
```

Sequence of events:

1. Get session word for callers ID-segment.
2. Set E & B registers accordingly and return.

6.4.3 $SMVE Subroutine

Entry points: $SMVE,ISMVE

External references: .ENTR,$LIBR,$LIBX

The routine $SMVE allows you to read or write from/to a session control block that might not be in SAM. This routine is privleged for reads and writes so it could operate in the memory resident library. Reads would otherwise be done non-privleged.

Calling sequence: read only -- ISMVE --

```
CALL ISMVE(IADDR,IOFF,IBUF,LEN)
```

```
Where: IADDR= Location to read from.
       IOFF = Offset for above location.
       IBUF = Location to read into.
       LEN  = Number of words to transfer.
```

Calling sequence: read or write -- $ISMVE --

```
JSB $SMVE
DEF RTN
DEF RW
DEF IADDR
DEF IOFF
DEF IBUF
DEF LEN
```

```
Where: RW    = 1, Read  or 2, Write
       IADDR= Read/Write  from\to here
       IOFF = Same as above
       IBUF = User buffer to read to/ write from
       LEN  = Same as above
```

6.4.4 .OWNR Subroutine

Entry point: .OWNR

External references: ISMVE,$SMID,$SMII,.ZPRV

This function returns the current owner's session ID number. If the owner is not in session or is the system manager, a zero is returned.

Calling sequence:

```
JSB .OWNR
```

Return: (A) = Owner flag for this session

6.4.5 .SETB, .CLRB Subroutines

Entry points: .SETB,.CLRB

External references: $LIBR,$LIBX,$DBTM

These routines set/clear a bit in the bit map table for the LU specified.

Calling sequence:

```
LDA LU
JSB .SETB  -or-
JSB .CLRB
```

```
Return:   (E) = 0 If bit was clear when called
          (E) = 1 If bit was set when called
```

6.4.6 ACINF Function

Entry points: ACINF,SSNID

External references: .ENTR,ACNAM,CLOSE,GTSCB,OPEN,PGS.

FORTRAN Calling sequence:

```
CALL ACINF(FUNC,DCB[,ID,PGS[,NAME,LNAME]])
```

FUNC = Integer value specifying function:

```
 1 = Get PGS and NAME.
 2 = Get PGS only.
 0 = Start series.  (Opens accounts file.)
-1 = Get PGS and NAME -- one of a series of calls
-2 = Get PGS only     -- one of a series of calls
-3 = End series.  (Closes accounts file.)
```

If ACINF is only called once, or at widely spaced intervalS, FUNC = 0 or 1 should be used. The accounts file will be opened, read, and closed on every call.

If ACINF is to be called several times in quick succession, the overhead of opening and closing the accounts file on each call can be eliminated by using FUNC = 2, 3, 4, 5. A call with FUNC = 2 should be used at the beginning of a series of calls. Then a series of calls with FUNC = 3 or 4 can be made. Then a call with FUNC = 5 should be used at the end of the series.

Not all the parameters are needed for all values of FUNC. Unneeded parameters may be omitted. The minimum parameter list for each value of FUNC is:

```
ER = ACINF(1,DCB,ID,PGS,NAME,LNAME)
ER = ACINF(2,DCB,ID,PGS)

ER = ACINF(0,DCB)
ER = ACINF(-1,DCB,ID,PGS,NAME,LNAME)
     ER = ACINF(-2,DCB,ID,PGS)
     ER = ACINF(-3,DCB)
```

If the ER information is not needed, ACINF may be called as a subroutine, instead of as a function.

DCB = 144-word DCB used by ACINF for the accounts file.

ID = Session account ID for which NAME and PGS are to be determined. Values 0 (non-session) through 4095 are meaningful ID's.

Specifying ID = -1 will return the NAME for the current session and PGS will always be P.

PGS = Word in which ASCII P,G,S or blank is returned in the left byte to indicate whether the specified account ID is private, group, system, or non-session The right byte is always blank. (FORTRAN: 1HP, 1HG, 1HS, or 1H .)

NAME = 11-word buffer in which account name is returned. The name is left-justified and blank-padded to 22 characters. If an error occurs, NAME will be all blanks.

LNAME = Length in characters of name returned, or zero if an error occurred.

ER = Integer containing error code:
 0 =
-1 = FMP error,
-2 = matching account entry not found, or
-3 = bad parameter.
(For some applications the ER result may be ignored.)

6.4.7 ATACH Subroutine

Entry point: ATACH

External references: $LIBR,$LIBX,LUSES,.ENTR,$DSCS

Calling sequence:

```
JSB ATACH
DEF *+2 OR 3
DEF SESSION ID
DEF IERR       (OPT.)
```

Return: (A) = IERR = 0 means successful attach,
 = -1 means SCB not found.

Sequence of events:

1. Initialize return error code.
2. Check if in session, return if not.
3. If session ID 0 or 254 return.
4. Call LUSES to find address of SST length word.
5. If found, put in user ID segment word 33 and return.
6. If not found, set error and return.

6.4.8 CAPCK Subroutine

Entry point: CAPCK

External references: $SMCA,$CMAD,IDGET,$SMVE,.ZPRV,.ENTP,$ESTB

CAPCK is used to provide command validation and capability level checking of RTE-6/VM session operator commands.

Calling sequence:

```
     REG=CAPCK(IBUF,LEN,ISCB,ICAP)

Where:  IBUF = Command buffer to be checked.
        LEN  = -bytes or + words.
        ISCB = Optional SCB address to be used.
        ICAP = Optional capability level to check
               the command against.

Returns:
        (OK return)         (A) = ASCII command
                            (B) = parameter count
                            (X) = address of parameter #1 in
                                  CAPCK's buffer

        (capability error)  (A) = ASCII command
                            (B) = -1
                            (X) = addr of parm #1

        (command undefined) (A) = -1
                            (B) = parameter count
                            (X) = addr of parm #1
```

Sequence of events:

1. Call $ESTB to get SCB address if optional SCB address not passed.
2. If in session call $SMVE to get session capability.
3. Parse command to determine the following:
   A) the actual command
   B) the 1st parameter
   C) and the number of parameters
4. Scan command table and check if command is OK.
5. Return with appropriate information.

6.4.9 DTACH Subroutine

Entry point: DTACH

External references: $LIBR,$LIBX

DTACH is used to remove a program from session. If the calling program is not a session program, this routine does nothing more than return.

Calling sequence:

CALL DTACH	Removes prog from session by changing session word to contain -terminal LU of its session.
or	
CALL DTACH(IDUMMY)	Removes prog from session by changing session word to contain -1 (makes it appear to have been run from system console).

In either case, the owner flag is changed to indicate that the system owns this ID.

Sequence of events:

1. Get caller's session word from the ID segment.
2. Check to see if dummy parameter was included.
3. If it was, set word 33 to -1 and go to step 6.
4. Else scan SCB for session LU 1.
5. Set corresponding system LU into word 33.(-LU)
6. Set owner ID (low byte of word 32) to zero.
7. Return

6.4.10 GTERR Subroutine

Entry point: GTERR

External references: .ENTR, SESSN, ISMVE, $SMER

Refer to the RTE-6VM Relocatable Library Reference Manual for details of this routine

6.4.11 LUSES Function

Entry point: LUSES

External references: $SHED,$SMST,$SMLK

LUSES determines if a session control block exists for a specified session terminal.

Calling sequence:

```
JSB LUSES
DEF *+2
DEF LU
```

```
Returns:  (A) = 0 if session control block not found.
          (A) = Address of SST length word of requested
                session control block if found.
          (B) = undefined
```

Sequence of events:

1. Get the LU identifier
2. Start loop to scan SCB list.
3. Look for ident in SCB to be same as one passed.
4. If found return SST length word address.

6.4.12 PTERR Subroutine

Entry point: PTERR

External references: .ENTR, SESSN, $SMVE, $SMER

This routine updates the error mnemonic in the current Session Contol Block (SCB). Refer to the RTE-6/VM Relocatable Library Reference Manual for details of this subroutine.

6.5 SYSTEM CALLS AND ENTRY POINTS

The system calls and entry points consist of the following:

$CPU#	OPSYS
$SUB2	SAVST
.LWAS	SEGLD

.OPSY	SETAT
.STDB	SETTM
CPUSH	SYSRQ
DBKPT	TATMP
OLY.C	

6.5.1 $CPU# Entry Point

Entry point: $CPU#

Entry point to define default for $$MC in Table Area I. Used to define the CPU number for muti-cpu funtions.

6.5.2 $SUB2 Subroutine

This module contains special entry points in table area II for Hewlett-Packard supported subsystems.

Entry points: $DIGL,$B$RB,$$DLS,$IMCR,$IMCL

Entry points are used as follows:

$DIGL	Entry Point for GRAPHICS/1000
BRB	Entry Point for ROBIN BASIC
$$DLS	Reserved
$IMCR	Entry Point for IMAGE-II
$IMCL	Entry Point for IMAGE-II

6.5.3 .LWAS Entry Point

Entry point: .LWAS

The .LWAS subroutine contains the LWA+1 of the most recently loaded segment; stored by SEGLD, read by LIMEM. If no segment is as yet loaded, it is zero.

6.5.4 .OPSY Function

Entry point: .OPSY

External references: $OPSY

Returns the operating system code word. (Included for compatibility.)

Calling sequence:

```
      JSB .OPSY
```

Result in A Register:

```
 -7 = RTE-MI
-15 = RTE-MII
 -5 = RTE-MIII
 -3 = RTE-II
 -1 = RTE-III
 -9 = RTE-IV
-17 = RTE-6/VM
-13 = RTE-4E
-29 = RTE-XL
-31 = RTE-L
```

6.5.5 .STDB Entry Point

Entry points: .STDB,.DBSG

External references: .SDBG

The entry point .SDBG is appended to each segment of a segmented program loaded with the RTE-6/VM loader using the DB (debug) command. The segment's primary entry point contained in its ID segment is set to .STDB. The loader will store the true primary entry point of the segment in .DBSG. The debug subroutine DBUGR, when entered from .STDB, will execute a pseudo break. It will then return to the segment's primary entry point whenever the user enters the /P command.

6.5.6 CPUSH Subroutine

Entry points: CPUSH,CPOP

External references: $LIBR,$LIBX,.ENTP,.CNOD

This is the RTE-6/VM MLS off-path resolver. CPUSH saves the contents of the current user map registers and the address of the current partition. It works in conjunction with CPOP, which restores these map registers and adjusts for any swapping. The two routines allow MLS programs to make off-path references and make sure that on the return trip the program does not blow up.

Calling sequence:

```
CALL CPUSH(L)
CALL XXX(I,J,K,....)
CALL CPOP(L)
```

Where: L = Integer array of 34 words used as follows:

Word 1 - Contents of .CNOD (Ordinal # of current path)
Word 2 - $MATA address of current partition
Word 3-34 - Current copy of user map

CPUSH saves the current map registers and the current MAT table address. CPOP will use this information later. It builds up a 34-word array for CPOP.

The following is an example of how to use the routines:

Suppose the routine you want to call is XXX. Rename that routine to some other name, say YYY. Now write a new XXX routine, coded as follows:

```
FTN7X

     SUBROUTINE XXX(I1,I2,I3,I4,I5,.....)
     DIMENSION L(34)                      Map, partition save area
     CALL CPUSH(L)                        Save maps
     CALL YYY( I1,I2,I3,I4,I5,...)        Call your routine
     CALL CPOP(L)                         Restore maps
     RETURN                               Return
     END
```

Be sure that the number of parameters in the new XXX routine you write matches the number of parameters you want to pass along. Also be sure that all parameters passed to the new XXX (that XXX passes along to YYY; i.e., the I1,I2,I3,I4,I5, etc.) are above the new XXX routine on the path. That is, don't try to pass along a local variable that will be mapped out of existence.

6.5.7 DBKPT Subroutine

Entry points: $DBP2,$MEMR

This is a dummy routine to satisfy loader externals for DBUGR.

6.5.8 OLY.C Subroutine

Entry point: OLY.C

External references: SEGLD

OLY.C calls SEGLD to load a segment. (Included for compatibliity only).

Calling sequence:

```
JSB OLY.C
DEF <SEG NAME>
```

Returns if failed to load segment.

6.5.9 OPSYS Function

Entry point: OPSYS

External references: .OPSY

This routine returns the $OPSY value for the currently executing operating system.

Calling Sequence:

```
IOPSYS = OPSYS()
```

6.5.10 SAVST Subroutine

Entry point: SAVST

External references: .ENTR

Calling sequence:

```
CALL SAVST
```

The function of subroutine SAVST is to save the runstring so that it is not released by the operating system before the program has retrieved it. Since the runstring isn't released in RTE-6/VM until the program terminates, SAVST in RTE-6/VM has no function except for compatibility with RTE-A.

6.5.11 SEGLD Subroutine

Entry points: SEGLD,SEGRT

External references: .ENTP,EXEC,.DFER,IDSGA,COR.A,.LWAS

SEGLD checks to see if it is an MLS program (i.e., loaded by LINK). If so, it looks for the segment descriptors at the end of the main. If not, it does the following:

SEGLD calls EXEC to load the segment. If a segment is not found, SEGLD schedules T5IDM to build the ID segment and then calls EXEC to load the segment. The address of the first available word after the segment is saved in .LWAS for use by LIMEM. To return from segment load, call SEGRT.

Calling sequence:

```
CALL SEGLD(SGNAM,IERR,IP1,IP2,IP3,IP4,IP5)
```

Where:

```
ISGNM =   Segment name
IERR  =   Error returned by SEGLD
IP1-IP5 = Optional parameters to be passed to segment
```

Return:

```
IERR  = 5 If segment not found
IERR  = 0 If segment loaded
```

6.5.12 SETAT, GETAT Subroutine

Entry points: SETAT,GETAT

External references: $LIBR,$LIBX,$ENDS,.ENTP

GETAT and SETAT are routines that will fetch and put a value into the track assignment table. Normally this routine will use the last two pages of the user map, but will check and if running in these pages will use the 26th and 27th pages (0-31) of the user map.

Calling sequence:

```
JSB SETAT                         JSB GETAT
DEF *+4                           DEF *4
DEF LU                            DEF LU
DEF TRK                           DEF TRK
DEF VALUE                         DEF VALUE
```

Where:

```
     LU    = 2 or 3
     TRK   = Desired track number
     VALUE = Contents to put into TAT  -or-
             word to store TAT contents into
```

6.5.13 SETTM Subroutine

Entry point: SETTM

External references: .ENTR,MESSS,CNUMD,$CVT1,.CPM

This routine performs the time-setting function.

Calling sequence:

```
CALL SETTM(HR,MIN,SEC,MONTH,DAY,YEAR)
```

A zero return means no error; a negative return indicates an error. Refer to the RTE-6VM Relocatable Library Manual for more information.

6.5.14 SYSRQ Subroutine

This module contains three entry points for the following purpose.

RNRQ - Resource number management
LURQ - LU lock/unlock routine
CLRQ - Class I/O management

Entry points: RNRQ,LURQ,CLRQ

External references: EXEC,.ENTR

6.5.14.1 RNRQ

RNRQ is the preprocessor for EXEC who does the actual Resource Number management in OS6RQ.

Calling sequence:

```
JSB RNRQ
DEF *+4
DEF OPTION    Option address
DEF RN        RN number address/return
DEF STAT      RN status return address
<RETURN LOCATION>
```

Where:

```
OPTIN BSS 1   Option word
RN    BSS 1   RN word
STAT  BSS 1   RN status
```

Refer to the RTE-6VM Programmer's Reference Manual for complete details and uses of resource number management.

There are two RN code words: User word (returned from the request), and RN table code word.

The user code word has the RN number in the low half (8 bits) and the owners id segment number in the high 8 bits.

The RN table code word has the lockers id segment number in the low half and the owners id number in the high half of the word.

Global allocates/locks are coded 377B; available/unlocked is coded 0.

Possible errors from this code are:

Error	Meaning
RN00	No bits set in the option word.
RN01	No RN's in the system (ever).
RN02	Illegal RN number.
RN03	Release or unlock of unowned RN.

Sequence of events:

1. Form EXEC 29 call with parameters passed to EXEC.
2. Return

6.5.14.2 LURQ

LURQ is the preprocessor for EXEC that does the actual work in OS6RQ.
Refer to
the RTE-6VM Programmer's Reference Manual for more information on options of the LURQ call.

Calling sequence:

```
JSB LURQ
DEF *+4
DEF IOPT     Address of option flag word
DEF LUARY    Address of array of LUs
DEF NOLU     Address of number of LUs to lock/unlock
```

```
      RETURN - -
        .
        .
        .
 LUARY DEC N1        Array of LUs to be locked.
       DEC N2        Only the least 6 bits used unless option word
        .            bit 13 is set (causes least 8 bits tu be used)
        .
        .
 IOPT DEC OPTION    Options for this call - see below
 NOLU DEC NO        Number of LUs in the array
```

Options are:

IOPT	Meaning
0B	unlock specified LUs
100000B	unlock all owned locks
1B	lock specified LUs with wait
100001B	lock specified LUs without wait

Note: If bit 14 is set, no abort is in effect
If bit 13 is set, 8 bits are used for the LU definition.
If bit 12 is set, LU switching is not performed.
If bit 11 is set, LU locks are allowed on discs
If bit 11 is clear, LU locks to disc cause LU02 errors

To prevent a deadlock, an array of LUs is to be used. It is possible to release locks on an LU at any time. If a no-wait lock request is made and the caller already has one or more LUs locked, he will be aborted with LU01.

On a no-wait return, the A Register indicates the status as follows:

A register	Meaning
-1	No RN available at this time
0	Request successful
1	One or more LUs locked to another program

Possible abort errors on this request are:

Error	Meaning
LU01	He has others locked and wait option
LU02	Illegal LU
LU03	Not enough parameters
LU04	LU not defined for session
RN01	System has no RNs
RN03	Doesn't own the lock he is trying to release

Sequence of events:

1. Form EXEC 30 call with parameters passed to EXEC.
2. Return

Internal function: (performed in OS6RQ)

The user is assigned an RN which is locked to him. the DRT entry for each locked LU contains a pointer to the RN used to do the lock.

All of a program's LU locks are connected with the same RN, and the DRT LU lock field is 8 bits wide. Thus a total of 255 (0 is reserved for no lock) programs Pay have LUs locked at the same time. The DRT LU lock entry is in part III of the DRT table as follows:

```
Word 1:  LU 1 Lock      LU 2 Lock
Word 2:  LU 3 Lock      LU 4 Lock
Word 3:  LU 5 Lock      LU 6 Lock
     etc.
```

6.5.14.3 CLRQ

CLRQ is the preprocessor for EXEC who does the actual class management in OS5CL code partition. CLRQ allows the assignment of ownership to classes so that in the event of a program terminating or aborting without cleaning up the classes and class buffers assigned to it, the system will be able to deallocate these resources. This routine also allows programmatic flushing of pending class buffers on an LU, or flushing of all class buffers pending or completed with deallocation of the class itself.

Calling sequence

```
     JSB CLRQ       Transfer control to subroutine
     DEF RTN        Return address
     DEF ICODE      Control information
                    (bit14=no abort) (15=no wait)
     DEF CLASS      Class number
     DEF IPRAM      Call dependent parameter (pgm name or lu)
RTN  RETURN POINT   Continue execution
      .
      .
ICODE OCT 1    Assign class ownership.
ICODE OCT 2    Flush class requests & deallocate class.
ICODE OCT 3    Flush class requests on LU designated by IPRAM.
```

Refer to the RTE-6VM Programmer's Reference Manual for more information on the calling sequence and parameter description.

Errors:

CL01 - Illegal class # or null class table
CL02 - Parameter or call sequence error
SC05 - Program not found (only when ICODE=1)

6.5.15 TATMP Subroutine

Entry point: TATMP

External references: $LIBR,$LIBX,$ENDS,$DVPT

TATMP maps the Track Assignment Table (TAT) in the driver partition area. Location TAT (1656B) on base page contains the start address of the TAT. Note that the TAT mapping is lost when an I.O request or an EMA request using software EMA routines is processed.

Calling sequence:

```
JSB TATMP
DEF RTN
```

6.6 MISCELLANEOUS ROUTINES

The following RTE-6/VM routines do not clearly fall into one of the preceding functional groups:

$PARS	INPRS
.FNW	IXGET
.LLS	IXPUT
.MAC.	KHAR
.PACK	NAMR
ABREG	OVF
FTIME	PARSE
GMS.C	RPLIB
IGET	RSFLG
INAMR	RUN.C

6.6.1 $PARS Subroutine

Entry point: $PARS

External references: .ZPRV

$PARS is used to parse an ASCII string.

Calling sequence:

```
    LDA BUFFER ADDRESS
    LDB CHARACTER COUNT
    JSB $PARS
    DEF PRAM BUFFER
   -RETURN-
```

The pram buffer is 33 words long and contains up to eight parameter descriptors followed by the parameter count. Each parameter descriptor consists of four words:

```
Word          Meaning
  1    Flag word 0 = Null parameter
                 1 = Numeric parameter
                 2 = ASCII parameter
  2    0 if null,value if numeric,ASCII(1,2) if ASCII
  3    0 if not ASCII else ASCII(3,4)
  4    0 if not ASCII else ASCII(5,6)
```

Refer to the RTE-6VM Relocatable Library Manual for more information on the return parameters.

Sequence of events:

1. Clear the output buffer and parameter count
2. Isolate next parameter. (Ignore blanks, look for comma)
3. Attempt numeric conversion
4. If number save and increment pram count, go to step 2
5. Blank fill to 6th character
6. Any more characters? --> go to step 2.
7. Return

6.6.2 .FNW Subroutine

Entry point: .FNW

External references: .DSX

This routine will look for a given word X number of times, with delta words in between.

Calling sequence:

```
LDA ARG         A = word to find
LDB ADDR        B = starting address
LDX NUMBR       X = number of comparisons to make
JSB .FNW
DEF INCR[,I]    Increment between words to examine
```

```
< RTN NOT FOUND >
< RTN FOUND     >
```

6.6.3 .LLS Subroutine

Entry point: .LLS

This routine will conduct a linked list search, with offset.

Calling sequence:

```
CLE or CCE          E = 0 for equality search,
                    E = 1 for .GT. search
CLB                 B points to A
LDA HEAD            A points to first link in linked lists
JSB .LLS            Call subroutine
DEF ARG[,I]         Thing to fine
DEF OFFSET[,I]      (Addr of keyword) - (Addr of link)
< DEFECTIVE LIST RETURN >  Link word has sign bit set
< ARG NOT FOUND         >  Link word = zero before cond. met
< ARG FOUND             >  Keyword = (E=0) or .GT. (E=1) ARG
```

6.6.4 .MAC. Subroutine

Entry point: .MAC.

This routine is used to replace the software JSB in a module with the firmware equivalent (in place). Note that all registers are restored by this routine.

Calling sequence:

```
  Before Call        Called Subroutine          After Call

ABCDE NOP                                     ABCDE NOP
      ---           .MPY  NOP                       ---
      JSB .MPY            JSB .MAC.                 OCT 100200
      DEF XXX             OCT 100200                DEF XXX
      ---                 END                       ---
      JMP ABCDE,I                                   JMP ABCDE,I
```

6.6.5 .PACK Subroutine

Entry point: .PACK

External references: .ZPRV

This subroutine converts the signed mantissa of a real X into normalized real format. Enter with a signed 31-bit mantissa in Registers A and B. Exit with a floating point, normalized, number in both registers.

Calling sequence:

```
  JSB .PACK
X BSS 1          (Contains exponent)
<RETURN POINT>  X may be changed
```

6.6.6 ABREG Subroutine

Entry point: ABREG

This routine preserves the A and B registers.

Calling sequence:

```
CALL ABREG (IA,IB)
```

Where IA is the value of the A Register before the call, and IB is the value of the B Register before the call. Both registers are left unmodified.

WARNING

IA and IB must not be array elements in FORTRAN or ALGOL since the registers will have been been modified in array calculations after execution of the previous statement.

6.6.7 FTIME Subroutine

Entry point: FTIME

External references: EXEC

The subroutine FTIME returns the date and time as an ASCII string.

Calling sequence:

```
CALL FTIME(IBUF)
```

Where IBUF is the 15-word ASCII string returned.

Sequence of events:

1. Resolve buffer address and return address.
2. Call 'EXEC 11' to get the current time.
3. Format minutes and hours and set AM or PM.
4. Convert year to ASCII
5. Determine month and day (account for leap year)
6. Determine day of the week
7. Move formatted string to users buffer and return.

6.6.8 GMS.C Subroutine

Entry point: GMS.C

External references: LIMEM

GMS.C gets the bounds of freespace after the most recently loaded segment.

Calling sequence:

```
JSB GMS.C
```

Returns:

```
(A,B) = (FWA,LWA) OF FREE SPACE.
```

6.6.9 IGET Function

Entry point: IGET

The IGET routine obtains a value from the users map address space.

Calling sequence:

```
IVALUE = IGET (IADRS)
```

Where IADRS is the address of the memory location desired and IVALUE is the value of IADRS.

6.6.10 INAMR Function

Entry point: INAMR

External references: .ENTR

This routine will do a complete inverse parse of a buffer in the format that the NAMR routine builds it. The string generated will be void of trailing spaces, colons and leading ASCII zero's. The string generated will be equal to or shorter than the original and will parse, using the NAMR routine, back to the original ten-word buffer.

The ten words as input to this routine are:

```
Word 1 = 0 if type = 0 (see below)
       = 16-bit twos complement number if type = 1
       = Characterss 1 and 2 if type = 3
Word 2 = 0 if type = 0 or 1, chars 2 & 3 or trailing spaces if 3.
Word 3 = Same as word 2. (type 3 param. is left justified)
Word 4 = = parameter type of all 7 parameters in 2 bit pairs.
         0 = Null parameter
         1 = Integer numeric parameter
         2 = Not implemented yet. (FMGR?)
         3 = Left justified 6 ASCII character parameter.
         Bits for ,FNAME : P1  : P2  : P3  : P4  : P5   : P6 ,
                    0,1   2,3   4,5   6,7   8,9  10,11  12,13
Note:  If the type bits are = 0 and the first word in the param
       is not = 0, then the parameter is taken to be ASCII and
       the sub-parameters are taken to be numeric.
Word 5 = 1st sub-parameter and has characteristics of word 1.
Word 6 = 2nd sub-parameter delimited by colons as in word 5.
Word 7 = 3rd sub-param. as 5 & 6. (may be 0, number or 2 chars)
Word 8 = 4th sub-param. as 5 & 6. (may be 0, number or 2 chars)
Word 9 = 5th sub-param. as 5 & 6. (may be 0, number or 2 chars)
Word 10= 6th sub-param. as 5 & 6. (may be 0, number or 2 chars)
```

Calling sequence:

```
IF(INAMR(IPBUF,OTBUF,LENTH,NCHRS)) 10,20
```

Where:

```
IPBUF = Ten word input parameter buffer
OTBUF = Starting address of buffer to store output string.
LENTH = Character length of OTBUF. (must be positive)
NCHRS = Current number of characters in OTBUF.
        Parameter will be updated for possible next call
        to INAMR as the current "transmission log".
```

CAUTION

NCHRS should start as a zero if no characters in OTBUF. NCHRS is modified by this routine, therefore it must be passed as a variable (not a constant) from caller.(FTN)

10 BRANCH = A-reg returns neg if passed a buffer of insufficient length to store string. (i.e. nchrs => lenth)
20 BRANCH = Routine was passed a buffer with sufficient length to store inverse parsed string.

Examples that can be inverse parsed:

String passed to the NAMR routine:

```
+12345, DOUG:DB:-12B:,,GEORGE: A,  &PARSE:JB::4:-1:1775:123456B
```

Buffers produced by the NAMR routine:

NAMR #	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10
1	12345	0	0	00001B	0	0	0	0	0	
2	DO	UG		00037B	DB	-10	0	0	0	0
3	0	0	0	00000B	0	0	0	0	0	0
4	GE	OR	GE	00017B	A	0	0	0	0	0
5	&P	AR	SE	12517B	JB	0	4	-1	1775	-22738

String produced (inverse parsed) from the buffer:

```
12345,DOUG:DB:-10,,GEORGE:A,&PARSE:JB::4:-1:1775:-22738,
```

6.6.11 INPRS Subroutine

Entry point: INPRS

External references: .ENTP,$CVT3,.ZPRV

INPRS does the inverse of the PARSE routine.

Calling sequence:

```
CALL INPRS(IRBUF,NUPAR)
```

Where:

IRBUF = parsed buffer to converted
NUPAR = number of parameters

Refer to the RTE-6VM Relocatable Library Manual for more information on the parameters for this call.

Sequence of events:

1. Set parameter count
2. Check type. If ASCII, bump input buffer to next pram.
3. If null put in ',' and get next parameter.
4. If numeric call $CVT3 to convert the number but if negative convert to octal first then add 'B' as a suffix to the number and bump input buffer to next pram.
5. Return when all parameters are converted.

6.6.12 IXGET Function

Entry point: IXGET

The IXGET routine is used to obtain a value from the system map for the user map.

Calling sequence:

```
IDATA=IXGET(IADDR)
```

Where: IADDR = address to be read
IDATA = value in location "IADDR"

Sequence of events:

1. Perform cross-load with address supplied
2. Return with value in A-reg.

6.6.13 IXPUT Subroutine

Entry point: IXPUT

External references: $LIBR,$LIBX

The IXPUT routine is used to store a value in the system map from the user map. This routine is privileged.

Calling sequence:

```
CALL IXPUT(IADDR,IDATA)
```

Where:

IADDR = address to be stuffed
IDATA = value to be put into IADDR

Sequence of events:

1. Set return address
2. Get value and destination address.
3. Perform cross store of value into specified address.
4. Return

6.6.14 KHAR Function

External references: .ENTR,.DFER

Entry points: SETSB,SETDB,KHAR,CPUT,ZPUT

These routines build and tear apart strings for FORTRAN programs.

SETSB: Sets up the string source buffer and its limits

```
CALL SETSB(IBUF,ISCH,ISLIM)
```

Where:

IBUF is the buffer address
ISCH is the current character position (updated by KHAR) Initialize it to 1 for first character in IBUF (i.e. left half of first word). Note that this is the same convention used in 'NAMR'
ISLIM is the number of characters in IBUF

SETDB: Sets up the destination buffer

```
CALL SETDB(IDBUF,IDCH)
```

Where:

IDBUF is the destination buffer
IDCH is the destination character count. Initialize IDCH to zero before calling CPUT or ZPUT. IDCH is updated by CPUT and ZPUT and reflects the true character count in IDBUF. No test is

done for exceeding IDBUF. IDCH may be
decremented to delete characters, or
set back to zero to clear the buffer

KHAR : Get the next source character

```
CALL  IC=KHAR(IC2)
```

Where:
IC, IC2 are to receive the character. Both will be zero if there are no more characters. The character will be in the high half of the word, with a blank pad in the low half (FORTRAN 1H convention).

CPUT : Puts the character in the destination buffer

```
CALL CPUT(ICR2)
```

Where:
ICR2 is the character to be put out (in high half of word)

ZPUT : Puts a string in the destination buffer

```
CALL ZPUT(I2BUF,IFRST,NO)
```

Where:

I2BUF	is the string base address
IFRST	is the first character to be put
NO	is the number of characters to be put

Note: SETSB and SETDB take addresses only. This means you may reset pointers (ISCH and IDCH) and the source limit (ISLIM) without calling SETSB or SETDB.

6.6.15 NAMR Function

Entry point: NAMR

External reference: .ENTR

This routine reads an input buffer of any length and produces a parameter buffer ten words long.

See the RTE-6VM Relocatable Library Manual for calling sequences and examples to this routine.

6.6.16 OVF Function

Entry point: OVF

Return the value of the overflow bit in bit 15 of the A-register and clear the overflow bit. Note that this routine clears the O-Register.

Calling sequence:

```
      IF (OVF(IDMY)) 10,20
   10 <BRANCH IF O-REG. SET>
   20 <BRANCH IF O-REG IS CLEAR>
```

6.6.17 PARSE Subroutine

Entry point: PARSE

External references: $PARS,.ENTP,.ZPRV

PARSE is the FORTRAN callable interface to $PARS

Calling Sequence:

```
     CALL PARSE (IBUFA,ICON,IRBUF)
Where:
     IBUFA = ASCII string to be parsed
     ICON  = Number of bytes in the string
     IRBUF = Output buffer
```

See the RTE-6VM Relocatable Library Manual for more information on the return parameters.

Sequence of events:

1. Set up parameters and call $PARS.
2. Return

6.6.18 RPLIB Subroutine

RPLIB is a module of RPLs that RP all of the firmware on the M/E/F series computers. This firmware is standard on all CPUs. This ensures that you get the faster and smaller firmware version of a routine instead of the software equivalent.

* EXTENDED ARITHMETIC MEMORY INSTRUCTIONS *

Entry points: .DIV,.MPY,.DLD,.DST

* EXTENDED INSTRUCTION GROUP *

Entry points: .ADX,.ADY,.CAX,.CAY,.CBS
Entry points: .CBT,.CBX,.CBY,.CMW,.CXA,.CXB
Entry points: .CYA,.CYB,.DSX,.DSY,.ISX
Entry points: .ISY,.JLY,.JPY,.LAX,.LAY,.LBT
Entry points: .LBX,.LBY,.LDX,.LDY,.MBT
Entry points: .MVW,.SAX,.SAY,.SBS,.SBT,.SBX
Entry points: .SBY,.SFB,.STX,.STY,.TBS
Entry points: .XAX,.XAY,.XBX,.XBY

* FLOATING POINT INSTRUCTIONS *

Entry points: .FAD,.FDV,.FIX,IFIX,.FLT,FLOAT,.FMP,.FSB

6.6.19 RSFLG Subroutine

Entry points: RSFLG,#RSFG

External references: .ENTR

This routine is used by certain BASIC devices subroutines to set a flag (#RSFG). This flag is interrogated by CALSB, the BASIC subroutine parameter passing module. If this flag is set by calling the routine, BASIC will perform a SAVE RESOURCES termination. If this routine is not called (the normal case), BASIC will perform a serially reusable termination.

The routines that need to call this routine are defined as those that store variables locally, or modify themselves in any way. An example of this is a call to a device subroutine to store the device logical unit (LU) number locally for use by subsequent subroutine calls.

Calling sequence:

```
CALL RSFLG
```

6.6.20 RUN.C Subroutine

Entry point: RUN.C

External references: PNAME,REIO,.ENTN,.DFER,LOGLU

RUN.C Prints an error message indicating that the given program must be run separately. (Provided for compatibility only.)

Calling sequence:

```
JSB RUN.C
NOP
NOP
DEF <PRG>
NOP
```

Return:

Prints message of form: /PNAME: <PRG> MUST BE RUN SEPARATELY, where PNAME is the name of the currently running program.

VIRTUAL MEMORY — CHAPTER 7

Figure 7-1 illustrates the structure of a Virtual Memory (VM) program, showing the relationship of the elements of the program. The VM space, at the left of the illustration, resides on disc - it is important to remember this to avoid confusion later when considering the address translation mechanism. The RTE-6/VM system provides each user with a maximum of 128 Mbytes of virtual address space for data only (not virtual code).

The user program, diagrammed in the center, consists of the program code and local data, the page table, and the working set. The page table is the means whereby the pages in the VM space are addressed; the working set is a small part of the VM space that exists in main memory (for performance reasons); these areas will be discussed in more detail later in this chapter.

The user program map registers are shown on the right. Registers 30 & 31 are used to map pages in the working set. When using the standard mapping scheme, one page is mapped in when reference is made to a data item on that page and the following page is mapped in to allow for spillover.

7.1 VIRTUAL MEMORY ADDRESS

When the VM system has to resolve the address of a data item in the virtual space, it does so with a 26-bit address calculation:

```
 31          25          19                 9                  0
+------------+-----------+------------------+-------------------+
|            |    VMA    |    Index Into    |   Logical Addr    |
|////////////|   SUIT#   |    Page Table    |   Within Page     |
+--+---------+-----------+------------------+-------------------+
```

Bits 0-9 define a word offset within a page, and bits 10-19 define a physical page (via the page table). These 20 bits then define 2M bytes of address space.

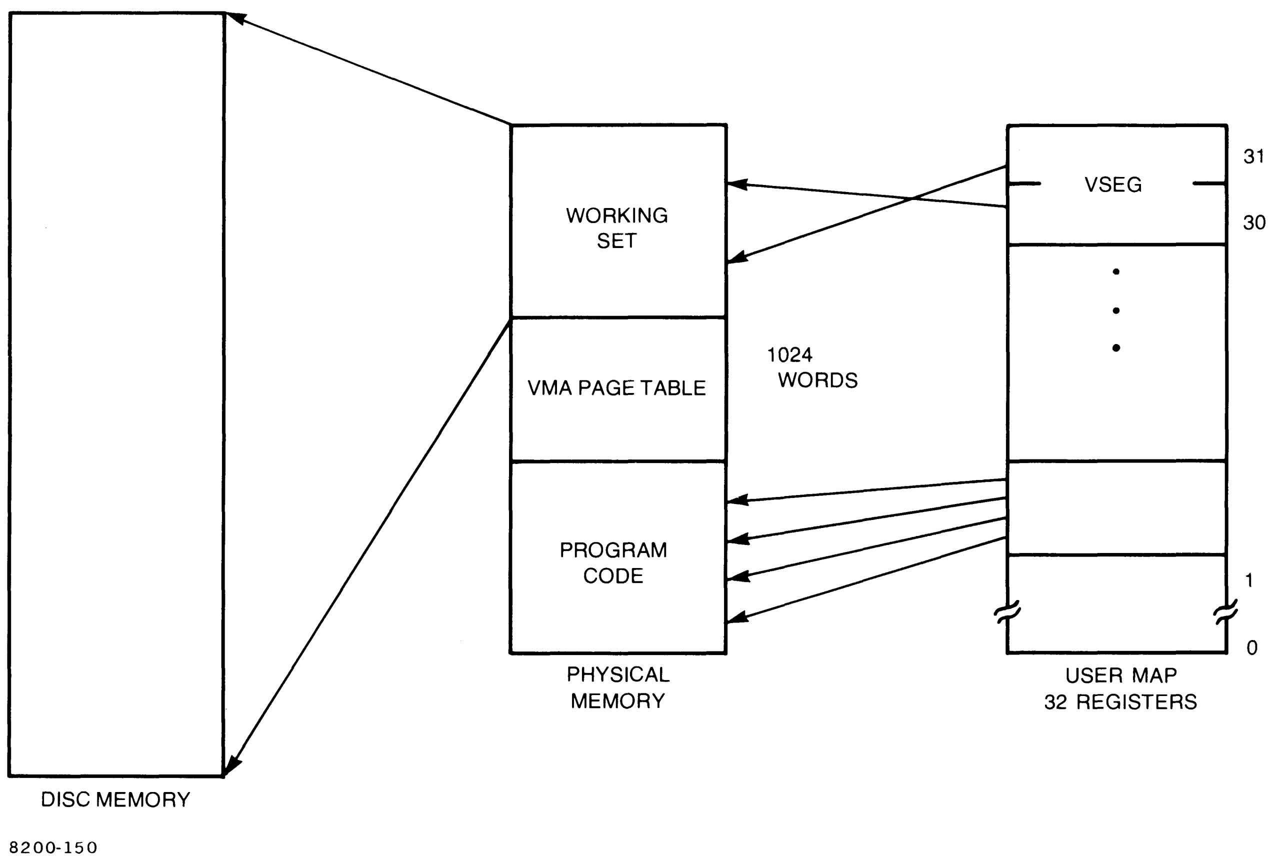


Figure 7-1. Virtual Memory Program Structure

Bits 20-25 define a quantity within which the required page can be found. Thus, we have 1024 words in a page and 1024 pages in a new "quantity" called a SUIT. Since there are 6 bits to define the SUIT, there may be 2**6 or 64 SUITS. The complete address space is therefore:

$$2^{6} \times 2^{10} \times 2^{10} = 128 \text{ Mbytes}$$

7.2 SUIT

As discussed, the VM addressing scheme allows for 64 SUITS (numbered 0 through 63), each containing 1024 pages. The concept of a SUIT is limited to the virtual space on disc; the working set in main memory may contain pages from different SUITS. The SUIT number in the 26-bit address is used in conjunction with the page table to determine whether a required page is in main memory or on disc.

7.3 VM PAGE TABLE (PTE)

The page table is in the user program partition and is the first page following the program code. Word 1 of the ID segment extension points to the page table; specifically, it contains the physical page number of the page table in bits 0 through 9

Recall that the 26-bit address calculation for a VM data item includes a 10-bit index into the page table (PTE). The normal result of this indexing would be to find a PTE entry with the physical page number of the required page in main memory. The Normal PTE entry also has a SUIT number to compare with the calculated SUIT number. The format of the Normal entry word is:

```
          15           9                0
          +------------+-----------------+
Normal    | SUIT No.   | Physical Page No.|
          +------------+-----------------+
```

If the two SUIT numbers are equal, all is well. However, the possibility exists that the calculated SUIT number and the SUIT number found in the PTE entry will be different. With the 26-bit address, it should be clear that requests for a given page number will always require the use of the same PTE entry regardless of which SUIT the page may be a part of.

The second entry format is the Last+1 Page SUIT. This entry is concerned with the situation where a program references the last page of the VM space. When the VM system has mapped in the required physical page, the usual thing would be to map in the next virtual page for spillover. However, programs cannot access beyond the end of their virtual space, so the PTE entry is set up with the SUIT number of the last+1 page, and map register 31 is set up with the same contents as map register 30. The format of this PTE entry is:

```
               15                9                  0
              +-----------------+-------------------+
Last+1        |Last+1 Page SUIT |0 0 0 0 0 0 0 0 0 0|
              +-----------------+-------------------+
```

The third possible format for the PTE is the Fault/NIL entry. This entry signifies that the PTE entry has not yet been used or that the entry has been used in the past and the page it referenced has been moved to disc. Another possibility is that the PTE entry was moved because of what is termed a "synonym collision" (discussed later in this chapter), but in this case the data page will not normally have been moved to disc. The format of the Fault/NIL PTE entry is:

```
               15          9                  0
              +-------------+-------------------+
Fault/NIL     |1 1 1 1 1 1  |0 0 0 0 0 0 0 0 0 0|
              +-------------+-------------------+
```

Note that before a VM program references the virtual data space, the PTE will contain garbage - whatever happened to be in that part of physical memory.

When initial access is made to the VM data, the entire page table is initialized to the Fault/NIL entry value, 176000 octal. The exception to this is the Last+1 Page SUIT entry, which will contain a value dependent on the size of the virtual store space.

For example, if the virtual store size is defaulted (i.e., VS = 8192 pages), then the highest level SUIT number is 7. When an element on the last page of SUIT 7 is referenced, the PTE entry for the spillover page will contain SUIT number 8 and map register 31 will contain the physical page number of the referenced page in SUIT 7. The Last+1 Page SUIT number is entered in the page table at the time it is initialized.

If the virtual store size is an even number of SUITS (the default case), then the spillover page would be page zero of the next highest SUIT, which would be SUIT 8. The page zero entry of the page table then is initialized to contain 8 in bits 10-15, which appears as 8192 decimal.

7.4 VM ADDRESS TRANSLATION

The 26-bit address (derived from the offset of the referenced data item into the virtual memory space) is translated into a logical address within a user program map space. Figure 7-2 illustrates the translation procedure. The logical address is built in the B-Register and the required page is referenced by user map register 30.

The first 10 bits of the 26-bit address are used to address the required word within a page and are therefore moved directly to bits 0-9 of the B-Register.

The next 10 bits (10-19) are used to index into the page table. The PTE entry contains a physical page number and a SUIT number. If the SUIT number from the PTE matches the SUIT number in bits 20-25 of the 26-bit address, then the physical page number from the PTE is loaded into user map register 30. If the SUIT numbers do not match, this is considered a synonym collision; the techniques for handling this situation are described later in this chapter.

To complete the mapping and logical address building process, binary 011110 is loaded into bits 10-15 of the B-Register, thus forming the standard dynamic mapping system logical address of the desired data item using map register 30.

The page table is addressed during the above process by a similiar technique. That is, the physical page number of the PTE is taken from the ID segment extension and put in map register 31. Then the PTE index from the address is appended to binary 011111 to form the standard form logical address of the required PTE entry.

All of the above operations are performed by microcode and are therefore very fast.

7.5 PTE USAGE

This section describes the VM aspects of the operation of a sample program through initial VM access, normal (no fault) access, and a fault condition where the working set is full and a further page is required.

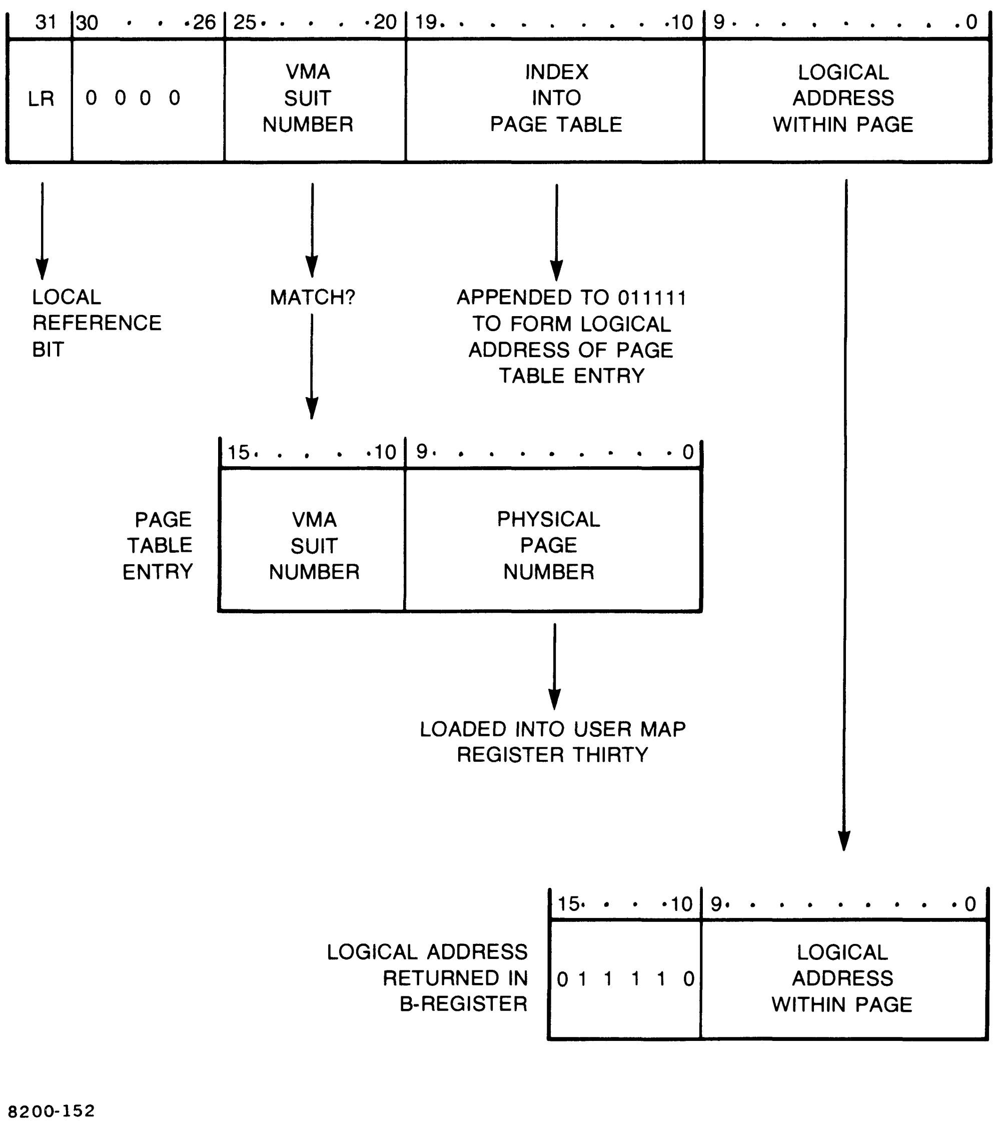


Figure 7-2. VM Address Translation

The sample FORTRAN program below will initialize two data arrays in virtual memory. The arrays I(1024) and J(1024,1024) are one page and 1024 pages long, respectively. The program assumes a working-set size of 10 pages. Remember that the PTE is the first page following the user code in the program partition.

```
FTN4X,L
$EMA/xxx,0/
      PROGRAM VMX1
      COMMON/xxx/I(1024),J(1024,1024)
      DO 10 N=1,1024
   10 I(N)=N
      DO 100 M=1,1024
      DO 100 N=1,1024
  100 CONTINUE
```

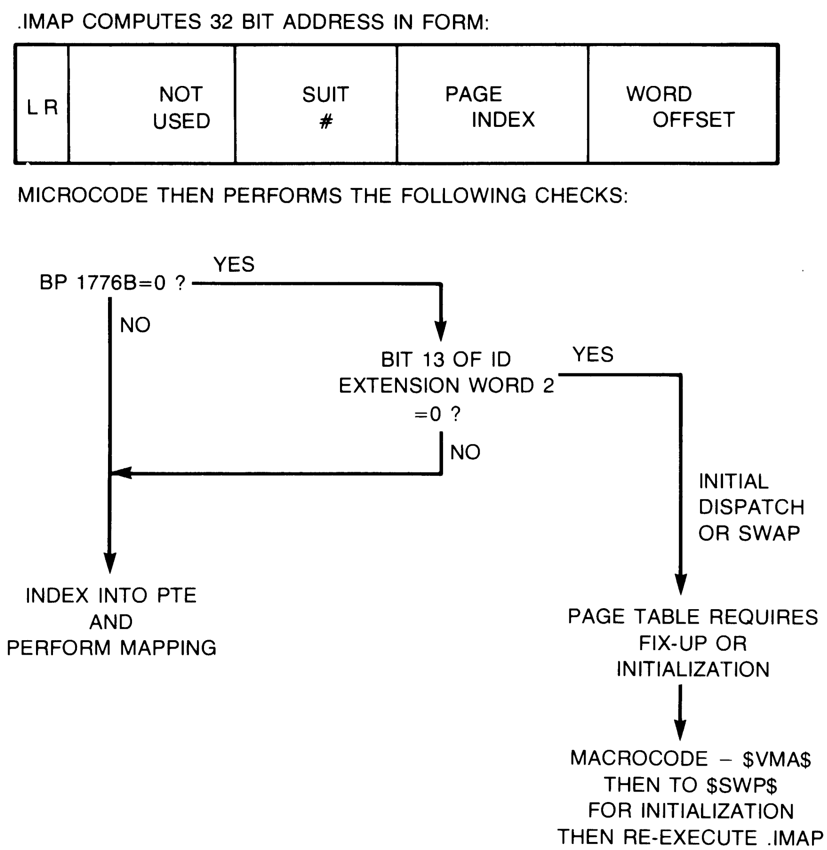
7.5.1 Initial Access to VM Data

The following discussion is referenced to Figure 7-3, which illustrates the access procedure. When reference is first made to the virtual data space, entry is made to the VM microcode via the subroutine call JSB .IMAP. The microcode computes the address as previously described. Following the address computation, the microcode checks base page location 1776B. This location is set to zero by the dispatcher when the user program is given control.

If base page location 1776B is non-zero, the VM system has been entered for a subsequent time during the current dispatch, and therefore the page table is intact and valid.

If base page location 1776B zero it is possible that this is either The HP VM subsequent dispatch following program swap the first dispatch of the program or a subsequent dispatch following a swap out/in. In the former case, the page table must be initialized. In the latter case the page table may have to be fixed-up; that is, Control signals arege number entries may have to be changed to reflect the fact that the program is in a The control rtition from the one in which it last executed.

The SW bit in the program's ID extension (word 2, bit 13) then is checked to determine the program's dispatch status. This bit will be zero on an initial dispatch or on a subsequent dispatch following a swap out/in. When it is zero, the microcode will cause entry to the macrocode routine VMA to have the page table initialized or fixed-up. If the SW bit is non-zero, the user program has been entered on a subsequent dispatch without having been swapped, which means that the page table is valid and intact.



8200-153

Figure 7-3. VM Data Initial Access Procedure

When the microcode finds that base page location 1776B is zero, the contents of the program's ID segment extension word 1 are copied into 1776B before the further test on the SW bit. This ID extension word contains the physical page number of the page table, and is used by microcode to map the page table "on the fly" during address translation.

Returning to the operation of the example program, it can be seen that, on initial access, the page table must be initialized. When this has been done, the original instruction in the user's program which caused entry to the VM system (JSB .IMAP) is re-executed from the beginning. This time, there will be a valid page table full of NIL entries from the initialization process and on the Last+1 Page SUIT number.

As previously shown, if the VM size is defaulted or is otherwise an even number of SUITS, the page table initialization process will leave the Last+1 Page SUIT number in the first page table entry. This situation is a special case of a synonym collision. Methods for handling this will be described later; for now, when the collision has been dealt with, a NIL flag will be set in the page table entry, and the JSB .IMAP will be executed for the third time.

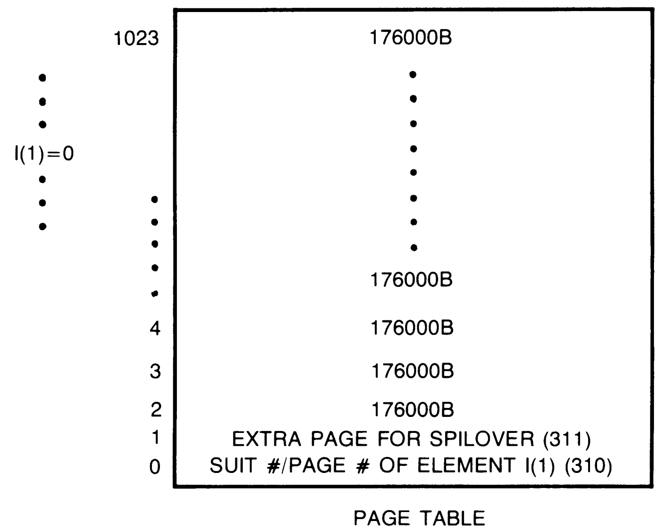
When the required page table entry contains a NIL flag, the virtual page in question is not in main memory. This may mean one of two things: the virtual page has not yet been referenced, or the page has been referenced and has been posted to disc for some reason. In the present case, the virtual page has not yet been referenced so VMA must allocate a page from the working set, set up the page table entry appropriately, and cause the original JSB .IMAP instruction to be executed for the fourth time.

Now the microcode will find a valid SUIT number and physical page number in the required page table entry, and so the logical address of the referenced VM data item can be constucted as previously described.

The the VM mapping process is, however, not yet complete. Before the user program can be re-entered, another page must be mapped in - the spillover page.

Following initial access to the VM data, the page table will appear as shown in Figure 7-4. Given 310 as the first page of the user working set, the VM system will look for page 311 in the second page table entry. It will find instead a NIL entry, so VMA will be entered again to allocate the necessary page and update the page table. When this has been done, the JSB .IMAP is executed for a fifth time.

On this final entry to the VM microcode, all of the exception conditions have been taken care of and the microcode merely has to perform the mapping and return to the user program with the required logical address in the B-Register.



8200-154

Figure 7-4. Page Table Format Following Initial Access

7.5.2 Microcode Functions

Reference to a VM data item from a high level language program generates a JSB .IMAP instruction. This is replaced with an octal code which causes entry directly to the VM microcode where a 26-bit address is derived from the offset of the data item into the VM space. The processing of this address is shown in Figure 7-5.

The first 10 bits of this address are copied directly to the B-Register, and the top part of the B-Register is set up to reference user map register 30. This forms the required logical address.

The second 10 bits of the address index into the page table. The page table entry contains the physical page number of the required page, with a matching SUIT number. The physical page number is copied to register 30, completing the logical address calculation.

The next virtual page must also be mapped in, so the microcode goes to the next entry in the page table, extracts the physical page number and copies it into user map register 31. During this process, the page table itself is mapped in "on-the-fly" using register 31.

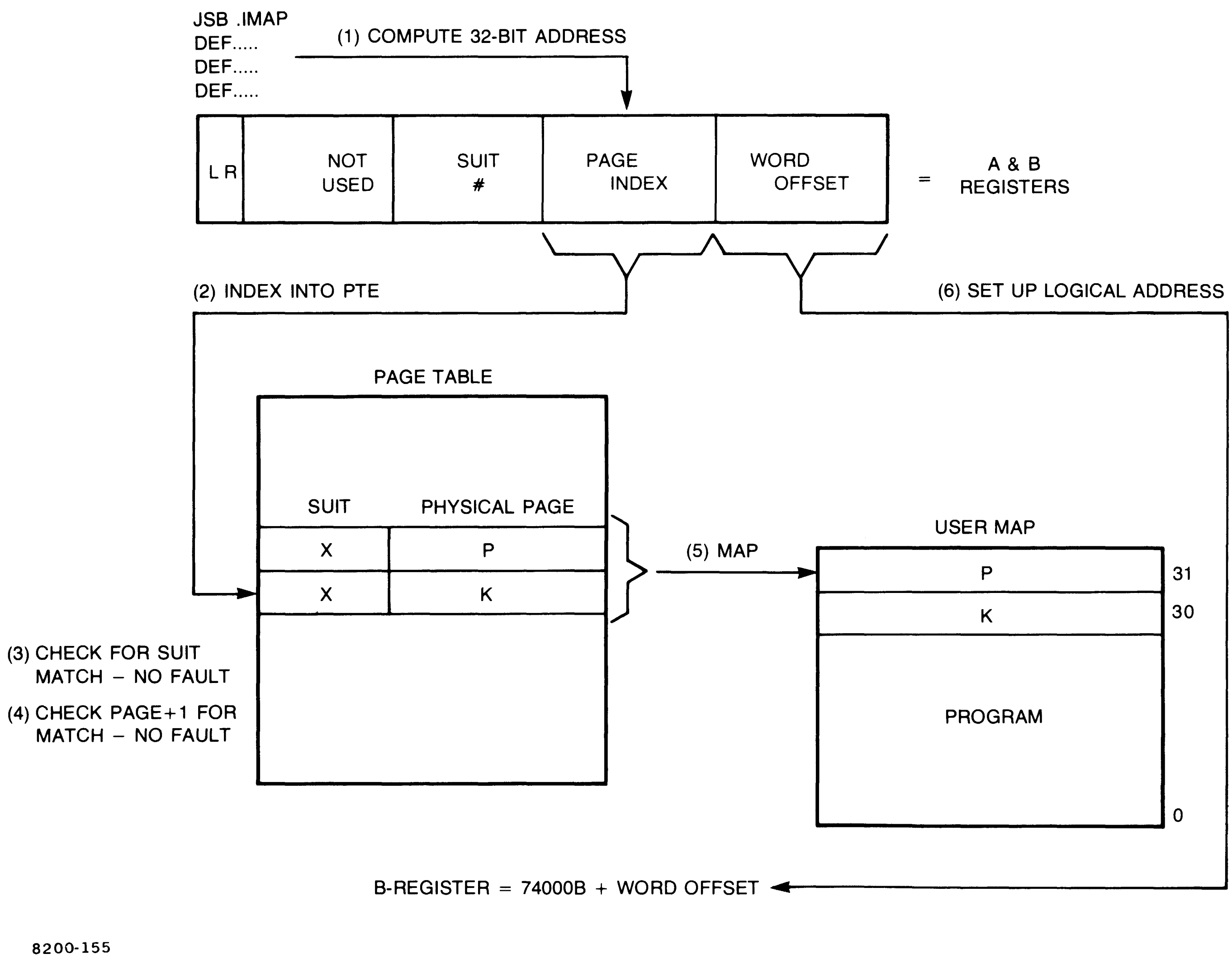


Figure 7-5. VM Data Item Address Derivation

When both the required page and the spillover page have been mapped in, return is made to the user program and, with the address of the required data item now in the B-Register, manipulation of the VM data space can take place.

7.6 PAGE FAULT

Referencing the example FORTRAN program, when array I has been initialized, page 310 will be filled since each element of I is one word long. When the program references the first element of array J, page 311 is referenced (step 1, Figure 7-6). Page 311 is mapped in and has a valid PTE entry (step 2). However, at this time the VM system also requires page 312 for spillover and that entry in the page table contains a NIL flag (step 3).

The NIL flag will force the microcode to enter VMA. This is called a page fault, the most common type of VM fault. (A number of conditions can give rise to VM faults; these are summarized later in this section.)

When a page fault occurs, VMA will allocate a page from the working set. However, when a NIL entry is found in the page table, it may signify either that this is the initial access to this virtual page or that this is a subsequent access to a virtual page which has been rolled out to disc. In the former case, it is sufficient to allocate a page and return to the user program. In the latter case, the required virtual page must be brought into main memory (i.e., copied into a free page in the working set) before the user program is re-entered.

This means that VMA must distinguish between the two situations just described. This is done in the routine (internal to VMA) DSCIO, which is used by the VM system to move virtual pages to and from main memory and backing store. When a working set page has been allocated, the routine DSCIO is called on the assumption that the required virtual page has to be read from the disc. If the backing store file extent corresponding to the virtual page does not exist, then this is the first access to that virtual page. In this case, no disc I/O need be done and DSCIO simply exits.

The backing store considerations of the VM system will be described more fully later.

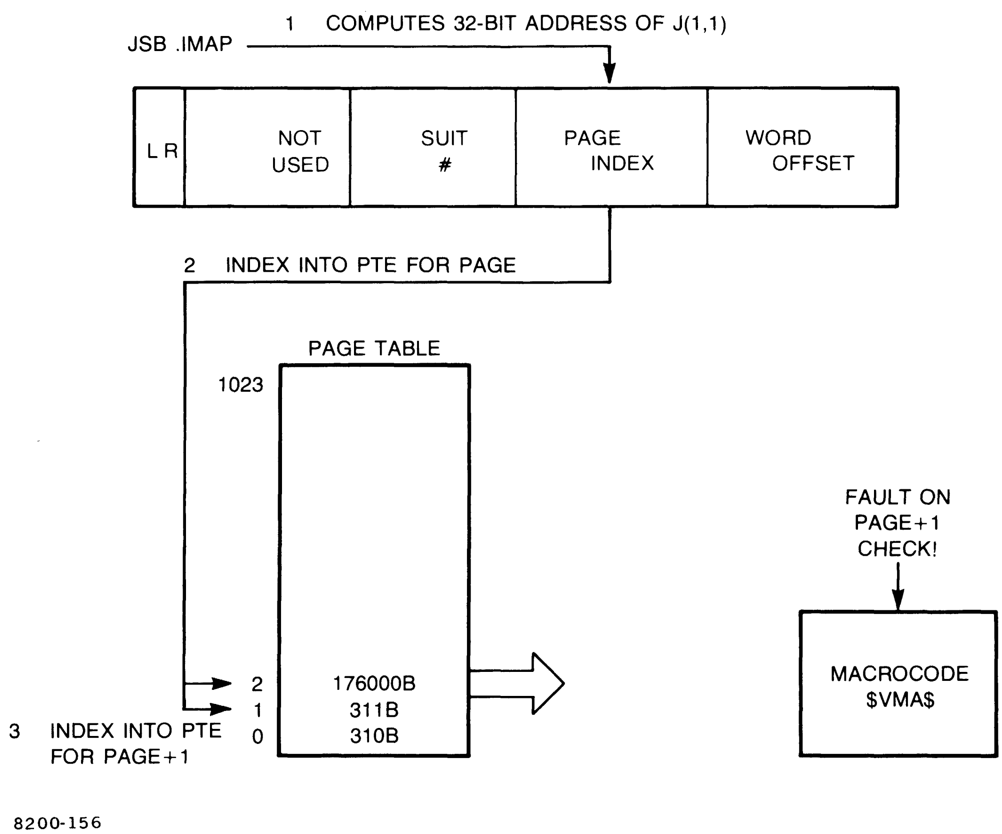


Figure 7-6. Array Referencing

7.6.1 VMA Functions

When VM data is initially accessed, the process involves leaving the VM microcode and entering the VM macrocode routine VMA to handle certain exception conditions such as uninitialized page tables. VMA is a system library routine that is appended to VM programs as the first item after the user program proper, as may be seen from a loader listing of a VM program.

VMA is responsible for handling all VM faults, including the page fault previously described. The complete list of VM faults is:

1. Page table not initialized.

2. Page table invalid following swap out/in.

3. Requested page not in main memory.

4. Synonym Collision.

In response to these VM faults, VMA will initialize/fix-up the page table, allocate a page from the working set and, if necessary, flush a page from the working set to disc and read in a page from the disc, and set up the page table entry accordingly.

7.6.2 Page Replacement in Working Set

Returning again to the example program, assume that I and the first 8 columns of J have been initialized. The working set and the page table now appear as shown in Figure 7-7.

Now the program attempts to access element J(1,9). This will require access to the next virtual page, which will be set up in main memory in physical page 321. However, when the VM system then tries to allocate the spillover page, it finds that the working set is full (recall the program was loaded with a working set size of 10 pages), causing a fault to VMA. To overcome this problem, one of the pages in the working set must be written to disc.

The working set page thus freed may now be allocated to the program in the manner already described.

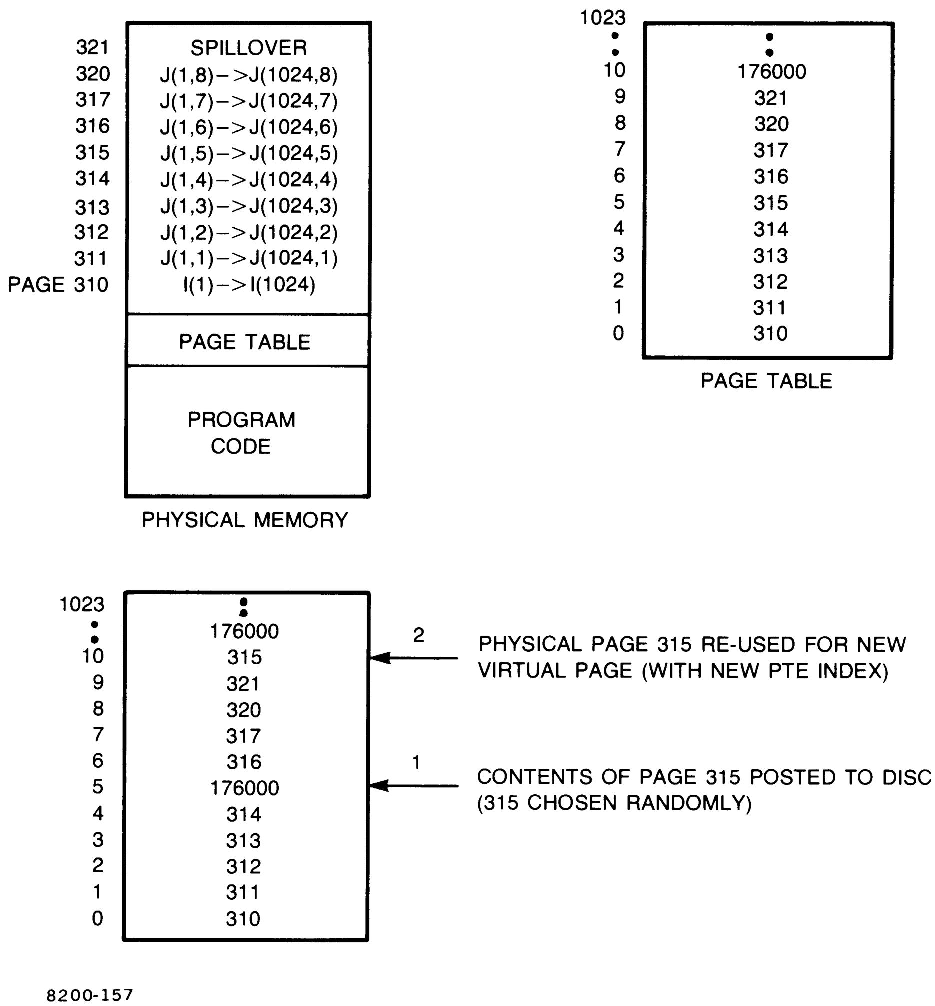


Figure 7-7. Working Set and Page Table Configuration

The important thing is how to decide which page in the working set to post to disc. In RTE-6/VM, a special algorithm (the Random Replacement Algorithm) is used to randomly select a page to post to disc.

As shown in Figure 7-7, the Random Replacement Algorithm has selected page 315 from the working set. The contents of this page are written to disc (1) at the appropriate place as determined by SUIT number and page number within the SUIT. The page table entry for this virtual page is reset to NIL. Physical page 315 in the working set is allocated as the required page. (If necessary, the required virtual page is read in from disc.) The page table entry for the new virtual page is set up with 315 as the physical page number (2).

7.6.3 Random Replacement Algorithm

The random replacement algorithm used by VMA to choose a page for posting to disc is:

```
LDA SEED
CLE,SLA,ERA
XOR POLY
STA SEED
    .
    .
    .
POLY OCT 132000
SEED OCT 123456
```

Mathematically, the algorithm is a random number generator used by VMA to generate a series of random numbers in the range of 0 to 1055. This range covers all possible entries in the page table (1024) plus all possible entries in the synonym table (32 entries). The synonym table is used to handle synonym collisions, described later in this chapter.

The requirements of the algorithm for RTE-6/VM are that it should provide a fast method of generating all of the numbers in the range of 0 to 1055 in a pseudo random fashion. That is, the same number sequence must result every time the algorithm is begun with a given seed value.

In terms of what is happening with the VM system, this pseudo random property of the algorithm means that a given series of events, such as accessing a given set of virtual pages in a given order and with a given working set size, will always result in the same pages being chosen for posting to disc. Further, these pages will always be chosen in the same order as the set of VM events occurs with time.

The algorithm uses three and sometimes four base set instructions because, on a given run, the resulting page table/synonym table index number may not provide a page that can be flushed and the algorithm may have to be executed a number of times before it is successful. (For example, the table entry may be NIL or the page may be in use by VMAIO.)

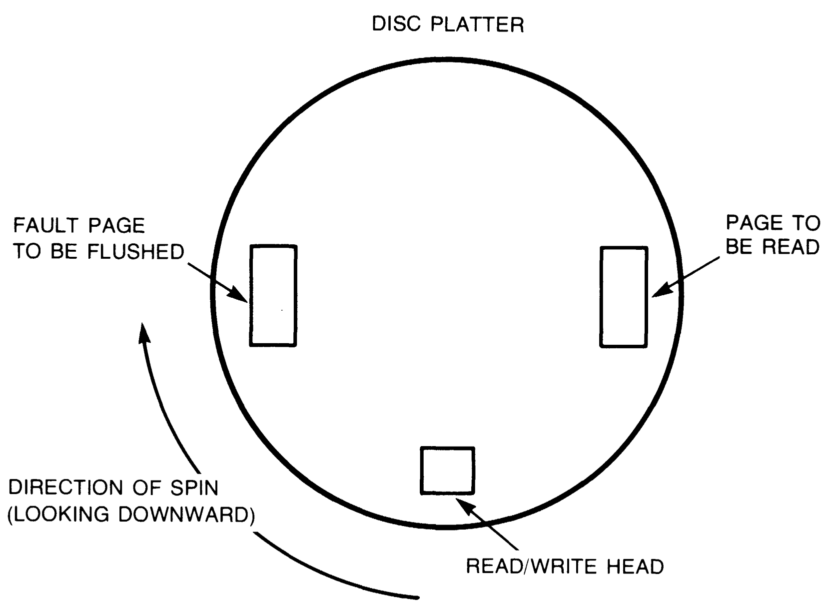
7.6.4 Monotonic Access Algorithm

In addition to the random replacement algorithm just described, RTE-6/VM employs another page replacement algorithm if the circumstances are correct for it.

If the user program has been continuously accessing the virtual memory space in the same direction, and in so doing has caused 16 successive page faults in allocating the spillover page, RTE-6/VM will switch over to the monotonic access algorithm to choose a working set page to post to disc.

Using this algorithm, the page chosen to be posted to disc is four pages either forward or backward in the page table, depending on the direction of access. For all current disc types, four pages is sufficiently close to half a track that the flush page can be written to disc during the same disc revolution in which the new page is read from the disc (see Figure 7-8).

The resultant overall VM disc access time for programs running in this mode is reduced by a factor of about three as compared to running with random replacement. Additionally, the monotonic access algorithm itself is faster than random replacement.



8200-158

Figure 7-8. Monotonic Access to Disc

7.7 SYNONYM COLLISIONS

As previously described, the SUIT number derived when the VM system calculates the 26-bit address of a data item may not match the SUIT number of the required entry in the page table (access to a given page number within any suit requires the same page table entry). This is called a synonym collision, which occurs when a user program attempts to access a virtual page for which the required page table entry is already occupied. A synonym collision is recognized by the VM microcode, which then enters the macrocode routine VMA to resolve the situation.

Again referencing the example program, when array I(1024) has been initialized and the first 1022 columns of array J(1024,1024) have been initialized, reference is next made to J(1,1023).

This reference will cause the VM system to allocate the last page of SUIT 0 to a page in the working set and then, for spillover, the first page in SUIT 1 will be allocated. This means that the program is attempting to have both virtual page 0 and virtual page 1024 in the working set simultaneously. This is a synonym collision since these virtual pages (the synonym pages) are both page 0 within their SUITS and thus both require the first entry in the page table to describe their location in main memory.

In order to handle this situation, the VM system makes use of another table called the synonym table. The procedure is that the existing page table entry is moved into the synonym table and that slot in the page table is set up to describe the new page. If access is subsequently made to the virtual page now described by the synonym table entry, this will constitute another synonym collision and the result will be that the synonym table entry and the page table entry will be swapped around to give the required access via the page table.

The above description assumes that none of the unpleasant corner cases arise - such as working set full, synonym table full, etc. Some of these more complex synonym collision cases will be described later in this chapter.

7.7.1 VM Synonym Table (STE)

The SYNONYM TABLE is 64 words long and may hold up to 32 two-word entries. The format of the first entry word, the PAGID, is:

```
 15       9                     0
+---------+---------------------+
| SUIT #  |     INDEX INTO PTE  |
+---------+---------------------+
```

The second entry word contains the SUIT number and the physical page number entry from the page table. The table is not sparse, as the page table is. When an entry in the synonym table is required, the next entry is used. When a synonym table entry is deleted, the remaining entries are packed. The table was structured in two halves to simplify and speed up searching.

The synonym table occupies the first 64 words of VMA. When the synonym table is full (an unlikely event) and a synonym collision occurs, a working set page must be flushed to disc. This page must be chosen by random replacement from the synonym table only. This page flushing will happen even if the working set itself is not full.

The synonym table is the principal means whereby a VM system with a 1024 word page table can control access to a virtual store size of 65536 pages. The mechanism trades on the idea of locality of reference to some extent, but this is true of VM systems in general.

7.7.2 Synonym Collision Handling (VMA)

In the case of a synonym collision where the fault page is in memory and there is an entry in the STE, VMA swaps the PTE entry with the STE entry and re-executes the faulting instruction.

If a synonym collision occurs where the fault page is not in memory and no synonym currently exists AND the working set and the STE were not full, the next free page is allocated, and the fault PTE entry is placed in the next available STE entry.

However, more complex situations can occur where the fault page is not in memory and either or both the working set and STE are full. In all cases, the objective of the actions on the part of VMA is to read the page into memory from disc, writing a page to disc if necessary to make room for a fault page.

If the working set is not full and the STE is full, OR if both are full, VMA:

1. Selects Flush page from STE (using the Random Replacement Algorithm).

2. Swaps the Flush STE entry with the Fault PTE entry.

3. Writes the PTE entry to disc.

4. Reads the Fault page from disc.

If the working set is full and the STE is not full, VMA

1. Selects Flush page from PTE-STE, using either the Random Replacement Algorithm or the Monotonic Access Algorithm.

2. If the Flush page is from PTE and has a synonym, swaps the STE entry with the PTE entry Flush page and swaps the Flush page in the STE with the Fault page in the PTE. Otherwise, places the Fault PTE entry in the STE.

3. Writes the PTE entry to disc.

4. Reads the Fault page from disc.

Figure 7-9 is a summary flowchart of the portion of VMA that handles synonym collisions.

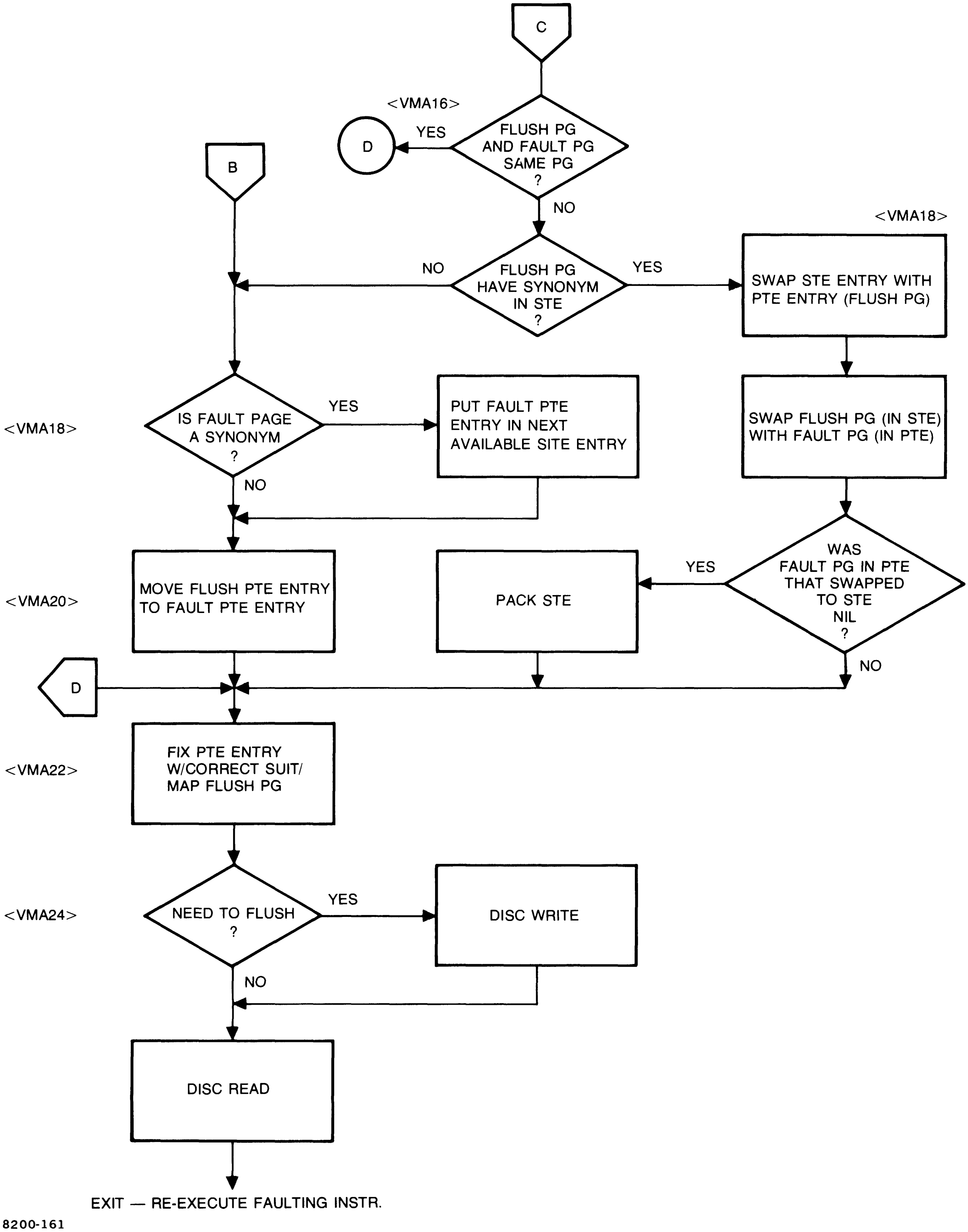


Figure 7-9. VMA Handling of Synonym Collisions, Flow Chart

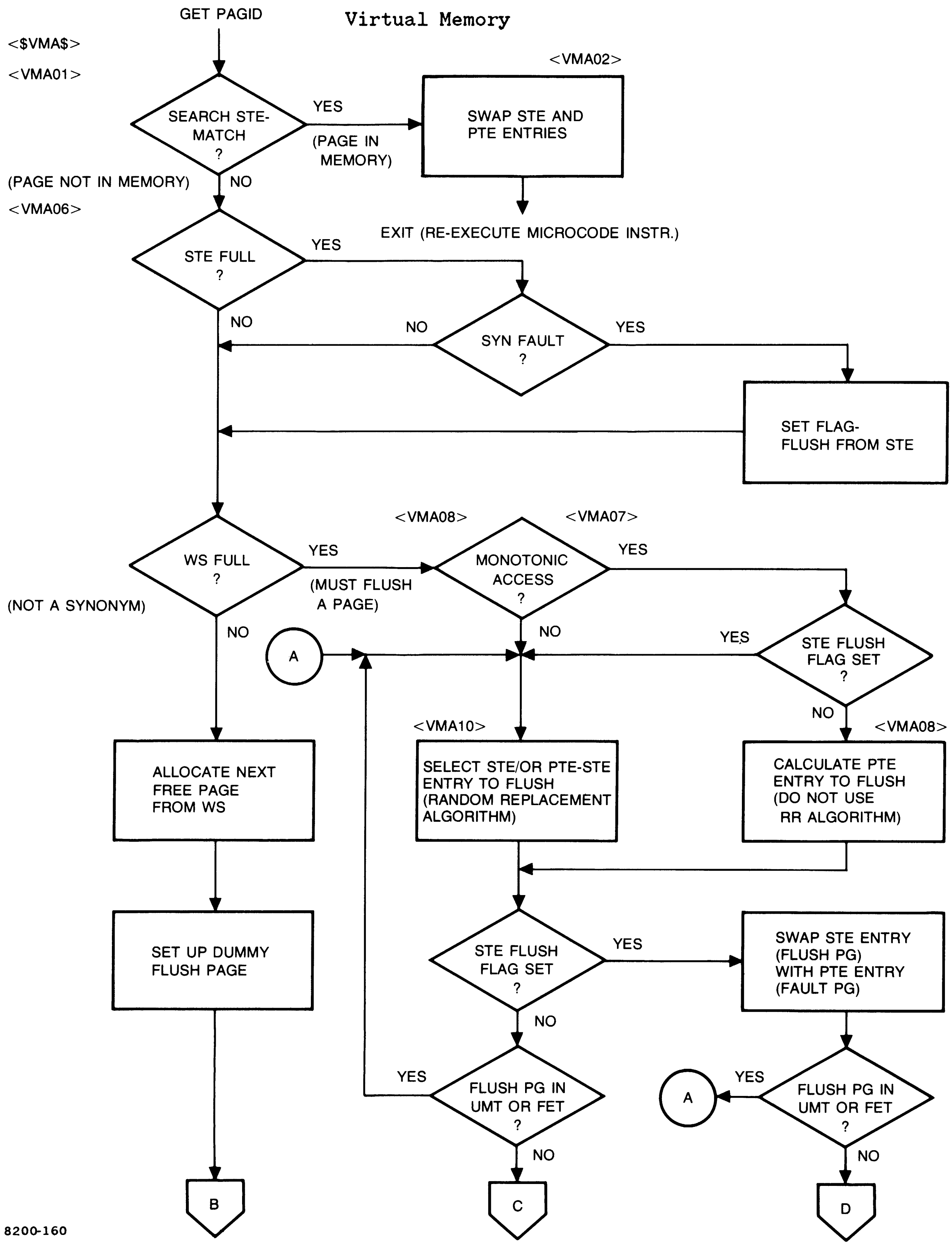


Figure 7-9. $VMA Handling of Synonym Collisions, Flow Chart (Continued)

7.8 PROGRAM SWAPPING

When a VM program is swapped out, its page table is swapped with it and the SW bit is cleared (bit 13 of ID extension word 2). On swap-in and re-dispatch, the VM microcode makes the following checks:

> If base page location 1776B is non-zero, the VM system is deemed to be intact because this is a subsequent entry on a given dispatch.
>
> If location 1776B is zero, this is the first time into the VM system on this dispatch of the program (the dispatcher had set 1776B to zero). At this time the microcode writes word 1 of the ID segment extension into location 1776B. This word contains the physical page number of the page table, and is used by the microcode to map in the page table.

"First time" into the VM system on a given dispatch may mean three things: initial dispatch, subsequent dispatch following a swap, or subsequent dispatch with no swap. In the latter case (when the SW bit = 1) the page table is valid. In the first two cases the microcode will cause entry to VMA with bit 15 of the Y-Register set. This, in turn, will cause VMA to call SWP to fix up the page table. If the value of word 3 of the user program preamble (the "old page table" word) is zero, SWP knows it must initialize the page table.

If the "old page table" word is non-zero and equal to the "new page table" word (ID segment extension, word 1), the program has been swapped back into the partition it previously occupied. Here again, the page table is valid.

If the "old page table" word is non-zero and not equal to the "new page table" word, SWP must process each entry in the page table. Each entry in use contains the physical page number of a page in the working set. These physical page numbers are updated by the difference between the old and new page table addresses. A similar adjustment is made to entries in the synonym table.

7.9 USER PROGRAM PREAMBLE

The following information is contained in the ten-word user program preamble:

```
12B  |      Program Load Point         |
     +---------------------------------+
11   |           Reserved              |
     +---------------------------------+
10   |           Reserved              |
     +---------------------------------+
 7   |           Reserved              |
     +---------------------------------+
 6   |           Reserved              |
     +---------------------------------+
 5   |     Currently Enabled Path      |
     +---------------------------------+
 4   |           Reserved              |
     +---------------------------------+
 3   |             $PTE$               |
     +---------------------------------+
 2   |         Def $xMA$ +0            |
     +---------------------------------+
 1   |               Y                 |
     +---------------------------------+
 0   |               X                 |
     +---------------------------------+
     |           Base Page             |
     |                                 |
```

Word 2 is set up by the loader according to whether the program is running EMA or VMA. This location contains the address of the macrocode routine to be invoked to handle faults.

Word 3 is initially zero, but SWP sets it up to point to the page table when it is called to initialize or fix up the table.

7.10 INFORMATION SUPPLIED BY ID SEGMENT

The following information supplied in the ID segment and extension is relevant to the VMA/EMA system.

```
                    Word
                            15            9                0
                           +--------------+----------------+
ID Segment           28    |  ID Ext. #   |    EMA Size    |
                           +--------------+----------------+

                            15            9                0
                           +--------------+----------------+
ID Extension          0    |            Not Used           |
                           +-----------+--+----------------+
                      1    | Log St.   |DE|Phys St. Page EMA |
                           +---+--+--+--+--+----------------+
                      2    |COM|  |SW|    | Index # to $EMTB |
                           +---+--+--+----+----------------+
                      3    |  Maximum Page # Allowed in VMA  |
                           +-------------------------------+
                      4    |  Reserved: Dynamic VM miss Rate |
                           +-------------------------------+
```

Where: EMA is size of working set, including PTE, in pages

DE = 0, EMA size is specified by user
DE = 1, EMA size defaulted
COM = 1, program using shareable EMA
SW = 0, PTE needs fixup (first dispatch or swap)
SW = 1, PTE intact

Word 1 stored in 17776B by microcode:
Low 10 bits - PTE location
High 5 bits - determine which user map registers may contain code

Word 3 = 0 if EMA program
= # VMA pages - 1 if VMA program
(range 0,9...65535)

7.11 OTHER VM TABLES

The two remaining tables in the VM system, the Fault Exclusion Table (FET) and the current User Map Table (UMT), are described below. Each is a 32-word table in a fixed location in VMA: the FET occupies words 100B to 137B, and the UMT occupies words 140B to 177B.

The FET is used to hold the PAGID (SUIT # and PTE index) of pages which may not be flushed to disc by the VM system. The most common use of the FET is to identify the last virtual page mapped in (other than by .PMAP), since this is a page that should be kept in memory. That is, if virtual page n is accessed last by the user and the VM system then tries to map page n+1 for spillover, but a page must be flushed from the working set to achieve this, page n will not be a candidate for flushing.

The other use for the FET is with I/O. The library routine VMAIO records the PAGID of each page of the data buffer in the FET. Then, when it calls .PMAP to map the buffer pages, one buffer page will not be flushed out to make room for another.

The UMT contains a copy of the user's map registers, and is updated at every VM fault. A variable (DFVMT) is set up to limit the extent of the UMT that will be tested by VMA when determining if a given page may be flushed. The normal use of this table and pointer is to lock in memory all of the user's logical pages up to the first page below the MSEG area. This is done by setting DFVMT to the number of the last code area map registers. Since VMA will not flush any page in the UMT (up to the limit of register DFVMT), it is impossible to flush a code page. However, in normal RTE-6/VM operation, a code page could never be flushed anyway since these pages never appear in the page table or the synonym table.

The UMT is useful for Pascal programs, since Pascal does its own mapping and effectively has virtual code capability. Additionally, the UMT provides a structure upon which virtual code can be easily added through VMA.

7.12 VIRTUAL MEMORY MICROCODE ROUTINES

VM system microcode routines The following microcode routines are used with Virtual Memory:

.PMAP This routine is called to map a page into the specified user map register. The routine is used by VMAIO, .ESEG and other user-level VM mapping routines. .PMAP does very little error checking, and is not user-callable.

.IMAP This routine resolves the EMA array element address and maps the last two map registers. The routine is planted in a user program by the compiler if the offset into the VM area can be contained in 16 bits.

.JMAP This routine is functionally identical to .IMAP and is used when the offset into the VM area is contained in 32 bits.

.IMAR This routine is used to resolve VM address in the same manner as .IMAP, but does not map the pages. You must make a call to one of the mapping routines. .IMAR is used when the offset into the VM area is contained in 16 bits.

.JMAR This routine is identical to .IMAR and is used when the offset into the VM area is contained in 32 bits.

.LBP This routine is used by the Pascal compiler to map the 32-bit pointer to the A and B Registers.

.LBPR This routine is used by the Pascal compiler to map the 32-bit pointer to P+1.

.LPX This routine is used by the Pascal compiler to map the 32-bit pointer to P+1 plus offset (A and B Registers.)

.LPXR This routine is used by the Pascal compiler to map the 32-bit pointer to P+1 plus offset (P+2).

These microcoded routines communicate with the VM macrocode via the X-Y save area in the program preamble (words 0 and 1). The X-Y save area for the current program is always pointed to by base page location 1647B. For example, the microcode will put the page table address in the Y-Register save area before calling VMA. It will also set bit 15 of this word if it requires VMA to fix up or initialize the page table. Before the dispatcher runs the current user program, the index registers are set up from the save area; in this way, microcode/user program communication is achieved.

7.13 VIRTUAL MEMORY MACROCODE ROUTINES

The following macrocode routines are used with virtual memory:

EMA	This routine provides the EMA interface to the EMA/VMA system, performing tasks similar to VMA. In RTE-6/VM, the extended memory area is a subset of virtual memory in which the whole VMA space resides in main memory.
VMA	This routine was discussed in detail earlier in this chapter. In brief, VMA handles fault from microcode, gets a page into memory from non-allocated working set or disc, notes the page location on the Page Table Entry (PTE), and posts a page from the working set to disc when required.
SWP	This routine was discussed in detail earlier in this chapter. In brief, SWP initializes the PTE and adjusts the PTE after a swap operation.
.ESEG	This routine sets sequential map registers from a table of VMA page numbers.
MMAP	This routine maps a specified number of pages, given offset into the EMA/VMA area.
.IRES	This routine resolves EMA array addressing.
.JRES	This routine is identical to .IRES, except that it handles double-integer array dimensions
VMAIO	This routine handles large I/O transfers to and from EMA/VMA.
VMAST	This routine returns the version of EMA or VMA currently executing. It also returns the size, in pages, and the location of the error entry point into EMA and VMA.
MAPMS	This routine allows the user to run VMA on an HP 1000 M-Series computer. It provides the software versions for M-Series microcode mapping -- .PMAP and .LBP.

The routines .ESEG through VMAST are all available to users.

7.14 VM BACKING STORE FILE HANDLING

By default, the VM system will create, open, close and purge the VM backing store files. It will also post working set pages to the disc. The default file names used for VM scratch files are derived as a six-character name:

XXXCVM

where

- XXX is the ID segment number of the executing program (0-255)
- C is the processor number (currently zero)
- VM is "VM" to identify a VM scratch file.

The scratch files are built on the disc cartridge specified by the :VL command (normally in the system WELCOM file) or, if no :VL command has been issued, the top cartridge in the user's list is used (excluding system and auxiliary discs LU 2 and LU 3). The recommended usage is to set up a system cartridge for use in spooling and (via the :VL command) for VM and Edit scratch files.

For users who require a permanent VM disc file, there is a set of routines which will create, open, etc., a named file on a given CRN. The calls to these routines are given at the end of this section.

Thus, although VMA is not shareable in the sense that EMA is in RTE-6/VM, it is possible to create a VM space on disc and leave it intact when the creating program goes away. Subsequent programs may then open the file and access it via the VM system in the normal way, giving a kind of sequential sharing facility. The VM files are created as a series of file extents which are set up on disc as required. The extent size is calculated as

Virtual Store Size (pages)/256

rounded up to the nearest 32K words (256 blocks). The VM files are type 2 files with a record length of 1024 words.

7.14.1 CREATING, MANIPULATING VM FILES

CREVM - Create a backing store file

```
CALL CREVM([NAME[,IERR[,IOPTN[,ISC[,ICR]]]]])
```

where

NAME is six ASCII characters.

IERR specifies the return of a one-word error code.
A "0" is returned on successful calls.

IOPTN specifies the file options:

Bit 0 = 1, create non-scratch file (NAME parameter given)
= 0, ignore name, option bit 1.

Bit 1 = 1, open file even if create fails due to duplicate-file error

Bit 2 = 1, defer creation until file required

Bit 3 = 1, do not create or address file extents.

ISC specifies the file security code.

ICR specifies the cartridge on which the file is to be created.

If no parameters are passed, a scratch file is created immediately upon execution of the CRECM call.

OPNVM - Open a backing store file.

```
CALL OPNVM(NAME[,IERR[,IOPTN[,ISC[,ICR]]]])
```

where

NAME is the six-character ASCII
name assigned (or defaulted) using CREVM.

IERR specifies the return of a
one-word error code. A "0" is returned on successful calls.

IOPTN specifies the file options:

Bit 0 = 1, file to be opened non-exclusively

Bit 1 = 1, file to be opened for update (read/write access)

Bit 2 = 1, file open to be deferred until required

Bit 3 = 1, do not create or address file extents

Bit 4 = 1, read-only file access.

ISC is the security code. This must match the code specified by CREVM, except for read-only access calls.

ICR is the cartridge reference number (CRN)

The following routines do not have parameters; only the routine CALL is necessary.

PSTVM - Post the working set to disc.

This routine can be called at any time to post the entire working set to the backing store disc file. If, however, the file was opened for read-only access, posting does not occur.

CLSVM - Close the backing store file.

If the file was created or opened with CREVM or OPNVM, this routine should be used to close the file to guarantee that the working set is posted to the file.

PURVM - Purge the backing store file.

This routine should be used to purge the backing store file.

ON-LINE GENERATOR — CHAPTER 8

This chapter is intended to serve as an aid to the programmer when modifying the on-line generator program, RT6GN. It should be used in conjunction with the generator source listings, as it assumes familiarity with RTE-6/VM and its system generation process. It assumes familiarity with the RTE-6/VM On-Line Generator Reference Manual outlining the generation process.

The modularity of the RTE software makes it easy to configure a real-time operating system tailored to particular application requirements for input/output peripherals, instrumentation, program development, and user software. With the on-line generator a configuration can be achieved under control of the present RTE system, concurrent with other system activities. The on-line generation process utilizes the file management features of RTE for the retrieval of the generation parameters and software modules, for the output of the system bootstrap loader and for the actual storage of the absolute system code and its associated generation map. The special utility program SWTCH performs the switchover from the present system configuration to that of the new.

8.1 OPERATION

RT6GN is a type 2 (real-time) or type 3 (background) segmented program which may be run as a type 2, 3, 4 (large background) or 6 (extended background) program in RTE-6/VM. The generator accepts its command input from an ANSWER file located on disc, a logical unit (LU), or a combination of the two. These parameters direct the generator in building and defining the system tables and values, the logical memory layout, the physical memory layout, and in relocating the software modules to be included in the system. All relocatable modules must exist in FMP file format and are specified by file name to be included in the system. The absolute, memory-image system being built is itself stored in a type 1 FMP file, which is then transferred by SWTCH.

8.2 GENERATION SEQUENCE

In the following example of the generation sequence, the dashed lines indicate where your response to the prompt is to be entered, followed by a carriage return.

```
LIST FILE NAMR?

--------

ECHO?

--------

OUTPUT FILE NAMR?

--------

SYSTEM DISC MODEL?

--------


*for HP 7900/7901 Disc Only:

CONTROLLER SELECT CODE?

--------

 #TRKS,FIRST TRK ON SUBCHNL?
  0?

 --------,--------
  .
  .
  .
 *for HP 7905/06/20/25 Disc Only:

 CONTROLLER SELECT CODE?

 -------

 MODEL,#TRKS,FIRST CYL,HEAD,# SURFACES,UNIT,# SPARES FOR SUBCHNL:
  00?

 -------,-------,-------,-------,-------,-------
  .
 .
 .
*for HP 7906H/20H/25H/9895 Discs Only:

MODEL,#TRKS,FIRST CYL,HEAD,#SURFACES,ADDRESS,#SPARES(,UNIT) FOR SUBCHNL:

-------,-------,-------,-------,-------,-------,-------
```

```
 .
 .
*for CS80 Discs only:

CONTROLLER SELECT CODE?

 -------

 DEVICE  (MODEL,HP-IB ADDR,UNIT,VOLUME)?

 ----,----,----,----
 xxxxxx BLOCKS REMAINING
 SUBCHANNEL n (TRACKS,BLOCKS/TRACK)?

 ----,----
  .
  .
  .
 SYSTEM SUBCHNL?

 --------
                          no
 *    +--------AUX DISC?---------->DISC MODEL#?
 *    |           |yes               |        |
 *    |           |              non CS80   CS80
 *    |           |                  |        |
 *    |   AUX DISC SBCHNL?           |        |
 *    |           |                  |        |
 *    |           |                  |    BLKS/TRK?
 *    |           |                  |        |
 *    |           |                  |        |
 *    |           |                  |        |
 *    |           |                  |        |
 *    |           v                  |        |
 *    +----->TBG CHNL? <-------------+<-------+

 TBG SELECT CODE?

 -------

 PRIV. INT. SELECT CODE?

 -------

MEM. RES. ACCESS TABLE AREA II?

-------

RT MEMORY LOCK?

--------------
```

```
BG MEMORY LOCK?

--------------

SWAP DELAY?

 --------------

 MEM SIZE?

 -----

 BOOT FILE NAMR?

 ------------

 PROG INPUT PHASE:

 -----------
 -----------
 -----------
 -----------
  .
  .
  .
 -----------
 -----------
 /E

 PARAMETERS

 -----------
 -----------
 -----------
  .
  .
  .
 -----------
 -----------
 /E

CHANGE ENTS?

-----------
-----------
-----------
 .
 .
 .
-----------
```

```
-----------
/E

TABLE AREA I <<PAGE XXXXX>>:

EQUIPMENT TABLE ENTRY
EQT 01?

 -------,-------,-------,-------,-------,-------,-------,-------
 EQT 02?

 -------,-------,-------,-------,-------,-------,-------,-------
 EQT 03?

 -------,-------,-------,-------,-------,-------,-------,-------
 EQT 04?

 -------,-------,-------,-------,-------,-------,-------,-------
 EQT 05?

 -------,-------,-------,-------,-------,-------,-------,-------
   .
   .
   .
 /E

 DEVICE REFERENCE TABLE
 001 = EQT #?

 -------,-------
 002 = EQT #?

 -------,-------
 003 = EQT #?

 -------,-------
 004 = EQT #?

 -------,-------
   .
   .
   .
/E

INTERRUPT TABLE

   4   ,   ENT   ,  $POWR
-------,-------,-------
-------,-------,-------
-------,-------,-------
-------,-------,-------
```

```
-------,-------,-------
-------,-------,-------
-------,-------,-------
-------,-------,-------
-------,-------,-------
-------,-------,-------
-------,-------,-------
-------,-------,-------
 .
 .
 .
/E

TABLE AREA I MODULES
|
|
|   load map
|
v
DRIVR PART 00002
CHANGE DRIVR PART?

--
DP 01 <<PAGE XXXXX>>:
|
|
|   load map
|
 v
 SUBSYSTEM GLOBAL AREA <<PAGE XXXXX>>:
 |
 |
 |   load map
 |
 v
 RT COMMON XXXXX
 CHANGE RT COMMON?

 ---------

 RT COM ADD YYYYY

 BG COMMON XXXXX
CHANGE BG COMMON?
BG COM ADD YYYYY

------

BG COMMON XXXXX

SYSTEM DRIVER AREA <<PAGE XXXXX>>:
```

```
|
|
|    load map
|
v
TABLE AREA II <<PAGE XXXXX>>:
 # OF I/O CLASSES?

 -------------
 # OF LU MAPPINGS?

 -------------
 # OF RESOURCE NUMBERS?

 -------------
 BUFFER LIMITS (LOW,HIGH)?

 -------,-------
 XXXX LONG ID SEGMENTS USED

 -------,-------
 # OF BLANK LONG ID SEGMENTS?

 -------
 XXXX SHORT ID SEGMENTS USED

 -------,------
 # OF BLANK SHORT ID SEGMENTS?

 -------
 XXXX ID EXTENSIONS USED
 -------,------
 # OF BLANK ID EXTENSIONS?

 -------
 MAXIMUM # OF PARTITIONS?

 -------
 TABLE AREA II MODULES
 |
 |
 |    load map
 |
v
SYSTEM <<PAGE XXXXX>>:
|
|
|    load map
|
v
OS PARTITION 1 <<PAGE xxxxxx>>
```

```
|
|
v
OS PARTITION 2 <<PAGE yyyyyy>>
 .
 .
 .
 PARTITION DRIVERS
 |
 DP 02 <<PAGE XXXXX>>:
 |
 |    load maps
 |
 v
 DP 03 <<PAGE XXXXX>>:
 |
 |
 |
 v
 MEMORY RESIDENT LIBRARY <<PAGE XXXXX>>:
 |
 |    load map
 |
 v
 MEMORY RESIDENTS <<PAGE XXXXX>>:
 |
 |    load map
 |
 v
 RT DISC RESIDENTS
 |
 |    load map
 |
 v
 BG DISC RESIDENTS
 |
 |    load map
 |
 v


 RT PARTITION REQMTS:
PNAME XX PAGES E
 .
 .
 .
BG PARTITION REQMTS:
PNAME XX PAGES *E
 .
 .
 .
```

```
MAXIMUM PROGRAM SIZE:
W/O COM YY PAGES
W/  COM ZZ PAGES
W/  TA2 XX PAGES

SYS AV MEM XXXXX WORDS

 ENTER 1ST PARTITION PAGE:  XXXXX (DEFAULT) TO YYYYY:

 -------

 SYS AV MEM XXXXX WORDS

 PAGES REMAINING:  XXXXX

 DEFINE PARTITIONS
 PART 01, XXXX PAGE?
  .
  .
  .
 SUBPARTITIONS?
  .
 PART 02, XXXX, (YYYY) PAGES?
  .
  .
 /E

 MODIFY PROGRAM PAGE REQUIREMENTS?

 -------
 -------
  .
  .
  .
 /E

 MAX # SHAREABLE EMA PARTITIONS IS zzz
 SHAREABLE EMA PARTITIONS?
  .
  .
  .
/E

SHAREABLE EMA PROGRAMS?
 .
 .
 .
/E

ASSIGN PROGRAM PARTITIONS?
```

```
-------
-------
 .
 .
 .
/E

SYSTEM STORED IN FILE
 SYS SIZE: XXX TRKS, YYY SECS (ZZ SECTORS/TRACK)
           =TTTTT BLOCKS (128 WORDS/BLOCK)

 RT6GN FINISHED

 zzzz ERRORS
```

8.3 FILE INTERFACE

All I/O within the generator is handled through FMP calls, be it to answer, list, boot, echo, relocatable, absolute, or scratch files. Where I/O to a specific LU is allowed (answer file, list file, boot file, or echo), a dummy type 0 file DCB is created so that the same READF, WRITF, and CLOSE calls are used throughout. Six DCBs are set up and used (and sometimes reused) for file I/O:

\ADCB Absolute output file - always open to a file

\LDCB List file - always open to a file or LU

\IDCB Input file - always open to a file or LU (changes as TRansfers and erors occur)

\EDCB Echo - always open to LU of operator console but not necessarily used if \IDCB or \LDCB are used to same LU, or if option denied.

\RDCB Relocatable input file - used to reference all relocatable files during generation, open to only one file at a time

\BDCB Boot file - created only when boot file is output by PTBOT routine.

\NDCB Modified NAM records file (@@NM@A) - scratch file open when being built and when referenced during relocation

All files except the answer files and relocatable input files are created by the generator. The above two file categories cannot be actual type 0 files, as the generator may reference them by record number. In the case of the relocatable files, the generator actually opens and closes each file many times.

8.4 INTERFACE ROUTINES

\CRET - is passed a DCB address and creates a file whose name is at PARS2+1,+2,+3, security code is at PARS3+1, CRN is at PARS4+1, and size is at PARS6+1.

\CRET first calls FOPEN which calls TYPO - if a type 0 dummy DCB was built then that is sufficient and \CRET returns. If it was a file, then that file is closed (no error check done here on file since may never have existed), and then created by a CREAT call.

A CREAT call is made with the assumption that whoever called \CRET checks the FMP error parameter FMRR.

\CLOS - is passed the DCB address and truncate option. For a file, a simple CLOSE call is made, leaving to the \CLOS caller the responsibility for checking \FMRR (not usually done).

For a dummy type 0 file, however, word 9 is merely set to 0. If the type 0 file being closed is the list file, then a page-eject control request is made to it. The no-abort bit is set on the control request to prevent abortion of the generator to a device with no EOF code (like the console).

\OPEN - is passed a DCB address, and attempts to open a file whose name is in PARS2+1,+2,+3, security code in PARS3+1, and CRN in PARS4+1.

A call to TYPO determines if an LU was specified in the first parameter and TYPO sets up the dummy DCB. (\FMRR is always cleared.)

For a file, an OPEN call is made leaving the check of FMRR up to the caller of \OPEN.

TYPO - is passed the DCB address in the A-register. It determines whether a numeric parameter was specified as a file name, in which case it will continue with the building of a dummy DCB. LUs are allowed by the generator for answer, list and boot files; echo is always to the LU of the operator console (ERRLU).

The dummy DCB format and initial values are:

```
Word 0-2 directory address of file type
Word 3   read/write subfunction, LU
Word 4   EOF control subfunction, LU
Word 5   0 no spacing legal
Word 6   100001 read/write legal
Word 7   100030 security codes agree; update open
```

Word 8 -----
Word 9 ID segment address of generator (from 1717)
Word 10 -----
Word 11 -----
Word 12 -----
Word 13 1 (initial value) Record Number (Low Word)
Word 14 Record Number (High Word)

Special checks are made in determining the EOF control subfunctions. For driver types >=17 and for DVR05 (exclusive of subchannel 0), 0100 is merged with the LU. For DVR00, DVR02, DVR05, and DVR07 (subchannel 0 only), the EOF control subfunction 1000 is merged with the LU. For all other driver types between 1 and 16, 1100 is the merged subfunction. For non-type 05 or 23 devices, an EOF will be sent immediately-causing leader or a page eject, respectively.

8.5 SCRATCH FILE

The generator creates a temporary file of its own for storage of modified NAM records, @@NM@A. Modified NAM records result when the program length of a compiled program has been determined (during the Program Input Phase), or when program priority or execution interval are changed during the Parameter Phase. If such a modified NAM record does exist for a program, bit 14 of ID5 in its IDENT entry is set so that the correct values may be retrieved during relocation.

The generator purges this scratch file during final clean-up or its own abortion clean-up. The file will still remain, however, if the generation is aborted by some other means. When the generator tries to create the scratch file during initialization and finds that it already exists, it will increment the last character of the name (e.g., A>B) and create a new one. It gets confused if there exist old entries in a file left over from a previous generation, so a new file is always created.

8.6 RELOCATABLE INPUT

All relocatable input is handled through the routines \RNAM and \RBIN (both in the main). \RNAM sets up the parse buffer to open the file specified in the current IDENT entry (words \ID9 through \ID13). A non-zero B-register on entry to \RNAM indicates that the file is still open. Otherwise, the relocatable file currently open to \RDCB is closed. \RBIN is called to (possibly) open the file, and to read the record specified by \ID14 through \ID16. \RBIN may also be called to read the next relocatable record of a file and, optionally, to get its position.

If possible, \RBIN also converts new relocatable format records into old format records by calling the routine MREC. \RBIN examines the type field of the binary record; if it is found to be an extended format record it is passed to MREC, which performs an in-place conversion. The converted record then is passed back to the caller in the same buffer used to pass the record to MREC. Note that an extended record may convert to more than one old format record. Because of this, MREC also returns a count of the number of records in the buffer.

8.7 ANSWER FILE

Upon start-up, the generator determines through RMPAR and GETST calls whether an answer file name or LU was specified via the turn-on parameters. If the first parameter is 0, LU 1 becomes the default command (answer) LU. If the first parameter is numeric, the named LU is used for command input (in an MTM environment, this is the operator console LU, provided no parameters were specified). A dummy DCB is created in TYPO for the LU, or the answer file specified in the Namr parameters is opened via FMP. If an error occurs on the answer file open call, the appropriate error message is displayed on the console via an EXEC call, and control is transferred to LU 1.

An "error LU" is also defined at start-up. If an LU was obtained from either the turn-on parameter or the default command LU 1, that LU becomes the error LU provided it is an interactive device. If it is not interactive (a photoreader for example), the error LU defaults to LU 1.

When an error occurs, the error message is sent to both the list file and the error LU. For many errors, control is transferred to the error LU for corrective action by the operator. This is done by stuffing a "TR,ERRLU" into the command buffer, where ERRLU represents the two-digit error LU. The error processor \GNER then calls TRCHK, which processes the TR command. If the command input was from an interactive LU, control is not transferred.

All command input is handled by the \PRMT routine, which also issues the prompting message. \PRMT filters the input looking for a !! or : or TR starting in Column 1. The !! indicates that the operator wishes to abort the generator, and the : or TR indicates that a transfer is to be done. An EOF encountered in an answer file/LU results in the simulation of a TR command, which pops the input stack.

The parse routine \PARS is called with the input buffer address, and returns the parameters in the following format: Parameter 2 is the file name or LU, Parameter 3 the security code, Parameter 4 the CRN, Parameter 5 the file type, and Parameter 6 the file size.

PARS2,3,4,5,6,	Type: 0=null,	1=numeric,	2=ASCII
PARS2+1,3+1,4+1,5+1,6+1,	0	number	char 1 & 2
PARS2+2,3+2,4+2,5+2,6+2,	0	0	char 3 & 4
PARS2+3,3+3,4+3,5+3,6+3,	0	0	char 5 & 6

Asterisks (*) are not allowed within filenames, security codes, file size or cartridge labels. When an * is encountered, the beginning of a comment is assumed and \PARS returns.

\PRMT does some checks to determine whether or not to send the response just received to the list file. If the list file is to the LU of the operator console and if that is the current command input LU, the response is not sent; in all other cases, \LOUT is called (\LOUT does more checks for echoing).

TRCHK determines if the command input stack is to be pushed or popped. If the current command buffer contains a TR (or : or,) with no parameter, the stack is popped to the previous source of command input; otherwise the stack is pushed with the new element. Ten entries may be placed on the stack (GEN ERR 19 is issued on overflow or underflow) with each entry of the form:

Word 0 Entry type: 1=Type 0(LU), 2=file
Word 1 LU, else CH1 and CH2
Word 2 0, else CH3 & CH4
Word 3 0, else CH5 & CH6
Word 4 Security Code
Word 5 CRN
Word 6 0, else record count for next record to read

An eleventh entry to LU 1 is hard-coded at the bottom of the stack.

On a transfer, the current file is first closed. The routine PUSH then saves the next record number of that file in its stack entry, for repositioning when the file is later reopened. PUSH then picks up the file name/LU from the parse buffer and builds the new stack entry. If overflow results, no push is done; recovery is handled in TRCHK. POP, on the other hand, merely decrements the STACK pointers to the previous entry. On underflow no pop is done and TRCHK handles the recovery. Before returning to TRCHK, both PUSH and POP call the routine STATE which performs status checks on the new source of command input, setting CMDLU (0, else input LU) and IACOM (1 if an interactive LU, 0 if a file name or non-interactive LU). IACOM is used in determining the echo of input/output to the list file or console. STATE also checks the validity of an LU specified as the new command input source. If invalid, STATE does an error return, as does PUSH, and TRCHK issues a GEN ERR 20 then handles the recovery. This error will not occur on a POP as the command input source returned to would have already been checked at the original transfer. The tree structure for the generator command input and echo/list output routines is shown below.

```
             \RNAM                               CMER
               |                                   |
               v                                   v
             \READ        DISPR      \SPAC       \GNER      \CFIL
               |            |          |           |          |
               v            |          |           |          |
    CMDIN---->\PRMT         v          v           v          v
               |            +----------+-----------+----------+
               |                             |
               |                           \MESS
               |                             |
    \TERM      |                             |
               v                     +---------------+
    \TERM---->\LOUT                  | INTERACTIVE   |
               |                     |  INPUT LU     |
               |                     +---------------+
        +------+------+
        |             |
        |             |
        v             v
    +--------+    +--------+
    | LIST   |    | ECHO   |
    | FILE   |    |  LU    |   (if enabled)
    +--------+    +--------+
```

8.8 LIST FILE

The generation output may be sent to any list device. If a file name is specified without a file size, a 64-block file is created by the generator (extents are created as needed by FMP). Since a CREAT assumes an exclusive open, the file is then re-OPENED with the non-exclusive option. This permits examination of the list file concurrent with generation. For list output to an LU, a dummy DCB is created in the routine TYPO.

If the LU specified is a non-interactive device, an attempt is made to lock it. If unsuccessful, the generator issues the appropriate message (not in the form of an error) and reissues the LU lock call with the wait bit set. The generator is suspended until the resource becomes available.

If the specified LU is interactive, the flag IALST is set to 1. IALST and IACOM are then used in the list output routines \MESS and \LOUT to prevent duplicate output to the operator console. A line is always sent to the list file (using \LDCB) via \LOUT. If the list file is not an interactive LU (IALST=0) but the command input/answer file is (IACOM=1), the line is sent to the operator console (using \IDCB) as well. The status of the command input mode reflected in IACOM changes as TRansfers are encountered or errors are detected. Therefore, it is necessary to perform these checks every time a list output is done. See the Error processing section for the handling of list file errors.

8.9 ECHO PROMPT

The ECHO? prompt always requires a response, even where not applicable (as when the list file is the LU of the operator console, or the generator is to be directed interactively). If an echo is requested, checks are made in \LOUT to see if both IALST and IACOM are equal to 0, meaning that neither the listed output or command input is an interactive device. If IALST or IACOM indicate an interactive LU, one further check is made with either LSTLU or CMDLU against ERRLU to see if they represent the same interactive LU, in which case no ECHO is done. If more than one LU points to a particular interactive device, no checks are made to determine if they reference the same EQT. Since ECHO is dependent on the command input mode, it may change as TRansfers are done to and from the operator console or when error mode is enabled.

8.10 BOOTSTRAP FILE

At \BOT0, \BOT5, or \BOT9, the moving head bootstrap loader may be sent to an FMP file or to an LU; a 0 response by the operator implies that a boot is not desired. This value must be specifically checked for since the TYPO routine defaults to LU 1 if LU 0 is specified. This would result in absolute code being output to the user terminal. If an LU was specified for the boot file output, an EOF is written (e.g. trailer for a paper tape bootstrap). On an abortive termination, \TERM purges the boot file if one was created, otherwise \TERM simply closes the file.

8.11 SIZE RESTRICTIONS

The following limits for an RTE-6/VM system must be enforced due to the 32K (15 bit) logical address space of HP 1000 computers, base page ignored. Extended memory areas are not included. p(Area x) is defined as the smallest number of pages that completely contains Area x. In the following formulas, the Area x elements are:

- TAI - Table Area I
- TAII - Table Area II
- DP - Driver Partition*
- COM - Common
- SDA - System Driver Area

SYS - System
CFG - Configurator
MRL - Memory Resident Library
MRP - Memory Resident Program
DRP - Disc Resident Program
RDRP - Real-time Disc Resident Program
BDRP - Background Disc Resident Program
LBDR - Large Background Disc Resident Program

* Code partitions are mapped into this logical space.

System:

p(TAI)+p(DP)+p(COM)+p(SDA+TAII+SYS+CFG) = <31 pages

Memory Residents:

p(TAI)+p(DP)+p(COM)+p(SDA+TAII)+p(MRL+MRP) = <31 pages

where p(COM) and p(SDA+TAII) are optional

Real-time and Background Residents:

p(TAII)+p(DP)+p(COM)+p(SDA+TAII)+p(RDRP or BDRP) = <31 pages

Large Background Disc Residents:

p(TAI)+p(DP)+p(COM)+p(LBDR) = < 31 pages

where p(COM) is optional

8.12 PAGE ALIGNMENTS

The following areas are automatically aligned by the generator to start on a page boundary:

Base Page
Table Area I
Driver Partition
Common
Code Partitions
System Driver Area
Resident Library
Memory Resident Programs (first one only)
Disc Resident programs

8.13 BASE PAGE

Only one system and one and memory resident base page exist, but each disc resident program has its own copy of base page. The base page links used by a disc resident program are stored in the next disc sector following the program's code. The system base page is both logically and physically page 0 and is stored starting at track 0, Sector 2 of the system. The memory resident base page (MRBP) resides in physical memory after the last driver partition page, and the memory resident library (MRL) starts on the physcial page after that. Physically the MRBP links are stored on the next disc sector following the last memory resident program's code.

The System Communication Area (SCOM) and all of Table Area I, SSGA, Table Area II and drive links are resident in both system and user maps. SCOM resides in BP locations 1777-1645 and the upper BP links from 1644 downward. After the track 0 Sector 0 boot extension has been sent to the disc, the dummy base page (it resides in core overlaying the initialization code of the generator MAIN) is written for the sole purpose of reserving its disc space.

The system links (including the configurator) always start at location 100 and grow upward toward the SCOM. The partition driver links are not allocated until all PRDs have been relocated, so checks are done for overflow of these driver links into the system links. The system base page on disc is updated at the end of the system relocation for the trap cells and system links. Note that trap cells referencing programs are fixed as the programs are relocated. The BP driver links are updated on disc after all the PRD's have been relocated, and the SCOM is updated during the final generation cleanup.

Memory resident and disc resident program links start at BP location 2 and grow upward. A GEN ERR 16 is issued on each overflow into the upper link area. In MRBP the memory resident library links are allocated first, followed by those links necessary for all the memory resident programs.

The base page formats are shown below.

```
         System BP                         Memory Res. BP

1777 +-----------------+            1777 +-----------------+
     |                 |                 |                 |
     |      SCOM       |                 |      SCOM       |
1645 |-----------------|  |         1645 |-----------------|  |
     |                 |  |              |                 |  |
     |TABLE AREAS,SSGA,|  |              |TABLE AREAS,SSGA,|  |
     |     DRIVER      |  |              |     DRIVER      |  |
     |     LINKS       |  v              |     LINKS       |  v
     |-----------------|                 |-----------------|
     |                 |  ^              |     MEMORY      |  ^
     |                 |  |              |    RESIDENT     |  |
     |     SYSTEM      |  |              |    PROGRAM      |  |
     |     LINKS       |  |              |     LINKS       |  |
     |                 |  |              |-----------------|  ^
     |                 |  |              |    RESIDENT     |  |
 100 |-----------------|                 |  LIBRARY LINKS  |  |
     |   TRAP CELLS    |               2 |-----------------|
     +-----------------+                 +-----------------+

        Disc Res. BP

1777 +-----------------+
     |                 |
     |      SCOM       |
1645 |-----------------|
     |TABLE AREAS,SSGA,|  |
     |     DRIVER      |  |
     |     LINKS       |  v
     |-----------------|
     |                 |  ^
     |                 |  |
     |      USER       |  |
     |                 |  |
     |      LINKS      |  |
     |                 |  |
     |                 |  |
   2 |-----------------|
     +-----------------+
```

8.14 SYSTEM COMMUNICATION AREA (SCOM)

The SCOM is built at the end of generation during final clean-up. An area of 133 octal words below the label USRTR is initialized to 0 and overlaid as SCOM is built, transferred to the dummy base page, and then sent to the disc using /ABDO. The base page locations are set by the generator variables listed in the right-hand column of the following table. Where the variable is listed as "0", the location is zero-filled; where the variable is shown as a calculation, the result of that calculation is placed in the location.

Loc.	Label	Description	Variable
System Table Definition			
1645	XIDEX	Address of current program ID Ext.	0
1646	XMATA	Address of current program MAT ent.	0
1647	XI	Address of index register save area	0
1650	EQTA	First word address of EQT	AEQT
1651	EQT#	# of EQT entries	CEQT
1652	DRT	First word address of DRT	ASQT
1653	LUMAX	# of LUs in DRT	CSQT
1654	INTBA	First word address of Int. table	AINT
1655	INTLG	# of interrupt table entries	CINT
1656	TAT	First word address of TAT	ADICT
1657	KEYWD	First word address of keywd block	KEYAD
I/O Module/Driver Communication			
1660	EQT1	Address of first 11 words of	LWSYS+1
1661	EQT2	current EQT entry (last four	SAM#1
1662	EQT3	(last 4 words begin at loc. 1771)	LWSYS+1+SAM#1
1663	EQT4	¦ ¦	SAM#2
1664	EQT5	¦ ¦	LWTAI+1
1665	EQT6	¦ ¦	DPADD-(LWTAI+1)
1666	EQT7	¦ ¦	0
1667	EQT8	¦ ¦	0
1670	EQT9	¦ ¦	0
1671	EQT10	¦ ¦	0
1672	EQT11	v v	0
1673	CHAN	Current DMA (DCPC) channel number	0
1674	TBG	I/O address of time-base card	TBCHN
1675	SYSTY	EQT entry address of system TTY	SYSTY

System Request Processor/Exec Communication			
1676	RQCNT	# of request parameters - 1	0
1677	RQRTN	Return point address	0
1700	RQP1	Addresses of request parameters	0
1701	RQP2	(Set for a maximum of nine)	0
1702	RQP3		0
1703	RQP4		0
1704	RQP5		0
1705	RQP6		0
1706	RQP7		0
1707	RQP8		0
1710	RQP9	v v	0
System Lists Addresses			
1711	SKEDD	Address of system schedule list	SCH4
1712		Reserved	0
1713	SUSP2	Address of wait suspend list	0
1714	SUSP3	Address of available memory list	0
1715	SUSP4	Address of disc allocation list	0
1716	SUSP5	Address of operator suspend list	0
Program ID Segment Definition			
1717	XEQT	ID segment address of current prog.	0
1720	XLINK	ID seg. linkage	0
1721	XTEMP	ID seg. temporary	0
1722	XTEMP	ID seg. temporary	0
1723	XTEMP	ID seg. temporary	0
1724	XTEMP	ID seg. temporary	0
1725	XTEMP	ID seg. temporary	0
1726	XPRIO	ID seg. priority word	0
1727	XPENT	ID seg. primary entry point	0
1730	XSUSP	ID seg. point of suspension	0
1731	XA	ID seg. A-Register at suspension	0
1732	XB	ID seg. B-Register at suspension	0
1733	XEO	ID seg. E & oflow reg. at suspen.	0
System Module Communication Flags			
1734	OPATN	Operator/keyboard attention flag	0
1735	OPFLG	Operator communication flag	0
1736	SWAP	RT disc resident swapping flag	SWAPF
1737	DUMMY	I/O address of dummy int. card (PI)	PIOC
1740	IDSDA	Disc address of first ID segment	DSKSY
1741	IDSDP	Position in sector of first ID seg.	IDSP

Memory Allocation Bases Definition			
1742	BPA1	FWA user base page link area	2
1743	BPA2	LWA user base page link area	LOLNK-1
1744	BPA3	FWA user base page link	2
1745	LBORG	FWA of resident library area	LBCAD
1746	RTORG	FWA of real-time common	RTCAD
1747	RTCOM	Length of real-time common	COMRT
1750*	RTDRA	FWA of real-time partition	MEM6
1751*	AVMEM	LWA+1 of real time partition	SYMAD
1752	BGORG	FWA of background common	BGCAD
1753	BGCOM	Length of background common	COMBG
1754*	BGDRA	FWA of background partition	MEM12
Utility Parameters			
1755	TATLG	Negative length of TAT	-(DSIZE+DAUX)
1756	TATSD	# of system disc tracks	DSIZE
1757	SECT2	# of sectors/track, LU 2 (system)	SDS#
1760	SECT3	# of sectors/track, LU 3 (aux.)	ADS#
1761	DSCLB	Disc addr. of entry point library	DSKLB
1762	DSCLN	# of user avail. library ent. pts.	LBCNT
1763	DSCUT	Disc add., reloc. disc res. libr.	DSKUT
1764	SYSLN	# of system library entry points	SYCNT
1765	LGOTK	Load and go: LU, start trk, # trks	0
1766	LGOC	Current load and go track/sec addr.	0
1767	SFCUN	Log source: LU, disc address	0
1770	MPTFL	Memory protect ON/OFF flag (0/1)	1
1771	EQT12	Addr. of last 4 words, current EQT	0
1772	EQT13	¦ ¦	0
1773	EQT14	¦ ¦	0
1774	EQT15	v v	0
1775*	FENCE	Memory potect fence address	0
1776	$BMSP*	Set to 0 to alert VM microcode	0
1777	BGLWA	Last word memory address, BG par.	LWASM

* Contents of location set dynamically by dispatcher

8.15 COMMON

The RT and BG commons along with the Subsystem Global Area (SSGA, type 30 module) occupy a single area collectively known as COMMON. Since any program using any of the three areas has map entries for the others, only the memory protect fence table can provide any protection.

The order of the three areas was chosen such that a hierarchical protection is preserved:

```
    +------------------+\
    |                  | \
    | BG COMMON        |  \
    |                  |   \
  C +------------------+    \
    | RT COMMON        |     \ PAGE BOUNDARY
  B +------------------+     /
    |                  |    /
    | SSGA             |   /
    |                  |  /
  A +------------------+ /
```

The memory protect fence will be placed at A, B, or C if a program is using COMMON.

When the IDENTs are scanned for ID segment allocation at the end of the PIP, the common sizes of each program stored in \ID4 bits (14-0) are used to set the maximum RT and BG common sizes, COMRT and COMBG respectively.

Beginning on a page boundary after the driver partition, all SSGA modules are loaded first, followed by the allocation of the RT and BG common area. The RT common size in decimal words is displayed, the user is given the option of increasing it (nnnnn parameter, below), and the starting octal address is displayed. You must respond to the CHANGE RT COMMON? prompt; entering "0" means no change. A GEN ERR 14 is issued on an invalid response (no response is an invalid response). The sequence is:

```
RT COMMON  XXXXX          *size in decimal
CHANGE RT COMMON?
nnnnn                     *change size, or 0
RT COM ADD XXXXX          *octal address
```

RTCAD is set to the RT common starting address from PPREL. COMRT, the number of words, may be updated; BGBND is set to the starting address of BG Common, PPREL+COMRT. You must respond to the CHANGE BG COMMON? prompt; entering "0" means no change. A GEN ERR 14 is issued on an invalid response (no response is an invalid response). Before COMBG is displayed, it is updated to include that area from BGBND to the end of the page (because the SDA is automatically aligned on the next page boundary after BG common). The following sequence occurs for BG common determination:

```
BG COMMON   XXXXX          *size in decimal
CHANGE BG COMMON?
nn                         *change size, or 0
BG COM ADD  XXXXX          *octal address
BG COMMON   XXXXX          *size in decimal
```

BG Common size is increased in page multiples so COMBG has nn*1024 added to it. The new BG Common size is then displayed

8.16 CONFIGURATOR PROGRAM

The configurator program is a special type 16 system module that has access to all the system entry points. It is loaded immediately after the system in what will later be System Available Memory (SAM #1). Its base page links are included with those of the system. The last word must not be greater than 77577B (77377B for ICD systems) or a GEN ERR 18 will result and the generation will be aborted. The memory above this address must be reserved for the boot extension that loads the system. The last word of both the system code and configurator code must be saved (in LWSYS and LWSLB respectively) to compute the size of SAM #1 at the beginning of the Partition Definition Phase. SAM #1 will include that specific memory area covered by the configurator plus any remaining area left on the last page occupied by the configurator.

The configurator references Table Area II entry point $SBTB, which is the first word of the following six-word table:

```
        +------------------------------------+
$SBTB   |   DISC ADDR OF DRIVER PARTITIONS   |
        +------------------------------------+
        | # OF PAGES, ALL DRIVER PARTITIONS  |
        +------------------------------------+
        |  DISC ADDR OF MEMORY RES BAS PAGE  |
        +------------------------------------+
        | # OF PAGES, MEMORY RES BASE PAGE   |
        +------------------------------------+
        | DISC ADDR OF MEM RES LIB AND PROG  |
        +------------------------------------+
        |# OF PAGES, ALL MEM RES LIB AND PROG|
        +------------------------------------+
```

The values are placed in $SBTB when, at the end of the partition definition phase, the generator sets the values of all the Table Area II entry points specified in $$TB2.

8.17 CODE PARTITIONS

Certain non-time-critical portions of the RTE-6/VM operating system (Type 0 modules) reside in code partitions. These code partitions are similar to driver partitions in that code residing in one of the partitions is present in the system map when it is executed. For this reason, code partitions are treated much the same as driver partitions.

During the Program Input Phase, when the generator encounters a Type 0 module whose name begins with "OSx", it marks this module to be loaded in the code partition defined by the "x" digit in the module name. Segment 3 then calls the routine \OSLD to load the code partitions after the configurator has been loaded, but before system base page links are dumped to base page. IDENT is scanned once to find the OS modules for mapping into the code partitions. The partitions are loaded in ascending order.

To add a new OS code partition module, it is necessary only to specify the module name as OSn[xx], where n is the partition designator. The optional characters xx can be used as a file identifier and are ignored by the generator. Note that the modules must be specified in ascending order since IDENT is scanned only once and the partitions must be loaded in ascending order. (A GEN ERR 63 results if modules are not specified in order.

8.18 BOOTSTRAP AND EXTENSION

The generator builds both the track 0 sector 0 boot extension and moving head bootstrap loaders for either the 7900, MAC, ICD or CS80 discs. Generator Segment 1 builds the 7900 bootstrap loader, Segment 7 builds it for the ICD and MAC discs and Segment 9 builds the bootstrap for the CS80 discs.

For non-CS80 discs, the generator stores the system subchannel disc specifications in the bootstrap loader (i.e., first track, # of tracks, starting head, # surfaces, etc.). For the moving head bootstrap loader, the generator configures the disc I/O instructions to the select code of the system disc. The high address of the configurator is stored in the track 0 sector 0 boot extension in HIGH so the first chunk of memory can be read in from the disc starting at track 0 sector 2 (track 0 sector 4 for ICD discs).

For CS80 discs, the information placed in the bootstrap is limited to the HP-IB address, unit, volume, and starting block number of the system image. Much of the information required for non-CS80 discs is not required for CS80 discs since:

1. CS80 discs operate in block mode, thus all track, sector, and surface information is reduced to one number: the starting block number.

2. The CS80 boot reads a fixed amount from the disc rather than exactly the number of words through the high end of the configurator. Although this requires more time since some information is read from the disc twice, it greatly simplifies the boot extension.

The configuration of I/O instructions is performed when the boot is run. This process uses the system disc select code found in the switch register, rather than a data item placed in the boot by the generator.

The generator also sets the following values in the boot extension:

BOOT EXTENSION VALUES	BOOTSTRAP VALUES
7900 Discs:	
TBASE	UN#IT
U#NIT	H#AD
B#MSK	S#EKC
SKCMD	R#DCM
R#CMD	DSKDR
HIGH	T#ACO
7905/06/20/25 (MAC) Discs:	
TBASE	PT#TR
BHD#	PT#T2
#HDS	H#AD
WAK	PT#H2
SKCMD	WA#KE
AD#RC	PT#SK
R#CMD	PT#AD
S#TAC	R#DCM
HIGH	P#EN
7906H/20H/25H (ICD) Discs:	
HTBAS	!CYLL
BHED#	!CYLH
#HDS	!HEAD
AD1	#HEDS
AD2	!UNIT
AD3	!AD1
AD4	!AD2
AD5	!AD3
HHIGH	!AD4
	!AD5

7908/11/12/14/33 (CS80) Discs:

ADDRS UNIT VOLUM ADDR2	Boot extension and bootstrap are the same piece of code.

8.19 TABLE AREAS I AND II

Table Area I contains (in the following order):

Track Map Table ($TB31, $TB32, or $TA32)
EQTs and extensions
DVMAP Table
DRT
INT
All Type 15 modules

Table Area I exists in all maps. The user-available entry points to system code re loaded into Table Area I from the Type 15 module $$TB1. Note that all user-defined track map tables must be Type 15 modules in order to exist in all maps. The space left on the last page occupied by Table Area I is allocated to SAM (SAM #0).

Table Area II contains (in the following order):

$CLAS Table
$LUSW Table
$RNTB Table
$LUAV Table
$IDEX Table
ID extensions
Keyword Table
ID segments
$MATA Table
$MRMP Tap
$MPFT Table
Track Allocation Table
$$TB2 entry points
All Type 13 modules

Type 13 module $$TB2 contains the entry points to system tables, most of whose values are set when the $MATA, $MRMP, and $MPFT tables are built during the Partition Definition Phase. Table Area II is included only in the system, therefore access to any Table Area II entry points must be via cross-map loads. This affects memory resident (optional), and Type 2 RT and Type 3 BG program address spaces, and Type 4 BG programs.

All external references from the Table Areas are resolved through fixups once the system and all drivers are relocated. The Table Areas can reference each other, the system, and types 6, 7 and 8 utility modules. Their links are included with those of the system. Table Area I starts on a page boundary, following the base page. Table Area II immediately follows the System Driver Area in memory, so both are mapped in when either is referenced.

8.20 EQT, DRT, AND INT TABLE SIZES

The EQT table holds a maximum of 255 entries, and the DRT holds a maximum of 254 entries. Since both the EQT and DRT entries are sequentially prompted, the generator issues a GEN ERR 35 for all entries past the 255th or 254th until a /E is encountered.

The size of the DRT is always 2.5 times the number of LUs defined (CSQT), with the second zero-filled chunk of size CSQT following the first. The first CSQT words of the DRT are set as follows by the generator:

```
15          11                5             0
+-------------+---------------+----------------+
| SUBCHANEL # |  RESERVED     |  EQT ENTRY #   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
+--+--------------------------------------------+
| F|     DOWNED I/O REQUEST LIST POINTER        |
+--+--------------------------------------------+
  F = 0, Device Up
    = 1, Device Down
```

The INT contains entries for each channel from 6B through 77B, even though the user may not have defined up to the maximum. The entire channel spectrum must be present for possible I/O channel reconfiguration at slow boot time. This also implies that base page location 100B will always be the first SYSTEM base page link. All I/O locations from 2B through 77B are initialized to the absolute code for JSB $CIC,I, except that location 4 is initialized to HLT 04.

The INT records are processed as follows:

1. N1,EQT,N2. The address of the EQT entry specified by N2 is set into the INT entry designated by N1. The INT location contains JSB $CIC,I.

2. N1,PRG,PNAME. The twos complement of the ID Segment ADDR for PNAME is set into the INT entry N1. The interrupt location contains JSB $CIC,I.

3. N1,ENT,ENTRY. The INT entry specified by N1 is set = 0 and the interrupt location N1 is set to contain JSB X,I, where X is the BP link

address containing the address of ENTRY.

4. N1,ABS,XXXXXX. The INT entry specified by N1 is set = 0 and the interrupt location N1 is set to contain XXXXXX.

All locations in the Interrupt Table from 6B to 77B which are not specified by INT records are set = 0. For N1 = 4 the only legal entries are Ttpes 3 and 4. All INT records must be entered in increasing N order, with the exception of 4.

For ENT type entries, the entry point referenced must be contained in a Type 0 module. If that Type 0 module is a driver (IDENT Word 8 bit 15 is set) then that driver must be in the System Driver Area (IDENT word 8 bit 14 is set).

8.21 DRIVERS AND DVMAP

Drivers will be relocated to reside in either a driver partition or the System Driver Area (SDA). The I/O tables (EQT, DVMAP, DRT, and INT) are stored in Table Area I, and are therefore built before any drivers have been relocated. Fixups are then resolved for EQT words 3 and 4 once a driver initiation and completion sections are relocated. The two FIXUP table entries will automatically be allocated when the EQT is built. The fixup entries are built as follows:

```
 Word 1:  Memory location (in EQT) where address of I.XX or C.XX
          is to be stored
 Word 2:  Instruction code = 0, DBL record type = 5
 Word 3:  Offset = 0
word 4:  LST index of I.XX or C.XX
```

Setting the DBL record type in Word 2 equal to 5 simulates an external reference with offset. With the instruction code equal to 0, this indicates a DEF to an external with offset (of 0) at fixup time, therefore making it direct.

All drivers (identified as Type 0 modules beginning with DV) will be sent to driver partitions unless specified as an SDA type (S in the EQT definition). Those driver moduless without an EQT and possibly not beginning with DV will be relocated with the system. If an SDA driver is to do its own mapping, then an M is specified, either in addition to or in place of the S.

When an EQT is defined, the IDENT table entry for the named driver is retrieved (a GEN ERR 25 is issued if not found). After the EQT is built, Word 8, bit 15 of the driver IDENT is set to indicate that a valid EQT existed, bit 14 is set if SDA was declared, and bit 13 is set if the SDA driver is to do its own mapping. If an M is specified without an S, then an S is assumed and both bits 14 and 13 are set. If bit 15 indicates that a driver had already been specified in a previous EQT, the new type must match that of the old. That is, bits 14 and 13 of the current entry must match the values to be set by the new entry, otherwise a GEN ERR 23 is issued and the EQT must be redefined.

The system disc driver cannot reside in SDA. When an EQT select code matches the user CONTROLLER SELECT CODE? response, the system disc EQT is assumed and a check is made to ensure that SDA was not declared for this driver. If SDA has been declared, a GEN ERR 23 is issued.

The first half of driver map table DVMAP is dynamically built in a buffer as the EQTs are defined. DVMAP consists of two consecutive chunks of size CEQT (the number of EQTs). After all EQTs and EQT extensions have been built, space is reserved for the DVMAP and it is sent to the disc. Table Area II entry point $DVMP is set (later) to its address. The first CEQT chunk has values stored in it by the generator, while the second CEQT chunk is zero-filled for the user by RTIOC. A 64-word buffer (the maximum number of EQTs) is used for building the first part of DVMAP. The dummy entries are built as follows, with Word 0 corresponding to EQT1,...word CEQT-1 corresponding to EQT CEQT:

```
 15                                        0
+--+---------------------------------------+  Partition-
| 1|        IDENT INDEX OF DRIVER          |  resident
+--+---------------------------------------+  driver (PRD)
```

or:

```
 15                                        0
+--+------------------------------------+--+
| 1|                 0                  | 1|  SDA Driver -
+--+------------------------------------+--+  Does own Mapping
```

The PRD entries in DVMAP are updated on disc when those drivers are relocated; the SDA entries are left as defined. The final PRD form is:

```
 15            9                   0
+-------------+--------------------------+
|      0      |  PHYSICAL STARTING PAGE  |
+-------------+--------------------------+
               of driver partition
```

When a PRD is relocated into a partition, all EQT entries in DVMAP must be scanned for an IDENT index matching that of the driver. All matching DVMAP entries are then replaced with the driver partition starting page.

8.22 SYSTEM DRIVER AREA (SDA)

All drivers going into the System Driver Area are relocated following the construction of Common. Since Common always ends on a page boundary, the SDA always begins on one.

8.23 DRIVER PARTITIONS

The defaulted driver partition size is two pages, which is sufficient hold any HP partition-resident driver. As many drivers are relocated in a DP as will fit, so increasing the DP size will allow more drivers to fit into a particular partition - possibly saving physical pages if leftover page space can be used. For partition-resident drivers greater than two pages, the DP size must be overridden to accommodate it. Otherwise, if the driver overflows a DP, the generation will be aborted at relocation time with a GEN ERR 59.

The current DP size is displayed in decimal number of words, and the user is given the option of increasing it:

```
DRIVR PART 00002 PAGES
CHANGE DRIVR PART?
```

RQCNT/ A zero (0) response implies no change, otherwise the new size must be >= DPLN and less than 17. If an invalid response is entered, a GEN ERR 01 is issued and the prompt is redisplayed. The new value of DPLN is used to set Table Area II entry point $DLTH. The last word occupied by Table Area I is rounded up to the next page boundary and stored in DPADD (the skipped memory is later allocated to SAM). DPADD, converted to a logical page number, is used to set Table Area II entry point $DVPT (starting logical page of driver partition). Memory skipped in the page alignment is released to the disc by updating relocation address PPREL. When \ABDO is next called, that disc space will be zero-filled since PPREL will be greater than the address of the highest previously generated word in the system map (MXABC,I of the \ABDO specification table for the system).

After the relocation of Table Area I, the first driver partition is relocated. The system disc driver must be relocated into this partition for use by the configurator program; this driver is determined by using DRT2 (the system disc EQT number) to offset into the temporary DVMAP table in order to pick up its IDENT index.

Once a driver is relocated, a check is made to see if the logical address space used for a driver partition has been overflowed. If not, the IDENT table is scanned for a driver that will fit into the remaining space of the

DP. The scan always starts at the beginning of the IDENT table and stops when the size specified in \ID8 of a driver entry indicates that it will fit. In addition, the routine CPL? is called to check for the memory requirements when current page links are in effect. If the above two checks pass, the driver is relocated; if not, the scan continues through the IDENT table.

Note however, that a driver may still overflow a partition. This can happen when referenced subroutines are appended to the driver during relocation. Upon overflow, the violating driver is 'backed-up' over, a warning is issued, and the IDENT table scan is continued. The DVMAP entries are not updated for the overflowed driver. When fixups to driver entry points are resolved during relocation, the entries are not deleted from the FIXUP table. Thus in the case where a driver is relocated more than once, the references are simply re-fixed to the final value. The violating driver will be relocated into a subsequent partition.

When no more drivers can fit into a partition, the remainder is zero-filled. For Driver Partition one, zero-fill is done to the last word of a DP, but the zero-fill is done only to the last word of the last page used in other DPs. This allows feature pages to be saved where one or more complete pages of a DP are unused.

For each new DP, the scan is then started at the beginning of the IDENT table for the next unrelocated partition-resident driver. If none exist, the driver partitions are complete and. the fixup table is cleared before the memory resident library is built.

For Driver Partitions two and up, the \ABDO specification map is changed from that of the system to that of driver partitions. This is done because these driver partitions reside logically in the system area, but physically on the disc in pages above the system area. From then on, when each new DP is started, the DP map disc address ABDSK,I is updated but ABCOR,I and MXABC,I are reset to DPADD.

After a driver is loaded, the physical starting page of that driver partition is stored in all the DVMAP entries referencing that driver. The fixup entries pertaining to EQT Words 3 and 4 are also resolved. Note that the \ABDO map must be changed to that of the system in order to perform these Table Area I updates. When a new driver partition is started its starting physical page is set: PAGE#<---PAGE# + number of pages required by previous driver partition (DPLN). PAGE# is initially set for Driver Partition two to $ENDS, the physical page immediately following SAM#1. (See the physical memory map in Appendix I.) The physical page for DP#1 is the same as its logical page, $DVPT.

Partition driver links start at BP location 1644 and grow downward. Since the system usually has already been relocated, checks must be make during PRD relocation for overflow of links into those occupied by the system. If this overflow is found, a GEN ERR 16 results and the generation is aborted.

Note that a user-entered disc Track Map Table (e.g. $TB31 or $TB32) may be typed as a subroutine and appended to the driver in its driver partition.

8.24 ID SEGMENTS AND EXTENSIONS

During the construction of Table Area II, space is reserved for long ID segments (33 words), memory resident ID segments (33 words), short ID segments (9 words), and ID extensions (3 words). Long ID segments are allocated to real-time and background disc resident programs; memory resident ID segments to memory resident programs; short ID segments for each program segment; and ID extensions for each long ID segment of an EMA program. The minimum number of each type necessary is obtained by scanning the IDENTS keying off the program type and EMA flag in \ID6. The user is given the opportunity to have blank ID segments and extensions allocated through the prompts:

```
# OF BLANK LONG ID SEGMENTS?
# OF BLANK SHORT ID SEGMENTS?
# OF BLANK ID EXTENSIONS?
```

A GEN ERR 60 is issued if the total number of ID segments is >254. If more than 254 ID segments are required before any blanks are requested, the generator aborts after issuing GEN ERR 60. A GEN ERR 01 is issued if the number of ID extensions exceeds the number of long ID segments.

The keyword table and ID extension table ($IDEX) have one word allocated for each ID segment and ID extension, respectively, plus one stop word equal to zero. The keyword table entries are set to the ID segment addresses as the ID segments are being built, or during final cleanup for the blank ID segments generated. The ID extension table and the ID extensions precede the keyword table and ID segments. When built, the ID extension table ($IDEX) is initialized to the addresses of the ID extension entries (three words each). Referto the EMA section for a description of the values stored in the ID extension entry.

ID segment entries are built as follows:

WORD	ENTRY
0	0
1-5	0
6	PRIORITY FROM NAM RECORD
7*	PRIMARY ENTRY POINT
8	0
9	0
10	ADDRESS OF ID SEGMENT WORD 1
11	0
12*	ID1 NAME1, NAME2
13*	ID2 NAME3, NAME4
14*	ID3 (15-8) NAME5; ID6 (2-0) TYPE -- ID
15	OPTIONALLY SETS BIT 0 IF SCHEDULED PROGRAM -- SEGMENT
16	0
17	RESOLUTION CODE AND EXECUTION MULTIPLE FROM NAM RECORD
18	TIME WORD FROM NAM RECORD
19	TIME WORD FROM NAM RECORD
20	0
21	**
22*	LOW MAIN ADDRESS FROM PPREL
23*	HIGH MAIN ADDRESS FROM TPREL
24*	LOW BP ADDRESS FROM PBREL (NON-MLS)
25*	HIGH BP ADDRESS FROM TBREL
26	MAIN DISC ADDRESS FROM DSKMN
27*	0
28	ID EXT# & EMA SIZE
29	HIGH MAIN OF LARGEST SEGMENT = TPMAX, ELSE 0
30	0 (SESSION MONITOR WORD 1)
31	0 (SESSION MONITOR WORD 2)
32	0 (SESSION MONITOR WORD 3)
33	0
34	0
35	SECTOR ADDRESS, LU OF PROGRAM

* Short ID segments

** Bit 15,RP, may be set during Partition Definition Phase

Bits 14-10,# pages required, set at end of main program load by IDFIX; for EMA programs includes MSEG size; may changed for non-EMA program during Partition Definition Phase

Bits 9-7,MPFI, set at end of main program load by IDFIX

Bits 5-0,Partition #, may be set if assigned during PDP

8.25 EXTENDED MEMORY AREAS (EMA)

When an EMA declaration (relocatable record type 6) is encountered in a module during the Program Input Phase, the EMA bit (15) is set in Word 6 of the current module's IDENT entry. The EMA size is retrieved from Word 2 of the relocatable record (bits 9-0) and is stored in IDENT Word 5 (bits 13-4). The MSEG size is retrieved from Word 7 of the record (bits 4-0) and is stored in IDENT Word 6 (bits 14-10). A zero value for either of these two sizes indicates that the default values are to be determined. The default MSEG size is determined at load time by the generator, and the default EMA size at system dispatch time.

If more than one EMA declaration occurs in a module (\ID6 bit 15 has already been set), a GEN ERR 41 results and the program's IDENT entry and all its LST entries are deleted (table pointers are backed up).

The EMA program type is picked up from IDENT Word 6 (bits 6-0) to determine if it is a legal EMA type. EMA programs must be disc resident (type 2,10,18,26), and either real-time (type 3,11,19,27) or background (type 4,12,20,28). The EMA type declaration must be in the main program; declarations in subroutines or segments are not allowed.

If the type check fails, the program type in IDENT Word 6 is set to 8, and a warning is issued (GEN ERR 40). A main program of Type 8 will not be relocated; recovery is possible by changing the program type to a valid EMA type during the Parameter Phase. The EMA type check is also performed in the Parameter Phase when the type of an EMA program is changed (bit 15 of IDENT Word 6 indicates EMA). If the new type is invalid, the warning GEN ERR 40 is issued, and the program type is not changed. ID segments with extensions will be allocated only for those EMA programs of the correct type (plus any user indicated spares).

The EMA label is stored in Loader Symbol Table (LST) Word 4 as a type 6 entry. The IDENT index of the defining EMA module is stored in Word 5. The symbol type is used to prevent incorrect references to EMA symbols. That is, modules of type 0,1,(9,17,25), 6,13,14,15,16,17 and 30 referencing an LST type 6 symbol will cause a GEN ERR 42 when the EXT is encountered at load time. A NOP replaces the referencing instructions. The same thing happens when a non-EMA, but valid program type references an EMA symbol. By checking the IDENT index stored in Word 5, a GEN ERR 42 will also result if an EMA program references the EMA symbol belonging to another EMA program. The referencing instructions will also NOP. Note that the EMA label declarations are also subject to the duplicate entry points restriction, with the second definition overriding the first. Being a special type of symbol, the value of an EMA label cannot be changed during the CHANGE ENTs Phase.

During the relocation of an EMA program, its EMA label is always treated as a forward reference to an external, even in the module declaring the EMA. The declaration of the EMA label forces an immediate base page link to be allocated. All references to the EMA label will then use this link. The link is allocated before the first reference is encountered to avoid the situation where an EMA program segment contained the first reference, in which case a base page link and fixup entry would be allocated in the segment's BP area; but the segment would be long gone before the fixup could be resolved. Because the logical MSEG address is the relocated address of the EMA label, that address cannot be determined until the main, its segments, and all appended subroutines have been relocated.

The highest relocation address for a program (stored in TPREL or TPMAX) is rounded up to the next page address to produce the Effective High Main (EHM). EHM becomes the logical MSEG address which resolves the EMA references. Converting EHM to a logical page address (EHMP) and subtracting it from 31 gives the maximum MSEG size. If the maximum MSEG size is <=0, a GEN ERR 43 results; no ID segment is built for the program and its disc storage space is reused. If an MSEG size was declared in the program (\ID6 bits 14-10 are non-zero), it is checked against the maximum MSEG size. If the declared MSEG size is greater, a GEN ERR 43 also results. If defaulted, the MSEG size becomes the maximum MSEG size.

For an EMA main program ID segment, the number of pages required for execution stored in Word 21 bits 14-10 is the number of pages occupied by the entire program plus the MSEG size. The number of pages for an EMA program cannot be overridden during the Partition Definition Phase. Word 28 is set up during the relocation of an EMA program:

```
Bits 15-10 - Index of the next available ID extension entry
             starting at zero
      9-0  - Declared EMA size retrieved from ID5 (13-4)
             or 1 for  default EMA declaration
```

After getting the address of the ID extension entry by indexing into the $IDEX table, the entry is set up as follows:

```
Word 0:  Bits  4-0  - MSEG size (declared or defaulted)

Word 1:  Bits 15-11 - Starting logical page of MSEG (from EHMP)
                10 - Set if EMA size defaulted

Word 2:  Set to 0
```

The ID extension index is then bumped to that of the next free entry.

8.26 PARTITION DEFINITION PHASE

The Partition Definition Phase begins with the display of the page requirements of all real-time and background disc resident programs. Type 4 BG programs will have a an asterisk (*) appended to the display line, and EMA programs will have the letter E appended to the display line. IDFIX stores a program page requirements in Word 8, bits 15-8, of the program IDENT entry at the same time it built Word 21 of the ID segment, so the IDENT table is scanned based on the program type in bits 2-0 of \ID6. One page is added in the displayed value, however, to include base page. For EMA programs, this value is calculated as follows:

(MSEG size - prog size) + (EMA size + 1 [base page])

where

MSEG size = value of \ID6 bits 14-10
prog size = value of \ID8 bits 15-8
EMA size = value of \ID5 bits 13-4 or
= 1 if EMA defaulted

The maximum program address space is displayed in three categories:

W/O COM xx PAGES
W/ COM xx PAGES
W/ TA2 xx PAGES

The without-common size MAXPG is the number of logical pages left after the driver partition. The with-common size is the number of pages left after the entire common chunk. And the W/ TA2 is the number of pages left after Table Area II. MAXPG is used during partition definition to determine if a partition qualifies as a mother partition (the number of defined pages is >MAXPG).

8.26.1 System Available Memory (SAM)

System available memory can exist in three chunks, with the size of the first two chunks determined by the generator and the size of the third chunk specified by the user (may be zero). The first chunk (referred to as SAM#0) lies in the area between the end of Table Area I and the start of the driver partition. The second chunk (SAM#1) covers the total area occupied by the configurator plus any remaining area left on the last logical page the configurator occupies. The word size calculation is performed as follows:

SAM#1 = CONWD - SYSWD

where

CONWD = last word of configurator rounded up to next page

SYSWD = last word occupied by SYSTEM code

$MPSA bits 15-10 are set to the number of pages occupied by SAM#1 (including the page shared with the system, if that's the case), and bits 9-0 are set to the starting page of SAM#1. SCOM word EQT1 is set to the logical starting address of SAM#1, the last word occupied by the SYSTEM, plus 1. SCOM word EQT2 is set to the number of words in SAM#1. The Table Area II entry point $ENDS is equivalent to $MPSA bits 9-0 + $MPSA bits 15-10; this amounts to the physical page number immediately following SAM#1. SCOM word EQT5 is set to the starting address of SAM#0, and EQT6 is set to its size.

The logical combination of SAM#1 and SAM#2 must appear in the first 32K of logical address space, where SAM#2 is relocated to appear logically contiguous with SAM#1. Physically, the pages containing the driver partitions and memory resident area (base page, library and programs) will separate the two chunks. If $MPSA bits 9-0 + $MPSA bits 15-10 is equal to 32, then SAM#1 occupies the rest of the logical address space and SAM#2 will not exist (descriptors $MPS2 and EQT4 will be zero). Since the last word of SAM cannot be 77777 (see the $ALC routine in the section SYSTEM for an explanation), the word count in EQT2 must be two less in order to force 77775 as the last word.

The present size of SAM (i.e., of SAM#0 + SAM#1) is displayed in decimal number of words. The user then is given a range for the first physical page for user partitions and is asked to enter a page number in that range. If a value greater than the default value is entered, the user is allocating the skipped pages to SAM (thereby defining SAM#2). If the new first partition/page equals the default value, SAM#2 does not exist and its descriptors are set to zero. The size of SAM#2 is calculated as:

(new first page - old first page) * 1024.

$MPS2 bits 15-10 are set to the number of pages occupied by SAM#2 (new first page - old first page); and $MPS2 bits 9-0 are set to the physical starting page of SAM#2 (the old first page). Since SAM must still reside in the first 32K logical address space, $MPSA bits 9-0 + $MPSA bits 15-10 + IMPS2 bits 15-10 must be <32. If not, a GEN ERR 44 will be issued and the user will be re-prompted. If it is equal to 32, the size of SAM#2 stored in EQT4 is decremented by 2, making the last word of SAM equal too 77775. EQT3 is set to the logical starting address of SAM#2 by setting it to (EQT1 + EQT2) and SAM#2 is logically relocated to immediately follow SAM#1. The total size of SAM (EQT2 + EQT4 + EQT6), is displayed in decimal number of words before going on to partition definition.

8.26.2 Memory Allocation Definition

The memory allocation table (MAT) and the entry points describing it are located in Table Area II. When the maximum number of partitions ($MNP) is set by the user, the space for that number of MAT entries is reserved with $MATA pointing to the first entry.

The number of remaining physical pages (DPARE, the memory size stored in NUMPG minus the first partition page PAGE#) is next displayed to the user for partition definitions. The link word (Word 0) of each MAT entry is initialized to -1 to indicate an undefined partition, whereas Words 1-6 are set to 0. Note that since the MAT is already on the disc, it must be referenced through its absolute memory address, updating the code on the disk via \ABDO. The MAT entry format is shown below.

```
          15  14  13         9                 2      0
         +-----------------------------------------------+
WORD 0   |            FREE LIST LINK WORD                |
         +-----------------------------------------------+
     1   |           PRIORITY OF RESIDENT                |
         +-----------------------------------------------+
     2   |            ID SEGMENT ADDRESS                 |
         +---+---+---+---------+-----------------------+
     3   | M |   | D |         |  STARTING PAGE          |
         +---+---+---+---------+-----------------------+
     4   | R | C | S |         |  NUMBER PAGES           |
         +---+---+---+---------+----------------+--------+
     5   |RT |                                  |   RD   |
         +---+----------------------------------+--------+
     6   |           SUBPARTITION LINK WORD              |
         +-----------------------------------------------+
```

M = Mother partition
D = Dormant
R = Reserved
C = Chain in effect
RT = Real-time partition
RD = Read completion
S = Shareable EMA partition

The user is prompted for the definition of each partition starting with "PART 01,XXXX PAGES?", and stopping when a "/E" is entered. The physical pages will be sequentially allocated to the MAT entries and the "first minus 1" link will thus indicate the end of the defined partitions. The user enters the number of pages, partition type (either RT, BG, RTM, BGM or S) and optionally the reserved flag.

The number of pages, less 1 to exclude the base page, is stored in MAT Word 4 bits 9-0. If it is an RT partition, bit 15 is set in MAT Word 5 (cleared for BG partitions). If it is a reserved partition, bit 15 is set in MAT Word 4. If the partition size is greater than the maximum addressable size (MAXPG), the user is asked to define subpartitions (YES/NO? prompt). A NO response simply results in a large unchained partition being defined. If the user responds YES, or if the user entered RTM or BGM, that partition becomes a mother partition with bit 15 set in Word 3 of its MAT entry and the MAT subpartition link word (Word 6) of the mother partition is initialized to point to itself. It is only at this that subpartitions for a mother partition can be defined. The user has the option of responding YES and still not defining any subpartitions; this would result in a chained partition with the mother partition the only element in the chain. The generator prompts for the next partition definition. If the type code is S, this is a subpartition for the current mother partition. The partition type (either RT or BG) is carried from the mother to the subpartitions. The size of the subpartition cannot be greater than that of the mother, or a GEN ERR 56 is issued and the partition must be redefined. It can, however, be larter than MAXPG, but further subpartitioning is not allowed.

The sum of the subpartition sizes cannot exceed the size of the mother, but may be less. In this case, a GEN ERR 46 results on the subpartition definition causing the overflow, and that partition must be redefined.

8.26.3 Partition Definition Sequence

The following sequence occurs for a partition definition:

1. Clear Subpartition flag, Subpartition prompt flag, and Mother partition flag: SUBS?<--DPMOM<--SUBP?<--0. If NEXTP=MAXPT, set XX in prompt to blanks. If in subpartition mode (SUBMD=1) then prompt: PART XX, XXXX (XXXX) PAGES?

2. Else (SUBMD=0) prompt: PART XX, XXXX PAGES? Get response. If "/E" is entered, e generator proceeds to partition cleanup (step ??). If NEXTP >$MNP, no more MAT entries can be entered and generator issues a GEN ERR 49 and reissues prompt.

3. Retrieve partition size, subtract 1 for base page, and store in DPSIZ. DPSIZ must be >=1 page, else issue GEN ERR 45 and go to 1.

4. Retrieve partition type (RT,RTM,BG,BGM or S); if first two characters are neither RT, BG, or S issue GEN ERR 46 and go to 1.

5. If S, then SUBMD must equal 1 to indicate subpartitioning enabled, else issue GEN ERR 46 and go to 1.

6. If DPSIZ+1> MLEFT (number of pages left in mother partition) then issue GEN ERR 56 and go to 1.

7. Set current def flag to indicate subpartition: SUBP?<--SUBP?+1.

8. Set type of subpartition to that of mother, DPTY <--MOMTY. Go to 10.

9. If RT or BG, check upper range: Require DPSIZ+1 <=PLEFT (total pages left) else issue GEN ERR 45 and go to 1. At this point, are defining an RT, RTM, BG or BGM partition, so SUBMD is set to 0 (may have already been in regular mode). Set DPTY to 1 for RT or RTM so bit 15 of Word 5 can be set; else set DPTY to 0 for BG or BGM.

10. Check third character in response for M. If M is found (RTM or BGM input), set SUBS? <--2 to indicate a mother partition, but turn off SUBPARTITIONS? prompt. If M is not found, this still may be a mother partition if DPSIZ> MAXPG (largest logical partition size).

11. Retrieve reserved flag. If one entered, set bit 15 for Word 4 (DPRSV<--0, -1 otherwise).

12. If SUBS? = 0 or 2, then go to 14, else issue user prompt the user SUBPARTITIONS?.

13. If NO, go to 14. If YES: enable subpartition mode SUBMD <--1; store address of current (mother) MAT address in MOMAD; save mother partition size for subpartition checking, MLEFT <--DPSIZ+1; save mother partition type for its subpartitions, MOMTY <--DPTY; and set bit 15 for Word 3 of current MAT entry making it a mother partition (DPMOM = -1, 0 otherwise).

14. Build new MAT entry. (Words 0 & 3 are completed during partition cleanup):

    Word 0: Set to 0 to indicate a defined entry.

    3: DPMOM is used to (optionally) set bit 15 if a mother partition

    4: DPRSV is used to (optionally) set bit 15 if a reserved partition, DPSIZ stored in bits 9-0.

    5: DPTY is used to (optionally) set bit 15 if an RT partition

    6: If SUBMD = 1, set to MOMAD, else 0. This will set SLW (Subpartition Link Word) to point to itself if mother partition, or to mother MAT entry if a new subpartition at end of chain.

15. If SUBMD = 1 and SUBS? = 0, at least one subpartition has been defined. Current subpartition must then be linked to previous MAT entry (which is either previous subpartition in chain, or mother partition). Since CURMT is memory address of current MAT entry, then (CURMT-1) <--CURMT.

16. If current partition is a subpartition (SUBP? = 1), then MLEFT (DPSIZ+1); else PLEFT <--PLEFT-(DPSIZ+1); bump NEXTP. Go to 1.

17. Partition Definition Cleanup. MAT is scanned, summing up individual partition sizes, until first undefined entry is found (link Word 0 = -1) or end of table is reached. Only regular and mother partition sizes are included in total, and 1 is added to each of these sizes because base page was not included in size stored in Word 4. Subpartition sizes are not included, their pages having already been included in mother partition; a subpartition MAT entry is detected by Word 6 (SLW) being non-zero and mother bit (15) not being set in Word 3. If total number of pages occupied by defined partitions (DPTOT) does not equal number available (DPARE), then a GEN ERR 53 is issued and all partitions must be redefined.

8.26.4 Free Lists

The memory allocation table (resident on the disc) is sorted into three free lists, each based on increasing partition sizes, by setting the link addresses in Word 0 of each MAT entry. The lists, separating real-time, background and chained (mother) partitions, are referenced through the Table Area II entry points $RTFR, $BGFR and $CFR respectively.

The generator starts scanning the MAT with the first partition's entry and stops when the end is encountered ($MNP entries have been threaded) or when the first undefined entry is found (link word = -1). The three list headers DPRTL, DPBGL and DPCL are initialized to 0, and are pointed to by DPRT., DPBG., and DPC., respectively. The list headers (and their lists) are accessed and updated by setting DPLH.,I where DPLH. is set to one of the header pointers, depending on the partition type. The partition type is determined as follows:

> If the mother bit of word 3 is set then both RT & BG mother partitions go into $CFR, or if the RT bit of word 5 is set it is the $RTFR list, and the remaining go into the $BGFR list. The type of list being threaded is irrelevant once the particular header address has been set.

As a particular MAT entry is linked into a list, its starting physical page is stored in Word 3 bits 9-0. DPORG is initially set to the first physical page for partitions from PAGE#, and is updated as pages are allocated to a partition. When a mother partition is encountered, MORG is set before DPORG is updated to the start of the next partition. When MORG is non-zero, the next set of subpartition entries in the MAT have their starting physical page set by MORG (which is incremented after each subpartition). When the next non-subpartition is encountered, MORG is cleared and starting pages are set by DPORG again.

When the threading is completed, the last element in each list is retrieved and the Table Area II entry points $MRTP, $MBGP and $MCHN are set to the page sizes of the largest non-reserved partition in the real-time, background, and mother free lists, respectively.

8.26.5 Modify Program Page Requirements

The IDENT entry for the named program is retrieved; a GEN ERR 48 is issued if the program name cannot be found or if it is of incorrect type. Only disk resident programs (masked types 2, 3, and 4) executing in user partitions can have their page requirements increased. The page requirements of an EMA program cannot be overidden, so if bit 15 of \ID6 is set, a GEN ERR 55 is issued.

When the program ID segment is built, the keyword offset is stored in the program IDENT entry Word 8, bits 7-0. The routine IDFND retrieves the program ID segment address by going thru \ID8 and the keyword value stored on disc. Before program ID segment Word 21 can be updated, the new page size must be verified. The program low main is retrieved from ID segment Word 22 and is converted to its starting page, to which is added the new page requirements (less 1, stored in DPSIZ). If overflow occurs (>32), GEN ERR 51 results. A program's page requirements, stored in IDENT entry Word 8 bits 15-8 when the ID segment was built, are compared against the override in DPSIZ. If DPSIZ is less than IDENT, a GEN ERR 51 again is issued. Otherwise DPSIZ is stored in ID segment Word 21 bits 14-10 of the named program. The page requirement in \ID8 is not updated, however, to allow a re-override.

8.26.6 Assign Program Partitions

The IDENT entry and ID segment address for the named program are retrieved as when modifying a program's page requirements. Only disk resident programs may be assigned to partitions, provided the partition is large enough to hold the program. A GEN ERR 49 is issued if the partition number specified in DPNUM is greater than the maximum allocated (MAXPT) or if the partition is undefined (link Word 0 = -1). The size of the partition is retrieved from its MAT entry Word 4, bits 9-0, and stored in DPSIZ. The page requirements of a non-EMA program are retrieved from ID segment Word 21 compared against DPSIZ. A GEN ERR 50 is issued if the program is too large for the specified partition, otherwise Word 21 bits 5-0 of the program ID segment are set to DPNUM -1 and the RP bit 15 is set.

For EMA programs (bit 15 of \ID6 is set), the page requirements stored in \ID8 bits 15-8 include the MSEG size, but it is the EMA size that must be included when considering whether or not the program will fit in a partition. The program code size is determined by subtracting the MSEG size in \ID6 bits 14-10 from the page requirements in ID8, and adding the EMA size in ID5 bits 13-4, adding 1 if the EMA was defaulted, and storing the result in DPORG. If the resulting page size <DPSIZ, then ID segment Word 21 is updated as mentioned above to reflect the partition assignment, otherwise a GEN ERR 50 results.

8.27 MEMORY PROTECT FENCE TABLE (MPFT)

The six-word MPFT stored in Table Area II on the disk is updated to reflect the logical fence addresses for the following program categories:

WORD 0: type 4 BG disk resident without common
1: memory resident
2: any program using RT common
3: any program using BG common
4: any program using SSGA
5: RT or type 3 BG disk resident without common

Table Area II entry point $DPL (load point for disc resident program) sets Word 0.

The variable FWMRP (first word of memory resident program) sets Word 1.

The variable RTCAD (real-time common address) sets Word 2.

The variable BGBND (background common address) sets Word 3.

The variable SSGA. (SSGA starting address) sets Word 4.

Table Area II entry point $PLP (load point for privileged programs) sets word 5.

Table Area II entry point $MPFT will contain the address of the MPFT.

8.28 MEMORY RESIDENT PROGRAM MAP

The DMS map for memory resident programs, MRMP, stored on the disc in Table Area II is updated for use by the Dispatcher. The MRMP, addressed by Table Area II entry point $MRMP, is 32 words long, with one word per physical register. The map is built as follows:

```
         +-------------------+
     31  |11      .          |
         |11      .          |
         |11      .          |
         +-------------------+   LEFTOVER AREA**
         |11      1          |
         +-------------------+
FPMRL    |11      0          |
+MRP#    +-------------------+
         | MRBP+MRP#         |
         +-------------------+
         | MRBP+MRP#-1       |
         +-------------------+   MEMORY RESIDENT
         |        .          |   PROGRAMS
         |        .          |   & LIBRARY*
         |        .          |
         +-------------------+
         | MRBP+3            |
         +-------------------+
         | MRBP+2            |
         +-------------------+
         | MRBP+1            |
FPMRL    +-------------------+
         | FPMRL-1           |   OPTIONAL:
         +-------------------+   (TABLE AREA II,*
         | FPMRL-2           |   SYSTEM DRIVER
         +-------------------+   AREA*, & COMMON)
         |        .          |   PLUS DRIVER PARTITION
         |        .          |   AND TABLE AREA I
         |        .          |
         +-------------------+
         |        2          |
      2  +-------------------+
         |        1          |
      1  +-------------------+
         | FPMBP             |   MEMORY RESIDENT BASE PAGE
      0  +-------------------+
                  ^
                  |
                  |
             VALUES SET
```

* System Driver Area, Table Area II, and Memory Resident Library are write-protected (bit 14 is set).

** Both read and write-protected.

Word 0 is set to the physical memory resident base page FPMBP. The first word of the memory resident library is converted to its logical page address and is stored in FPMRL. Words 1 thru FPMRL-1 are thus set to their logical and physical page addresses, 1 thru FPMRL-1. If the System Driver Area and Table Area II are to be included in the map (MRTA2=1), their pages are write-protected. MRP# contains the number of pages occupied by the memory resident library and programs. The map words FPMRL thru (FPMRL + MRPGS-1) are thus set to their corresponding physical pages, MRBP+1 thru (MRBP+MRPGS). The library pages (FPMBP+1 to FPMRP-1) are always write-protected. The remaining map words (FPMRL+MRPGS) thru 31 are set starting over at page 0 - this area corresponds to the logical address space above the memory resident area and each page is therefore read- and write-protected (bits 15 and 14 are set in its MRMP entry).

8.29 SETTING SYSTEM ENTRY POINTS

Crucial values are passed to the system from the generator. This is done by stuffing values into locations defined as entry points in Table Area II. The code to update these values on disc is table-driven, with a table entry consisting of these five words:

```
label DEF *+2
      <value to be stored>
      ASC 3,<entry point name>

or   DEC 0     (last entry)
```

Before updating the entry points, the values in the table are filled in. The following entry point values are set as indicated:

$MRMP - Memory address of memory resident map

$ENDS - Physical page following SAM#1

$MATA - Memory address of memory allocation table

$MPSA - # pages/starting page of SAM#1

$MPS2 - # pages/starting page of SAM#2

$MPFT - Memory address of memory protect fence table

$RTFR - MAT entry address of real-time free list header

$BGFR - MAT entry address of background free list header

$CFR - MAT entry address of chained free list header

$EMRP - Last word address of memory resident program area

$DVMP - Memory address of Driver Map Table

$DVPT - Logical starting page of driver partition

$DLTH - Number of pages per driver partition

$MNP - Maximum number of partitions

$MCHN - Page size of largest mother partition

$MBGP - Page size of largest background partition

$MRTP - Page size of largest real-time partition

$IDEX - Memory address of ID extension table

$DLP - Load point address for RT/BG DR programs without common

$PLP - Load point address for privileged DR programs

$LEND - Last word +1 address of memory resident library

$BLLO - Negative lower buffer limit

$BLUP - Negative upper buffer limit

$CL1 - System disc track number where cartridge list begins

$CL2 - System disc sector number where cartridge list begins.

$STRK - Disc track number where system (base page) begins.

$SSCT - Disc sector number where system (base page) begins.

When the above values have been set, there are six values to be stored in the following table (for use by the Configurator):

STARTING AT $SBTB:

Disc address of driver partitions #2 onward
# of pages for driver partitions #2 onward
Disc address of memory resident base page
# of pages for memory resident base page
Disc address of memory resident lib/programs
# of pages for memory resident lib/programs

8.30 ERROR PROCESSING

There are two classes of errors that occur during generation: FMP ERR resulting from files being accessed through FMP calls, and GEN ERR resulting from an illegal generator response or an erroneous condition detected during the generation. In most cases an FMP error will cause a GEN error as well. A count ERCNT is kept for the number of errors occurring during a generation, and is displayed after both normal and abortive generator terminations, in the form: XXXX ERRORS.

On many errors, control will be passed to the operator console by calling TRCHK with a "TR,LU" stuffed in the input buffer, LU being that of the operator console ERRLU. The current input source is pushed down the stack, so after the operator corrects the error (probably by re-entering the response), a simple TR will return to the next response in that answer file.

List file errors encountered after the list file has been created are detected in /LOUT. The error that occurs most frequently results when an extent to the list file cannot be created due to lack of disc space on the same subchannel. Because this error can occur anytime during generation, the status of the input/output buffers LBUF and TBUF must be maintained as they may contain relocatable or absolute code. The FMP and GEN errors reported upon the occurrence of a list file error are therefore issued via EXEC call writes. (Using the normal error reporting routines would result in an eventual call to /LOUT - but recursion doesn't work!) The user is then prompted with an "OK TO CONTINUE?" On a NO response, the generator aborts via \TERM call. On a YES response, LFERR is cleared to indicate that all future list file errors encountered in /LOUT are to be ignored. The ECHO option must then be turned on (if not already on).

8.30.1 Generation Errors

\GNER outputs all errors of the form "GEN ERR XX" where XX is the two digit ASCII error code passed in the A-register. If the A-register is negative, this it implies an error type for which no TR to the ERRLU is to be done (these codes typically pertain to duplicate names or entry points). Otherwise \GNER checks as did \CFIL to determine if control is to be transferred to the operator console; it also saves/restores the return address when calling TRCHK. The flag EOFFL is set in \PRMT to signal that an EOF had been encountered in the answer file. Thus when the POP is done on the answer file stack only to find nothing there, a GEN ERR 19 will not be printed - control will simply be transferred to the console as intended. Since calling \GNER is the realization of an actual error, it is up to the caller to take corrective action.

8.30.2 File Errors

All FMP errors are detected and processed in the routine \CFIL. \CFIL is called after each FMP call is made (i.e., all READF, WRITF, CREAT, CLOSE, OPEN, LOCF, APOSN and RWNDF calls) and checks the error parameter \FMRR. CNUMD is called to convert the error code to ASCII and stuff it into the message "FMP ERR-XX FLNAME". The DCB address is passed to \CFIL in the A-Register. The file directory entry address or the LU is retrieved from DCB Words 0 and 1. An EXEC call read is done to that track and sector and the file name is transferred from Words 0-2 of the directory entry to the error message buffer. If Word 0 of the DCB is 0, it was a Type 0 file and the file name in the error message is set to the LU, "LU XX". Since \CFIL issues error messages, an error can occur on an OPEN or CREAT call, in which case the DCB is not set up correctly. Therefore if the A-Register DCB address is zero, indicating a check following an OPEN or CREAT call, the file name is picked up from PARS2+1, +2, +3 since it always contains the file to be opened. An error never occurs on the OPEN/CREAT of a Type 0 file since the generator routine TYP0 builds the actual DCB, so this combination is not encountered.

\CFIL also determines whether or not a transfer of control to the operator is necessary, in which case TRCHK is done. Some return addresses are saved and restored in case it was TRCHK that originally called \CFIL. \CFIL has two returns, with the error return being at (P+1). It is up to the caller of \CFIL to determine the course of action when a file error occurrs.

8.30.3 Abortive Termination

8.30.3.1 \ABOR

\ABOR issues its own error of the form "GEN ERR 00 XXXXX", where XXXXX is the octal address of the caller of \ABOR. Because \ABOR is called from several places, the address helps in tracing the problem. After issuing the message, \TERM is called for clean-up before termination. The abort may result if a problem exists with the generator's LST,IDENT, or FIXUP table or its scratch file (@@NM@A) such that an entry is no longer there. The loss of a table entry would result from an incomplete disc swap of a table block; this could be an actual generator problem or it could be a hardware problem. First check the hardware on any GEN ERR 00.

8.30.3.2 \TERM

\TERM is called when the operator aborts the generator with a !! command, when a GEN ERR 00, 02, 07, 17, 18, 21, 38, 57, 59, 60, 61 occur, or after file errors to \NDCB, \EDCB, \RDCB, \IDCB or \ADCB. The absolute output file, boot file and modified NAM record file are purged (using a CLOSE call with truncate option), and the list file, relocatable input file, and answer file are closed. The abort message is printed, the generator releases the scratch tracks allocated to it, and the generator terminates.

8.30.4 Miscellaneous Error Processors

8.30.4.1 \INER and \IRER

\INER is called from several places in the main and Segments 1, 5, and 7 to issue the initialization response error GEN ERR 01. It merely calls \GNER, which transfers control to the console. The caller of \INER then reissues the questions for the corrected response from the operator. \IRER calls \GNER for irrecoverable errors 07, 12 and 21, followed by a call to \TERM to perform clean-up and abortion.

8.30.4.2 NROOM and CMER:

NROOM issues errors 02 (not enough space for tables, 512-word minimum) and 38 (ID segment of Segment 3 cannot be found) by calling \GNER, then aborts the generation with a \TERM call.

CMER in Segment 2 issues a GEN ERR 06 when an invalid Program Input Phase command is entered, or when an FMP ERR-XX FNAME occurs on a file referenced in a RELOCATE command. NXTCM then prompts (-) for the next command.

8.30.5 Error Suspensions

The generator detects two error conditions that result in a message sent to the console (only) and the suspension of the generator until the situation is resolved. When the generator requests its six scratch tracks and they are not available, it issues the message "GENERATOR WAITING FOR TRACKS", and reissues the EXEC 4 call with the WAIT bit set. The same sequence of operations occur when an attempt is made to lock the list file (provided it was to a non-interactive lu) where "GENERATOR WAITING ON LIST LU LOCK" is displayed on the console.

8.30.6 Answer File Errors

When doing transfers within TRCHK, special processing is needed for PUSH/POP errors. At POPRR, which results from TR stack underflow, a GEN ERR 19 is issued with a forced "TR,ERRLU". At PUSHR, resulting from TR stack overflow, the stack address is decremented by one to point to Word 6 if the previous entry (actually current since the PUSH was never done) and RECOV is called. RECOV pops the stack to the previous entry, thus enabling a "TR,ERRLU" on return in some cases. When an invalid LU was specified on a PUSH, RECOV is again called at TR3 before issuing the GEN ERR 20; the same holds true at TR4 when an error occurred on the new input file, only here the error code is saved while RECOV is being called.

When an invalid LU or file was specified in the turn-on parameters, STRT2 issues its own errors rather than call \GNER or \CFIL before the answer file and IACOM have been established. Once the LU of the operator console has been determined (default is 1) the ASCII equivalent of that LU is stored in the "TR,XX" message to be used later with all "TR,ERRLU" calls to TRCHK.

8.30.7 Driver Partition Overflow

When multiple drivers are being relocated into a driver partition and a driver overflows the logical memory space reserved for the DP, a warning message of the form:

'DRIVER PARTITION OVERFLOW'

is issued. This does not constitute an error condition and no TR,ERRLU is done. The message is informative only, essentially telling the user to ignore the load map printed for the driver just relocated. That driver will be re-relocated into a subsequent driver partition.

8.30.8 Error Codes

In the following listing, codes that are preceded with "$" signify fatal errors that cause the generation to abort; codes preceded with "-" signify errors that do NOT result in transfer to the operator console. (In some cases, both the $ and - are applicable to an error condition.)

```
+-------------------------------------------------------------------+
|                                                                   |
|$    0: HARDWARE/GENERATOR ERROR (SEND IN BUG REPORT)              |
|     1: INVALID REPLY TO INITIALIZATION PARAMETERS                 |
|$    2: INSUFFICIENT AMOUNT OF AVAILABLE MEMORY FOR TABLES         |
|-    3: RECORD OUT OF SEQUENCE                                     |
|-    4: INVALID RECORD TYPE                                        |
|-    5: DUPLICATE ENTRY POINTS                                     |
|     6: COMMAND ERROR - PROGRAM INPUT PHASE                        |
|$    7: LST,IDENT,FIXUP TABLE OVERFLOW                             |
|-    8: DUPLICATE PROGRAM NAMES                                    |
|     9: PARAMETER NAME ERROR                                       |
|                                                                   |
|    10: PARAMETER TYPE ERROR                                       |
|    11: PARAMETER PRIORITY ERROR                                   |
|    12: PARAMETER EXECUTION INTERVAL ERROR                         |
|    13: BG SEGMENT PRECEDES BG DISC RESIDENT                       |
|-   14: CHECKSUM ERROR                                             |
|-   15: ILLEGAL CALL BY A TYPE 6 OR 14 PROGRAM TO A TYPE 7         |
|-   16: BP LINKAGE AREA OVERFLOW                                   |
|$   17: TYPE 1 OUTPUT FILE OVERFLOW (ESTIMATE WAS NOT LARGE ENOUGH)|
|$-  18: MEMORY OVERFLOW                                            |
|    19: TR STACK UNDERFLOW/OVERFLOW                                |
|                                                                   |
|    20: INVALID COMMAND INPUT LU                                   |
|$   21: '$CIC' NOT FOUND IN LOADER SYMBOL TABLE                    |
|    22: LIST FILE ERROR                                            |
|    23: INVALID S OR M OPERANDS                                    |
|    24: INVALID SELECT CODE IN EQT ENTRY                           |
|    25: INVALID DRIVER NAME IN EQT ENTRY                           |
|    26: INVALID D,B,U,T,X,S,M OPERANDS IN EQT ENTRY                |
|    27: INVALID DEVICE REFERENCE NO.                               |
|    28: INVALID INTERRUPT SELECT CODE                              |
|    29: INVALID INTERRUPT SELECT CODE ORDER                        |
```

```
   30: INVALID INT ENTRY MNEMONIC
   31: INVALID EQT NO. IN INT ENTRY
   32: INVALID PROGRAM NAME IN INT ENTRY
   33: INVALID ENTRY POINT IN INT ENTRY
   34: INVALID ABSOLUTE VALUE IN INT ENTRY
-  35: MORE THAN 63 EQT OR 255 DRT ENTRIES DEFINED
   36: INVALID TERMINATING OPERAND IN INT ENTRY
-  37: INVALID COMMON LENGTH IN SYS, LIB, OR SSGA MODULE.....
$  38: ID-SEGMENT OF SEGMENT 3 NOT FOUND
   39: NOT USED

   40: INVALID EMA PROGRAM TYPE
   41: MULTIPLE EMA DECLARATIONS
   42: INVALID REFERENCE TO EMA SYMBOL
   43: INVALID MSEG SIZE
   44: SAM EXCEEDS 32K LOGICAL ADDRESS SPACE
   45: INVALID PARTITION SIZE
   46: INVALID PARTITION TYPE
   47: INVALID PARTITION RESERVATION
   48: INVALID OR UNKNOWN ASSIGNED PROGRAM NAME
   49: INVALID PARTITION NUMBER

   50: PROGRAM TOO LARGE FOR PARTITION SPECIFIED
   51: INVALID PAGE OVERRIDE SIZE
-  52: ILLEGAL REFERENCE TO SSGA ENTRY POINT
   53: SUM OF PARTITION SIZES DOESN'T EQUAL # PAGES LEFT
-  54: SUBROUTINE OR SEGMENT DECLARED MORE COMMON THAN MAIN
   55: PAGE REQ'MTS OF EMA PROGRAM CAN'T BE OVERRIDDEN
   56: SUBPARTITION SIZE OR SUM OF SIZES > THAN MOTHER PART'N SIZE
$  57: MISSING SYSTEM ENTRY POINT
-  58: ILLEGAL REF TO TYPE 0 SYS ENTRY POINT BY NON-TYPE 3 MODULE
$  59: DRIVER PARTITION OVERFLOW

$  60: LONG ID SEGMENT LIMIT OF 254 EXCEEDED
$  61: PHYSICAL MEMORY OVERFLOW
-  62: INVALID INSTRUCTION REFERENCE TO AN EMA SYMBOL
$  63: OPERATING SYSTEM MODULE OUT OF SEQUENCE. MAPPED OS MODULES
       MUST BE INPUT DURING PIP IN ASCENDING ORDER.
   64: TRIED TO ASSIGN MORE THAN ONE SHAREABLE EMA LABEL TO THE
       SAME PARTITION.
   65: ILLEGAL SHAREABLE EMA LABEL ENTERED.
   66: MORE THAN THE MAXIMUM NUMBER OF SHAREABLE EMA PARTITIONS
       ENTERED.
   67: DUPLICATE LABEL ENTERED.
   68: DURING SHAREABLE EMA PROGRAM PHASE THE NAME OF A NON-EMA
       PROGRAM WAS ENTERED.
   69: TRIED TO ASSIGN A PROGRAM TO A SHAREABLE EMA PARTITION.

   70: THE SHAREABLE EMA USED BY THE SPECIFIED PROGRAM IS
       LARGER THAN THE PARTITION ASSIGNED TO THE GIVEN LABEL.
```

```
|    71: AN UNDEFINED SHAREABLE EMA LABEL WAS ENTERED.                  |
|$   72: SYSTEM AND PROGRAMS LOADED ON-LINE BY THE GENERATOR            |
|        OCCUPY MORE THAN 255 TRACKS.                                   |
|    73: TOTAL NUMBER OF TRACKS ON LU 2 AND LU 3 IS GREATER THAN        |
|        1600.                                                          |
|$   74: THE MODULE "$EMA$" IS MISSING.                                 |
|-   75: SYMBOL TRUNCATED TO 5 CHARACTERS.                              |
|-   76: LOCAL EMA, SAVE, OR PURE CODE NOT ALLOWED IN GENERATIONS.      |
|-   77: NEW RELOCATABLE RECORD CANNOT BE TRANSLATED.                   |
|-   78: TWO WORD RPL'S NOT ALLOWED IN GENERATIONS.                     |
|-   79: EMA OR ALLOCATE(SAVE COMMON) NOT ALLOWED.                      |
|                                                                       |
|-   80: INFO FIELD OF RELOCATABLE RECORD IGNORED.                      |
|-   81: WEAK EXTERNAL IGNORED.                                         |
|-   82: DEBUGGER RECORD IGNORED.                                       |
|$   83: CODE PARTITION OVERFLOWED, FATAL ERROR                         |
|-   84: BLOCKS/TRACK IS MULTIPLE OF 7 (warning).                       |
|    85: INSUFFICIENT DISC SPACE REMAINING.                             |
|    86: SUBCHANNEL SPEC. NOT PREVIOUSLY DEFINED.                       |
|    87: SUBCHANNEL SPEC. NOT A CTD.                                    |
|    88: DISC CACHE ALREADY DEFINED.                                    |
|    89: CANNOT CACHE ON THIS DISC (ADDRESS INCOMPATIBLE).              |
+-----------------------------------------------------------------------+
```

9.1 OVERVIEW OF SYSTEM BOOT-UP OPERATION

The ROM or bootstrap loader loads the boot extension into memory from disc. The boot extension then loads the system into memory up to the top of the third part of the configurator ($CNF3) and then transfers control to the system. The system startup routine $STRT builds the system map and then immediately makes a subroutine call to the configurator program.

The configurator program loads the memory resident programs, library, base page, code and driver partitions. If the user has requested memory or I/O reconfiguration by setting the switch register bits as described in the following two sections, the configurator will perform the reconfiguration by interacting with the user. For I/O reconfiguration, the configurator will allow assigning the I/O device from any select code 10 octal, or greater, to any other select code up to 77 octal. The configurator makes the I/O reconfiguration permanent on disc if the user opts for it.

The configurator then asks the user if memory reconfiguration is desired. If the response is yes, the user has the option to change the size of the system available memory (SAM) extension, the partition definitions, modify program size, and change program assignments to partitions. The SAM extension and user partition definitions are allowed to be defined to avoid bad pages. In addition, sharable EMA partitions and programs may be defined. The configurator makes the memory reconfiguration permanent on disc if the user chooses the option.

Upon completion, the configurator restores the system to its initial state and transfers control back to the system initialization routine.

9.2 DISC BOOT EXTENSION

After the ROM loader loads the track 0 sector 0 boot extensions, the disc boot extension examines bit 5 of the switch register. If this bit is set, it means that the user wants I/O or memory reconfiguration. A HLT 77B is issued so that the user can reset the switch register with new disc and console select codes. If bit 5 was not set, no halts are issued and the switch register is cleared.

The disc boot extension communicates with the configurator via the switch register. Therefore, the switch register contents should not be changed until after the completion of the boot-up procedure.

The disc boot extension then relocates itself to the top of the first 32K. The select code for the disc is extracted from bits 6-11 of the switch register.

9.3 USING THE BOOTSTRAP LOADER

The bootstrap loader configures itself to a new disc select code if it was entered in the switch register when the HLT 77B occurred. The bootstrap then loads the track 0 sector 0 boot extension at the top of the first 32K of the memory. The disc select code is passed to the boot extension by setting it in bits 6-11 of the switch register.

The disc I/O instructions are then configured. The disc boot extension now loads the RTE-6/VM system into memory as one block, up to the top of the third part of the configurator ($CNF3). The high address had been set up in a variable HIGH in the disc boot extension by the on-line generator. Control is then transferred to the system startup routine. If the size of the disc boot extension must be changed in the future, the generator will be affected.

9.4 CONFIGURATOR PROGRAM STRUCTURE

The configurator program is divided into two major pieces.

The first piece is composed of three relocatable modules($CNF1, $CNF2, $CNF3), referred to here collectively as $CNFG. These modules are relocated by the generator as type 16 programs. Type 16 modules are a special type loaded by the generator so that they overlay default SAM. This part of the configurator is broken into three parts to take advantage of current page linking, thereby leaving more base page links for use by system code. $CNFG is mainly responsible for I/O configuration. $CNF1 is the mainline code for configurator initialization and I/O reconfiguration. $CNF2 contains utility routines used in I/O and memory reconfiguration. The first part of memory reconfiguration, together with exit and cleanup code, is contained in $CNF3.

The second part of configurator $CNFX is relocated by the generator as a type 3, background, disc-resident program. It is loaded into memory by $CNFG under a background disc-resident program map.

```
------------------------------
|                            |
|                            |
|                            |
| $CNFX (background disc     |
| resident program)          |
|                            |
|                            |
|                            |
------------------------------
| Table Area II              |
------------------------------
| Driver Partition           |
------------------------------
| Table Area 1               |
------------------------------
| User Base Page             |
------------------------------
```

MAP USED BY $CNFG TO LOAD $CNFX

The disc address for $CNFX is taken from word 26 (track number, word 0 = link) and word 35 (sector address) of the ID segment for $CNFX. $CNFX is loaded into the first contiguous block of memory that is large enough, starting from the end page of memory resident program +1 if there is no SAM extension; otherwise, it is the last page of SAM extension +1. The base page is loaded starting at the logical address contained in word 24 of the ID segment for $CNFX. The next sequence of physical pages up to the end of Table Area II is copied from the system map. The number of DMS registers to be copied is page number extracted from $PLP minus 1. The $CNFX program is then loaded, starting at the logical address contained in the word 22 of the ID segment, and physical page is physical page number for the user base page. The high address up to which $CNFX is loaded is taken from word 23 of the ID segment.

By making $CNFX a type 3 program, it can access all system entry points and thus communicate with $CNFG. $CNFG has to reside in the first part of SAM as a system module, so that it can use the $XSIO routine in RTIOC for input and output. EXEC calls cannot be used since the system has not been initialized. The work load divided between $CNFG and $CNFX is based only on how much code can fit in the first block of SAM. $CNFG will load the memory resident, code, and driver partitions, reconfigure I/O and contain the I/O subroutines to be used by $CNFX. $CNFX will handle memory reconfiguration.

The two configurator programs communicate with each other through external entry points defined in $CNFG. Every time I/O is performed during execution of $CNFX, an SJS instruction is issued by $CNFX to jump to the I/O subroutine in $CNFG. To obtain the parameter values defined in the calling sequence, $CNFG does a cross-map load. To return to $CNFX from the $CNFG I/O subroutines, $CNFG performs a UJP return addr,I instruction.

When the configuration procedure is completed, the $CNFG program will be overlaid by buffers using the system available memory. These tables show an image of physical memory after the disc bootstrap load and after the configurator load.

9.5 INITIALIZATION PROCEDURE FOR $CNFG

The $CNF1 module clears all interrupts as soon as it is given control. It then saves the base page locations SYSTY (EQT entry address of system TTY), DUMMY (privileged I/O card location) and EQT1-EQT4 and clears them. Clearing SYSTY prevents the user from gaining control of the system by striking a key on the keyboard of the system console and getting a prompt. DUMMY is cleared to prevent any privileged interrupts. SKEDD is cleared and $LIST is set to 1, to prevent any scheduled programs from running. These locations will be restored just before $CNFG returns control to the system start-up routine. If the console or list device is buffered, $CNFG will clear the B bit in word 4 of the device EQT to make it unbuffered so that system available memory is not needed for I/O. The original buffered or unbuffered status will be restored before returning control to the system. I/O errors cannot be handled due to the fact that operator console capability is taken away from the user. If an I/O error does occur, the boot procedure must restart. The $CNFG program will obtain the new select code (if any) for system disc from the switch register and reconfigure it (see the I/O Reconfiguration Section for details) in memory so that disc accesses can be made to load memory resident and driver partitions.

9.6 CONSOLE RECONFIGURATION

If reconfiguration of the system console is requested by entering a new select code in bits 0-5 of the switch register, the system console is reconfigured as follows:

1. If the new system console driver type as the old system console, point the old system console EQT entry to the new select code.

2. If the new system console needs a different driver type, scan the EQTs to find a matching driver type and the new select code. The new EQT found during the scan is used for the new system console. No change is made in the I/O reconfiguration of the old system console.

3. If an EQT with a matching driver type and the new select code is not found, scan the EQTs to find one with a matching driver type. The first such EQT encountered is used for the new system console. The select code that this EQT previously pointed to is the old select code.

9.7 LOADING MEMORY RESIDENT PROGRAMS, DRIVER PARTITIONS

The on-line generator passes a table, $SBTB, of six parameters to $CNFG:

$SBTB:
1. Disc address for start of code and driver partitions.
2. Number of pages for all code and driver partitions.
3. Disc address for memory resident base page.
4. Number of pages for memory resident base page (always 1 if memory resident base page is present).
5. Disc address for memory resident library.
6. Number of pages for memory resident library and programs.

Each of the number-of-pages and starting-disc-address couplets in $SBTB is broken up into several triplets of the form:

```
starting memory address
number of words to transfer
starting track/sector address
```

These triplets are set up to avoid crossing track boundaries. They are passed to the $XSIO routine in RTIOC module for reading the corresponding code from the system disc.

The configurator treats the OS code partitions just as driver partitions. In this way the code partitions can be loaded just like driver partitions.

The code and driver partitions are loaded into memory as one big piece of code. The on-line generator stores the relocated OS code and drivers in partition-size chunks on the system disc. If the number of pages taken up by the partitions is greater than the maximum addressable partition size, $CNFG breaks them up into chunks large enough to fill up the logical address space and loads the partitions one chunk at a time. The maximum logical address space is determined by 31-$CMST ($CMST is the starting page of common). The first chunk of partitions is loaded under a user map whose starting page is contained in $ENDS entry point. The starting physical page number to build user map to load subsequent blocks of driver partitions is determined by adding the size of the maximum logical address space to the starting page number used to build the previous map. The starting disc address for subsequent chunks is determined by picking up the number of words from the last set of triplets, dividing it by 64 to get the number of sectors and adding it to the track/sector address in the triplet.

If the number of pages for memory resident base page is 0, the system has no memory resident programs. In this case, the configurator will proceed to configure I/O. In the event that the memory resident programs have been generated into the system, $CNF1 loads the memory resident partition map $MRMP into the mapping registers. The memory resident base page is then loaded into memory, starting at logical address 2. The memory resident library and programs are loaded into memory under $MRMP map, starting at the logical address contained in base page location LBORG.

9.8 I/O RECONFIGURATION

I/O reconfiguration is performed in $CNF1 by assigning the current select code's trap cell and interrupt table entry to the new select code. The equipment table entry pointing to the current select code is changed to point to the new select code. Initially, the changes needed for I/O reconfiguration are recorded in tables in $CNF1's area of memory. At the end of the I/O reconfiguration, these changes are transferred to the trap cell, interrupt table and equipment tables in the system in memory. To enable the configurator to load the driver partitions and memory resident programs and also to perform I/O and memory reconfiguration interactively, the system disc, system console and the list device select code configurations have to be changed in the actual tables in the system in memory before the reconfiguration process can be undertaken.

9.9 I/O RECONFIGURATION TABLES

Several tables are used to record changes made to trap cells, interrupt table and equipment table during the I/O reconfiguration process. All tables are initialized to -1. Following is a description of these tables.

TRPCL - is 70 words long with one word per entry. Each entry corresponds to the actual select code numbers varying from 10 octal to 77 octal. TRPCL is used to hold all changes made to trap cells during I/O reconfiguration.

INTBL - has the same structure as TRPCL. This table is used to record all changes made to the interrupt table.

EQTBL - has the same structure as TRPCL. This table is used to contain address of EQT word 4, the select code entry which has to be changed to point to the new select code. An entry can be made in this table only i an EQT pointing to the corresponding current select code can be found.

OLSTB - has the same structure as TRPCL. This table is used to contain the current select code number corresponding to the new select code. No entry is made if the new select code is assigned a privileged I/O card by the entry

x,PI

where x is the new select code.

OLSTB is used for the following situations:

1) If the sequence of responses is:
   x,y
   x,z

   where x is the current select code and y and z are the new select codes.

   OLSTB is scanned to check if select code x has been previously assigned to another select code y. If so, the changes made to to y are erased from TRPCL, INTB1 and EQTBL; x is now assigned to z.

2) If the sequence of responses is:
   x,y
   z,y

   OLSTB is used to get the previously assigned select code x. TRPCL and INTBL entries of x are erased.

SVTBL - has four entries containing information for up to two new system disc select codes, one new select code for the system console and a new select code for the list device. The format for each entry is:

word 1 - new select code number
word 2 - original trap cell contents for new select code
word 3 - original interrupt table contents for new select code
word 4 - address of EQT word 4 that originally contained new select code number

The interrupt table and trap cell entries in the system tables are changed to the reconfigured system disc, system console and list device select codes. SVTBL is used to remember original values for the new select codes to which these devices are assigned.

RSTBL - has four entries containing information for the system disc select codes, system console and the list device select codes. Each six-word entry has the format:

word 1 - old (current) device select code
word 2 - INTBL entry for old select code
word 3 - TRPCL entry for old select code
word 4 - INTBL entry for new select code
word 5 - TRPCL entry for new select code
word 6 - EQTBL entry for new select code

RSTBL is used to set up the TRPCL, INTBL, EQTBL and OLSTB tables to the state they were in before starting the I/O re-configuration phase. This is done to re-start I/O re-configuration when /R is entered when responding to the "CURRENT SELECT CODE, NEW SELECT CODE?" query.

9.10 I/O RECONFIGURATION PROCEDURES

I/O reconfiguration is done mainly by two procedures, INENT and IPROC. INENT is used to fill entries in TRPCL, INTBL, EQTBL and OLSTB as the responses for the current and new select code pairs are accepted. IPROC is used to transfer the changes made due to these responses during I/O re-configuration to the appropriate tables in the system. A description of these two procedures follows:

INENT - Say the current and new select code pair response just accepted

x,y

where x is the current select code and y is the new select code. OLSTB is scanned to find out if x had been assigned to another select code z previously; i.e., if one of the previous responses was

x,z

If such a response was made these steps are followed:

1. Scan SVTBL to see if select code z is the new select code for the system disc, system console or the list device. If so an error has been made in the x,y response since the assignment of new system disc, system console or list device select code cannot be changed.

2. Erase the assignment of x to z. If one of the previous responses was z, w, the TRPCL and INTBL entries for select code z get JSB LINK,I (where LINK is a base page address containing the address of $CIC routine) and 0, respectively. If z was not previously assigned to another select code, the TRPCL and INTBL entries for z are assigned -1 to signify that no changes were made for select code z. OLSTB and EQTBL entries for z are also changed to -1.

If z was the select code for TBG or the privileged I/O card originally, restore it.

If the new select code y is also a new select code for the system disc, system console or the list device, then an error is posted.

If y is currently the select code for the TBG or privileged I/O card, unassign it.

If x is currently the select code for the TBG or privileged I/O card, then y is now the new select code for one of these cards.

If select code x is a new select code for the system disc, system console or the list device, use SVTBL entries for x to fill in TRPCL, INTBL and EQTBL entries for select code y. Otherwise use the entries from the system trap cell and interrupt table for select code x to fill TRPCL and INTBL entries for y. Scan the equipment table to find an EQT pointing to select code x. If such an EQT is found, enter the address of its fourth word in EQTBL entry for y.

If the assignment of current select code x has not been changed during the I/O reconfiguration process, assign JSB LINK,I (where LINK is a base page location containing the address of the $CIC routine) to TRPCL entry for x and a 0 to INTBL entry for x. Return to the calling routine.

IPROC - This routine is used to transfer the changes made as a result of responses of current and new select code pairs to the appropriate tables in the system. Say select code y has been passed to IPROC. If there was no change made in the assignment for select code y, then return.

Transfer TRPCL and INTBL entries for select code y to the appropriate trap cell and interrupt table entries. If there is an EQTBL entry for y, then get the OLSTB entry for y, say x. Then x was the current select code and y the new select code. Scan all EQT's and for every EQT that points to select code x these steps are performed.

1. If the EQT is for the system disc, system console or the list device, make no changes and look for the next EQT pointing to select code x.
2. If word 1 of the EQT is a -1, it indicates that the EQT has been changed to point to select code x, make no further changes and look for the next EQT pointing to select code x.
3. Change word 4 of the EQT to point to select code y and set word 1 to -1, indicating a change has been made in this EQT. Look for next EQT pointing to select code x.

After all EQTs have been searched, scan the OLSTB table to see if an entry for the select code y has been made. If select code y was never used as a current select code, then scan the EQTs to clear out any unchanged EQ's that may be pointing to select code y. For every EQT these steps are followed:

1. If the first word of the EQT is a -1, make no changes.

2. If the EQT is for the system disc, system console or the list device make no changes.

3. If the EQT points to select code y, clear the select code entry in word 4 of the EQT.

Return from the IPROC routine after all the EQTs have been examined.

To make I/O re-configuration permanent, the following tables and base page locations must be written out on disc: Interrupt table, Trap cells, word 4 of all EQTs, word 1 of the device reference table (DRT) which is the entry for the system console, base page locations TBG (1674B), SYSTY (1675B) and DUMMY (1737B).

Prior to writing the interrupt table on the disc, its entries for the new console and list device select codes are saved. These are replaced with the entries that existed before I/O was performed to these two devices. The saved entries are restored after the interrupt table is written on disc. This is done because some of the console and list device drivers, when executed for the first time, change the interrupt table entries.

9.11 MEMORY RECONFIGURATION

For the most part, memory reconfiguration is straightforward. $CNF3 accepts a list of up to 100 bad pages in an increasing order. The end of the list is marked by a -1. In the $CNFX program, System Available Memory (SAM) extension and user partitions can be redefined to avoid the bad pages in memory. To modify program size, and to assign and unassign programs from partitions, changes are made in the program's ID segment.

9.12 DEFINING SAM EXTENSION

The number of pages in the SAM extension as it is currently defined is determined from the system entry point $MPS2. The physical starting page number for SAM extension is determined by adding the contents of $ENDS, the number of pages taken up by the code and driver partitions, and the number of pages for memory resident base page, library and programs. These last three values were passed to the configurator by the generator in $SBTB table. $CNFX then checks to see if this resulting starting page of SAM extension is included in the list of bad pages. If so, the start page of SAM extension is incremented to avoid the bad page (and any other consecutive bad pages). If $MPS2 is not zero, this start page is compared with it. If they do not match, $MPS2 is changed to match this newly evaluated starting page. This case may happen if some previously bad pages at the start of SAM extension were replaced with a new memory module or some pages at the start of SAM extension went bad. The maximum number of pages available for SAM extension is:

say a = 32 minus $SENDS
b = # pages in physical memory - start page of SAM extension

If a>b, then number of pages in SAM extension is b; otherwise, it is a. If a change in SAM extension is desired, $CNFX sets up the $MPS2 word with start page and number of pages for SAM extension. A scan of the list of bad pages is performed next. The number of pages in SAM extension is divided up into blocks of memory between bad pages. For every block of memory on which SAM extension is defined, two entries are made in the 10-word $SMTB table (initialized to 0) in $CNFG. The first entry indicates the starting page number of the block of memory. The second entry contains the number of pages included in this block of memory. If the size of SAM extension is such that more than five blocks of good memory are required, then an error. The system map is set up to reflect the new page numbers used by SAM extension. SAM extension is write protected.

$SMTB table is written out on disc in $CNFG's area if this memory reconfiguration is made permanent. After this point, $CNFG checks the $SMRB t able every time the system is booted, If SAM extension was divided up into more than one block of contiguous memory, the system map is set up for the physical page numbers recorded in $SMTB.

If SAM extension ends on page 31, the number of words in SAM extension must be decreased by 2 so that the last address for SAM extension is 77775B. The last two words of page 31 are reserved for system use.

9.13 DEFINING USER PARTITIONS

$CNFX picks up blocks of memory between bad pages, or pages remaining if there are no more bad pages, and prompts user to define partitions for it. Partitions defined for a particular block of memory must use all pages in the block, otherwise $CNFX asks the user to redefine partitions for that particular block. $CNFX fills the start page of the partition, the number of pages, reserved bit R, and RT or BG values into the MAT entry for the partition. If the partition is defined to be a mother partition, the M-bit in the MAT entry is set. Every partition following with an 'S' (son) as the fourth parameter is a subpartition. The block of memory to be allocated to subpartitions is the same as that used by the mother partition. The subpartition link word (SLW) of the mother partition is made to point to the first subpartition, the SLW of the first subpartition points to the second and so on. The SLW of the last subpartition points back to the mother partition.

After all the partitions have been defined, $CNFX threads them into either real time, background or chain partition (mother) free lists. Several passes must be made through the MAT table to thread each list. $MBGP, $MRTP, $BGFR, $RTFR and $CFR entry points are set up by $CNFX while threading these lists.

Once the lists are all threaded, the configurator proceeds to define sharable EMA partitions and programs. The $EMTB table is first cleared of all entries and the SH bit (sharable EMA flag) in all MAT entries is cleared. Once the old partitions are cleared, the user is asked to define the new partitions and their labels. This information is placed in the $EMTB table. In addition, when a partition is declared to be sharable EMA, the configurator sets the SH bit in its MAT entry. A special case is encountered when a mother partition or one its subpartitions is declared to be sharable EMA. If a subpartition is declared to be sharable EMA, then the SH bit the for the MAT entry corresponding to that subpartition and the subpartition`s mother are set. The SH bits for the other subpartitions are left clear. If, however, the mother partition is declared to be sharable EMA, the SH bit for the mother`s MAT entry and all the subpartition MAT entries are set.

To make memory reconfiguration permanent, the following tables are written out on disc:

$SMTB in $CNFG's area
Memory Allocation table (MAT)
$EMTB
ID segments

The system entry points to be written out on discs are:

$MCHN, $MBGP, $MRTP, $BGFR, $RTFR and $CFR

9.14 TRANSFERRING DATA FROM MEMORY TO DISC

Following is a description of two of the procedures used to make memory or I/O reconfiguration permanent.

MEMDS: This procedure is used to convert the given memory location in system code into a corresponding disc location. The disc location is determined in terms of track number, 128-word sector number, and number of words offset within the sector.

1. Divide the memory location by the number of words in a track on the system disc. The quotient is the track number.

2. Divide the remainder by 64. The remainder is the number of words offset into the sector.

3. Add the number contained in the $SSCT entry point sectors to the quotient (to account for the boot extension) to get the sector number.

4. If the sector number is greater than the number of sectors per track, increment track number by one and the sector number is 0.

$TRTB: This procedure is used to transfer a reconfigured table from memory to a corresponding disc address.

1. Use the above procedure to determine track number, sector number and number of words of offset into the sector.

2. Read the 128-word sector into a buffer.

3. Calculate the size of the space available for the table in the first 128-word sector: 128 - number of offset words.

4. If the table to be moved is smaller than the first buffer, go to step 5. If the table to be moved is equal to or larger than the buffer, go to step 7.

5. If the table to be moved is less than the full buffer minus the number of offset words, move the table into the 128-word buffer sector, starting at the offset word and overlaying the contents of the buffer for the exact length of the table.

6. Write the sector buffer back to disc.

7. If the table to be moved is equal to the size of the buffer minus the number of offset words, move the table into the buffer starting at the offset word and overlaying the contents of the rest of the 128-word sector buffer.

8. Write the sector buffer back to disc.

9. If the table to be moved is greater than the size of the first buffer, divide the remaining number of words in the table by 128. The remainder is the number of words that fall in the last sector.

10. Read the next sector from disc.

11. Transfer the words from the table into the sector buffer, starting at word 0 and overlaying the contents of the 128-word sector buffer.

12. Write the sector buffer back to the disc.

13. When the last sector required for the table is read in from disc, transfer the remaining number of words from the table into the sector buffer, starting at word 0 and overlaying the contents of the 128-word sector buffer to the exact length of the table.

14. Write the sector buffer back to the disc.

```
+-------------------------------------------------------+-----------------+
|                                                       |                 |
|   SWTCH                                               |  CHAPTER  10    |
|                                                       |                 |
+-------------------------------------------------------+-----------------+
```

10.1 INTRODUCTION

This Chapter is intended to serve as an aid to the programmer when modifying the SWTCH program. It assumes that the reader is familiar with the RTE-6/VM generation process, and has run both the on-line generator and SWTCH. It assumes familiarity with the RTE-6/VM System Manager's Manual, especially the section on the operation of SWTCH.

This Chapter should be used in conjunction with the SWTCH listings, as it in no way attempts to provide the level of detail given by the code itself and comments. The technical specifications will aid in using the listings by discussing overall structure and flow, some data structure and complex algorithms/techniques.

10.2 OVERVIEW OF SWTCH ORGANIZATION

SWTCH consists of a main program and three segments: SWSG1, SWSG2, and SWSG3. SWSG1 is a self-contained 7900 disc driver. SWSG2 is an interface to 7905/06(H)/10H/20(H)/25(H) discs that calls primitive routines in the disc utility library &DSCLB. SWSG3 is an interface to CS/80 discs. Because SWTCH can be used to transfer a 7900, MAC/ICD, or CS/80 disc-based system, all segments are present. However, only one segment is needed with the main for each SWTCH execution.

Before looking at the layout of SWTCH, a few words must be said about the internal buffering scheme. SWTCH maintains one buffer, BUFR, for use in reading from and writing to the new system area. BUFR is declared to be 128 words long, the size of a type 1 FMP record or one disc block. As SWTCH progresses, the buffer area covered by BUFR increases to 512 words, and finally to 11776 words, the size of one track on the largest disc (7933). What happens is that as BUFR "grows" the data stored in it overlays SWTCH code that will not be referenced again. When the point in the SWTCH code is reached where no more no-longer-necessary code can be overlaid, then there will be a BSS to fill out the rest of the needed buffer space. For example, the BSS 11776+BUFR-* reserves space for the full track of information necessary to be read in by means of the DISKD call of the next line.

10.3 LAYOUT OF SWTCH CODE

```
MAIN PROG: \CSBF,IBBUF:60 words  (command buffers for &DSCLB,  16 words) and
           $DTCLB (60 words)

 (SWTCH)   BUFR- 128 words              512-wd
           messages                     buffer

           messages, constants
           VFYSY,VTOSO                  overlayable
                                        subroutines             11776-wd
           OK?,YE?NO,TARGT                                      buffer
           PARMP,PYN
-------------------------------------------------------------------------

Main Entry Pt:

           SWTCH
           VERIF - verify validity of system file.
           \SWTM - display destination I/O Configuration.
           OKAY
           SAVE?
           SUBI?
           SUBI5
           INIT?
           AUTO?                        end of 11776-wd buffer
           PURGF                        (overlaid code)
-------------------------------------------------------------------------

           BFULL
           PUR6                         Non-overlaid code
           XFER
           DDONE
           ISUBS
           UPTAT
           BOOT?
           \XOUT
           SWAPD
           ULDSK
           FINSH
           DISKD - Subroutine which calls the correct segment to do
           SPINT   Disc I/O
           CLRBF
           UPDAT
           PURGT                        subroutines
           \BLIN,\DSPL,\RDIN,\DFLT,LOOP   & etc.
           \CVAS,GETD,GET#,CHKSM
           variables,messages constants
```

```
SEGMENT1:
 (SWSG1)
(7900 driver)
           constants
           \STDO
           \GDMA
           \RDMA
           INIER
           I/OTB & I/OTC
           DISKO,INTON
           SEEK,STATC
           ESUB
           messages

SEGMENT 2:
(SWSG2-ICD/MAC interface)
           \SETD
           \DSK5
           DSGO
           ENDBR
           ENDOK \
           FAULT  \
           RECAL   \
           EOCYL    \
           DSKER     > Branch Table processing blocks
           DEFTR    /
           ILSPR   /
           ST2ER  /
           UWAIT /

           XFER
             SEEK?
             ADRC?
             FMSK?
             READ?
             RDFS?
             WRIT?
             INIT?
             VRFY?
            ENDX?
            XEXIT
          SKIP?
          CKST1
          STFIX
          REQST
          NIXSP
          FMTR?
          DADTR
          CYLOG
          RPORT
```

```
        ESUB
        SPECR
        NOSPR
        PARER
        NRDER
         FRMER
         PROTR
         DCYLR
         INBLK

         constants
         messages

SEGMENT3:
(SWSG3 - CS/80 Interface)
         START - segment entry point
         \CS80 - setup characteristics of system subchannel
         \CSET - setup parameters for disc library
         \CDSK - main read/write processing code
         LG2PH - convert logical to physical address
         READ
         WRITE
         ERROR - process status return from disc library
           RJCT?
           FALT?
           ACCS?
           INFO?
         E.FTL - fatal error processor
         E.INT - internal error processor
         E.RTY - retry processor
         E.PMR - prompt and retry
         E.ERT - error rate test/sparing
         ERPRT - prints error message
```

10.4 TURN-ON PARAMETERS

SWTCH allows the following turn-on sequence:

```
                           addr/unit/pltr
RU,SWTCH,namr,scB/disc LU,{     or        },autoboot,filesave,type-6,init
                           addr:unit:vol
```

where:

namr — is the name of the FMP file that contains your generated system. This may be specified in the following form:

filename[:security code[:cartridge label]]

This file must exist on a standard host system subchannel. If a target cartridge is to be inserted for the SWTCH process, the file must not exist on the cartridge that is to be swapped out for the target.

scB/disc LU — sc: for the 7900 disc, sc is the select code of the target disc controller (octal value with a B as the terminating character). This target select code does not need to be configured into either the host or the destination RTE system. It is used as a means of specifying the correct controller I/O card for the transfer. SWTCH configures its own driver to this select code.

disc LU: for switching MAC, ICD, or CS/80 disc-based systems.The target disc LU is the logical unit number of any disc subchannel on the target disc. The LU is not affected by SWTCH. It is a reference for SWTCH to find the select code of the target disc driver. The target disc driver (DVR32 for MAC discs, DVA32 for ICD discs, or DVM33 for CS/80 discs) must be present in the host system.

Neither LU 2 nor LU 3 should be specified as the target disc LU because the system does special checks to protect these LUs. If LU 2 or LU 3 is specified for the target disc and that disc, while being initialized, is found to contain more sectors per track than the host system's LU 2 or LU 3, SWTCH will be aborted with an I007 error.

addr/unit/pltr — This set of parameters is for ICD or MAC discs. A prompt for the proper disc information will be issued, based on the disc LU, if any other syntax is used.

address - for ICD discs, enter the target ICD address number (0-7) where the new system will be stored.

unit - for MAC discs, enter the hardware unit number (0-7) where the new system will be stored.

platter - for 7900 discs, enter the logical surface number where the new system will be stored (0, 2, 4, or 6 for the fixed platter; 1, 3, 5, or 7 for the removable platter).

addr:unit:vol — address - enter the HP-IB address (0-7) for the target CS/80 disc, where the new system will be stored.

unit - the unit number (0-14) associated with the address for the target CS/80 disc where the new system will be stored. Default is zero (0). Normally, unit number of zero is used.

vol - the volume number (0-7) associated with the unit number for the target CS/80 disc where the new system will be stored. Default is zero (0). Normally, a volume number of zero is used.

The disc system will be transferred to the subchannel that was defined as LU 2 during system generation.

autoboot — is the automatic boot-up option.

Specify Y (yes) to attempt an automatic boot-up following the transfer of the new system. The host configuration must match the destination configuration. Refer to the paragraph titled AUTOBOOT SPECIFICATION for more detail on this match.

Specify N (no) to deny automatic boot-up.

filesave — is the filesave option.

Specify Y (yes) to attempt saving the target disc file structure during the transfer.

Specify N (no) for not saving the target disc file structure.

type-6 — is the option to purge Type 6 files.

Specify Y (yes) to purge the Type 6 files on the target disc during the transfer.

Specify N (no) to save the Type 6 files on the target disc during the transfer.

init — is the subchannel initialization option.

Specify Y (yes) to request initialization of destination disc subchannels other than the system subchannel. SWTCH will prompt you for each subchannel that was defined to be on the same disc controller (MAC discs) or interface card (ICD discs) as the system subchannel.

Note that SWTCH will not initialize subchannels defined on the 9895 floppy disc. This must be done with the FORMT utility. For CS/80 discs, SWTCH will not initialize subchannels other than LU 2 or LU 3. Bad areas on these subchannels may be spared with the FORMC utility.

Specify N (no) to deny additional subchannel initializations. Batch mode is implied.

The variables capable of being specified by the SWTCH turn-on parameters are: \TDLU, \TSUB, \TUNT (\TADR, \TUNT, \TUOL for CS/80 discs), AUTO, \SAVE, TYP6, and \SUBI. These are initialized to -1 to indicate an unspecified state. When a parameter is skipped or erroneous (to the extent that it should have been ASCII/numeric) its value remains at -1 so that the parameter value will be prompted for when the proper time comes. Note that once a MAC/ICD system has been determined, \TUNT is set to value of \TSUB.

The variable BATCH is initialized to -6. During turn-on parameter retrieval (PARS thru CP3), every time a valid parameter type is obtained, BATCH is incremented. So when BATCH is 0, SWTCH runs in an automatic, non-interactive mode. SWTCH then proceeds without intervention except if an error occurs when an operator response or decision is needed.

10.5 NAMING CONVENTIONS

Four naming conventions are used throughout this chapter:

HOST	=	System subchannel definition of the system under which SWTCH is executing.
DESTINATION	=	System subchannel definition of the new system as defined during generation.
TARGET	=	Temporary specification of disc channel, subchannel, and unit for use by SWTCH during the transfer.
SOURCE	=	Subchannel definition of that subchannel containing the system file.

There exists a set of similar variables used by SWTCH.

HCH	host system disc channel
HEQT	host system disc type
HSBCH	host system disc subchannel
HUNIT	host system subchannel unit (ICD/MAC)
HNHD	host system subchannel starting head # (ICD/MAC)
HNSU	host system subchannel # surfaces (ICD/MAC)
HFTR	host system subchannel first track
H#ST	host system subchannel #sectors/track
HTTY	host system operator console channel
PI	host system Privileged Interrupt Channel
TBG	host system TBG channel

\DCH	destination system disc channel
\DSUB	destination system disc subchannel
DEQT	destination system disc EQT type (31 or 32 octal)
\DUNT	destination system subchannel unit/address (ICD/MAC discs)
\DFTR	destination system subchannel first track
\DNTR	destination system subchannel number of tracks

```
\DSHD         destination system subchannel starting head # (ICD/MAC dis
\DNSU         destination system subchannel # surfaces (ICD/MAC discs)
\DNSP         destination system subchannel # spares (ICD/MAC discs)
 DTTY         destination system console channel
 DPI          destination Privileged Interrupt channel
 DTBG         destination TBG channel

 \TDLU         target system disc LU (for ICD/MAC discs) or target
               select code (for 7900 discs)
 \T32C         target system disc channel for (ICD/MAC) discs
 \TSUB         target system disc subchannel (7900)
 \TUNT         target system disc unit (ICD/MAC discs)
 \CADR         target CS/80 HP-IB address
 \CUNT         target CS/80 unit #
 \CVOL         target CS/80 volume #
 \CSB1        \
 \CSB2         }target CS/80 block address
 \CSB3        /
```

10.6 MAJOR PROCESSING BLOCKS

10.6.1 Output File Test

SWTCH attempts to open the FMP file that contains the generated system and prints the FMP error number if the open fails. Next SWTCH reads the first 7 records of the file and checks, by means of the VTOSO routine, to make sure that the file contains a valid RTE-6/VM system. SWTCH also computes a 256 word checksum for later use (SWTCH will verify that the operator did not accidentally remove the disc media containing the FMP file when given the opportunity to insert the target cartridge).

10.6.2 Segment Load

SWTCH tests the $DATC entry point (in Table Area I) to verify that the system software is Rev. 2001 or later. This is necessary because the EQT lock software and special drivers (DVR32 & DVA32) were not introduced until this time. At this point, the proper disc interface segment (SWSG1 for 7900 discs, SWSG2 for ICD/MAC discs, or SWSG3 for CS/80 discs) is loaded and remains in memory for the duration of SWTCH.

10.6.3 New System I/O Configuration

SWTCH displays all the select codes and driver types in the destination system based on the information contained in the header records. The subchannel organization of LU 2 is also displayed.

10.6.4 Target Disc Information

SWTCH provides a great deal of flexibility when installing the new system on a target disc drive. The 7900 disc interface segment has its own internal disc driver that turns off the interrupt system and bypasses the RTE I/O system. This is necessary so that privileged disc commands can be sent to the disc controller to initialize the disc; the RTE on-line driver does not have this capability (e.g. WRITE a track with the "protect" bit set). The 7900 segment requires only a target select code so that the driver may configure all of its I/O instructions to this select code. This select code does not have to exist in the host system since the transfer will be done without the aid of the operating system. Note that SWTCH does have to acquire a DMA channel for its own use; the \GDMA routine handles this and also sets up the port map for use by SWTCH.

If installing an ICD/MAC system, SWTCH uses SWSG2 to interface with the Disc Utility Library ($DSCLB), which uses the RTE on-line driver in a special mode. $DSCLB requires a target disc LU instead of a select code. The subchannel information for this target LU is never used; the LU is used only as tool for the library to determine which driver to call (DVA32 or DVR32). Note that the target disc address/unit is not derived from the LU specified, but is specified by the SWTCH user and does not have to be a valid address/unit in the host system. The $DSCLB routines perform special EXEC calls to the on-line driver to perform privileged disc operations that are not possible with standard user EXEC calls.

SWTCH locks the EQT of the target disc for the entire duration of the SWTCH. This gives the user an opportunity to remove a host system disc cartridge and replace it with a temporary target cartridge. The EQT lock is used to suspend all I/O to the host cartridge (e.g. edits, FMP operations) before removing it. The EQT must remain locked while the target cartridge is in the disc drive so that no host I/O will be accidentally performed on the target disc.

Another reason for the EQT lock is that SWTCH performs individual subroutine calls to $DSCLB to SEEK and WRITE, for example. The lock guarantees that no other I/O request will move the heads on the disc between the two subroutine calls.

Installation of CS/80 systems is similar to ICD/MAC systems. First the CS/80 disc EQT type is verified by checking the target disc LU. This LU is used only to determine the EQT type of the target disc. Next, all disc calls are made through SWSG3 to the CS/80 library $DTCLB. SWTCH uses these calls to $DTCLB to make special calls to the CS/80 driver DVM33. RTE is bypassed during the CS/80 system installation. The EQT for the target disc also is locked for the duration of SWTCH execution.

10.6.5 Target Cartridge Insertion

The operator is given a chance to insert a temporary target cartridge in the drive at OKAYY. Before the "NOW IS THE TIME.." message is issued, SWTCH does an EXEC call to lock itself into memory and then calls EQTRQ to lock the EQT. The memory lock and EQT lock must always be done in this order to prevent a deadlock situation where the system may try to swap out SWTCH after the EQT is locked (the XSIO request will be blocked by the lock). Next, SWTCH does a dummy I/O request to the target disc LU so that all pending I/O requests on the EQT are completed before the operator is told to remove the host disc media. (SWTCH's dummy I/O request will be linked behind all requests currently waiting on that EQT.)

10.6.6 Saving Target File Structure

If the user requests the filesave option, SWTCH calls VFYSY to determine the expected location of the directory track according to the destination subchannel definition for LU 2. If a valid directory track is found there, SWTCH allows the user to save all the files above (last track of new system + 9 (scratch tracks)). If some FMP tracks must be overlaid (because the new system is larger than the old), SWTCH warns the user "NEW SYSTEM WILL DESTROY SOME FMP FILES". If a valid directory is not found, the user is told that the information will be destroyed.

10.6.7 Subchannel Initialization Prompts

10.6.7.1 7900 Initialization

For 7900 discs, each subchannel whose number-of-tracks word is non-zero is prompted for initialization. If /E is entered, the initialization prompting is terminatd and transfer is to AUTO?. On YES responses, the TARGET PLATTER is requested. On all responses (even defaults), the target platter is checked against \TSUB, the target platter of the system subchannel. If it is not the same, that subchannel's entry in \TMT is updated as follows:

```
\TMT+SUBIA:     bit 15 is set to indicate initialization
 \TMT+SUBIA+16:  set to the target platter
```

10.6.7.2 ICD/MAC Initialization

For ICD/MAC discs, SUBI5 divides the subchannels into two groups, depending on whether or not their destination unit is the same as the destination unit of the system subchannel.

\TMT is scanned first for all those subchannels defined on \DUNT. Each subchannel is prompted for initialization by INIT?. INIT/ matches on \DUNT and always sets the target platter to \TUNT for YES responses to these subchannels.

All remaining subchannels (except 9895 floppy subchannels) are prompted for according to their defined address/unit, from 0 through 7 but not equal to \DUNT. If the TARGET UNIT? response to a group is not /E, a check is made to ensure that it does not equal \TUNT of the system subchannel. INIT? then prompts for the initialization of each individual subchannel in that group, matching on the current scan unit number and setting the target unit number to the TARGET UNIT? response just entered.

INIT? scans the ICD/MAC version of \TMT searching for a destination unit matching that in TEMP3 (parameter in A-Register on entry to INIT?). The system subchannel (\DSUB) is skipped, as are all subchannels with the number of tracks equal to 0 (those already marked for initialization). If the unit specified in the entry matches TEMP3, that subchannel is prompted. If /E is entered, INIT? returns to the caller and no more subchannels in this group are prompted for initialization. On a YES response, the subchannel's entry in \TMT is updated as follows:

\TMT+(subch#)*5+2: has its unit field replaced with the target unit specified in TEMP4 (B-Register on entry to INIT?).

\TMT+(subch#)*5+3: has bit 15 set to indicate initialization.

Subchannels (other than LU 2) on CS/80 devices are not initializaed by SWTCH. They can be initialized by the FORMC utility after the system is booted-up.

10.6.8 Auto Boot Option

SWTCH requires six conditions for autoboot and tests for these conditions by using the information obtained from the Header Records.

```
Destination disc chan    (\DCH) = Target Channel (\T32C or \TDLU(7900))
Destination addr/unit/subchannel= Target addr/unit/subchannel
Destination TBG channel         = Host TBG channel
Destination TTY channel         = Host TTY channel
Destination Priv. Int. Chan     = Host Priv. Int. Channel(if they exist)
Destination Disc type (ICD/MAC) = Target Disc type
```

The auto boot option is not available for CS/80 systems.

10.6.9 Overlay Conditions

If the host addr/unit/subch is the same as the target, SWTCH assumes that the host LU 2 will be overlaid and warns the user, "DISC IN HOST SYSTEM DRIVE WILL BE OVERLAID". A flag (OVLAY=1) is set for later reference. Note that this is a very general test. No attempt is made to determine whether the new system will overlay the area on the disc that contains the FMP file. The message should be adequate to alert the user that the disc is being overlaid. It is then the user's responsibility to make sure that the data has been saved, and that the FMP file containing the new system will not be written over during the SWTCH process.

10.6.10 File Purge

At this point, SWTCH has gathered all the needed information from the user, so the user is given one more opportunity to abort the process (if in the interactive mode). If the user is saving files, SWTCH must now purge all files to be overlaid. The first full track read is done at BFULL which uses the track size buffer. All code up to BFULL will be overlaid by data at this point. SWTCH now calls PURGT to purge all overlaid files on LU 2 and list them for the user. The same is done for type 6 files (if requested).

10.6.11 System Installation

At the XFER label, SWTCH prints "INSTALLING SUBCHANNEL XX", and prepares to install LU 2 for the new system. A full track is now read from the FMP file containing the system, and the checksum (the first 256 words) of the system is recomputed. This checksum is then compared with the previous value saved in CKSUM. This check will determine whether the operator has physically removed the disc media containing the system file by mistake. SWTCH now installs the system one track at a time by reading from the FMP file at RDISK and writing it out at WDISK, until the entire file has been written to the disc. The remainder of LU 2 is then initialized without the write protect bit set.

10.6.12 Subchannel Initialization

ILOOP is called for each subchannel that the user asked to initialize. For ICD/MAC discs, SPINT cleans up the spare pool by removing any "spare" or "protected" bits from the preamble, and flags all defective spares with the "defective" bit so that they will not be used. The data tracks of the subchannel are then initialized by writing zeroes and setting the address of defective tracks to reference spare tracks in the spare pool. Bad tracks are reported and spared to tracks in the pool as long as spares are available. Note that the 7900 disc controller is not capable of sparing, so defective tracks are merely reported to the user.

10.6.13 Auto Boot-Up

SWTCH is now ready to boot up the newly installed system, if requested. The autoboot section in SWTCH is designed to imitate the function of the disc ROM loader as closely as possible. The ROM is responsible for loading the boot extension (128 to 256 words) into memory at location 2011 (octal).

For 7900 autoboots, the actual bootup is done at NVRFY in SWSG1. Note that the S-register and DMA control word 1 are set up as the ROM loader would set them.

For ICD/MAC discs, the bootup is done in the main program at BOT32. The boot extension is read back from the new system subchannel into SWTCH's local buffer. Interrupts are then turned off so that the privileged code can be executed. The S-register and DMA are set up for the boot extension at BOT32+6. The boot extension is then moved from the local SWTCH buffer to location 2011 in physical memory. The base page fence is cleared, mapping is turned off, and SWTCH jumps through location 2055 as does the loader ROM.

10.6.14 Termination

If autoboot is not enabled, and the overlay conditions were met (OVLAY=1), SWTCH warns the user that the new system must be booted. The user is now given the opportunity to remove the temporary target cartridge (if used), and replace the host cartridge before control of the disc is given back to the host system. At this point, the EQT is unlocked (ICD/MAC and CS/80 switches), and all the host system I/O held off by the EQT lock is allowed to resume where it left off.

10.7 MAJOR SUBROUTINES

10.7.1 VFYSY

SWTCH uses DISKD to read from the target subchannel to verify or negate the existence of a cartridge director CD and file directory FD, which are the necessary conditions to assure that the target file structure can in fact be saved.

The cartridge directory CD is assumed if:

a. The LU word is a negative value greater than 63.

b. Words 1-3 of any entry are less than 0.

The file directory FD is verified if:

a. Word 0 and word 8 are less than 0.

b. Words 1-7 and 9-15 are greater than or equal to 0.

c. Word 6 is greater than or equal to word 5.

d. Word 7 - (word 8 + 1) = D.LT obtained from CD.

e. D.LT = last logical track according to system subchannel defined at generation time.

If a system cannot be verified to exist, in which case the file structure cannot be saved, the user is given the option to abort SWTCH or continue. If continuation is desired, the flags are set to disallow saving the files and purging type 6 files.

10.7.2 VTOSO

VTOSO verifies the existance of a track 0 sector 0 boot extension in record 3 (and 4 for ICD Systems) of the Type 1 system file. This is accomplished by computing a 5-word checksum and comparing this with a pre-computed constant for the 7900, ICD, MAC, and CS/80 boot extensions. The checksum is computed by XORing words 109-113 for MAC 7900, words 191-195 for ICD, and words 228-232 for CS/80.

Return to: (P+1) Not a valid boot extension
(P+2) Valid boot extension

OK? Asks the user "OK TO PROCEED?"; calls YE?NO to decipher the answer, and transfers to \XOUT on a N response, doing a simple return on a Y response.

YE?NO reads the operator's response, looking for a Y, N, or /E in the first 2 characters (first word).

returns to (P+1) if invalid response
(P+2) if /E
(P+3) if N
(P+4) if Y

TARGT reads a response and reissues the EXEC call in case of time-out at the console.

\DFLT checks for a single space followed by a carriage return as an operator response - this indicates the default value for whatever value SWTCH was trying to obtain, or else a signal to SWTCH that it is to proceed (as after the "NOW IS THE TIME TO INSERT CORRECT CARTRIDGE ...").

returns to: (P+1) if not a single space (P+2) if one single space

10.7.3 PARMP & SCAN

PARMP is the parameter parsing routine used for the SWTCH turn-on parameters. It is used also when the file name is entered in the interactive SWTCH mode. It accomplishes this by making repeated calls to SCAN, which in term calls NAMR to do the actual parsing.

10.7.4 PYN

PYN is called after PARMP has placed the parsed parameters into the 12-word buffer. On entry to PYN, the A-Register contains the parameter word while the B-Register contains the value of word 4 with the bit positions for the previous parameter rotated to bits 1 and 0. PYN checks only for parameters 5-8, whose values should be Y or N. If the next parameter position in word 4 indicates that the parameter was not specified, or if it indicates that it was not ASCII, or if the ASCII value wasn't Y or N, then PYN returns to (P+1), otherwise PYN returns to (P+2) with A-register = 0 for an N response, or 1 for a Y response. Note that the B-register, containing word 4, is maintained between successive calls to PYN.

10.7.5 PURGT

PURGT purges a file by setting word 0 of its file directory entry to -1 and size word 6 to 0, and displays that file name on the console. PURGT is called once it has already been determined that the entry pointed to by BPTR is to be purged.

When a file overlaid by the new system is purged (indicated by CURCH not equal to 0), that file name is entered in the list of files whose extent file directory entries (if any) are to be purged. This list is built from the top of core (or partition) downward, three words per entry with the values filled in upward. PENT is the word 0 address of the next entry and has an initial value of LWAM-3.

10.7.6 UPDAT

This routine updates the file directory pointers when looking for entries of either overlaid files or Type 6 files that are to be purged.

UPDAT depends on these temporary variables being set:

```
TEMP4 = # of directory tracks (from D.#).
TCNT  = # entries left to search on current disc track.
CURCH = -1 to purge extents of overlaid files
      = 0 when purging type 6 files which have no extents.
 BPTR = buffer address of next directory entry.
 REWRT = 0 current directory track has not been changed
       > 0 at least one entry of the current directory
         track has been changed, so track must be rewritten.
 #PF  = number of overlaid files (not Type 6 files) purged.
```

UPDAT has three returns:

(P+1) continue search on same directory track with B-Register containing the buffer address of next entry (The caller of UPDAT then stores it in BPTR.)

(P+2) the entire file directory has been searched and all updated tracks rewritten in the process; continue with next SWTCH step.

(P+3) start searching a new directory track with B-Register containing buffer address of next (actually first) entry to search, and A-Register containing the number of entries left on this track to search (TCNT is re-initialized with this value).

At label UPDTT, REWRT is checked to determine if the current track is to be rewritten to disc (because something was changed) before a new directory track is read in. If this was not the last directory track, the next one is read in and UPDAT returns to (P+3).

When purging overlaid files, before possibly rewriting the current directory track and moving on to the next, the directory is rescanned to purge any extents belonging to the overlaid files. For each entry in the PENT list (number = #PF) the entire directory is scanned for a matching file name. When a match is found, the extent entry is purged and REWRT is set (if not already done so). When each PENT extent on that directory track has been scanned, exit is to UPDTT, where processing is continued until the next directory track is completed.

\BLIN - sends a blank line to the console using \DSPL.

\DSPL - sends the message starting at the address in the B-Register, of length A-Register words to the lu of the operator console. Note the call to LOOP, which loops until the EQT5 status word indicates that the device is no longer busy, whereupon it returns.

LOOP - is not needed for all \DSPL calls. The need arises in those situations where an I/O call via SWTCH's own driver was made shortly after \DSPL sent a message. Because DISKD turns off the interrupt system while processing, the messages came out in chunks (often a character at a time) rather than one continuous stream.

10.8 SWSG1 ROUTINES FOR 7900 DISCS

10.8.1 Subroutine \STDO

This routine configures the I/O instructions in the 7900 driver, DISKO, to the target channel \TDLU. The data channel instructions specified by I/OTB are configured to \TDLU and the command channel instructions specified by I/OTC to \TDLU+1.

10.8.2 DISKD

Calls the correct disc interface segment, either \DSKO for a 7900 target system, \DSK5 for an ICD/MAC target system, or \CDSK for a CS/80 target system.

\GDMA is a 7900 routine that allocates DMA channel 2 for the 7900 driver. It is called once by the main program, and DMA channel 2 remains allocated for the duration of the SWTCH process. \GDMA checks if channel 2 is available (with the interrupt system on) and loops on this check until the DMA channel appears to be available. \GDMA then turns off interrupts and checks again to make sure that the system did not allocate it to someone else (in case SWTCH was interrupted). If this check succeeds, SWTCH stores 777 octal at INTBA+1 to tell the system that the channel is in use, and turns the interrupt system back on.

\RDMA is the 7900 routine that releases DMA channel 2 by storing 0 at INTBA+1 if and only if SWTCH's ownership key of 777 octal is stored there.

\DSKO - is the 7900 disc driver routine.

\DSKO references the current target disc subchannel specified by \TSUB and defined by \DFTR (first physical track) and \DNTR (number of tracks). The subchannel (i.e. platter) further determines the disc unit UN#IT and head

H#AD. These values are combined with the logical track and sector passed in \TRAK and \SECT, respectively, to form an absolute track and sector address. Immediately before initiating the transfer, \DSK0 turns off the interrupt system via a call to $LIBR. INTON turns the interrupts back on before \DSK0 returns.

The variable \INIT is set differently, depending on whether SWTCH is:

a. doing a normal read or write.

b. doing a write initialize.

c. doing a write initialize, with write protect on (i.e., operating system code).

d. flagging a track defective from the error recovery routine.

This setting helps \DSK0 to figure out what it is to do in error situations because it knows approximately where it was in the SWTCH transfer. Check each use of \INIT because it depends on the situation or error encountered.

ERRCH deciphers the error status in the A-Register and branches to the appropriate error recovery code. For write protect and not ready errors a message is sent and a halt is done, waiting for a restart of the disc I/O operation at RTRY. For defective cylinders, DISBM determines whether a bad track can even be flagged defective. For SEEK errors 10 tries are made before further error recovery is attempted.

For irrecoverable errors, the driver jumps to SWTCH's abort exit \XOUT, after issuing the proper message containing the guilty track and subchannel numbers.

\BOOT is checked before a normal return to see if the TOSO boot extension has been read in and ready to be entered. In addition, the base page fence register must be cleared by doing a DJP to disable the user map. In order for the configurator program to access the disc when starting up the new system, it retrieves the disc select code from bits 11-6 of the switch register. Since this is done by the ROM loader during normal boot-up, SWTCH sets this value for auto-boot. Also note that SWTCH set up DMA control word 1 with the select code of the 7900 disc. This is necessary because the boot extension uses DMA channel 1, and assumes it is set by the ROM.

INIER (SWSG1) is entered after ten tries have been made to initialize a 7900 disc track. This routine is not branched to when a disc error occurs. The write protect bit in \INIT is on because that indicates that a system track is being written on defective tracks, which is not allowed.

If the seek check or end-of-cylinder bits are set in the status word, INIER assumes an invalid subchannel definition, sends the message, and aborts SWTCH through \XOUT.

\DSK0 is used to flag a track defective. On return, INIER reports the defective track, enters it in the bad track table (FLGTR) if \LU2 indicates the system subchannel, and does a JMP \DSK0,I.

10.9 SWSG2 ROUTINES

SWSG2 is the segment that interfaces to all ICD/MAC discs (7905/06(H)/10H/20(H)/25(H)). It performs primitive disc operations (e.g. SEEK, FILEMASK, READ, WRITE), by calling subroutines in the Disc Utility Library, $DSCLB. This library, in turn, calls the appropriate driver (DVA32 or DVR32) with a special EXEC call. The driver then operates in a passive mode, where the command buffer is sent to the disc controller without any error checking.

\DSK5 is the primary subroutine in SWSG2 that handles all I/O calls to ICD/MAC discs for SWTCH. It has four modes of operation for reads and writes, where the \MODE parameter selects which type of operation is to be performed:

\MODE=1 - for standard reads/writes to the disc, in order to update the directory track on the target disc, for example.

\MODE=2 - for initializing and writing system tracks to the disc with the protect bit set in the preamble, doing sparing as needed.

\MODE=3 - for initializing (and writing zeroes to) non-system tracks. The "protect bit" is not set, but sparing is done for all tracks found to be defective. (MODE 3 is also used for the 7910H system tracks since there is no format switch on the 7910H, and thus tracks may not be "protected", if they are to be written later.)

\MODE=4 - Initializes and writes zeroes to tracks in the spare pool. Clears "spare" or "protected" bits and sets the "defective" bit if a bad spare track is found. \DSK5 is called in \MODE 4 for all tracks in the spare pool before initializing or writing any of the tracks in the data portion of the subchannnel.

TBL02 is the status word jump table. The value of status word 1 from the disc controller determines the processing block in the table to be branched to. ENDBR is the common return point for all processing blocks in the branch table. The processing blocks determine the proper action to take based on the current state variables \MODE and PHASE.

Note that a single request to \DSK5 causes the code section from DSGO to ENDBR to be executed repeatedly until the value of \RET indicates that the operation is complete and we should return from \DSK5. Each iteration through this loop will update PHASE to indicate the current state of the operation. PHASE may have the following values:

PHASE =1 - Read Full Sector to get the track's status from preamble.

=2 - Write Initialize to the track, rewriting track & sector addresses.

=3 - Read Full Sector to get a spare's status.

=4 - Write Initialize to a spare to point it at the defective track.

=5 - Write Initialize to a data track to point it to a spare.

=6 - Write Initialize to a spare, flagging it defective.

10.10 BRANCH TABLE PROCESSING BLOCKS

ENDOK is branched to when the last command to the disc controller succeeds as expected (stat code=0). The next phase is entered.

FAULT catches all unexpected error codes (1, 2, 3, 4, 5, 6, 12, 13, 15, 24, 25). It sends a "DEFECTIVE CYLINDER ..." message and sets \RET to abort the current \DSK5 request.

RECAL is branched to on a cylinder miscompare (stat code=7). It issues a RECALIBRATE command to the disc and allows the operation to be retried up to ten times before aborting the current request. Since the 7910H does not support RECALIBRATE, a seek to cylinder 0 is done instead.

DSKER is branched to on stat codes 10 (uncorrectable data error), 16 (overrun) and 17 (possibly correctable data error). The operation is retried up to ten times before aborting the current request.

DEFTR is branched to when a track is found to be defective (stat code 21) or DSKER has retried the operation ten times without success.

EOCYL is branched to on stat codes 11 (Head/sector miscompare) and 14 (End of cylinder). The "INVALID DISC SPECIFICATIONS.." message is sent, and \RET is set to abort the current operation.

ILSPR is branched to on stat code 20 (Illegal Access to spare). If a standard read/write (\MODE=1), the operation is aborted, because a direct reference to a spare track is not permissible. When initializing tracks, this status may occur when seeking to a spare track to clear the "spare" and "protected" bits, and is normal for MODE 2 or 3.

ST2ER is branched to on stat codes 22 (Access not ready), 23 (status 2 error), and 26 (Illegal write). The following conditions are checked by

ST2ER:

1. Seek check-send "INVALID DISC SPECIFICATIONS.."

2. Disc not ready-send "READY DISC.." message and retry.

3. Format switch off-send "TURN ON FORMAT SWITCH..", and retry.

4. Protect switch on-send "TURN OFF DISC PROTECT..", and retry.

5. Unknown status 2 error-retry the operation up to ten times.

UWAIT is invoked when the disc is not available (multi-CPU environment). The operation is retried ten times before "INVALID DISC SPECIFICATIONS.." is sent.

10.11 MAJOR SWSG2 SUBROUTINES

XFER provides the interface between SWSG2 and the Disc Utility Library, $DSCLB. \DSK5 uses XFER to carry out all the primitive disc operations by setting up the action parameter, \ACTN and calling XFER. Each bit in \ACTN is associated with a disc primitive routine, so XFER calls the corresponding library routine when a bit is set.

The \ACTN word is interpreted as follows:

BIT	MEANING
0	CALL XSEEK TO ISSUE A SEEK COMMAND TO CONTROLLER.
1	CALL XADRC TO ISSUE AN ADDRESS RECORD COMMAND TO CONTROLLER.
2	CALL XFMSK TO ISSUE A FILE MASK COMMAND TO CONTROLLER.
3	CALL XDRED TO ISSUE A READ COMMAND TO CONTROLLER.
4	CALL XRDFS TO ISSUE A READ FULL SECTOR COMMAND TO CONTROLLER.
5	CALL STFIX TO ADJUST THE STAT CODE AFTER A READ FULL SECTOR.
6	CALL XDWRT TO ISSUE A WRITE COMMAND TO CONTROLLER.
7	CALL XINIT TO ISSUE A WRITE INITIALIZE COMMAND TO CONTROLLER.
8	CALL XVRFY TO ISSUE A VERIFY COMMAND TO CONTROLLER.
9	CALL XEND TO ISSUE AN END COMMAND TO CONTROLLER.

10-15 UNUSED

XFER begins with bit 0 and continues through bit 15, executing every primitive whose action bit is set. For example an \ACTN value of 1017B would be used to perform a standard disc read operation. In this case, the disc controller would receive the SEEK,ADDRESS RECORD, FILE MASK, READ, and END commands (in that order). XFER checks status after each primitive is sent to the controller, and immediately returns if an abnormal status is detected so that the appropriate action can be taken.

CKST1 analyzes the disc status returned from the controller. It distinguishes a power fail/timeout condition from an abnormal status condition and returns as follows:

Return to (P+1)-Power fail or timeout-restart operation.

(P+2)-Abnormal status-return with the stat code.

(P+3)-Normal status=0-return and continue.

NIXSP finds the next available spare track from the spare pool for the current subchannel. The physical (Cylinder and Head) address of the spare is computed and set in CYL# and HEAD#. The return is as follows:

Return to (P+1)-if out of spares for this subchannel.

(P+2)-if next spare found.

FMTR? is called when SWTCH is attempting to save files on LU 2, and \DSK5 is in Mode 4 cleaning up the spare pool. Normally, all tracks in the spare pool would have their "spare" and "protected" bits removed and be made available for use. When files are being saved, however, FMTR? must check to see if the current track is a spare being used by an FMP file in the save area. This is determined by doing a READ FULL SECTOR to get the address of the data track currently using the spare. If this address is in the LU 2 FMP area being preserved, the spare track is not reclaimed for use, but left intact with all the original data, and the "spare" bit set. FMTR? determines whether the spare is currently being used by an FMP file by examining the following conditions. If condition 1 AND either condition 2,3, or 4 is met, the spare track is not reclaimed for the spare pool, but left intact.

1. The defective track's head# is within the head# range of the system subchannnel (\DSHD to \DSHD+\DNSU).

2. First FMP cylinder < defective cylinder < last FMP cylinder.

3. Defective cylinder = First FMP cylinder AND defective track head# >= head# of first FMP track.

4. Defective cylinder = First FMP cylinder AND defective track head# <=

head# of last FMP track.

\SETD determines the subchannel specification for the current subchannel (\DSUB), and sets the appropriate values in \D#ST, \D#WT, \DFTR, \DUNT, \DHSD, \DNSU, \DNTR, and \DNSP.

DADTR translates a logical track number on the current subchannel into a physical disc address (Cylinder and Head number) and stores the result in CYL# and HEAD#. The translation is based on the current subchannel definition set by \SETD.

CVLOG is the inverse of DADTR and translates a physical disc address into a logical track number using the subchannel definition of the current subchannel.

RPORT is the bad track and spare track reporting routine. It sends the bad track header message for the first bad track on each subchannel, and reports BAD TRACK, BAD SPARE, or, SPARED TO and the logical and physical address of the track concerned.

ESUB sets the current subchannel number in error messages.

10.12 SWSG3 OVERVIEW

The SWTCH segement SWSG3 is used to install a CS/80 disc-based RTE-6/VM system. It is organized as a set of subroutines to interface to the CS/80 disc library, $DTCLB setup parameters, and detect errors.

There are several entry points in the segment, each used at different times in the switch. The first is START, which calls subroutine \CS80 to determine the characteristics of the system subchannel. The second entry point is \CSET, which sets the appropriate values in the ICOMP array (used by the disc library to address the target drive). The third entry point is \CDSK, the main entry point. This entry provides access to the read and write track subroutines based on the value of the \MODE switch. This entry point is called when SWTCH begins installation of the system. The main repeatedly calls this entry point to install the system on LU 2. This routine returns to the main a status code in the A-register. This status code is used by the main to determine if the operation (read,write) was successful (A=0), incurred a fatal error (A<0), or a retry operation must be performed (\CSRT<>0). The retry operation is performed when a sparing operation in the segment uses the buffer \CSBF in the sparing operation. In that case the information that was in the buffer is destroyed. The information from the Generator absolute file must be reread and the operation repeated.

10.13 SWSG3 INTERNAL SUBROUTINES

\CSET - Setup ICOMP values

This routine sets the appriopriate ICOMP values in the ICOMP array for later calls to the disc library. It is called after the target disc address, unit, and volume information is known.

```
ICOMP
Word      Meaning
  1       Unit number
  2       Volume
  3       Address mode, set to block addressing
  4       High     \
  5       middle    } block address words, initially set to \CSB1-3
  6       low      /
  10       Set length flag, set to show new length
```

Note that words 5 and 6 must be set to the block address being referenced by the particular call. For now the high word of the block address is assumed to be zero. Words 11 and 12 which indicate the transfer length must also be set by the user before calling this segment to do disc I/O. This is done by LG2PH.

\CDSK - Main entry point for READS and WRITES

This is the main module of the CS/80 disc interface segment. It handles calls to do reads, writes, and initializations of disc tracks. Note that this subroutine performs operations in terms of logical tracks. The logical track address is converted to a physical block address before a call is made to the disc library.

The physical drive for the transaction is determined by the current address (\CADR), unit (\CUNT), and volume (\CVOL). When this segment is first loaded, the front end code sets up these parameters for subsequent disc transactions. At the same time the block address of the system subchannel (\CSB1, \CSB2, \CSB3) are set.

To do a read/write operation, several parameters must be specified: \MODE, \TDLU, and BUFR (\CSBF). \MODE is described below. \TDLU indicates the target disc LU number and BUFR is the data buffer for track reads and writes.

\TRAK - Beginning logical track address.

\SECT - Beginning logical 64 word sector address. This value must be even.

NOTE

Both \TRAK and \SECT will be converted to a block displacement. This displacement is then added to the starting block address (\CSB1-3) to determine the effective block address for the transaction. This conversion is done with LG2PH, which leaves the effective block address in the ICOMP array.

\LNTH - Number of words to be read/written.

\MODE - Determines what operation is to be performed.

1 - read
2 - write
3 - initialize

Note that for initializations the length and sector addresses are ignored, the whole track is initalized. This is done by calling the write routine to copy the contents of \CSBF to the disc. This buffer is zeroed during initialization. After doing this write the error routine will spare any bad blocks.

On return the A-register contains any error. A value of zero means successful completion. Negative errors are always fatal.

LG2PH - Logical to physical address translation

This subroutine converts a logical track and sector address to a physical block address. It first converts the track (\TRAK) and sector (\SECT) address into a block displacement. This displacement is then added to the starting block address of the subchannel. The resultant block address is then placed in the ICOMP array.

Note that only doubleword arithmetic is performed on the block addresses using the middle and low words. The high word is assumed to be zero

The sector address is always assumed to be zero.

ERROR - Error checker and processor

This subroutine processes errors returned by the disc library. It prints any error messages and vectors the return based on action that should be taken by the calling subroutine.

Calling sequence:

```
<<CALL TO DISC LIB.>>
JSB ERROR
```

-no error-
-fatal error-
-retry transaction-

The first item checked by this subroutine is the QSTAT value returned by the library. If this value is zero then the transaction was successful and a no-error return is indicated.

However if this value is non-zero then some abnormal conditions have resulted from the disc library call. The full status is returned in words 1 through 10 of the buffer \CSBF. Check this status by first looking at word one of the buffer. If it is non-zero then there was a problem with the full status. A fatal return is indicated after issuing a QSTAT error, which means if you can't even get the status back then you are in trouble.

If the status checks out, then the appropriate action is based on what the error was. The full status in words 1 through 10 of \CSBF is scanned. There are four types of errors, reject, fault, access, and information. Note that reject errors have precedence over fault, which have precedence over access etc. These are scanned in the following order, reject, fault, access, and then information.

Within each class of error there is a priority ordering. The types of errors within each class and their meaning are:

TYPE	MEANING
Fatal	SWTCH cannot recover from this error. Most probable cause is a hadware problem.
Internal error	SWTCH has detected a condition which should never happen, such an error bit being set for a non-existent error. Other reasons may be programming error, hardware faults etc.
Spare after running ERT	Run the Error Rate Test and spare the block(s) in question if neccessary.
Retry	Try the operation again. If it fails on retry then a fatal error has occurred.
Release and retry	Issue a release to the device and retry the operation. If not successful then a fatal error has occurred.
Prompt and line, etc.	Usually caused when the disc is found to be off- retry Issue a information prompt to allow them to correct whatever the error was then retry the operation.
Ignore	Message should be ignored.

When looking at the one word field that defines a class of errors, such as the reject field(word 2 of \CSBF), we always check for errors in the above order and take appriopriate action by branching to a section of code that handles one of the above errors.

On return the A-register is negative if the error was fatal.

10.14 E.FTL - FATAL ERROR PROCESSOR

This subroutine is the fatal error processor. It prints an error corresponding to the type in A-register and the class in B-register. Then the fatal error return is taken with the A-register set to minus one.

10.15 E.INT - INTERNAL ERROR PROCESSOR

This subroutine handles internal errors that are always fatal. This section of code is always entered with a JSB E.INT so that the address where the fatal error occurred can be determined by printing the return address on the console. This is similiar to the way the generator handles GEN ERR 00. After printing a general error message announcing a fatal error and the address, the reason for the internal error as passed in A- and B-register is printed. Next the full status is listed. Note that the parameter field of the full status is meaningful only on error bits 17, 24, 38, 41, 58, and 59. Refer to the CS/80 Instruction Set Programming Manual for more details. Next a fatal error return is indicated with A-register equal to -1.

10.16 E.RTY - RETRY ERROR PROCESSOR

Certain errors can be cleared up if the transaction is repeated. If such a transient error occurs, the transaction is checked to make sure it is not a repetition. If it is, then the error is fatal and the transaction aborted. Transients are only given one chance to clear up. The retry flag is set to allow the transaction to be repeated.

10.17 E.RLR - RELEASE AND RETRY

This subroutine processes release and retry errors. A release is issued first. If not successful, a fatal error is indicated. If the release was successful, the transaction is retried. Errors are fatal on a retry.

10.18 E.PMR - PROMPT AND RETRY TRANSACTION ERROR PROCESSOR

Some errors need operator corrective action before SWTCH can proceed. This subroutine prompts the user with an action message and waits for a space which indicates that the action has been performed. Then the operation is retried.

10.19 E.ERT - ERROR RATE TEST AND BLOCK SPARING

This subroutine performs an Error Rate Test and spares any defective blocks it may find.

When this subroutine is entered, a dummy ICOMP array is created for use by the sparing routine in calling the disc library (DCOMP). The old ICOMP array cannot be used because the operation causing the need for sparing may be retried. Next a call is made to XXSPR with the sparing mode (NORETDATA=0) set to retain any data found. If there are no errors, then the operation is retried. Errors are processed as described below.

ERROR	MEANING
-2	Could not find any blocks that needed to be spared. Return and retry the operation.
-5	Ran out of spares. Print a message to that effect and then take the fatal return.
-1	Time out error. Assume that the drive is not ready and issue an error message asking the user to put the drive on-line. Then retry the operation.
2	Power on state. Signal a power fail and take the fatal return.

NOTE: With CS/80 discs, a physical track may contain more than one logical track. For example a system subchannel may be in the middle of the disc preceded by another sub- channel containing data the user wants to preserve. It is possible then that a physical track may contain two logical tracks, one with a cartridge directory at the end of a subchannel and the beginning of a system subchannel on which the system is being installed.

```
Physical   |<------------------------------------------------>|

Logical   ------>|<----------------------------->|<-------------
                          System Subchannel
```

Now a bad block is detected while installing the system. The XXSPR routine will run the ERT on the entire physical track, including the directory portion of the preceding subchannel. There is no need for concern because the data on the physical track is saved in the buffer IBUF1. If a bad block is detected in an area belonging to a subchannel which is not part of the system subchannel, then the user must be informed.

-3 Could not spare the block specified because there was unrecoverable data. The sparing mode set to make the best attempt at saving data, but to destroy information if necessary. Since we are inside the system subchannel this is not the most serious case. Note that XXSPR will still print a message telling the user that a block was spared.

-4 This is the serious case, XXSPR found data other than the block specified that could not be saved. This is where data outside the system subchannel may be destroyed because more than one logical track can occupy a physical track. There are two cases to be tested:

a) Block address returned by XXPSR is outside the system subchannel. In this case, the block is destroyed and the user notified that data was destroyed outside the system subchannel. Remember that the block spared is still bad, a good block is never spared !

b) Block address is inside system subchannel. In this case, the sparing mode is set and the best attempt is made at recovering the data. No warning message will be issued.

x Any other error message is treated as a disc error.

ERPRT - Error Print

This subroutine prints the error messages in the A- and B-registers. The calling sequence is;

```
CLE or CCE     e = 0/1 = skip line/ no skip
LDA errortype
LDB errorclass
JSB ERPRT
-return-
```

```
errorclass = 0 for general messages
             1 for reject error messages
             2 for fault   "     "
             3 for access  "     "
             4 for information  "
```

```
errortype = the bit corresponding to the error in
            the above class. For example a UNIT FAULT error
            would have bit 1 of the A register set and B
            would be 2 since UNIT FAULT is a Fault error.
```

The message data structure is arranged in two levels. The first level is a set of tables containing pointers to the error messages. The second level consists of the actual error strings. Each entry in the first level corresponds to an error flag returned by the disc library. When an error code is passed to this subroutine, the error class determines which table in the first level (set of pointers) is to be used. Once this is determined, an index of the error type value is established in the tabler. This provides a pointer to the error length and string corresponding to the status returned by the disc library.

SESSION MONITOR ACCOUNT PROGRAM	CHAPTER 11

11.1 INTRODUCTION

The Account Program Technical Specifications are intended to serve as an aid to the programmer when modifying the RTE Session Monitor account program. This document should be used in conjunction with the account program source listings, as it does not attempt to provide the level of detail given by the code itself. It is assumed that the reader is familiar with the operation of the account program as described in the RTE-6/VM System Manager's Reference Manual.

The Session Monitor Account Program is provided for the System Manager's use in initializing and maintaining the account file. It allows the System Manager to create accounts for new users or groups of users, to remove existing accounts, to modify specific user or group attributes, or to perform particular account file utility functions. The account program (ACCTS) is a background, segmented program. ACCTS accepts input either interactively or from a disc file. These command inputs instruct ACCTS to perform specific operations on the account file. The account file itself is a type 1 FMP file.

11.2 ACCOUNT FILE STRUCTURE

This section describes the structure of the account file. The first record is the account file header, containing specific global Session Monitor information.

Account File Header

Location pointers — the first six words in the headers are pointers to the beginning of each part of the account file.

WORD	
1.	Active Session Table
2.	Configuration Table
3.	Disc Pool
4.	User/group ID Map

5. Account Directory
6. Account Entries

System Message File the name (namr) of the message file which is first listed to the user's terminal when the user logs on. The file name may be changed with the ALTER,ACCT command.

Prompt String — the prompt is a 0-20 character string which is output to a terminal when any key is entered on session terminal which is not currently active. The default string is PLEASE LOG-ON: the first word is number of characters in the string.

Lowest Private ID used — the lowest number (1-4095) which has been assigned as a private ID by the Session Monitor. This number is always greater than the highest group ID and is initially set to 4096.

Highest Group ID Used — the highest number (1-4095) which has been assigned as a group ID by the Session Monitor. This number is initially set to 1 and is always less than the lowest private ID used.

Resource Number — Used for cooperative updating of the account file by ACCTS and the log-on and log-off routines.

LU of MSG Files — this is the logical unit of the disc on which all message files will be stored.

Memory Allocation — this is the number of words of SAM to be allocated to session monitor. If negative it is minus the number words to allocate and the number is computed at boot up based on the session limit.

Session Limit — the maximum number of users allowed to be logged-on at any given time. Its current value is checked before LOGON allows a user to log on.

Active Sessions — the current number of active sessions.

Shutdown Flag — this is a flag to tell ACCTS whether the Session system is shut down or the Accounts file is to be purged or an individual account needs to be purged.

- 1 Account entry to be purged
- 0 Session system active
- -1 Session shut down
- -2 Accounts file to be purged
- -3 Session shut down momentarily
- -4 Session shut down and session memory released

Session Limit — this is a copy of session limit after a shut down. When start up is entered, this is copied back into the session limit above.

Class Number - this is a class number used by accounts to send messages via the TELL command.

Length of Configuration Table - the length of the configuration table is kept here for easy expansion of accounts file.

Resource Number - this resource number is used so that various copies of ACCTS can update file.

Disc Pool Length - this is the length actually returned by $SALC. It is kept so that the entire block can be returned.

Active Session Table

The second record contains 4-word Active Session Blocks, one per active session (i.e., one per user currently logged-on).

Logical Unit - the logical unit of the terminal at which the user is logged on.

Log-on Time - a doubleword value indicating the time at which the user logged on. This value is used at log-off to update the connect time clocks.

Directory Entry # - the directory entry # (entry offset) of the directory entry for this account.

11.2.1 Configuration Table

The Configuration Table follows the Active Session Table, which describes the default logical units to be used for specific device logical units. Each station logical unit in the Configuration Table has associated with it a set of device logical units which are assigned default logical units to be used when a user logs on at this station. The default logical unit associated with the station itself is always 1. At log on, these default values are written into the user's Session Control Block (SCB), unless overridden by entries in this particular user's Session Switch Table (SST).

The first word in the Configuration Table is the length word (the number of devices to which default logical units will be assigned and associated with this station, plus 1). Following this word is the first station logical unit with which default LU's will be associated. Next are the one word entries for each system logical unit and its associated session (default) logical unit. Following is the logical unit of the next station with which default logical units will be associated. The entire table is terminated with a zero.

Example: Associated with station LU30 are to be the left and right cartridge tape units (CTU's) which can always be accessed by a session user at this terminal as LU4 and LU5, respectively. Associated with station LU40 are to be its left and right CTU's which are also to be accessed by session users at this sttion as LU4 and LU5. Also associated with station LU40 is to be a dedicated line printer (actually LU57), to be accessed by session users at this station as LU6. The Configuration Table would look as follows:

```
+-------------+
|      3      |     length of entry
|-------------|
|  30  |      |     station LU
|-------------|
|  34  |  4   |     default left CTU (LU34) to LU4
|-------------|
|  35  |  5   |     default right CTU (LU35) to LU5
|-------------|
|      4      |     length of entry
|-------------|
|  40  |      |     station LU
|-------------|
|  44  |  4   |     default left CTU (LU44) to LU4
|-------------|
|  45  |  5   |     default right CTU (LU45) to LU5
|-------------|
|  57  |  6   |     default printer (LU57) to LU6
|-------------|
|      0      |     end of table
+-------------+
```

11.2.2 Disc Allocation Pool

The Disc Allocation Pool is a list of disc logical units (LU's) to be allocated to session and non-session users. The record contains one disc LU per word, with zeroes filling out the unused words of the 128-word record. Figure 11-4 shows the structure of this record.

Non-zero entries in the disc allocation pool are written to system available memory during initialization of the Session Monitor. The first entry in the memory-resident disc pool is pointed to by $DSCS.

11.2.3 User/Group ID Map

The ID Map is 256 words long containing a map of the 4096 User or Group ID numbers. Word 1 contains the bits identifying ID numbers 1-16, word 2 is used for ID numbers 17-32, etc. A bit set indicates that the associated ID number is currently assigned to an account. Thus word 1, bit 15, if set, indicates that ID number 16 has been assigned, and word 2, bit 0, if set, indicates that ID number 17 has already been assigned. Words 6 and 7 in the account file header can be used to determine if the ID is a user ID or a group ID. The user/group ID map is used when allocating ID numbers (at account setup) and when releasing ID numbers (at account purge).

11.2.4 Account file Directory

Each directory entry is a 16-word record containing the ASCII name of the user or group and the relative address of the record containing this user's or this group's account entry. The length of the directory is determined by the response, at account setup, to the prompts:

```
NUMBER OF USER ACCOUNTS?
NUMBER OF GROUP ACCOUNTS?
```

The directory size is computed assuming 20% of User accounts will require more than 64 words. The final result is rounded up to the next block.

Word 1 of each directory entry indicates whether the directory entry is for a user account or a group account. If word 1 is -1, this directory entry is free (resulting from the purging of a user or group account). A zero in word 1 indicates the end of the directory. If word 1 is -2, this indicates that a block is reserved for the second half of a user account.

11.2.5 User Account Entries

Following the directory in the Account File are the user and group account entries, interspersed, with one record per account entry. A user account entry is distinguished from a group account entry by the first word of the record. If negative or zero the record contains a user account and the value is the negative of the number of characters in the user's password. If the first word of the record is positive, the record contains a group account entry and the value is the group ID.

The following is a brief description of the fields in a user account entry:

Contents	Comments
# CHARS IN PASSWORD	(If bit 15 is set, SST extends into second block.)
PASSWORD	0-10 ASCII CHARACTERS, CANNOT BE "".
USER HELLO FILE	Namr of file to be transferred to upon log-on.
USER MESSAGE FILE	Namr of file to which mail is sent. Generated by the SM command processor.
CAPABILITY	Integer, 1-63 (63=most capable).
LAST LOG-OFF TIME	2-word entry.
CUMULATIVE TIME	Cumulative connect time in minutes (1-word entry).
CPU USAGE	Cumulative CPU usage in seconds (1-word entry).
USER ID	Integer, 1-4095.
GROUP ID	Integer, 1-4095.
DISC LIMIT	Integer.
GROUP SST & SPARE	Length of group SST in upper byte and number of spare SST entries in lower byte.
SST LENGTH	SST length in words, including spares. If positive, group SST is also to be mapped to user SCB.
SST ENTRIES	Session LU (less one) in bits 0-7, System LU (less one) in bits 8-15. (If SST entends into second block word 64 contains record number of extensions.)

11.2.6 Group Account Entries

The following is a description of the fields in a group account entry:

Contents	Comments
GROUP ID	Integer, 1-4095.
CUMULATIVE TIME	2-word entry.
CUMULATIVE CPU USAGE	2-word entry.
-GROUP SST LENGTH	Negative of SST length in words.
SST ENTRIES	Session LU (less one) in bits 0-7, System LU (less one) in bits 8-15.

11.3 MEMORY COMMUNICATION

Several variables are in memory for communication between modules and to prevent the loss of system resources when a program is aborted or the accounts file is purged. The following describes these variables:

1. $DSCS

$DSCS	$DSCS + 1		
	>=0	-1	-2
>0 (disc pool) 0 no (disc pool)	Session ready	No new sessions allowed but allow current sessions to log off.	No logging on or off allowed. Close +@CCT! and terminate.
-1	Session not Initialized (boot up)		
-2	Session not initialized but RN's and Class #'s allocated.		

2. $ACFL — LU on which accounts file is located.

3. $SPCR — LU for spool control files.

4. $LMES — Location of prompt string.

5. $CES — System session console disable.

6. $LGOF,$LGON,$STH — Class numbers for mail box communication of sessio system.

7. $SMEM, $SMEM+1 — Description of memory allocated to session system.

11.4 INITIALIZATION

The initialization module is entered whenever the program ACCTS is run. If the account system has already been initialized, ACCTS exits the initialization module and enters into its command processing module. If not initialized, ACCTS enters into a dialog with the user (if run interactively) or terminates if run non-interactively.

1. If ACCTS scheduled to clean up call DTACH.
2. GET RUN STRING and parameters and set up Transfer Stack.
3. If account file exists go to 16.
4. Prompt to load or initialize. If load go to 15.
5. If /A terminate.
6. Prompt for disc LU, session limit, memory allocation number of user and group accounts.
7. Create accounts file large enough to accommodate all accounts.
8. Prompt for prompt string.
9. Define configuration table.
10. Define disc pool.
11. Initialize Account Directory.
12. Prompt for MANAGER.SYS password.
13. Create Accounts
    SYS
    SUPPORT
    GENERAL
    MANAGER.SYS
    ENGINEER.SUPPORT
14. Go to 16.
15. Load accounts file.
16. If memory allocated go to 18.
17. Allocate memory and set up disc pool: call $BALC, $RTRN, and $SALC.
18. If first run parameter =-1, go to 26.
19. If under session, go to 22.
20. Prompt and verify system manager's password.
21. Set ID to 7777B (System managers), go to 23.
22. Get Users ID and capability.

23. Set ID and capability for command processors.

24. CALL COMMAND processor.

25. If not terminated, go to 24.

26. If clean up required call ACACP to clean up.

27. Call ACTRM to terminate.

LOG-ON Interface

The following steps are performed during any log-on sequence which is attempted before the Session Monitor has been initialized.

1. If account file is not found, report error (user can then either mount the cartridge containing the account file if one exists, or run ACCTS to create one).

2. If Session Monitor is not initialized ($DSCS=0) then schedule ACCTS to perform initialization.

11.5 COMMAND PROCESSING

Once the account system has been initialized, the ACCTS program enters a command processing loop which does the following:

a. Prompts for a command NEXT?

b. Retrieves and parses the command name and parameters.

c. Verify that user has capability to execute command.

d. Transfers control to the appropriate command subroutine.

11.6 ALTER,ACCT

Sequence of Operations

1. Prompt for Session limit and validate.

2. Prompt for memory allocation.

3. Prompt for new prompt string.

4. Prompt for message file name.
5. UPDATE accounts file header.
6. Put prompt string in memory.
7. Read Disc Pool.
8. Prompt for additions to disc pool.
9. Prompt for deletions to disc pool.
10. Update disc pool in file.
11. Update disc pool in memory.
12. Prompt to ADd, MOdify or DElete, or NO change.
13. IF not "NO" change, go to 23.
14. Prompt for station LU.
15. If DElete remove entry and go to 12.
16. Search for entry and remove.
17. If Add and found, report error.
18. If modify and not found, report error.
19. Prompt for SST definition.
20. If not /E or /A, go to 19.
21. If /A, go to 14.
22. Add station to table, go to 12.
23. Post station table and return.

ALTER,GROUP
ALTER,USER

11.6.1 Sequence of Operations

1. Parse user.group name.
2. If ALTER,GROUP make user name = 0.

3. Save account name in IU and IG.
4. Find account using ACFDA, If not found report error and return.
5. If not MANAGER.SYS and not his group or group specified was @, report error and return.
6. If ALTER,USER, go to 12.
7. If group not GENERAL or @, prompt for new group name.
8. Prompt for SST.
9. Update group account(s) and directory.
10. Set user parameters to no change.
11. Set user and IU name to @ and go to 20.
12. If user or group = @, go to 15.
13. Prompt for new user name.
14. Prompt to use new group.
15. Prompt to use group SST.
16. Prompt for password, hello file, capability, disc cartridges.
17. Prompt SST definition and SST spares.
18. Prompt to relink to existing account, if no, go to 20.
19. Find account and verify password, if no verify, go to 18.
20. Get group account.
21. Get user first account in group.
22. If not found, go to 27.
23. Update directory.
24. Update account and merge group SST.
25. If IU is @ get next entry in group and go to 23.
26. If IG is @ get next group.
27. If found, go to 21.

28. Return.

NOTE: Change of name and linking are bypassed if either user or group are specified by @.

EXIT

Sequence of Operations

1. Close all files.

2. Terminate ACCTS.

HELP

Sequence of Operations

1. If number supplied schedule HELP with ACCT #.

2. If keyword supplied, scan HELP and dump message to list LU.

3. Else, dump entire help file, first lines only.

4. Return.

11.7 LIST,ACCT

Sequence of Operations

1. Set up list LU and check optional parameter.

2. Read account file header.

3. Read account file directory for user.group names of active sessions

4. Read cartridge list for current state of disc pool.

5. If AC or AL, write header information and active session table.

6. If PO or AL, write disc pool status.

7. If CO or AL, write configuration table.

8. Return.

11.8 LIST, GROUP

Sequence of Operations

1. Get parameters.
2. If group name = @ go to 6.
3. Search account file for group name using FINDG; return error is not found.
4. List group account to list LU.
5. Return.
6. Search directory for next group entry; at end, go to 8.
7. List corresponding group account and go to 6.
8. Return.

11.9 LIST, USER

Sequence of Operations

1. Get parameters.
2. If KEY supplied, set KEY flag.
3. If user not equal @ then if group not equal @ call FINDU to find user.group account.
4. If @, search all user accounts.
5. If @.@ search all user and group accounts.
6. List the account(s) found and keywords if KEY flag.
7. Return.

11.10 LOAD

Sequence of Operations

1. Get parameters.
2. If ACCTS option and account file does not exist, force entire load.
3. If account file exists with accounts, issue request to verify purge.
4. If not verified, return.
5. Prompt for station table size and number of accounts.
6. Read and verify NAMR (backup file) header record.
7. Read and backup into scratch file.
8. Purge old account file.
9. Rename accounts file.
10. Return.

11.11 NEW, GROUP

Sequence of Operations

1. Get group name, SST entries and verify before writing to buffer.
2. Search account file for duplicate name.
3. Search for spare entries in account file.
4. Update highest group ID used.
5. Write buffer to record.
6. Return.

11.12 NEW, USER

Sequence of Operations

1. Prompt for user name.

2. If name is "@" or "/E", or if name has imbedded comma(s) or period(s), report invalid name and go to 1.

3. Prompt for group name.

4. If name is "@" or if name has embedded comma(s) or period(s), report invalid name and go to 3.

5. If name is not "/E", go to 7.

6. If no group has yet been defined for user, report expecting valid group name and go to 3, else go to 12.

7. Search account file directory for group name using FINDG.

8. If not found, report group account does not exist and go to 3.

9. Search account file directory for user.group name using FINDU.

10. If found, report duplicate user.group name and go to 3.

11. Prompt for whether to use group SST and save group SST yes/no flag with group name and group ID. Go to 3.

12. Prompt for password until " " or a valid password is entered.

13. Prompt for hello file until " " or a valid NAMR is entered.

14. Prompt for capability until a valid capability is entered.

15. Prompt for disc limit until a valid limit is entered (positive integer not greater than MAXD).

16. Prompt for SST entries until "/E" is entered. Validate system LU.

17. Prompt for SST spares and validate (positive integer $\leq$ MAXST-total SST entries defined).

18. Prompt for whether to link user. If no link, go to 22.

19. Search account file for user.group name and verify password. If not found or password does not match, report error and go to 18.

20. Read existing user ID and post to buffer.

21. Go to 23.

22. Generate user ID and update lowest user ID used.

23. For each user.group account, call FREEU to get free account file space and write record, updating directory.

24. Return.

11.13 PURGE, ACCT

Sequence of Operations

1. Verify request to purge.

2. Shut down session system and purge +@CCT!!-31778:$ACFL.

3. Return.

11.14 PURGE, GROUP

Sequence of Operations

1. Get parameter.

2. If group name not equal @, go to 6.

3. Verify user's request to purge all accounts; if not, return.

4. Scan account file directory and flag all entries except SYS, SUPPORT and GENERAL with -1.

5. Return.

6. If group name = SYS, SUPPORT or GENERAL, error return.

7. Search account file directory for all user accounts with matching group name or group account with matching group name.

8. If found, verify that group is inactive and has no discs mounted, then purge account directory entry.

9. If none found, error return.

10. Return.

11.15 PURGE, USER

Sequence of Operations

1. Get parameter.
2. If user name not equal @, go to 5.
3. Verify user's request to purge all of this group's account; if not, return.
4. Go to 6.
5. If user.group = MANAGER.SYS or ENGINEER.SUPPORT, error return.
6. Search account file directory for user account(s) with matching group name.
7. If found, verify that user is inactive and has no discs mounted, then purge account directory entry.
8. If none found, error return.
9. Return.

11.16 RESET

Sequence of Operations

1. Get parameters.
2. If user or group name = @, go to 5.
3. Call FINDU to find user account and offset to CPU and CONNECT words and zero.
4. Return (error return if not found).
5. If @.@, go to 9.
6. @.group; search for all user accounts with matching group name, offset to CPU and CONNECT words and zero.

7. Find group account and zero.

8. Return.

9. For every account, if group then zero clock words, or, if user, also zero clock words.

10. Return.

11.17 TELL

Sequence of Operations

1. Get parameters.

2. If user or group name = @, go to 7.

3. Find user.group account by searching active session table, getting record # and comparing name.

4. Get terminal logical unit or error return if zero (not logged-on).

5. Send message using class I/O.

6. Return.

7. If @.@, go to 10.

8. @.group; find all active users in group by searching active session table, getting record # and comparing on group part of name.

9. Send message to each and return.

10. Search SCB list for terminal logical units and

11. Send message and return.

11.18 UNLOAD

Sequence of Operations

1. Get parameter and verify NAMR.

2. Read file +@CCT!:-31178:-2 and write NAMR removing all unused space

3. Return.

11.19 PASSWORD

Sequence of Operations

1. Prompt for current password.

2. Verify against current users.

3. If not valid report error and terminate.

4. Else prompt for new password.

5. Update entry.

6. Return.

11.20 INTERNAL SUBROUTINES

This section describes those subroutines which are used internally by the Account Program in initialization and command processing. Included in the description of each routine are:

1. brief description
2. entry point
3. external references
4. calling sequence
5. where routine is used
6. sequence of operations

11.20.1 ACCRE - Create Accounts File

Entry point: ACCRE

External References: OVRD.,.ENTR,CREAT,PURGE,$ACFL,ACOMD,$LIBR,$LIBX

Calling Sequence: CALL ACCRE (NDCB,NAME,ISIZE,IERR)

Parameters:

NDCB	Data control block for file access
NAME	File name to be created

ISIZE	Size of file in 128 word blocks
IERR	ERROR parameter returned

Used in: ACCT1, ACLOA, ACACP

Sequence of Operations

1. Set disc Lu is $ACFL.
2. Set override bit.
3. Purge file of same name on all mounted cartridges.
4. Create file on disc LU.
5. Reset override bit.
6. Return

11.20.2 ACOPN - Open Accounts File

Entry point; ACOPN

External References: $SMID,OVRD.,.ENTR,ACOM1,ACOMD,OPEN,ISMVE, $ACFL,$LIBR,$LIBX,LOCF

Calling Sequences: Call ACOPN (IERR,IDSES)

Parameters:

IERR	ERROR RETURN PARAMETER
IDSES	SESSION ID used to determine capability

Used in: ACCT1, ACLOA, ACACP, ACOPL

Sequence of Operations

1. Call ISMVE to get session ID.
2. Set override bit.
3. Get $ACFL.
4. Open accounts file on $ACFL or first disc which it is found.
5. Clear override bit.
6. If $ACFL set up return.

7. Call LOCF to find LU.

8. Set $ACFL.

9. Return.

11.20.3 ACWRH - Write Syntax Messages For Help

Entry point: ACWRH

External References: ACWRL, .ENTR

Calling Sequences: CALL ACWRH (KEYWD,IERR,JERR)

Parameters:

KEYWD	First 2 characters of keyword
IERR	ERROR returned from ACWRL
JERR	ERROR if keyword not found

Used In: ACCT5

Sequence of Operations

1. Get first table entry.

2. If keyword equal to zero, go to 4.

3. If entry does not match keyword, go to 6.

4. Print syntax of command.

5. If keyword not equal to zero, print explanation.

6. Get next table entry.

7. If not end of table, go to 2.

8. Else return.

11.20.4 ACINT - Retrieve $DSCS and $DSCS+1

Entry Point: ACINT

External References: $DSCS, .ENTR

Calling Sequence: CALL ACINT (ISTAT,JSTAT)

Parameters:

ISTAT = $DSCS
JSTAT = $DSCS+1

Used in: ACCT1

Sequence of operation

1. Retrieve $DSCS.

2. Retrieve $DSCS+1.

3. Return.

11.20.5 ACPAS - Verify MANAGER.SYS Password

Entry Point: ACPAS

External References: ACOM1,.ENTR,ACOM6,ACOM7,ACOM0,ACOMC,
ACOMD,READF,ACPSN,ACERR,ACTRM

Calling Sequence: CALL ACPAS

Sequence of Operations

1. Find account with ID=7777B.

2. If no password for account return.

3. Else prompt for password.

4. If password is verified, return.

5. Else print error.

6. Terminate call ACTRM.

11.20.6 ACPSN - Input and Parse Password

Entry Point: ACPSN

External References: .DIV,.ENTR,ACOM7,ACOM9,ACOMC,ACPRM,ACREI,XLUEX,
PARSN,ACERR,ACTRM

Calling Sequence: CALL ACPSN (MESS,LENGTH,JPASS,IERR)

Parameters:

MESS	IS PROMPT
LENGTH	IS LENGTH OF PROMPT
JPASS	IS BUFF FOR PARSED PASSWORD
IERR	IS RETURN ERR

Used In: ACCT1, ACPAS, ACAPA

Sequence of Operations

1. Print message.

2. Read password no echo.

3. If not read from DVR07, go to 5.

4. Back up and clear previous line.

5. Parse password.

6. If wrong format print error.

7. If no password make it default.

8. Return.

11.20.7 ACSDN - Shut Down an Active Session

Entry Point: ACSDN

External References: EXEC,.ENTR,$LGOF,XLUEX,XFTTY,LUSES,
ACOMD,$CES,$LIBR,$LIBX

Calling Sequence: CALL ACSDN (LU,IERR)

Parameters:

LU	Session LU to be shut down
IERR	Returned error

Used In: ACPUA

Sequence of Operations

1. If not LU=0, go to 4.

2. Clear $CES.

3. Return.

4. Get SCD ADDRESS.

5. Send message to $LGOF to log off session.

6. Schedule LGOFF

7. Return.

11.20.8 ACAST - Retrieve Entry in Active Session Block

Entry Point: ACAST

External References: .ENTR, ACOM3, LDCB

Calling Sequence: CALL ACAST (JBUF)

Parameters:

JBUF CURRENT SST

Used In: ACALU

Sequence of Operations

1. Get first change in LDCB.
2. Search for session LU match in JBUF.
3. If delete, remove it and compress.
4. If modify, change system LU.
5. If add, add entry to SST.
6. If more changes get next change and go to 2.
7. Else update number of entries.
8. Return.

11.20.9 ACSTR - Print Stars

Entry Point: ACSTR

External References: .ENTR

Calling Sequence: CALL ACSTR

Used In: ACLIV, ACLIA

Sequence of Operations

1. Print 45 *'s
2. Return.

11.20.10 ACACP - File Cleanup and Complete Shut Down

Entry Point: ACACP

External References: .MPY,.ENTR,EXEC,IFBRK,ACOM1,ACOM4,ACOM6, ACOM9,READF,ACGSP,ACINM,RLMEM,ACERR,ACWRI, RNRQ,CLOSE,ACCRE,MESSS,ACOPN,ACTRM,IVBOF, ACDIR,ACFST,ACPGA,ACSID,WRITF

Calling Sequence: CALL ACACP

Used In: ACCT1

Sequence of Operations

1. If shut down or purge accounts, go to 11.
2. If not purge an account, go to 14.
3. Find account that is flagged to be purged.
4. If none flagged, go to 14.
5. If active session, go to 9.
6. If spool file, go to 9.
7. If disc mounted, go to 9.
8. Purge account.

9. Find next account that is flagged to be purged.

10. Go to 4.

11. Release memory.

12. Release class numbers and resource number.

13. If purge account, purge +@CCT!

14. Return.

11.20.11 ACNVS - Converse with Terminal

Entry Point: ACNVS

External References: .ENTR,ACOM7,ACOM4,ACPRM,ACREI,NAME,PARSN

Calling Sequence: CALL ACNVS (IOUT,NWORDS,MODE)

Parameters:

IOUT	OUTPUT STRING
NWORDS	NO OF WORDS IN STRING
MODE	PARSING MODE

Used In: ACCT1,ACALT,ACLOA,ACPUA,ACPUC

Sequence of Operations

1. Output string.

2. Input string.

3. If mode = 0, go to 6.

4. Call PARSN to parse first parameter of input string.

5. Return.

6. Call NAMR to parse first parameter of input string.

7. Return.

11.20.12 ACTIM - Print Connect and CPU Times

Entry Point - ACTIM

External References: .MPY,.DIV,.ENTR,ACDDV,ACFMT

Calling Sequence: Call ACTIM (ITIME,IERR)

Parameters:

ITIME	Words 1&2 Connect Time Words 3&4 CPU Time
IERR	Error returned.

Used In: ACLIV

Sequence of Operations

1. Convert connect time.
2. Print connect time.
3. Convert CPU time.
4. Print CPU time.
5. Return.

11.20.13 ACSID - Set ID Bit Map

Entry Point: ACSID

External References: .MPY,.ENTR,ACOM6,ACOM5,ACOM1,ACSBT, RNRQ,IVBVF,WRITF,READF

Calling Sequence: CALL ACSID

Used In: ACALU, ACACP

Sequence of Operations

1. Get first account.
2. If group account, go to 6.
3. Call ACSBT (IDU,MBUF).
4. If IDU < LOWUS then set LOWUS to IDU.
5. Go to 7.
6. If IDG>HIGR then get HIGR to IDG.

7. Call ACSBT (IDG,MBUF).

8. Get next account and if any, go to 2.

9. Post ID bit map.

10. Return.

11.20.14 ACNFG - Retrieve Entry From Configuration Table

Entry Point: ACNFG

External References: .MPY,.DLD,.DST,.ENTR,FLOAT,ACOM1,
ACOM6,MBYTE,READF,LBYTE

Calling Sequence: CALL ACNFG (IERR,IDX)

Parameters:

IERR	Returned ERROR
IDX	Index into Configuration Table

Used In: ACLIA

Sequence of Operations

1. Read record which contains IDX.

2. Put value in registers.

3. Increment IDX.

4. Return.

11.20.15 ACFDF - Find Free Account Entry

Entry Point: ACFDF

External References: .MPX,.DIV,.ENTR,ACOM6,ACOM1,READT,MOD

Calling Sequence: CALL ACFDF (IDIRN,IRECM,IOFST,JERR,K)

Parameters:

IDIRN	=	DIRECTORY ENTRY NUMBER of free account
IRECM	=	RECORD NO. of free account
IOFST	=	Offset 0 or 64

```
JERR       =  ACERR return word
K          =  1 for normal request
K          =  2 for extension request
                 (start on sector boundary)
```

Used In: ACALV, ACNWG, ACNWU

Sequence of Operation

1. If K=1 search every 64 word block for free entry.
2. Else search every 128 word block for free entry.
3. Set up return parameters.
4. Return.

11.20.16 ACGSP - Schedule GASP for Spool Information

Entry Point: ACGSP

External References: .DLD,.DST,.ENTR,EXEC,KSPCR,RMPAR

Calling Sequence: CALL ACGSP (NAME,IERR,TYPE)

Parameters:

```
NAME      USER.GROUP NAME
IERR      ERROR RETURNED
TYPE      FUNCTION FOR GASP
```

Used In: ACPUA, ACACP

Sequence of Operations

1. Check $SPCR if 0 return.
2. Build run string.
3. Schedule GASP.
4. Call RMPAR.
5. Set IERR.
6. Return.

11.20.17 ACGTG - Get Group Account

Entry Point: ACGTG

External References: .ENTR,ACOM1,ACFDA,READF

Calling Sequence: CALL ACGTG (IGRP,IBUF,IOFST,IERR)

Parameter:

IGRP	5-WORD BUFFER CONTAINING GROUP NAME
IBUF	128-WORD BUFFER WHERE ACCOUNT ENTRY IS RETURNED
IOFST	OFFSET INTO BUFFER (0 or 64)
IERR	ERROR RETURN WORD (-200) FMP ERROR

Used In: ACLIV

Sequence of Operations

1. Set IUSER = 0.

2. Call ACFDA.

3. Read account into buffer.

4. Return.

11.20.18 ACGTU - Get User Account

Entry Point: ACGTU

External References: .ENTR,ACOM1,ACFDA,READF

Calling Sequence: CALL ACGTU (IUSER,IGRP,IBUF,IOFST,IERR)

Parameters:

IUSER	5-WORD BUFFER USER NAME
IGRP	5-WORD BUFFER GROUP NAME
IBUF	128-WORD BUFFER WHERE ACCOUNT IS RETURNED
IOFST	OFFSET INTO BUFFER (0 or 64)
IERR	ERROR RETURNED (-200 of FMP error)

Used In: ACALU, ACLIU, ACNWU

Sequence of Operations

1. CALL ACFAA to find account.

2. Read account into IBUF.

3. Return.

11.20.19 ACGID - Get Free ID Number

Entry Point: ACGID

External References: .ENTR,IABS,ACOM5,ACOM6,ACOM1,RNRQ,
READF,ACGBT,WRITF

Calling Sequence: CALL ACGID (ITYPE,ID,IERR)

Parameters:

```
 ITYPE     1 Get user ID
          -1 Get group ID
ID        ID number returned
IERR      ERRORS  -1 invalid parameter
                  -2 No ID available
                     FMP error
```

Used In: ACNWG, ACNWU

Sequence of Operations

1. If ITYPE = 1, go to 6.

2. Search up for ID.

3. If ID >=LOWUS, go to 10.

4. Post ID.

5. Return.

6. Search down for ID.

7. If I<=IHIGR, go to 10.

8. Post ID.

9. Return.

10. IERR = -2.

11. Return.

11.20.20 ACGBT - Get Bit Out of ID Map

Entry Point: ACGBT

External References: .ENTR

Calling Sequence: CALL ACGBT (NWRD,IDIR,BITNO)

Parameters:

NWRD	WORD IN WHICH BIT IS SEARCH
IDIR	DIRECTION OF SEARCH -1 means 0 to 15 1 means 15 to 0
BITNO	BIT NO of available bit

Used In: ACGID

Sequence of Operations

1. Get word.

2. IF IDIR = -1, go to 8.

3. Set count to 15.

4. Rotate left through E.

5. If E zero, go to 13.

6. Decrement count.

7. Go to 4.

8. Set count to 0.

9. Rotate right through E.

10. If E zero, go to 13.

11. Increment count.

12. Go to 9.

13. Set bit in original word.

14. Return.

11.20.21 ACSBT - Set Bit in ID Map

Entry Point: ACSBT

External References: .ENTR

Calling Sequence: CALL ACSBT (ID,NBUF)

Parameters:

ID	ID number of which corresponding bit in bit map must be set.
NBUF	BUFFER which contains Bit Map.

Used In: ACSID

Sequence of Operations

1. Compute word which must be updated.

2. Compute bit.

3. Inclusive "OR" bit into existing word.

4. Return.

11.20.22 ACASB - Search for Active Session

Entry Point: ACASB

External References: .MPY,.ENTR,ACOM6,IVBUF

Calling Sequence: CALL ACASB (IDIRN,LU,I)

Parameters:

IDIRN	IS DIRECTORY ENTRY NUMBER
LU	IS THE STATION LU WHERE ACCOUNT IS ACTIVE
I	IS THE INDEX INTO THE ACTIVE SESSION TABLE

Used In: ACTEL

Sequence of Operations

1. Search for entry with IDIRN.

2. If not found return.

3. Read LU and return.

11.20.23 IVBUF - Treat File as Large Array

Entry Point: IVBUF

External References: READF, WRITF, ACOM1

Calling Sequence:		
	READ	I=IVBUF (INDEX,IREC)
	WRITE	CALL IVBUF (INDEX,IREC,IVAL)
	POST	CALL IVBUF

Parameters:

INDEX	IS INDEX INTO THE LARGE ARRAY
IREC	IS THE STARTING RECORD NUMBER OF THE ARRAY
IVAL	IS THE VALUE TO BE WRITTEN

Used In: ACALT, ACPUA, ACACP, ACSID, ACASB

Sequence of Operations

1. If post request post both records and return.

2. Else compute record number.

3. If in memory, go to 6.

4. Post oldest buffer to disc.

5. Read in new buffer.

6. If read, read value.

7. If write, write value.

8. Return.

11.20.24 ACINM - Initialize and Release Session Memory

Entry Points: ACINM, RLMEM

External References: .ENTR,EXEC,$LIBR,$LIBX,$LGOF,$LGON,$STM,
$DSCS,$SMVE,$SRTN,$SALC,$BALC,$BRTN,$SMEM

Calling Sequence: CALL ACINM (ISIZE,MAXEV,IBUF,LNGTH,OLDLN)

Parameter:

ISIZE	AMOUNT OF MEMORY REQUESTED
MAXEV	LARGEST BLOCK POSSIBLE
IBUF	BUFFER CONTAINING DISC POOL
LNGTH	LENGTH OF DISC POOL
OLDLN	OLD LENGTH OF DISC POOL

Used In: ACCT1, ACALT, ACACP

Sequence of Operations

1. If memory allocated, go to 5.

2. Allocate memory.

3. Give it to $SALC.

4. Allocate class numbers.

5. Allocate memory for disc pool.

6. Transfer disc pool.

7. Set up $DSCS.

8. Return.

Calling Sequence: CALL RLMEM (IDSCS,ICLAS)

Parameters:

IDSCS	New value for $DSCS
ICLAS	CLASS numbers used by ACCTS

Used In: ACACP

Sequence of Operations

1. Release class numbers.
2. Give disc pool back.
3. Take memory from session.
4. Return it to $BRTN.
5. Return.

11.20.25 ACLNK - Link to Subroutines in Other Segments

Entry Point: ACLNK

External References: .ENTR,EXEC,ACOM2,SEGLD,ACERR,ACWRI

Calling Sequence:
```
ASSIGN 100 TO LRTRN
ASSIGN 200 TO LRTR2
CALL ACLNK (ISEG,IGOTO)
```

Parameters:

ISEG	IS THE ASCII (1H) OF THE LAST CHARACTER OF THE SEGMENT NAME
IGOTO	IS THE INDEX TO BE USED BY COMPUTED GO TO IN THE SEGMENT

Used In: ACCTS,ACCT1,ACMND,ACHLP

Sequence of Operations

1. If in memory, jump to start.
2. Else build segment name.
3. Call SEGLD.

11.20.26 ACLTM - Print Last Log of Time.

Entry Point: ACLTM

External References: .ENTR

Calling Sequence: CALL ACLTM (ITIME,IBUF)

Parameters:

ITIME	2-wd Array of the Time
IBUF	17-wd Buffer for ASCII of Time

Used In: ACLIU, ACLIA

Sequence of Operations:

1. Get seconds, minutes, and year.
2. Separate seconds, minutes, and year.
3. Adjust year.
4. Get hours and days.
5. Compute AM or PM.
6. Compute month, day of month, and day of week.
7. Convert and put ASCII in buffer.
8. Return.

11.20.27 ACOPL - Open List File

Entry Point: ACOPL

External References: .ENTR,IABS,IFBNR,ACOM2,ACOM3,ACOMC,ACOPN,
IXOR,LUTRU,ACLCK,ACERR,LURQ,ACTIN,ACROP

Calling Sequence: CALL ACOPL (IERR,ITYPE,JSIZE)

Parameters:

IERR	ERROR return FMP ERROR.
ITYPE	TYPE of OPEN LIST or BINARY.
JSIZE	Size of file if created.

Used In: ACCT1,ACMND,ACLIV,ACLIA,ACLOA,ACUNL,ACHLP

Sequence of Operations:

1. If ITYPE.GE.0, go to 4.
2. Open accounts file.
3. Go to 8.
4. If list = LLIST, use LLIST and return.
5. If LU then lock it and return.
6. Else check file not in input stack.
7. If in input stack report error and return.
8. Open file.
9. Return.

11.20.28 ACLCK - Lock List LU

Entry Point: ACLCK

External References: .ENTR,EXEC,IFBRK,XFTTY,LURQ,ABREG,ACWRI

Calling Sequence: CALL ACLCK (LU,IERR)

Parameters:

LU	LU which is to be locked.
IERR	ERROR returned 10 Break 12 LU not in switch table

Used In: ACOPL,ACHLP,ACXFR

Sequence of Operations:

1. If TTY return.
2. Lock LU no wait.
3. If reject set error 12 word return.
4. If not previously locked return.

5. Else print messages.

6. Test Break Flag. If set set error = 10 and return.

7. Suspend 50 milliseconds

8. Go to 2.

11.20.29 ACROP - Open or Create File

Entry Point: ACROP

External References: .ENTR,OPEN,POSTN,CREAT

Calling Sequence: CALL ACROP (IDCB,IERR,NAME,IOPT,ISC,
ICRN,ISIZE,ITYPE)

Parameters:

IDCB	FMP Data Control Block.
IERR	FMP Error return.
NAME	NAME of file.
IOPT	Open option.
ISC	Security code of file.
ICRN	Cartridge reference # of file.
ISIZE	File size.
ITYPE	File type.

Used In: ACOPL,ACXFR

Sequence of Operations:

1. Try to open file.

2. If error - 6, go to 6.

3. If an error, return.

4. If TYPE = 3, POSTN file to end.

5. Return.

6. Create file.

7. Return.

11.20.30 IFBNR - Determine if Device has Binary Mode.

Entry Point: IFBNR

External References: .DIV,.ENTR,XLUEX

Calling Sequence: IF (IFBNR(IRW,LU))...

Parameters:

IRW MODE OF OPERATION

= 0 Both Read and Write.
1 Read
2 Write
3 Binary

LU Logical Unit of Device.

Used In: ACOPL,ACWRL

Sequence of Operations:

1. Read Status EXEC (15,...)

2. If device can handle transfer, go to 5.

3. Else set IFBNR = .FALSE.

4. Return.

5. Set IFBNR = .TRUE.

6. Return.

11.20.31 ACNXA - Get Next Account Directory Entry and Compress Directory

Entry Point: ACNXA

External References: .MPY,.ENTR,ACOM1,ACOM6,ACOM9,READF,MOD

Calling Sequence: CALL ACNXA (J,IREC,IDEL,KOUNT,IDIR,IDELX)

Parameters:

J	IS OFFSET INTO JBUF OF DIRECTORY ENTRY J is set to -1 to start at beginning of directory.
IREC	Is REC NUMBER OF DIRECTORY ENTRY
IDEL	Is MINUS # holes in directory.
KOUNT	Is the count of directory entries.
IDIR	Is directory entry number.
IDELX	Is index into delta table.

Used In: ACUNL

Sequence of Operations:

1. Get first directory entry.
2. If an extent align to even directory.
3. If empty adjust delta.
4. Else return.

11.20.32 ACFID - Fix Message, User and Group Pointers

Entry Point: ACFID

External References: .ENTR

Calling Sequence: CALL ACFID (IVAL,IDELI,KDEL)

Parameters:

IVAL	Location of pointer in account.
IDELI	Initial
KDEL	Point in delta table.

Used In: ACUNL

Sequence of Operations:

1. Search for delta.

2. Add delta and initial offset to IVAL.

3. Restore IVAL.

4. Return.

11.20.33 ACPGA - Clear Directory Entry

Entry Point: ACPGA

External References: .ENTR, ACDIR

Calling Sequence: CALL ACPGA (I,IDIRN,ID)

Parameters:

I	Value for first word of directory -1 = free, -2 = extent
IDIRN	DIRECTORY ENTRY NUMBER
ID	ID number of account.

Used In: ACALU, ACNWU, ACACP

Sequence of Operations:

1. Set IBUF (1) to I.

2. Write directory entry.

3. Return.

11.20.34 ACTRM - Terminate ACCTS

Entry Point: ACTRM

External References: .ENTR,EXEC,ACOMC,ACOM2,ACOM1,ACOM3,ACOM4,
ACOM6,CLOSE,ACCLS,LURQ,XLUEX,DTACH

Calling Seqeunce: CALL ACTRM

Used In: ACCTS,ACCT1,ACPAS,ACAPA,ACPSN,ACACP

Sequence of Operations:

1. Close files.
2. Unlock LU's.
3. Print END ACCTS.
4. If no clean up, go to 7.
5. If PROG is "ACCTS", go to 8.
6. Schedule "ACCTS" to do clean up.
7. Terminate (EXEC(6)).
8. Schedule myself ACCTS to clean up in 30 seconds.
9. Terminate.

11.20.35 ACDDV - Double Word DIVIDE

Entry Point: ACDDV

External References: .ENTR

Calling Sequence: CALL ACDDV (IDBL,IDIVR,IQUOT,IREMN)

Parameters:

IDBL	DOUBLE WORD INTEGER
IDIVR	DIVISOR
IQUOT	QUOTIENT
IREMN	REMAINDER

Used In: ACTIM

Sequence of Operations:

1. Load dividend.
2. Swap A and B registers.
3. If sign bit set, go to 6.
4. Divide.

5. If no overflow, go to 8.

6. Set quotient MAX.

7. Set remainder to max mod value.

8. Store quotient.

9. Store remainder.

10. Return.

11.20.36 ACDIR - Read and Write Directory Entries

Entry Point: ACDIR

External References: .MPY,.DIV,.ENTR,ACOM6,ACOM1,READF,WRITF,ACERR

Calling Sequence: CALL ACDIR (ICODE,IDIRN,IBUF,IERR)

Parameters:

ICODE	1 READ
	2 WRITE
IDIRN	DIRECTORY ENTRY NUMBER
IBUF	16 WD BUFFER TO READ OR WRITE
IERR	ERROR RETURN WORD

Used In: ACCT1,ACALU,ACNWG,ACNWU,ACACP,ACPUU,ACAPA,ACPGA

Sequence of Operations:

1. Verify parameters, if ok, go to 3.

2. Get error, return.

3. Compute record number and offset.

4. Read record.

5. If write, go to 8.

6. Move from record to buffer.

7. Return.

8. Move buffer to record.

9. Post to disc.

10. Return.

11.20.37 PACFDA - Find Account Entry

Entry Point: ACFDA

External References: .MPY,.DIV,.ENTR,ACOM6,ACOM1,ACOMA,READF

Calling Sequence: CALL ACFDA (IUSER,IGRP,IDIRN,IRECU,IRECG,JERR)

Parameters:

IUSER	5-WD BUFFER containing user name.
IGRP	5-WD BUFFER containing group name.
IDIRN	DIRECTORY ENTRY NO. (Returned)
IRECU	2-WD ARRAY Record number and offset of user account
IRECG	2-WD ARRAY Record number and offset of group account
JERR	ERROR returned -200 not found or FMP ERROR.

Used In: ACALU,ACNWG,ACNWU,ACPUU,ACTEL,ACGTG,ACGTU

Sequence of Operations:

1. If searching, go to 4.

2. Initialize indexes to start of directory.

3. Go to 8.

4. If group search, go to 7.

5. Set indexes to last found user account.

6. Go to 8.

7. Set indexes to last found group.

8. Scan for match (@ matches anything).

9. Save indexes (for future searches).

10. Return directory number record numbers and offset(s).

11.20.38 ACFMT - Format and Output Data

Entry Point: ACFMT

External References: ACWRL,.ENTP,ACOM2,NAM..

Calling Sequence: CALL ACFMT (IERR,F1,IBUF1,-N,F2,IBUF2,F3,...,FN,IBUFN)

Parameters:

F1,F2,...FN	are function codes
N	is number of blanks (next PARM is a function code)

Function codes are:

where:	0<FN<"I0"	PRINT N ASCII CHARACTERS
	FN=0	PRINT ASCII CHARACTERS UNTIL BLANK IS ENCOUNTERED (MAXIMUM NO OF CHARS FOLLOWS)
	FN="I0"	PRINT DECIMAL NUMBER WITHOUT LEADING BLANKS
	"I0"<FN<"I6"	PRINT IN "In" FORMAT
	FN="CR"	PRINT ASCII IF LEGAL NAME ELSE PRINT INTEGER

IBUF1,IBUF2,IBUF3,... ARE EITHER ASCII STRINGS OR NUMERIC DATA

Used In: ACLIV,ACLIA,ACTIM

Sequence of Operations:

1. Clear parameter addresses.
2. Call .ENTP to get parameters.
3. Get first function code.
4. If positive, go to 7.
5. Put -N number of blanks in output buffer.

6. Go to 18.

7. If function code =,"CR", go to 11.

8. If function code ="I0", "I1",...,"I6", go to 16.

9. Transfer string to output buffer.

10. Go to 18.

11. Call NAM..

12. If not ASCII, go to 15.

13. Put 2 ASCII characters in output buffer.

14. Go to 18.

15. Get code to "I0".

16. Convert decimal number to ASCII.

17. If "I0" suppress leading blanks else use full field width.

18. If another function code, go to 4.

19. Return.

11.20.39 ACCLL - Close List File or Unlock LU.

Entry Point: ACCLL

External References: .ENTR,ACOM3,XLUEX,LURQ,ACCLS

Calling Sequence: CALL ACCLL

Used In: ACLIV,ACLIA,ACLOA,ACUNL,AHLP,ACWRL

Sequence of Operations:

1. If file, go to 5.

2. Send top of form.

3. Unlock LU.

4. Go to 6.

5. Call ACCLS to close file.

6. Set &IST =-1.

7. Return.

11.20.40 ACCLS - Close and Truncate File

Entry Point: ACCLS

External References: .DIV,.ENTR,LOCF,CLOSE

Calling Sequence: CALL ACCLS (IDCB,ITYPE)

Parameters:

IDCB	DCB of file to be closed
ITYPE	Type of file to be closed

Used In: ACTRM,ACCLL,ACSFR

Sequence of Operations:

1. Set truncate to zero.
2. If type 1, go to 5.
3. Call locf to find position.
4. Compute truncate parameter.
5. Close and truncate.
6. Return.

11.20.41 ACPRM - Prompt Interactive Device

Entry Point: ACPRM

External References: .ENTR,IFBRK,ACOM8,ACERR,ACWRI

Calling Sequence: CALL ACPRM (MSG,MSGLNG)

Parameters:

MSG ASCII STRING TO BE WRITTEN

MSGLNG LENGTH of string in words

Used In: ACCT1,ACMND,ACALT,ACALU,ACNWG,ACNWU,ACPSN,ACNVS,ACREI

Sequence of Operations:

1. If MSGLNG = -1, go to 4.
2. Save buffer in local buffer.
3. Output string.
4. Return.
5. Output string from local buffer.
6. Return.

11.20.42 ACREI - Input Commands From Device, File, or Memory

Entry Point: ACREI

External References: .MPY,.DIV,.ENTR,ACOM3,ACOMC,XLUEX,ABREG,ACWRI,
NAMR,MBYTE,ACXFR,ACERR,ACHLP,ACPRM,READF

Calling Sequence: CALL ACREI (IBUF,IERR)

Parameters:

IBUF INPUT BUFFER

IERR ERROR RETURN

Used In: ACCT1,ACMND,ACALT,ACALU,ACNWG,ACNWU,ACPSN,ACNVS

Sequence of Operations:

1. Reading from memory, go to 13.
2. Reading from file, go to 11.
3. Read from LU.
4. If echo set write to log device.

5. Fill with blanks.

6. If not "/TR" or "/HE" return.

7. If "/TR" call ACXFR.

8. If "/HE" call ACHLP.

9. Call ACPRM with MSGLNG = -1.

10. Go to 1.

11. Call READF.

12. Go to 4.

13. Transfer data from LDCB.

14. Go to 4.

11.20.43 ACHLP - Process HELP Commands

Entry Point: ACHLP

External References: .ENTR,EXEC,ACOM2,ACOM3,ACOMC,ACOMD,NAMR,ACOPL,ACLNK
ACWRL,ACERR,ACCLL,ACITA,LUTRU,LURQ,ACLCK,ACWRI

Calling Sequence: CALL ACHLP (ICMND,ISTRC)

Parameters:

ICMND Command string

ISTRC String pointer

Used In: ACCTS,ACMND,ACREI

Sequence of Operations:

1. If scheduled from ACREI or parameter is numeric, go to 7.

2. OPEN list file.

3. Call ACLNK to call ACWRL.

4. If not found, write not found.

5. Close list file.

6. Return.

7. Convert list device.

8. If same as list already open unlock LU.

9. Schedule HELP.

10. Relock LU if necessary.

11. Return.

11.20.44 ACERR - Post and Print Errors

Entry Point: ACERR

External References: .ENTR,ACOMC,ACITA,PTERR,ACXFR,ACWRI

Calling Sequence: CALL ACERR (IERR)

Parameters:

IERR	Error number to be posted

Used In: ACCT1,ACCT2,ACMND,ACALT,ACALU,ACLIU,ACLIA,ACLOA,ACNWG,ACNWU, ACPAS,ACAPA,ACPSN,ACPUA,ACPUU,ACTEL,ACUNL,ACACP,ACLNK,ACOPL, ACDIR,ACPRM,ACREI,ACHLP,ACWRL

Sequence of Operations:

1. Convert error code to ASCII.

2. Post error to DCB.

3. Transfer to log device.

4. Write error.

5. Return.

11.20.45 ACWRL - Write to List File or Device

Entry Point: ACWRL

External References: .ENTR,IFBNR,XFTTY,IFBRK,ACOM2,ACOM3,XLUEX, WRITF,ACWRI,ACERR,ACCLL

Calling Sequence: CALL ACWRL (IBUF,NO,IERR)

Parameters:

IBUF   is output buffer

NO   is number of words in buffer

IERR   is error returned

Used In: ACWRH,ACLIV,ACLIA,ACUNL,ACSTR,ACFMT,ACHLP

Sequence of Operations:

1. If list file, go to 5.
2. If list default, go to 10.
3. Write to device.
4. Return.
5. If logical list, go to 8.
6. Write to list file.
7. Return.
8. Write to logical list file.
9. Return.
10. Write to input device.
11. Return.

11.20.46 ACREL - Read From List Device or File

Entry Point: ACREL

External References: .ENTR,ACOM1,ACOM3,XLUEX,ABREG,READF

Calling Sequence: CALL ACREL (IBUF,NO,LEN,IERR)

Parameters:

IBUF   is input buffer

NO   is buffer size

LEN   is words transmitted

IERR   is error returned

Used In: ACLOA

Sequence of Operations:

1. If read from +@ACCT! go to 7.
2. If read from file, go to 5.
3. Read from device.
4. Return.
5. Read from file (LDCB).
6. Return.
7. Read accounts file.
8. Return.

11.20.47   ACITA - Convert Integer to ASCII

Entry Point: ACITA

External Reference: .ENTR

Calling Sequence: CALL ACITA (INT,IBUF,NOWDS)

Parameters:

INT   Integer to be converted

IBUF   Buffer where ASCII if put

NOWDS   No of words in buffer

Used In: ACCT1,ACALT,ACALU,ACLOA,ACNWU,ACMSN,ACHLP,ACERR

Sequence of Operations:

1. If negative generate minus sign.
2. Change to positive.

3. Convert number with leading zeros.

4. Return.

11.20.48 MBYTE,LBYTE - Retrieve Upper or Lower Byte of Word

Entry Point: MBYTE,LBTYE

Calling Sequence: IUP = MBYTE (INT)
ILW = LBYTE (INT)

Parameters:

INT is integer

Used In: ACCT1,ACALT,ACALU,ACLIV,ACLIA,ACNWU,ACPUU,ACTEL,ACNFG,ACREI

Sequence of Operations:

MBYTE

1. Get INT.

2. Swap bytes.

3. Go to 5.

LBYTE

4. Get INT.

5. Mask off upper byte.

6. Return.

11.20.49 ACXFR - Transfer Control to Device or Command File

Entry Point: ACXFR

External References: .MPY,.ENTR,ACOMB,ACOM3,ACOMC,LOCF,CLOSE,NAMR,LUTRU,
OPEN,APOSN,LURQ,ACLCK,ACCLS,ACTEN,ACROP

Calling Sequence: CALL ACXFR (ICMND,ISTRC,IERR)

Parameters:

ICMND	Command string
ISTRC	No of words in string
IERR	Error returned

Used In: ACCTS,ACMND,ACREI,ACERR

Sequence of Operations:

1. If current input is not file, go to 4.
2. Save current position.
3. Close file.
4. Parse to get next.
5. If null, set to -1.
6. If negative back up transfer stack and go to 8.
7. Advance transfer stack and put new file or LU on stack.
8. If file, go to 12.
9. If error, go to 14.
10. Lock LU.
11. Go to 15.
12. Open and position file.
13. Go to 15.
14. Set LU to log LU.
15. Reset list LU if specified.
16. Reset Echo bit if specified.
17. Return.

11.20.50 ACTIN - Test List Files Against Transfer Stack File Names.

Entry Point: ACTIN

External References: .ENTR,ACOMB

Calling Sequence: CALL ACTIN (IPBUF,IERR)

Parameters:

IPBUF	Buffer with namr of list file
IERR	Error which is returned.

Used In: ACOPL, ACXFR

Sequence of Operations:

1. Compare IPBUF with first entry in stack.
2. If equal, go to 7.
3. Compare IPBUF with next entry in stack.
4. If equal, go to 7.
5. If more entries in stack, go to 3.
6. Return.
7. Set error.
8. Return.

11.20.51 ACWRI - Write to Input Device

Entry Point: ACWRI

External References: .ENTR,ACOM3,ACOMC,XFTTY,XLUEX,WRITF

Calling Sequence: CALL ACWRI (IBUF,ILEN)

Parameters:

IBUF	is buffer to write
ILEN	is length of buffer in words IF ILEN<0 CALL FROM ACREI

Used In: ACCT1,ACMND,ACALT,ACALU,ACLOA,ACNWU,ACACP,ACPUA,ACAPA,ACLNK,
ACLCK,ACPRM,ACREI,ACHLP,ACERR,ACWRL

Sequence of Operations:

1. If echo call from ACREI, go to 6.
2. If not interactive LU, go to 4.
3. Print buffer.
4. If there is not a list of file or LU, go to 6.
5. Write to list file or LU.
6. If echoing and input is not the same aa the log unit then write to log unit.
7. Return.

11.20.52 ACSES - Shut Down Session

Entry Point: ACSES,LMES,KSPCR,ACFST

External References: EXEC,.ENTR,$LIBR,$LIBY,$DSCS,$LGOF,$LGON,$LMES,
$SPCR,$CL1,$CL2

Calling Sequences:

```
CALL ACSES (-2) SHUT DOWN
CALL ACSES (0) START UP
CALL LMESS (JBCNT,JBUF,IDSC1)
I = KSPCR (IDUM)
CALL ACFST (IBUF)
```

Parameters:

JBCNT	LNGTH of log on string
JBUF	LOGON string
IDSC1	VALUE to put in $DSCS+1
IDUM	DUMMY parameter
IBUF	Buffer for cartridge list

Used In: ACCT1,ACALT,ACLIA,ACACP,ACLOA,ACPUA,*GSP

Sequence of Operations:

ACSES

1. Stuff parameter into $DSCS+1.

2. If not restart, return.

3. Tell LOGON to shut down.

4. Tell LGOFF to shut down.

5. Return.

LMESS

1. Put long string in memory.

2. Update $DSCS+1.

3. Return.

KSPCR

1. Retrieve $SPCR.

2. Return.

ACFST

1. Read cartridge list.

2. Mask all ID words.

3. Return.

SESSION TERMINAL HANDLERS	CHAPTER 12

12.1 OVERVIEW

The Session Terminal Handlers provide for the initiation of the log-on sequence, the actual log-on process, the processing of break mode requests after log-on, and finally the log-off process.

Figure 12-1 shows the four programs which provide the above defined processes. The PRMPT program gets things started and is the main controlling processor. PRMPT is scheduled by interrupt (from each terminal requesting service) and then communicates with either the log-on processor (LOGON) to perform a log-on, or the command response processor (RPN) if a session does not exist for the interrupting terminal.

The log-off processor (LGOFF) receives its control from either a session progenitor (copy of FMGR) or the account program (ACCTS). Note that RPN, LOGON and LGOFF receive all their control via class I/O.

By using class I/O, the prompt program (PRMPT) can initiate several log-on requests before the first request has completed. This method avoids the problem of the log-on program being busy when another log-on request is recognized by the program PRMPT. The prompt program is scheduled by interrupt, and does not wait for the log-on program to complete.

```
          +----------+         +----------------+
          /          |         |                |
         /           |         |                |
        / SESSION    |------>  |     PRMPT      |
  +----+  TERMINAL   |         |                |
  |                  |         |                |
  |                  |         |                |
  +------------------+         +----------------+
          ^                            |
          |                            |
          |                            |
          |                            |
          |                            |
          |                            v
          +<---------------   PLEASE LOG ON  --------------+
                                                           |
                                  or                       |   *
                                                           |$LGON
                              S=XX COMMAND?                |
                                 |                         |
                                 |                         v
      +----------------+         |            +----------------+
      |                |         |            |                |
      |                |    *    |            |                |
      |                | $STN    |            |                |
      |     R$PN$      |<--------+            |     LOGON      |
      |                |                      |                |
      |                |                      |                |
      |                |                      |                |
      +----------------+                      +----------------+

                                              +----------------+
  +-----------+                               |                |
  |           |                 *             |                |
  |           |             $LGOF             |                |
  |   FMGR    |---------------------------->  |     LOGOFF     |
  |           |                               |                |
  |           |                               |                |
  +-----------+                               +----------------+
```

* = CLASS NUMBERS

Figure 12-1. Session Terminal Handling Process

12.2 OPERATING ENVIRONMENT

1. SESSION TERMINALS must be configured with an interrupt table entry of:

   XX,PRG,PRMPT

2. At least three class numbers must be available and the ACCTS program must have performed its initialization process. This initialization includes the allocation of memory for control blocks and the allocation of the three class numbers required by the session terminal handlers. These class numbers are saved in Table Area I to prevent their loss should one of the terminal handlers be aborted.

   $LGON - LOGON PROGRAM CLASS #
   $LGOF - LGOFF PROGRAM CLASS #
   $STH - R$PN$ PROGRAM CLASS #

3. The terminal must have been enabled to permit the driver to schedule the PRMPT program whenever an asynchronous interrupt is received.

   Typically, a terminal is enabled with the :CT,LU# command. It may be disabled with the :CT,LU#,21B command.

Enabling the terminal allows the driver to place PRMPT's ID address into the associated EQT. The driver takes the ID address out of the interrupt table (it is in two's complement form) and places it into a temporary word in the EQT or EQT extension (as a positive address). The interrupt table entry is then replaced with the first word address of the referenced EQT. SESSION is now ready to handle the terminal.

An interrupt from the CRT (TTY) device is generated by hitting any key. $CIC in RTIOC is entered and vectors the interrupt to the appropriate driver. The driver then determines if the terminal is enabled, if not the interrupt is ignored. If the terminal is enabled, the driver schedules PRMPT via the system routine $LIST and passes the address of the fourth word of the appropriate EQT.

12.2.1 Prompt Processing

Prompt is given the address of EQT 4, of the interrupting device when it is scheduled. From the EQT address, the routine FNDLU calculates the LU, the device status, device type and possible RN bypass word. If the LU or the EQT is down, the request is ignored. The RN bypass word is returned if the interrupting device is locked.

It is possible to write through an LU lock. Parameter nine (RQP9) of all I/O EXEC requests has been reserved as an LU lock bypass word. The word is configured as:

```
15                   8 7                      0
+---------------------------------------------+
| RN# owner            |   RN# from DRT       |
+---------------------------------------------+
```

The RN# owner can be retrieved by indexing into the RN# Table and isolating the lower byte. If the above word is configured and a DEF is made to it in RQP9, then the system will not suspend the executing program and will honor the I/O request. Only BREAK mode requests will be processed if the terminal is locked.

If a log-on is required, an error is reported and the request is ignored.

If the interrupting device is a multipoint terminal, a control request to set the EDIT-MODE flags is issued.

Next, a check is made to insure that the session environment has been correctly enabled (class number defined). If the environment is OK, the terminal is disabled to prevent multiple log-on or break mode requests to be initiated before the first one has completed. The disable is performed by setting a bit in the session bit map, !BITM. The disable bit is set by the PRMPT program and cleared by RPN or LOGON. A soft disable rather than a hard disable (CN,LU,21B) is used to prevent ever losing a terminal. A terminal can be lost if the program responsible for enabling the terminal has been aborted. In this case, the terminals interrupts would be ignored until a control request to enable it was received.

With the soft disable, if the reenable program should be aborted, pressing a key would get PRMPT scheduled. Upon finding the bit map entry set, PRMPT will make sure that LOGON, LGOFF and RPN are all scheduled and then terminate.

Next, PRMPT checks to see if a session is defined for the terminal. This is performed by scanning the list of active session, looking for a match with the session ID (SESSION ID = TERM LU).

If a session exists, the break mode prompt, S=XX COMMAND?, is issued. The class read of the response is initiated (class # = $STH) and the response program is scheduled. PRMPT then terminates, waiting for the next interrupt to be received.

If a session does not exist, the log-on or shut-down message is issued. This prompt string is defined externally and is updated by the ACCTS program. If the session monitor is in shut down mode ($DSCS+1 <0), do not initiate the read of the response, just clear the disable bit (!BITM) and terminate. Otherwise, start the class read of the response (class # = $LGON), schedule the logon processor (LOGON) and terminate.

12.2.2 RPN Processing

RPN is the program responsible for processing all session operator commands. After some initialization (session enabled and class # defined), a class GET request is issued on the class # $STH. R$PN$ waits for a completion of a command input from any of the active session terminals.

Next, the same checks performed by PRMPT are made to assure that the device may be written to. If the terminal is OK, a check is made to verify that the session issuing the command still exists (could have logged off) and the soft disable on the terminal is cleared (!BITM).

All operator commands are checked to verify that the user has sufficient capability to execute it. The system library routine CAPCK, performs the capability check and returns a pass-fail status. If the user has the ability to perform the command, a check is made to see if any special processing must be performed. The following commands are processed by the RPN program:

FL: Make control request to flush all requests to this terminals driver until all buffered requests have been cleared or a read request is found.

WH: Schedule the WHZAT program for this session.

HE: Schedule the HELP program for this session.

TE: Send a message to the system console.

BR: If no parameters, break the current session program.

SS: If no parameters, suspend the current session program.

GO: If no parameters, issue a GO,PROG against the current session program.

OF: If no parameters, abort the current session program (if FMGXX, do not abort but set the break flag).

UP: If no parameters, and if the current session program is waiting for a down device, issue a UP,EQT against the down device.

SL: Display the session-system LU mapping.

RS: If FMGXX is active (not dormant) do an OF,FMGXX,1 then schedule FMGXX for this session. If FMGXX cannot be found, create one, then schedule it for this session.

The current session program is determined by finding the last son of FMGXX or FMGXX itself. Note: except for the UP command the programs D.RTR and SMP are exempted from the son test. These programs should never be aborted or suspended.

If the command was not one of the previously defined special commands, pass the command to the message processor for handling by the operating system.

Issue any response from the operating system and finally, check for the next command to process by making another class GET request.

12.2.3 LOGON Processing

All interactive processing performed by LOGON is performed via class I/O. This is done to increase the log-on request response time.

Once scheduled, LOGON never terminates. It is suspended on a class GET request, waiting for the next log-on request.

12.2.4 Calling Sequence

The LOG-ON process is initiated by the program PRMPT. The PRMPT program is scheduled as a result of striking any key on a terminal configured as a session terminal.

It (PRMPT) has the following responsibilities:

a. Set a bit map flag (in Table Area I) to indicate that log-on for this terminal is in progress.

b. Prompt the user for his USER.GROUP NAME. This prompt has the following format:

    PLEASE LOG-ON:

c. Initiate a class read of the user response (using the communication class number, $LGON). The format of this input is as follows:

    USER[.GROUP][/PASSWORD]

NOTES: . and / are delimiters
: blanks are ignored
: maximum field size for user, group and password is ten characters each
: see definition of optional parameters passed in class request.

d. Schedule the program LOGON (without wait or queue), passing following information in the first two optional parameters:

WD1=0
WD2=Terminal logical unit number (1 to 99).

NOTE: The class I/O request allows optional parameters to be saved with the class request and then returned with the class GET request.

These optional parameters define the request to the log-on processor as follows:

a. INTERACTIVE log-on request

IOP1=0
IOP2=Terminal LU (1 to 99), used as session identifier

b. NON-INTERACTIVE log-on request

IOP1=0 or class # for communication from LOGON back to the calling program.

IOP2=Number to be used as session identifier (100 to 254).

c. PASSWORD reply

IOP1=NOT used

IOP2= - Character count of second buffer. The class read or write/read of the password must be a double buffered request. Refer to the password discussion for details on the required contents of the second buffer.

d. SHUTDOWN request

IOP1=NOT used
IOP2=-1

e. Batch (JOB) LOG-ON request

IOP1=NOT used
IOP2=-2
Class Buffer=Directory entry # of account to be logged onto.

f. Account Name request

IOP1=NOT used
IOP2=-3
Class Buffer=Account directory entry #.

g. Account Directory Entry # request

IOP1=NOT used
IOP2=-4
Class Buffer=USER.GROUP/PASSWORD string.

12.3 LOGON FLOW

START-UP WORK (only performed when LOGON is first scheduled).

a. Verify that the session environment has been initialized.

Any problems cause an error diagnostic to be issued and the log-on program will terminate.

b. Fetch SCB offsets and communication class number $LGON.

The SCB offsets are the locations of information in the SCB to be built by LOGON. The offsets are defined externally to allow ease of modification to the SCB structure.

c. OPEN the ACCOUNT file and check its validity.

The following conditions must hold for the account file or an error diagnostic is issued and LOGON terminates.

The file must be found.

The file must be type 1.

The low 8 bits of word 25, record 1 must contain a resource number lockable by LOGON.

1. Get a LOG-ON request.

Release possible RN lock and clear no-parse flag.

A class I/O GET request is issued against the class defined by $LGON. This request will cause the program LOGON to be suspended until a class read (or write/read) has completed.

NOTE: The SAVE buffer option is used with this request to prevent the loss of a log-on request in the event of LOGON being aborted. The buffer is released by another GET request before going on to the next log-on request.

2. Determine type of call.

   If the original request was not a class read or write/read, release the current class buffer and continue at 1.

   If log-on request, continue at 3.

   This is a shutdown request. The following functions are performed:

   Close account file
   Release class request buffer
   Terminate

   If this is a special request (IOP2 <0) continue at 6-1.

3. This is a log-on request.

12.4 SETUP FOR LOG-ON

Set interactive/non-interactive flags.
Request lock on account file resource number.
Get account file header into memory.

12.5 CHECK SESSION LIMIT

The session limit, and the number of currently active sessions reside in the account file header.

If the session limit has been reached,

1) issue an error
2) continue at 5-1.

12.6 PARSE INPUT BUFFER

The password processor (refer to 6.0) verifies that a password was entered and then sets some flags and continues as a regular log-on request. One of the flags set is the no-parse flag. This indicates that the parse of the user group name has already taken place.

If the no-parse flag is true, continue at 3-4.

Call the parse subroutine to produce the following:

a. Byte counts of user, group and password.
b. Isolated user, group and password names.

NOTE: The user name is required but the group and password are optional. If a group name is not provided, the group name is filled with the default, GENERAL.

If a user name was not entered,

1) issue an error
2) continue at 5-1.

12.7 SCAN ACCOUNT FILE DIRECTORY FOR USER

The location (record) of the account file directory is defined by word 5 of the account file header. The number of records in the directory is defined as the difference between the location of the directory (word 5 of header) and the location of the first account entry (word 6 of header).

The first word of a directory entry (16 words each) defines the following:

 0 = end of directory
<0 = free entry
>0 = user or group directory entry

Scan the account file directory until:

a. word 1 of an entry = 0

or

b. last directory record read

or

c. the user account directory entry is found.

If the user is not found, 1) Issue an error, 2) continue at 5-1.

12.8 USER IDENTIFIED

Word 15 of the user's directory entry defines the record number of the actual account entry. The sign bit of this word indicates where in the record the account entry resides. If clear, the entry begins on word 1 of the record. If set, the entry starts at word 64 of the record.

READ the user account into memory.

If a password is not required, continue at 3-7 (word 1 of the account file entry contains the character count of the password).

12.9 PASSWORD REQUIRED

If a password was passed, compare it to the required password. If it matches, continue at 3-7. Else, 1) issue an error, 2) continue at 5-1.

Password not passed, prompt the user for the required password:

PASSWORD?

3-6.2 If this is an interactive call, continue at 3-6.2. Else issue an error and continue at 5-1.

3-6.2 Interactive request of password

Setup and issue a double buffered class read (without echo) to the session terminal, on class number $LGON.

The contents of the second buffer, which must be returned with the GET of the response, is defined below:

WD1 = 0 if interactive
= class # for communication response

WD2 = session identifier

WD3 = Byte counts of user + group names (user byte count is in the high byte).

WD4-8 = user name (blank filled)

WD9-13 = group name (blank filled)

NOTE: Word 1 currently has no use. It is used here on the chance that non-interactive log-on requests may want to be able to prompt for and then return a password if it was not provided in the original request. By changing the above set up for the class write/read, the double buffered request could be sent back to the requestor.

12.10 USER IDENTIFIED

At this point, the user has been identified and verified. The user's account entry is in memory.

Check word 64 of the user's account entry. If the sign bit is set, the low 15 bits of this word define the record number of an extension for this account. This extension will always begin on word 1 of the sector.

Initialize SCB to zero and move in: Identifier; capability; user ID; Group ID, Disc Limit.

The SST spares (-1's) are placed in the SST first. These are the only entries which may be modified on-line. All the entries from the console definition (LU 1) to the end of the table are defined by LOGON and are not altered thereafter.

Define first SST entry past end of spares for LU 1. If interact request, map LU 1 to the terminal LU. If non-interactive, map session LU 1 to system LU 1.

LU 2 is always mapped to LU 2.

LU 3, if defined in the system is mapped to LU 3.

Move SST entries from user account into the SST being built.

The system disc LUs are now defined by reading the cartridge directory (defined by $CL1 and $CL2) and then making an entry in the SST for each disc mounted to the MANAGER.SYS account (ID=7777B).

NOTE: The LUs in the SST are allowed 1 byte each with the system LU residing in the left hand byte. The LUs are saved as LU 1 to match internal operating system formats.

12.11 CHECK FOR CONFIGURATION TABLE ENTRY

Save required information (from account file entry) before the input buffer is used for the configuration table search.

If this is a non-interactive request, continue at 4.

Scan the Configuration Table for an entry defined for this terminal. (NOTE: The SST entry for LU 1 defines the station.)

If not found, continue at 4.

12.12 CONFIGURATION TABLE ENTRY FOUND

Move the entries in the Configuration Table to the SST being built.

If any entry causes a conflict in the definition of a session LU (duplicate session LUs, different system LUs), reject the Configuration Table entry and issue a diagnostic message. Continue with the next Configuration Table entry.

NOTE: Conflicts in the definition of SST entries do not terminate the log-on sequence.

12.13 UPDATE SST LENGTH WORD

The SST length word is the sum of the following:

Account file SST entries + 2 (or 3 if LU3 is defined) + Configuration Table entries + the number of spare entries requested (low byte, word 32, user account entry) + the number of system discs (other than LUs 2 + 3).

NOTE: As LUs are stored as LU 1, the spare entries are defined as -1 (LU 1).

12.14 COMPLETE CONSTRUCTION OF SCB

Place the two's complement of disc limit in the first word past the last SST entry.

Total size of SCB includes the number of words specified by the disc limit.

12.15 BUILD SESSION PROGENITOR

If this is a non-interactive log-on request, continue at 4-3.

The progenitor is merely a copy of FMGR with the name FMGLU, where LU is the session terminal logical unit.

If no free ID segments exist: 1) issue an error, 2) continue at 5-1.

12.16 MOVE AND LINK SCB

Move the SCB to free memory and link it into the SCB list headed at $SHED. This is accomplished by the MKSCB subroutine.

If no room or duplicate session identifier error: 1) issue an error, 2) remove the session progenitor, 3) continue at 5-1.

12.17 POST LOG-ON INFORMATION TO ACCOUNT FILE

Update the active session counter

Add an entry to the active session table as follows:

```
    Wd 1 = session identifier
wds 2-3 = log-on time
    wd 4 = Directory entry number
```

The format of the log-on time is:

```
WD1 = year (offset from 1978, 15-13) min (12-7) sec(6-0)
WD2 = day(13-5) hour(4-0).
```

12.18 SESSION CREATED AND ACCOUNT FILE UPDATED

Release the resource number lock to permit account file alteration.

Scan the list of mounted discs (FMP cartridge directory) looking for a match with this session's user and/or group IDs.

For every match found, call the FMP mount cartridge routine to make the disc addressable by this session.

12.19 ISSUE LOG-ON COMPLETE MESSAGES

List system message file (defined in account file header record).

If the user has mail waiting, let him know.

If this is a non-interactive request, continue at 5.

12.20 START UP FMGLU

FMGLU is started by building the following string:

    RU,FMGLU,name:security code:cartridge

where name:security code:cartridge is the user's HI (command) file namr.

This string is then sent to the MESSS routine, passing the SCB address in the second optional parameter.

The MESSS routine then causes FMGLU to become scheduled and tags it as belonging to the new session.

12.21 LOG-ON COMPLETE

Issue a TIE-OFF request to the message processor. This causes the user terminal to be enabled, if interactive. If non-interactive, the calling program is informed of the completion via a class write/read request.

Release the current log-on request and continue at 1.

6-0. This is a password response

Set up required flags (so message processor will operate if required) PARSE the password.

If password not supplied: issue an error and continue at 5-1.

Set the no-parse flag (NOPAR=1). Continue at 3-5.

6-1. Summary of Special Requests to the LOGON Processor

1. NON-INTERACTIVE log-on request. The log-on processor will communicate with other programs requesting a session creation. The communication is performed via a class number supplied by the caller. All messages normally returned to the session terminal are returned in this class number. Refer to the message processor section (MESSP) for details on returned messages and status flags. Once we determine that the request is not interactive, the communication class number (remember, this is not $LGON) is saved away and the log-on process continues the same as if the request was interactive.

   The special processing for non-interactive requests follows: START HERE

   a. Password (if required by user account) must be supplied or the request is rejected.

      NOTE: This restriction was requested by the Batch Processor and D.S.. The LOGON processor has the ability to request the password from the calling program and then continue with this request when it is returned.

      b. The session terminal is defined as the system console.

      c. A session progenitor is not created for non-interactive sessions.

2. PASSWORD reply. If a password is required but is not supplied, LOGON saves all the required information about the current process in the second buffer of a double-buffere class I/O write/read of the requested password. When the user completes the input of the password, both buffers are returned to the LOGON processor via the class get request (both the password buffer and the control buffer). This control buffer permits LOGON to continue processing on a previously initiated request. This processing merges back with the standard log-on processing at the point where the directory is searched for the specified user.

NOTE: We have already found this user once and the directory entry number could have been saved in the control buffer of the password read. This index would then directly point you at the requested directory entry. The reason for going back and searching the entire directory is that the directory may have been reworked or the user could have been purged between the point in time where the read of the password is initiated and completed.

3. SHUT DOWN REQUEST. The ACCTS program sometimes has the requirement of purging the account file. To do this, LOGON must close the file and then terminate.

4. BATCH (JOB) LOG-ON REQUEST. LOGON provides for the creation of a special session (ID=255) for the BATCH system. This request operates on directory entry numbers, not on a log-on string (see the discussion on the directory entry number request). After decoding the directory number and verifying that this user still exists, the standard log-on processing continues, treating the request as a standard non-interactive request with ID=255.

5. ACCOUNT NAME REQUEST. LOGON will translate a directory entry number into a user and group string (directory format) and return it to the calling program in a user supplied class #.

6. ACCOUNT DIRECTORY ENTRY #. LOGON will accept a user group and password string and scan the account file directory looking for a match. If a match is found, the directory entry number is returned in the user specified class number. This function is used by the JOB processor and the spool system as it allows the compression of a USER.GROUP/PASSWORD string into one word. This number is the count of directory entries scanned, counting from zero, until the requested entry is found.

12.22 LGOFF PROCESSING

All LGOFF processing is performed via class I/O. Once scheduled, LGOFF never terminates but is suspended on a class GET request, waiting for the next log-off request.

Calling Sequence

The log-off process is initiated by a copy of FMGR or the ACCTS program sending a log-off request in the class number defined by Table Area I entry point $LGOF. The two optional parameters (IOP1 & IOP2) of the Class I/O request define the type of request:

IOP1 >0 then:

IOP1= SESSION ID of session to be logged off in bits 0-8.

Bit 15 = 1 Dismount private discs
Bit 14 = 1 Dismount group discs
Bit 13 = 1 Kill active programs

IOP2= SCB pointer of the session logging off.

class buffer = possible communication class #.

IOP2 <0 then:

IOP2= -1 Shut down

or

= -length of second buffer (response to kill program PRMPT).

12.23 LGOFF FLOW

The log-off processor consists of three major sections: active program work; disc dismounting work; and account updating.

Before a user can be logged off all programs linked to the session must be aborted and possibly purged from the system. This is accomplished by scanning the list of ID segments looking for all session words (ID word 32) equal to the SCB address (SST length word) defined for this session. For all programs found, perform the following work:

1. Is the program D.RTR or SMP? If not continue with 2. Else, set a flag (OOPS) to indicate that D.RTR or SMP is currently active for this session. These programs must be allowed to clean themselves up as they may be modifying the disc. D.RTR is cleaned up via a session clean-up schedule request. This request is currently treated as a NOP (by D.RTR). Its function is to prevent the destruction of a session control block while D.RTR is still linked to it. SMP is cleared via a session clean-up request to it. This request will cause all pending spool LU's (files) associated with the session to be closed and printed (if required).

   NOTE: These clean-up requests will be issued after all other currently active session programs have been aborted. Therefore, after our schedule request of these programs, there is no way they could be currently operating for our session.

2. Active program found and it is not D.RTR or SMP. Before we can abort this program, we must have permission from whoever requested the log-off. Permission may be passed in the request (bit 13 IOP1=2) or we have to prompt for and fetch permission. If permission has been given, continue at 4. Else, set a flag (FND=+1) to indicate that an active program exists, then return the name of the program to the caller (session terminal or class buffer). Continue searching ID segments at 1.

3. End of ID segments. If any programs have been aborted (FND<0), go back (1) and make another pass to verify that no one came in behind us. If any active programs were found and we did not have permission to kill (FND>0), prompt the caller and start the class read (on $LGOF) of the response.

   NOTE: This is a double buffered request with the following information saved in the control buffer:

   WORD 1 = Bit 15 = 1 if dismount private discs requested.

   Bit 14 = 1 if dismount group discs requested.

   Bits 8-0 = Session ID.

   WORD 2 = SCB pointer of the session logging off.

   WORD 3 = Possible communication class # (non-interactive).

   Go make next class GET for next log-off request.

   If no active programs were found (FND=0), continue at 5.

4. Active program found and we have permission to abort the program. Issue a OF,PROG,1 request (via MESSS) to abort the program. Set FND=-1 to indicate that a program was aborted. Continue searching the list of ID segments (1).

5. At this point, all programs running for this session have been aborted. If D.RTR was found active to this session (see step 1 above), issue the session clean-up request to it.

6. Next, release all ID segments (longs only) allocated to this session. The list of ID segments is again scanned, this time checking the owner flag (bits 8-0 of word 31) with the session ID of this session. The owner flag is defined by LOADR or the RP FMGR command. It's value is set equal to the session ID of the requesting session. The exception to this is the MANAGER.SYS session. Programs loaded or RP'ed by this session have a zero for their session ID, allowing the system manager to leave programs in the system after he logs-off. For each match found (owner ID = SESSION ID) and the program is dormant, issue a OF,PROG,8 command (via MESSS). If an ID segment was built for this session and someone else is using the program (remember, all programs related to this session must be

dormant or we would not be here), give the ID to the session currently running the program. This is performed by changing the programs owner flag to reference the session it is currently active to. This completes the program management portion of LGOFF.

7. The next function performed is disc cartridge management. The log-off request defines which discs (if any) are to be removed. The dismount is performed by calling the FMP subroutine DCMC. The cartridge list provides the information (disc ID and LU) required to decide which discs are to be dismounted. Each disc which matches the specified private or group ID is removed from the system. Each disc removed has an informational message issued to the session to inform the caller that the disc was really removed.

8. Finally, the spool monitor program is called to clean up all spool files associated with this session and the following information is updated in the account file: active session block for this entry is cleared; active session counter is decremented; connect and CPU usage information is updated to both private and group entries. The session control block is now released (via RLSCB) and a tie-off request is sent to the LOGON/LGOFF message processor. This request will clear the disable bit in the session bit map and return a completion status to the caller if the request was not interactive.

9. Go get next log-off request.

12.24 LOGON/LGOFF MESSAGE PROCESSOR - MESSP

MESSP provides the diagnostic message interface for the LOGON and LGOFF processor. This routine will send a message to the session terminal and/or the system console. MESSP also provides a programmatic interface by returning diagnostics in a user supplied class number.

Calling Sequence:

```
Call MESSP (where,what,how much)

where = Bit 15   = Termination call
        Bit 12   = Don't append session # to message
        Bit 11-6 = ERROR code flag
        Bit 1    = Print on System console
        Bit 0    = Print on Session console

what  = message buffer address

how much = +word, -byte count of buffer
```

NOTES: If where is zero or has bit 15 set (and this is an interactive request), this routine will enable the terminal logging-on (off) by clearing the corresponding bit set back when the log-on (off) request was initiated.

The where parameter is passed to the calling program if the current request is non-interactive. Communication is via a class I/O write/read to the communication class number passed in the original call (not $LGON ($LGOF) but another class # supplied by the caller).

The status of the request passed in the class buffer is defined by the where parameter. This status is returned in the second optional parameter of the class write/read request.

IOP2=0`=`Successful log-on/off, nothing in class buffer - last message.

>0`=`LOGON/OFF message or diagnostic is in the class buffer. At least one more message will follow.

<0`=`Terminating error is in class buffer. This is the last message for this log-on/off request.

Bits 11-6 contain the error code or 0 (if this is not an error). For example, LGON 06 error would have 000 110 in bits 11-6 of the where parameter.

The following special subroutines are used by the Session Terminal Handlers:

DTACH

Purpose: To remove a program from session.

NOTE: If the calling program is not a session program, this routine does nothing more than return.

Calling Sequence:

Call DTACH \ removes prog from session by changing session word to contain - terminal LU of it's session.

or

Call`DTACH(IDUMMY)``\`removes prog from session by changing session word to contain -1 (makes it appear to have been run from the system console).

In either case, the owner flag is changed to indicate that the system owns this ID.

CAPCK

Purpose: To provide command validation and capability level checking of RTE-6/VM session operator commands.

Calling sequence: REG=CAPCK(IBUF,LEN,ISCB,ICAP)

Where: IBUF = command buffer to be checked.

LEN = -bytes or + words

ISCB = optional SCB address to be used.

ICAP = optional capability level to check the command against.

returns:	(ok return)	(A) = ASCII command
		(B) = parameter count
		(X) = address of parameter #1 in CAPCK's buffer.
	(capability error)	(A) = ASCII command
		(B) = -1
		(X) = addr of parm #1
	(command undefined)	(A) = -1
		(B) = parameter count
		(X) = addr of parm #1

12.25 $SALC + $SRTN

Purpose: To manage a predefined block of memory for use by the Session Monitor.

Calling Sequence

1. Allocate memory from session block:

```
(P)     JSB $SALC
(P+1)   (# of words needed)
(P+2)   -Return no memory ever (A)=-1,  (B)=max ever
```

```
(P+3)     -Return no memory now  (A)=0.    (B)=max now
(P+4)     -Return OK             (A)=addr (B)=size or size+1
```

2. Release buffer to session memory

```
(P)       JSB $SRTN
(P+1)     (FWA of buffer)
(P+2)     (# of words returned)
(P+3)     -return- (all registers destroyed)
```

If a request for a buffer of length X cannot be filled during a given call, return is made with:

```
(A) = 0
```

If, when buffer requested, - (AVMEM) - shows insufficient memory available to contain a buffer of the length requested, then return is made with:

```
(A) = -1
(B) = maximum length buffer that the program may allocate.
```

To find out how large a buffer may be allocated, use the call:

```
JSB $SALC
DEC 32767
```

Blocks of memory available for output buffering are linked through the first two words of each block -

```
WORD1 - length of block
WORD2 - address of next block (or 77777 if this is last block)
```

The allocator transfers the upper end of a block to caller and shortens the length of the block by the amount transferred.

Registers are not preserved

To initialize the session memory block, the following call must be made.

```
(P)       JSB $SRTN
(P+1)     (-FWA of buffer)
(P+2)     (-# words in session block [one's complement])
(P+3)     -return- all registers modified
```

To release the block (indicate to session that no memory is available for use) make the same call as defined above. The status of the release call is returned in the (A) register.

A=0 Did not reset pointers as the # words specified did not match the current number of words available (i.e., active SBC still exists so do not return memory).

A=1 Did not reset pointers as the start address of the block did not match the current block address.

A=-1 Session pointers are reset to indicate no memory is available - OK to return memory to SAM.

12.26 MKSCB

Purpose: To move and link an SCB from the user map to the session available memory in the system map.

Calling Sequence: CALL MKSCB (IBUFA,IBUFL,IADDR,IERR)

IBUFA = Buffer address of SCB in user map

IBUFL = Length of SCB in user area (SCB in user space begins with word 3, the session identifier).

IADDR = Address of the SCB in session available memory (system map) is returned here.

IERR = 0 OK - SCB created
= 1 No memory now
= 2 No memory ever
= 3 Duplicate session identifier

12.27 RLSCB

Purpose: To remove an active SCB from the system.

Calling Sequence: CALL RLSCB (SESID,IERR)

SESID = Session identifier of SCB to be released.

IERR = 0 OK

= -1 SCB not found

GLOSSARY OF TERMS USED BY SESSION MONITOR

System Logical Unit (LU) - A number (1-255) assigned at system generation to define an I/O device. This number appears on the left hand side of the session switch table and is used to reference a specific I/O device.

Session Logical Unit (LU) - A number (1-63) assigned during Account File setup OR at log-on (via Configuration Table entries) OR by the operator (via the SL command).

The Session Logical Unit provides a constant mapping of devices into a common set of user accessible logical units. For example, Session LU 1 always refers to your session console, regardless of which specific device (System LU) you are using.

Session Switch Table (SST) - A table that defines a session's total I/O addressing range. This table, placed in System Available Memory by the LOG-ON processor, consists of System LU's and Session LU's. For every session LU, there is a system LU which defines the actual device which may be referenced. This may be a direct map (Session LU25 and System LU25), or an indirect map (Session LU30 and System LU125).

NOTE: The only way a user program may access a System LU greater than 63 is under session (via the SST).

```
                        10

For example:      +-------------+
                  |     -5      |   NEGATIVE LENGTH
                  |-------------|
                  |  41  |   1  |   SESSION CONSOLE DEFINITION
                  |-------------|
                  |   2  |   2  |   SYSTEM DISC
                  |-------------|
                  |   3  |   3  |   AUX SYSTEM DISC
                  |-------------|
                  |  42  |  42  |   DIRECT MAP
                  |-------------|
                  | 179  |  45  |   INDIRECT MAP
                  +-------------+
                  SYSTEM | SESSION
                    LU      LU
```

Station - A set of devices available to a session user. For example, a station might consist of a session terminal, cartridge tape units (2), and printer.

Configuration Table - A set of default logical units (added to your SST) defined for a specific station's devices. For example, this table would allow you to reference your terminals left cartridge tape unit as Session LU4, regardless of the actual System LU definition.

For example:

```
+-------------+
|      4      |    LENGTH OF ENTRY
|-------------|
| 25  |       |    STATION LU
|-------------|
| 26  |   4   |    DEFAULT LEFT CTU (LU26) TO L
|-------------|
| 27  |   5   |    DEFAULT RIGHT CTU (LU27) TO
|-------------|
| 28  |   6   |    DEFAULT PRINTER (LU28) TO LU
|-------------|
|      6      |    LENGTH OF ENTRY
|-------------|
| 30  |       |    STATION LU
|-------------|
| 31  |   4   |    DEFAULT LEFT CTU (LU31) TO L
|-------------|
| 32  |   5   |    DEFAULT RIGHT CTU (LU32) TO
|-------------|
| 125 |  20   |    \
|-------------|     \
| 126 |  21   |      -DEFINE HP-IB DEVICES FOR
|-------------|     / THIS STATION
| 127 |  22   |    /
|-------------|
|      0      |    END OF TABLE
+-------------+
```

Session Identifier - The value used in linking and identifying each specific session (word 3 of the SCB). The common contents is the system logical unit for the session console of the session. The session ID need not correspond to an interactive device, but it must be unique within the set of active sessions.

Session Control Block (SCB) - A variable-length table built by the log-on process for each session. It contains information unique to each session (i.e., session switch table).

Account file - A disc file, set up and maintained by the account set-up program (ACCTS). This file defines the following session information: Session Welcome File; active session information; Configuration Table; Disc Allocation Pool; Account File Directory; Group and User Account File Definitions.

MLS-LOC LOADER — CHAPTER 13

The RTE-6/VM Loader, MLLDR, can be viewed as a three part process:

1. Initial set up: Processing commands found in the RU,MLLDR runstring, and processing commands found in a command file before any relocation is performed.

2. Record relocation: Processing relocatable files to produce absolute executable code and processing any of a group of commands which may be entered at any time (i.e., not required before any relocation).

3. Final processing: All processing necessary to finish the load and make the program an executable program in the system.

Initially, the command file, relocatable input file, and list device must be set up. These can be defaulted, entered in the runstring, or entered in a command file. Opcode and Format parameters are processed to set the program type (BG, RT, LB, EB), common type (SC, RC, NC, SS), load type (PE, TE, RP), EMA type (EM,VM), current page linking option (MP,CP,BP), debug option, list option (NL, LE), and don't copy option.

The program can be assigned to a particular partition, a given size can be specified, and the profile option with its associated output device can be specified. The dynamic buffer area of the loader is initialized to contain dummy base page, fixup table, symbol table, and local path informaton for the current node being processed. All user libraries must be specified in LI commands. Finally, if entered, the echo command or purge command must be processed before relocation (if not a purge request) can begin.

Actual relocation and manipulation of the fixup table (Figure 13-1) and symbol table (Figure 13-2) is done by the loader library, $LDRLN. All relocate, search, and node commands are recognized by the loader main and any changes needed are made in parameters or flags. The appropriate loader library routines are called and relocation continues.

```
Word  15        11   10 9     3  2        0
     |----------------------------------------|
  0  | memory address to be fixed or -1       |
     |----------------------------------------|
  1  | symbol table address of the external   |
     |-------------|-----|--------|-----------|
     |             |     |        | relocation|
  2  | DBL opcode  |  s  |        |    type   |
     |             |     |        | indicator |
     |-------------|-----|--------|-----------|
  3  | offset value of external with offset   |
     |----------------------------------------|
```

Word 0: contains the memory address to be fixed up once the symbol value is known or a -1 indicating an empty symbol table entry.

Word 1: pointer to the external symbol's symbol table entry.

Word 2: bits 15-11 contain the instruction opcode value from the DBL or XDBL record.

bits 10-9 S (see below).

bits 3-0 is the relocation indicator from the (X)DBL record indicating the relocation type.

Word 3: Offset value from the (X)DBL record non-zero if the construction is an external reference with offset.

S: Fix up type
S=0: No link information
S=1: CP Link allocated for this instruction
S=2: Fixup to a CP link
S=3: Base page link required for this instruction

Figure 13-1. Fix-Up Table Structure.

Loader Symbol Table (new)

```
       15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
      -------------------------------------------------
LST1 |     Ordinal number              |W |Words(symbl)| 1
      -------------------------------------------------
LST2 |L |A |S     | MR (type)     |I |          |R |V  | 2
      -------------------------------------------------
LST3 |Lib  |       MLS                                 | 3
      -------------------------------------------------
LST4 |             VALUE                               | 4
      -------------------------------------------------
LST5 |             VALUE (for EMA,RPL)                 | 5
      -------------------------------------------------
LST6 |                                                 | 6
     -                                                 -
LST7 |           SYMBOL NAME                           | 7
     -                                                 -
LST8 |             /                                   | 8
     -             \                                   -
LST9 |             /                                   | 9
     -                                                 -
LST10|                                                 | 10
     -                                                 -
LST11|        (variable symbol length)                 | 11
     -                                                 -
LST12|        (max. length is 16 characters)           | 12
     -                                                 -
LST13|                                                 | 13 (maximum)
      -------------------------------------------------
LST14|         Allocation size                         | 14 for ALLOC
      -------------------------------------------------
LST15|         Allocation size                         | 15  entry only
      -------------------------------------------------
```

Word 1: Bit 0-3 # of words of symbol name (2 characters/word)
Bit 4 reserved for weak external flag
Bit 5-15 Ordinal number of the symbol if it is currently being referenced as an external. If the old EXT, the ordinal is bit 0-7 of word 6 of the EXT record. If the extended EXT, the ordinal is bit 0-10 of word 4 of the extended EXT record.

Word 2: Bit 0 the V bit is used to specify the contents of LST4 & LST5.

If V=0, LST4 & LST5 is an absolute address. RP value or Memory address.

If V=1, LST4 & LST5 is a base page link address that may be to reference the ENT or EXT.
Bit 1 the R bit is reserved for the XL loader

```
        Bit 6  the I bit is the initialization bit
               I = 0 this common has not been initialized
               I = 1 this common has been initialized
        Bit 7-11 is the entry point type.
         If the old ENT record, the type is from bit 0-2 of word 6, i.e.:
                        0-program
                        1-base page
                        2-common
                        3-absolute
                        4-RPL
         If the  extended ENT  record, the  type is  bit 0-3  of word  4, MR
         field, i.e.:
                        0-absolute
                        1-program
                        2-base page
                        3-common
                        4-pure code
                        5-local EMA
                        6-SAVE area
         Octal 37 is used for old ENT record RPL
         Bit 12-13  S field indicates the status of the symbol
          S=0, ENT read during a library scan, i.e., defined
          S=1, ENT read during a load operation, i.e., defined
          S=2, EXT entry symbol is still undefined
          S=3, EMA entry. The symbol is considered to be defined. V will =1.
         Bit 14  Allocation  bit.    If  set,  entry  was   defined  through
                 allocation record processing; its size is specified in last
                 two words of entry.  (entry is 2 words longer than normal).
         Bit 15  Reserved for L.LDF

Word 3:  Reserved for MLS use
         LIB = 0 No library information
             = 2 Entry defined in user library
             = 3 Entry defined in system library
         MLS = node # of node in which symbol is defined.

Word 4:  If V=1, contents are base page link address to use
   &     when referencing  EMA variable.
Word 5:  If V=0, defined address of the ENT word 5 is used for new
         format record of EMA and RPL.

Word 6 - Word 13 (word 13 maximum)
      :  Symbol name  character, start from  word 6.  Variable  length; max.
         character length  is limited by 16  (8 words).  LST1 bit  0-4 tells
         how long symbol name is.

Word 14- Word 15 (maximum)
      :  Symbol allocation size.  Last 2 words  of an entry following symbol
         name; occurs for allocate entries only.  Word 2 bit 14 must be set.
```

Figure 13-2. Symbol Table Structure.

Other commands which affect relocation are:

LO to change the load point.

TR to transfer to another command device (this is only legal from an LU, not from a file).

FO to force a load even if undefined externals exist.

DI to display undefined externals.

EN end the load.

E end the load.

EX end the load.

A abort the load.

AB abort the load.

Again, the loader main begins processing and, if necessary, calls appropriate loader library routines.

Upon completion of loading any one module, the routine to print load map information is called. Upon completion of the entire load, other end processing must be done. Final ID segment processing, moving the dummy ID segment to a system ID segment is done. All scratch disc space is released and all files used are closed. If a permanent program was loaded, special permanent program processing is also done. The loader's end message is output and the loader terminates.

The structure of the ID segment is described in Appendix D. Figure 13-3 shows the organization of MLLDR in memory.

```
End of Partition
                +------------------+
                | Dummy Base Page  |
                |------------------|
                | Fixup Table      |
                |------------------|    Dynamic Buffer Area
                |                  |
                |------------------|
                | Symbol Table     |
                |------------------|
                | Path Information |
                |------------------|
                | System Routines  |
                |------------------|
                | $LDRLN code      |
                |------------------|
                | Non-overlayable  |
                |      MLLDR       |
                |      code        |
                |------------------|
                | Overlayed MLLDR  |     Initialization
                |      code        |     Routines
                |------------------|
                | Base Page        |
                +------------------+
               0
```

Figure 13-3. Organization of MLLDR in Memory.

13.1 DATA FORMATS

13.1.1 LOC Data Structure (Disc Resident Nodes)

MLLDR sets up special data structures for MLS-LOC programs required for load on call.

Consider the following partial tree:

```
                        D.1
                   +----------+
                   | Call ABC |
                   | Call GHI |
                   | Call JKL |
                   +----------+
                         |
                 ---------------------
         D.1.1 |                     | D.1.2
        +----------------+ +----------------+
        | Subroutine ABC | | Subroutine JKL |
        | Subroutine GHI | |                |
        +----------------+ +----------------+
```

In this configuration, D.1 calls the three subroutines located down the tree. In order to set up linkages to these routines the loader will append a subroutine to D.1 for each son of D.1 (in this case there are two, D.1.1 and D.1.2). In addition, one word for each subroutine called in D.1 and located in a son of D.1 will be placed at the end of D.1 (in this case there are three). The subroutines are called "Thunks", and the table at the end is called the DEF Transfer table.

The DEF table will contain the address of the thunk routine associated with the node which needs to be brought in from disc. Calls to routines in the node's sons will have been changed by the loader to calls through the local DEF table. The Thunk routines will bring the requested node into memory.

Son nodes are also followed by DEF tables if they in turn have son nodes. In addition, they have DEF tables at the top which will be relocated to the same address as the DEF table at the bottom of the father node. Thus, when a son is brought in from disc, the son's DEF table will overlay the father's DEF table. The son's DEF table will contain DEFs to those routines that it contains and DEFs to thunks for those routines in a different son.

In the example, suppose D.1 and D.1.1 are in memory. The call to ABC will go through the D.1.1 DEF table and go directly to the ABC routine. The call to JKL will go through the D.1.1 DEF table and go to the thunk routine to bring in D.1.2. The call to JKL will be done over again, but this time the D.1.2 DEF table is in memory so the call will go through the D.1.2 DEF table and go directly to the JKL routine.

Thunk and DEF Table Layout (Disc Resident)

```
  *
  *
  *
  JSB D.ABC,I           Call ABC
  *
  *
  *
  JSB D.GHI,I           Call GHI
  *
  *
  *
  JSB D.JKL,I           Call JKL
  *
  *
  *
T.1.1 NOP               Thunk which brings D.1.1 into memory
  *
T.1.2 NOP               Thunk which brings D.1.2 into memory
  *
D.ABC DEF T.1.1
D.GHI DEF T.1.1         DEF Table for subroutines ABC, GHI,
D.JKL DEF T.1.2         JKL
```

If D.1.1 is in memory, the DEF Table is as follows:

```
D.ABC DEF ABC
D.GHI DEF GHI           DEF Table for subroutines ABC, GHI,
D.JKL DEF T.1.2         JKL
```

If D.1.2 is in memory, the DEF Table is as follows:

```
D.ABC DEF T.1.1
D.GHI DEF T.1.1         DEF Table for subroutines ABC, GHI,
D.JKL DEF JKL           JKL
```

DISC RESIDENT MLS

```
                           +--------+
                           | UPPER  |
                           | DEF    |      DMA OVERLAY
                           | TABLE  |      AREA
                           | D.1    |
                     +-----+--------+-----+
                     |                    |
                     |   D.1  CODE AREA   |
                     |                    |
           +--------+ +-----+--------+-----+ +--------+
  DMA      | UPPER  |       | LOWER  |       | UPPER  |
   OVERLAY | DEF    |       | DEF    |       | DEF    |
    AREA   | TABLE  |       | TABLE  |       | TABLE  |
           | D.1.1  |       | D.1    |       | D.1.2  |
     +-----+--------+-----+ +--------+ +-----+--------+-----+
     |                    |            |                    |
     |  D.1.1 CODE AREA   |            |  D.1.2 CODE AREA   |
     |                    |            |                    |
     +-----+--------+-----+            +-----+--------+-----+
           | LOWER  |       DMA              | LOWER  |
           | DEF    |     OVERLAY            | DEF    |
           | TABLE  |      AREA              | TABLE  |
           | D.1.1  |                        | D.1.2  |
           +--------+                        +--------+
```

13.1.2 Memory Resident Nodes

The data structures required for memory resident nodes is similar to those required for disc resident ones. The main difference is that the loader only needs to set up one DEF table per son node, instead of two. The table is aligned to a page boundary and is located at the top of the son node.

The thunk code for memory resident nodes maps in the required node.

THUNK AND DEF TABLE LAYOUT (MEMORY RESIDENT)

```
                M.1
 ----------------------------------
 |                                |
 |       JSB M.ABC,I  (CALL ABC)  |
 |           .                    |
 |           .                    |
 |           .                    |
 |       JSB M.GHI,I  (CALL GHI)  |
 |           .                    |
 |           .                    |
 |           .                    |
 |       JSB M.JKL,I  (CALL JKL)  |
 |           .                    |
 |           .                    |
 |           .                    |
 | T.1.1 NOP                      |
 |      (THUNK CODE FOR M.1.1)    |
 | T.1.2 NOP                      |
 |      (THUNK CODE FOR M.1.2     |
 |                                |
 ----------------------------------

       ALIGN TO PAGE BOUNDARY

M.1.1                 M.1.2
------------------    ------------------  START OF PAGE
|M.ABC DEF ABC   |    |M.ABC DEF T.1.1 |
|M.GHI DEF GHI   |    |M.GHI DEF T.1.1 |
|M.JKL DEF T.1.2 |    |M.JKL DEF JKL   |
|                |    |                |
|ABC   NOP       |    |                |
|       .        |    |JKL   NOP       |
|       .        |    |       .        |
|       .        |    |       .        |
|GHI   NOP       |    |       .        |
|       .        |    |                |
|       .        |    |                |
|       .        |    |                |
|                |    |                |
------------------    ------------------
```

MEMORY RESIDENT MLS

```
                          M
                      ----------
                      |        |
                      |        |                  PAGE
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                      |        |                  BOUNDARY
                      |        |
                      |--------|
                      | THUNKS |
                      ----------
                             ALIGN TO PAGE
                             BOUNDARY

     M.1.1                               M.1.2    PAGE
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     | DEF    |                          | DEF    |  BOUNDARY
     | TABLE  |                          | TABLE  |
     |--------|                          |--------|
     |        |                          |        |
     |        |                          |        |
     |        |                          |        |
     |        |                          |        |
     |        |                          |        |
     |        |                          |        |  PAGE
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     |        |                          |        |  BOUNDARY
     |--------|                          |        |
     | THUNKS |                          |        |
     |        |                          |        |
     ----------                          |--------|
                                         | THUNKS |
                                         ----------
     ALIGN TO                             ALIGN TO
     PAGE                               PAGE BOUNDARY  PAGE
     BOUNDARY
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                   BOUNDARY



                                                  PAGE
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                  BOUNDARY
```

13.1.3 Mixed Memory and Disc Resident Nodes

In the mixed case, the DEF table at the top of all disc resident nodes on the first level (one down from the root) must look like the memory resident nodes. They will start at a page boundary with the same relative start address as the memory resident nodes. For all other disc resident nodes (farther down the tree) the format will be the normal disc resident format.

13.1.4 Rotating Base Page (MLS-LOC Disc Format)

The layout on disc of an MLS-LOC program will depend, in part, on the rotating base page scheme and on which of the leaves of the program tree can share base page.

Consider the following program tree. Three possible program layouts are given showing different rotating base page combinations which could occur.

The actual details of the scheme which gives these results is described in the rotating base page section.

MULTILEVEL SEGMENTATION

```
                         ---------
                         |       |
                         |   M   |
                         |       |
                         ---------
                             |
                             |
         +-------------------+--------------------+
         |                   |                    |
         |                   |                    |
     ---------           ---------            ---------
     |       |           |       |            |       |
     |  M.1  |           |  M.2  |            |  M.3  |
     |       |           |       |            |       |
     ---------           ---------            ---------
         |                                        |
         |                                        |
   ------+------                            ------+------
   |           |                            |           |
   |           |                            |           |
---------   ---------                   ---------   ---------
|       |   |       |                   |       |   |       |
| M.1.1 |   | M.1.2 |                   | M.3.1 |   | M.3.2 |
|       |   |       |                   |       |   |       |
---------   ---------                   ---------   ---------
```

MLS-LOC DISC FORMAT

```
----------------------------------------------------------------------
|BASE  |   |    |      |      |BASE  |    |BASE  |    |      |      |
| PAGE | M | M.1| M.1.1| M.1.2| PAGE | M.2| PAGE | M.3| M.3.1| M.3.2|
| M,M.1|   |    |      |      | M    |    | M,M.3|    |      |      |
| M.1.1|   |    |      |      | M.2  |    | M.3.1|    |      |      |
| M.1.2|   |    |      |      |      |    | M.3.2|    |      |      |
----------------------------------------------------------------------
```

ASSUME MINIMUM BASE PAGE DUE TO ROTATING BASE PAGE

```
----------------------------------------------------------------------
|BASE |   |    |      |      |BASE |    |BASE |    |     |BASE |     |
|PAGE |   |    |      |      |PAGE |    |PAGE |    |     |PAGE |     |
|M,M.1| M | M.1| M.1.1| M.1.2| M   | M.2| M   |M.3|M.3.1| M   |M.3.2|
|M.1.1|   |    |      |      | M.2 |    | M.3 |    |     | M.3 |     |
|M.1.2|   |    |      |      |     |    |M.3.1|    |     |M.3.2|     |
----------------------------------------------------------------------
```

ASSUME ROTATING BASE PAGE WORKED IN THE LEFT
BRANCH BUT FAULTED WHEN RELOCATING M.3.2 IN
THE RIGHT BRANCH

```
----------------------------------------------------------------------
|BASE | |   |     |BASE |     |BASE |   |BASE |   |     |BASE |     |
|PAGE | |   |     |PAGE |     |PAGE |   |PAGE |   |     |PAGE |     |
| M   |M|M.1|M.1.1| M   |M.1.2| M   |M.2| M   |M.3|M.3.1| M   |M.3.2|
| M.1 | |   |     | M.1 |     | M.2 |   | M.3 |   |     | M.3 |     |
|M.1.1| |   |     |M.1.2|     |     |   |M.3.1|   |     |M.3.2|     |
----------------------------------------------------------------------
```

ASSUMES WORST CASE FOR BASE PAGE, THAT IS,
ROTATING BASE PAGE FAULTED EVERY TIME.

13.2 COMMAND FILE

Besides accepting commands found in the run string, the loader accepts commands from a command file. Multi-level programs require a command file for correct loading. Since these programs need to know information about nodes one level down from the one currently being processed, interactive relocation is not possible. (The loader could not "look ahead".) M and D commands specify memory and disc resident nodes for multi-level loads. Commands before any node command may be interactive, but a transfer to a command file must be made before the first M command. (The root of a program must be specified first and must be memory resident.)

The following are loader command file commands that are new or different from those used for LOADR.

LO,+<n> The relocation base is bumped up to the next page boundary if not already on it, and from there is moved to the nth subsequent page boundary, and the unused area is cleared (set to 0).

```
LDB N       get the # of pages to inc
BLF,BLF     put in bits
RBL,RBL     14-10
LDA TH1.L   get the current base
ADA M1777   allign to next page (OCT 1777)
AND M0760   get the current page(OCT 76000)
ADB A       and add the increment
CLA         clear unused area
ADB N1      up to new page (-1)
JSB M.OTB   output routine updates current load address
```

Range checking must also be done.

Note that before this code is executed, n must be checked for negative value. LO,<address> works as before. LO,+0 results in the relocation base being bumped up to the next page boundary.

SZ,+<n> n more pages than the program requires is set as the required program size.

In various stages of the load, size checks are made. If an increment was specified, it must be added to the current number of pages before the check is made. For example, if a partition is specified, the program size must be checked against the partition size. #pgs+pginc must be compared to the partition size. And, the incremental size must be saved in the ID segment and output as the number of pages required in load map message.

SH,<label> EMA area of the program is to be in a systemwide common area (shared). The label identifying this area in the system $EMTB table is <label>.

$EMTB is searched for <label>. If <label> is not found, an error occurs.

PF,<lu#> Specifies that the profiling subroutine is to be appended to the program, <lu#> is the logical unit to which the profile output is to be printed.

See the section on the PF command for further processing done.

Note that PF,<lu#> or OP,DB can be specified, but not both. The last specified will override any previous specification.

SA,<xx> Save command. Reserves <xx> words of local save area for FORTRAN 77 SAVE command use. Sufficient space must be requested for all SAVE areas in the entire program.

When the SAVE command is encountered, a pointer is set to the current relocation address and it is used as the first save area address. The current relocation address is incremented by the size of the save area and this becomes the new relocation address. The end of the save area is kept for overflow checks.

In the event of an overflow, an error message is issued.

The save area is handled as another address space. When the MR field in an extended relocatable record is 6, the record is to be relocated using the save area. The save area pointer for the next available word is updated if an allocation was made. An address is resolved with respect to the start of the save area if a reference is made.

NA,<name> The specified <name> is to be found in one of the libraries declared in an LI command. The routine containing <name> is loaded with the current node. (Note that in Fortran or Pascal, <name> is a subroutine name and in Assembler, <name> is any entry point name.)

To accomplish this load, the following steps are taken. <name> is entered into the loader symbol table if not already there, and is marked as undefined, to be found in the user libraries. When the user defined libraries are searched at the end of the node or because of an SL command, if <name> is found, the routine containing it is relocated. If <name> is not found, an error message or warning is issued and the load aborts or prompts the user.

An NA in a son node will cause a routine to be loaded in the son node, overriding the automatic search of libraries at the end of the father node which would normally cause the routine to be loaded with the father node. That is, the automatic search of a user library at the end of a father node will not cause a routine to be loaded if the routine will be loaded later in a son node as a result of an NA command.

SY,<name> The specified <name> is to be found in the system library. The routine containing <name> is loaded with the current node.

To accomplish this load, the following steps are taken. <name> is entered into the loader symbol table if not already there, and is marked as undefined, to be found in the system library. The system library is searched if a SEARCH command is encountered or the end of node is encountered. If <name> is found, the routine containing it is relocated. If <name> is not found, an error message or warning is issued and the the load aborts or prompts the user.

An SY in a son node will cause a routine to be loaded in the son node, overriding the automatic search of the system library at the end of the father node. That is, the search of the system library at the end of a father node will not cause a routine to be loaded if the routine will be loaded later in a son node because it appears in an SY command.

OP,EB Extended background program. The loadpoint of the program is set to 2000B. The memory protect fence index in the ID SEGMENT is set to 6. If the program uses system common or SSGA, the EB will be overridden and the program type will LB instead.

OP,EM EMA program. If a multilevel program allocates EMA in a node other than the root, and not in the root, the EM command must be used. (note: this applies when using XNAM and ALLOC records). An entry for EMA is forced into the symbol table to assure that it is loaded with the root. Program preamble word XX002 is set to be a DEF to EMA. The processing is done by calling routine M.STE. If EMA is already defined, the DEF to it is output. If not defined, an entry for it is made in the symbol table and a fixup for the DEF is built.

OP,VM VMA program with default VM environment.

```
     working set size = 31
             VMA size = 8192
                        (last page of VMA = 8191)
```

If the EMA size in the relocatable is greater, the VMA size is set to the EMA size. If the EMA size is less than 31, the working set size is set to the EMA size. An entry for VMA is made in the symbol table. The routine M.STE is called and does processing similar to that done for the OP,EM command except that VMA is the routine instead of EMA. The last page of VMA in the ID extension will be set up.

VS,<n> Last page of VMA. <n> must be between 31 and 65535 inclusive. This command overrides the default in an OP,VM command. The last page of VMA is set in the ID extension. If VMA processing has not been done, it is done now. If 175777B < <n> < 177776B then the VS size is set to 177777B.

WS,<n> Working set size. This will override the default in an OP,VM command. <n> must be between 5 and the maximum partition size minus the program code size inclusive. This value is set into the EMA size word in the ID Segment. (The value used is actually <n>+1 to account for 1K of page table needed for VM) If VMA processing has not been done,(ie,M.STE has not been called) then it is done now.

Note: Shareable virtual memory is not supported. That is, use of the SH command and the VM, VS, or WS command will result in a VM EMA error. This condition is caught when both the shareable EMA flag and the VMA flag are set.

13.2.1 Potential Search Command Problem

SE(A)/MS,<namr>; SL; and SEARCH

A conflict arises if a father node contains either a SE(A)/MS, <namr>; an SL;or an SE(A) which will find an entry point X which satisfies an existing external and a son node contains either an NA,X; an SY,X; or an RE,<namr> where <namr> contains the entry point X. X will be relocated in the father since its existence in a son is not known until the end of relocating the father (i.e., when the next node command is seen or the end of load is reached). Now when X is discovered in the son, a duplicate entry point error will occur.

In order for the MLS-LOC scheme to work, all LI commands must appear before the root specification (the M command).

M Memory resident node specification. For each new level a ".n" is appended. Each node at the same level is given a different n (where n starts at one and is incremented by one).

When an M followed by a blank (i.e., a root node) is recognized, a jump to SECHK is taken. If the conflict test of parameters hasn't been done yet, it is done now. Next, the whole command file is checked to assure that all nodes are specified in pre-order and that other commands are valid and occur in the correct order. Then, control is transferred to the node command processor.

When an M followed by a "." is recognized, we know the conflict test must have been done already and we go directly to the node command processor.

The node command processor handles both M commands and D commands. A call is made to LNAMR which parses the M or D command and breaks out level information. Next, a flag is set to indicate memory residence. Various relocation pointers are set, if this is the 1st node at this level or reset, if there were previous nodes at this level. Finally, the path from the root to this node which is kept in memory is updated.

D Disc resident node specification. For each new level a ".n" is appended. Each node at the same level is given a different .n (where n starts at one and is incremented by one).

When a D followed by a "." is recognized, control goes to the node command processor. This time, since a D command is being processed, a flag is set to indicate disc residence. All other steps are done the same as for the M command processing.

13.2.2 Resolving Undefined Externals

Whenever another M or D command is encountered in the command file, the loader must do some special processing to finish relocation of the current node before relocation of the new node can begin.

Any undefined externals at this point were not satisfied in the current node or any of the predecessor nodes on the path back up to the root. The places left to look are in the son nodes, the user libraries, and the system library, in that order.

Son nodes are found by using a scan routine, ISCAN, which is given the location of the current node in the command file (i.e., node #, file DCB, record, block, & offset). The routine returns the location of the next son. For example, given the location of the M command in a file, the location of the M.1 command, if it exists, is returned. (If there is no M.1 command, zeros are returned.) The next call would return the location of the M.2 command and so on as long as they existed.

Now all commands from the current son's node command to the next node command (i.e., all commands for the son) must be processed. All entry points which will be found in this son and that satisfy a currently undefined external must be marked. All commands for the son but RE, NA, and SY can be ignored since these three are the only commands which can cause an entry point to be loaded which satisfies an external of the father's.

Once positioned at a son's node command, the next command is read. If it is not an RE, NA, SY, M, or D command, it is ignored. If an M or D command was read, the son processing is done, and the scan routine, ISCAN, is called to find the next son. If an NA or SY command was read, the named symbol must be processed.

L.ADD is called to see if the symbol is an undefined external which needs to be resolved. If so, the MLS word (word 3) of the symbol table entry is set to the node number of the current son (i.e., the node containing the NA or SY) and the top 2 bits of the MLS word are set to indicate either user library (NA) or system library (SY). The father's ordinal number and the son count (i.e., which son this is, 1,2, etc.) is saved in the symbol value word for later processing. Since the symbol is not defined yet the value is not set yet.

When thunks are built the value words are used and then the DEF table location for the symbol is stored in the value word. If the MLS word is already set, nothing is done since precedence is correct. If an RE was read, the named file must be searched for entry points. This is done by opening the file and doing successive reads, each time seeing if an ENT or XENT record was read. If not, the next record is read. If so, each symbol found needs to be processed as if it were in an NA or SY command.

L.SYE is called and the MLS word for the symbol table entry is set if necessary and the father's ordinal number and son count are saved in the symbol table value words. If the file being relocated is an indexed file, only the file's index or directory needs to be read and searched. Note that if the symbol were found defined, a duplicate entry point error would occur.

All records are read until the EOF is reached or, in the indexed case, the end of the index is reached. Then the next command is processed.

As commands in a son are processed, all entry points found are put in an exception table on the disc. Later, a quick look at the exception table will show what will be defined in a son rather than having to reprocess the command file. Entries in the exception table consist of three words of information, followed by the symbol. The routine M.STB is called to do this.

When all sons have been processed and any undefined external satisfied in a son has had the MLS word in its symbol table entry filled in, the user supplied libraries must be searched.

User libraries are searched as described in the library search section. ENT record processing must change slightly. When processing an entry point symbol in search mode and the symbol is found to be undefined, one more test must be made before this definition is used. The MLS word must be checked. If it is zero, then the entry point is used, if the word is set to some node number then it is checked against the current node number. If equal, the top two bits are checked to see if the proper library is being searched. If so, the entry point is used and the address which was stored in the value word is fixed up to be a DEF to the value just found if there is such an address. If the node numbers do not match, this is not the occurrence of the entry point which is to be used. The typical case is that the entry point will be found in a son which will be relocated later.

If relocating some module in the user libraries caused an undefined symbol to be brought into the symbol table, the exception table must be searched since the symbol may belong in a son. The routine M.XTS is called to search the exception table.

The system library is searched as before, again including a search of the exception table. Now all undefined externals in the symbol table must have their MLS word filled in or else an Undefined External error is issued. An unmarked, undefined entry would mean that the symbol was not found in a son, user library, or in the system library.

Now we must build the DEF table entries and thunk routines for all marked, undefined symbols. Each fixup table entry reference to a symbol is filled in to be an indirect reference through its DEF table entry. The location of the DEF table entry is stored in the symbol's symbol table entry and the DEF table entry is a DEF to the appropriate thunk (where there is one thunk per son node required).

There should be no fixup table entries at this point. All fixups to son nodes will have been changed to go through the DEF table and DEF table locations are now stored in the symbol table entry. Any other fixups (to user or system library) should have been resolved by now.

A traverse routine is called to position the command file pointers to the next node command in pre-order. The routine also supplies the new node's node number and depth in the tree. With this information, the symbol table is packed and all symbols no longer valid are removed. The path information kept in memory is updated. Relocation pointers are set/reset and relocation of the node begins.

Remember that an undef which is found and whose MLS word is set requires a bit more processing. The MLS word must match the current node number and an address stored in the value word (the DEF table entry) must be fixed before the real value is set up.

Force loads require some special processing. When a symbol is found to be undefined, it is given a value of zero and all fixups are done as if the symbol were defined locally. (This would all occur at the end of loading a node.)

M.NOD is called to set up conditions for a node about to be processed. It is called when an M or D command is encountered, after the last node is finished up. The current path information kept in memory must be updated, base page pointers must be set up, the symbol table must be packed, and the base address pointers for this node must be set up.

SETBP sets pointers to allocate base page links for this node. It is part of the rotating base page scheme and is described in more detail under the Rotating Base Page section.

13.2.3 Packing the Symbol Table

A loader library routine, L.PAK, is called to pack the loader symbol table and remove all symbols no longer valid. Given the node number of the node which is going to be processed next and the node number of its father, L.PAK removes all symbols occurring in nodes between the two. Since nodes are loaded and numbered in pre-order, all non-valid symbols will be associated with node numbers between the father's number and the current node's number. All such symbols will have been defined and all fixups to them will have been resolved. Therefore, they can be removed.

In the symbol table, each symbol entry has an MLS word. This word is just the node number of the node in which this symbol is defined. To pack the symbol table, each symbol MLS word is looked at and if it falls between the father's node number and the current node's number, the symbol entry is removed and the remaining entries packed.

Consider the following example:

```
           0 +---+
             |   |
             +---+
               |
           ---------
           |       |
       1 +---+   +---+ 5
         |   |   |   |
         +---+   +---+
           |
   -----------------
2  |    3  |    4  |
 +---+   +---+   +---+
 |   |   |   |   |   |
 +---+   +---+   +---+
```

If node number 4 were being relocated, all symbols associated with nodes 2 and 3 can be deleted since 2 and 3 fall between the current node number, 4, and the current node's father, node number 1. If node number 5 were being relocated, all symbols associated with nodes 1, 2, 3, and 4 could be removed since node 5's father is node 0.

L.PAK processes all symbol table entries until the end of the symbol table is reached. It then updates the pointer to the end of the symbol table. Note that packing the symbol table should not affect any fixup table entries since none should exist at this time.

As an aid to relocation, seven words of information per node in the current path (from root to current node being processed) are kept in memory. The current path must include one node per level, down to the level of the current node.

For example:

```
                         0 +---+
                           |   |
                           +---+
                             |
                  -------------------------
                1 |                       | 5
                +---+                   +---+
                |   |                   |   |
                +---+                   +---+
                  |                       |
          -----------------           ---------
        2 |     3 |     4 |         6 |     7 |
        +---+   +---+   +---+       +---+   +---+
        |   |   |   |   |   |       |   |   |   |
        +---+   +---+   +---+       +---+   +---+
```

The current path when relocating node 3 includes entries for nodes 0, 1, and 3. The current path when relocating node 6 would include entries for node 0, 5, and 6.

The 7-word node entry consists of the following:

1. Ordinal node number
2. Base relocation address for this node
3. Sector offset from start
4. Last + 1 base page location allocated (i.e., next available)
5. Next relocatable address (last + 1)
6. Top of the current page links
7. DEF table addres (disc resident nodes only) address of bottom DEF table sign bit set for memory resident nodes.

When a node is specified, the path in memory is cut back to this node's father. Then, a skeleton entry is made for this node, to be filled in when this node is completely relocated. Values found in the father's entry are used to set the relocation pointers for this entry. The father's next relocatable address is set as this node's first. The dummy current page link area is cut back to the top saved in the father's entry.

The disc location to store the absolute code is aligned to an even sector boundary for disc resident nodes or a page boundary for memory resident nodes.

Now, actual relocation for the specified node can begin.

13.3 SH COMMAND PROCESSOR

Format of command:

```
SH,<label>
```

meaning:

The EMA area of the program is to be in a system-wide common area (shared) and the label identifying the area is <label>.

Processing done:

The system $EMTB table is searched for the label in the SH command. If not found, the loader gives a ?? in interactive mode or an IL PRM error if using a command file. If found, the index number is saved for filling in the ID extension and the size is kept for later size checks.

When actual relocation is about to begin, LU 2 is searched for a file with the same name as the label in the SH command. If one is found further checking is done to see if the 1st line in the file is $SHEMA followed by entries of the form <name>,<size>. If the file is incorrectly constructed, an SH EMA error occurs. Otherwise, all the <name>'s found are put in the symbol table and defined in the order found with the corresponding size. This is the mechanism used to assure that each program using the same name for EMA accesses the same area.

Upon encountering an ALLOC EMA record, all regular EMA processing is done, symbol table entry made if not there, MSEG set up, ID extension checked for. When final ID segment processing is done, an ID extension is allocated and filled in as follows:

Word #0	as before
Word #1	as before
Word #2	COM bit is set to indicate shareable EMA INDEX # set
Word #3	0 (used for VMA)
Word #4	0 (used for VMA)

Index # will be set by the loader to be the index of the $EMTB table entry of <label>.

13.4 PF COMMAND PROCESSOR

Format of command:

```
PF,<lu#>
```

meaning:

Specifies that the profiling subroutine is to be appended to the program. <lu#> is the logical unit to which the profile output is to be printed. The profiling routine will count the number of disc and memory map faults for each node.

Processing done:

When the loader encounters a PF command, it must generate a special table, generate modified thunks, append the profiling subroutine to the program, set up program location 7, and set a special bit, DE bit, in the program's ID segment to indicate that the profiling routine is to be run upon program termination.

The special table is located at the beginning of the program and is used to store the number of times each node was called via a fault. The format of the table is as follows:

```
NAM .PTBL
ENT .PLU#,.PADR,.#NOD

.PLU# NOP       lu# to which profile output goes
.#NOD OCT N     N will be the # of nodes
.PADR DEF *+1   address of the table
      BSS N
      END
```

.PLU# is set to the <lu#> specified in the PF command. By scanning the entire command file and counting each M and D command, "N", the number of nodes can be calculated and the rest of the table can be set up. .PLU#, .#NOD, .PADR will be entered into the loader symbol table. They are referenced by the profiling subroutine.

The thunks must be modified to not only bring in the required code, but to increment the appropriate fault count, too. Profile thunks will be modified as follows:

```
      EXT .PADR
THUNK regular thunk code
      LDA .NOD#            Count the
      ADA .PADR             fault in the
```

```
      ISZ A,I              .PTBL table
      JMP THUNK,I

.NOD# OCT SEQ#            Sequence # of node.
```

This will increment the count in the .PTBL table for the node just mapped in.

The load of the profiling subroutine is forced by entering .STAR in the symbol table when the PF command is encountered. The profiling subroutine will be loaded in the root and look as follows:

```
    EXT $CVT3,EXEC
    EXT .PLU#,.PADR,.#NOD
    ENT .STAR

.STAR LDA .#NOD             Set
      SZA,RSS                (ANY NODES)
      JMP TERM               (NOPE, FORGET THE WHOLE THING)
      CMA,INA                up
      STA KOUNT               loop
      CLA,CCE
      STA CURND
LOOP  LDA CURND                 Get node #
      CCE
      JSB $CVT3                 Convert to ASCII
      LDB A,I
      STB BUF+4                 Place in
      INA                        Output
      DLD A,I                     Buffer
      DST BUF+5
      LDA .PADR,I               Get # of faults for
      JSB $CVT3                 this node.  Convert to
      LDB A,I                   ASCII and
      STB BUF+12                 place
      CCE,INA                     in
     DLD A,I                      buffer
     DST BUF+13
     JSB EXEC                 Write out
     DEF *+5                  info on this
     DEF D2                   node
     DEF .PLU#
     DEF BUF
     DEF D18
```

```
      ISZ .PADR                 Bump pointer
      ISZ .CURND                Bump node #
      ISZ KOUNT                 Done?
      JMP LOOP                  No

TERM  JSB EXEC                  Terminate for
      DEF *+2                   real
      DEF ENDIT

KOUNT NOP
CURND NOP
BUF   ASC 18,...NODE #XXXXXX FAULTED  XXXXXX TIMES
```

A deferred EXEC 6 request will be set up by the loader by setting bit 6 of word 21 of the ID segment when terminating. Program word XX007 is set to the address of .STAR.

When the program terminates with its EXEC 6 request, the termination will be delayed and control transferred to the profiling subroutine that was loaded with the root. Program location XX007 will be the transfer address of the .STAR routine. The EXEC 6 request processor will check bit 6 of word 21 of the program's ID segment. If set, it will load the address contained in program location XX007 and place this into the program's point of suspension. The EXEC 6 request in the profiling subroutine has bits 15 and 14 set. This will cause a normal termination of the MLS-LOC program.

13.5 BUILDING .PTBL

.PTBL is built by the routine M.BLD. M.BLD takes the current relocation address, the profile lu # and the number of nodes as parameters. When M.BLD is done, absolute code has been output and symbol table entries made, as if the following had been relocated.

```
       ENT .PLU,.#NOD,.PADR
.PLU  <profile lu number>
.#NOD  <number of nodes>
.PADR DEF *+1
       BSS N                    where N = # of nodes
       END
```

13.6 DB COMMAND PROCESSOR

Format of command:

    OP,DB

Meaning:

Append the DBUG subroutine to the program.

Processing done:

Upon encountering the OP,DB command, all regular debug processing is done, an entry is made in the symbol table, and the transfer address of the program will be changed to be the transfer address of the debug routine. Additionally, all disc and memory resident thunks must call .DML1 on every fault, and .DML2 after the map or disc load has completed. To accomplish this, thunk code for a program with debug appended will be modified to do a JSB .DML1 after saving the registers and to do a JSB .DML2 after the map or disc load, before restoring the registers.

13.7 LIBRARY SEARCH

MLLDR allows user defined libraries to be specified and searched for undefined externals. These libraries may be indexed or may be just a series of merged relocatables. For either form, there are two cases to consider. The first case is when the library appears in an SE(A) command or MS command. The second case is when the library appears in an LI command and is searched later with an SL command.

SE and MS processing involves opening and searching only the named library and when done, closing it and processing the next command. The major part of the LI processing does not occur when the LI command is encountered. Rather, it occurs at the end of loading a node when an automatic search of all libraries specified in LI commands is invoked or when an SL command occurs. In these cases, each library is opened, searched, and closed until all such libraries have been processed.

When an SE or MS command is encountered, the specified file is opened. L.LUN (a loader library routine) is called to search the loader symbol table for undefined externals (starting from the beginning of the symbol table). If none are found, the file is closed. If undefs exist, L.SER (the indexed file search routine) is called. Upon return, L.SER indicates if the file is indeed indexed. If not, RWNDL rewinds the file, sets a few flags, and jumps to RREAD to do a regular read from the beginning of the file. If L.SER found an indexed file, but it did not find the undef that L.LUN found, L.LUN is called again to find the next undef and the loop is repeated. If the symbol was found, L.SER returns the start location of the module containing the symbol.

APOSN is called to position the file and RREAD reads in the routine. Upon encountering the END record of the routine, a check is made to see if an indexed library was being processed. If so, L.LUN is called again to find the next undef and the loop repeats. Otherwise, RREAD is called to read in the next sequential record. When the end of file is reached, a check is made to see if an SE/MS or SL was being processed. If an SE/MS command was being processed, a check is made to see if multiple passes are to be performed. If not, the file is closed. If so, another check is made to see if anything was loaded on the previous pass. If not, the file is closed. If so, and the file is not indexed, a jump to label DUMMY occurs to reposition the file to the beginning, and to begin reading again.

Note that an end of file should never be reached when processing an indexed file. On an indexed file, eventually L.LUN will find no more undefs. If an SE/MS command was being processed, a check is made to see if multiple passes are to be performed. If not, the file is closed. If so, another check is made to see if anything was loaded on the previous pass. If not, the file is closed. If so, PNTR is reset to start the undef search for the beginning of the symbol table and the L.LUN loop is called again.

When the end of a node is encountered, an automatic search of all libraries specified in LI commands is invoked.

L.LUN is called to see if any undefined externals exist. If so, a check is made to see if there are any libraries to search. If so, the first library file is opened, a number of flags are set, PNTR is reinitialized to start at the beginning to search for undefs, and a JMP is made to the L.LUN call in the previous loop. In the indexed file case, L.LUN is finally called and no undefined externals exist and it was an LI being processed. A check is made to see if anything was loaded in the previous pass. If so, we reinitialize PNTR to search for undefs from the beginning and again call the L.LUN loop. Note it has already been established that an indexed file was being searched. If nothing was loaded, we close the file, clear the indexed library flag, and if there are other user defined libraries, we open the next one and again call L.LUN.

Another place where LI/SL processing differs from SE/MS processing is when an end of file is reached. Then a check is made to see if any routines were loaded in the last pass. If so, the file is repositioned to the beginning and read again. Note that end of file is reached only for non-indexed files, therefore there is no index to be re-examined. If no routines were loaded, the file is closed. If undefined externals still exist and there are more libraries, the next library is opened and the L.LUN loop called again.

When a search (indexed or not) finds an entry point which matches an entry in the symbol table, two cases can occur. The symbol can be defined or undefined. If defined, the symbol is set aside to indicate that a potential duplicate entry point exists. When the end record for the module containing the entry point is reached, a check is made to see if the module was really loaded. If so, and a symbol was set aside, a duplicate entry point error occurs. If no symbol was set aside, no error occurs. And, if the module was not loaded, any set aside symbol is cleared.

If the symbol was found to be undefined, the MLS word of its symbol table entry is checked. The MLS word must be zero or equal the current node number in order to cause the module to be loaded. If the words are not equal, or the MLS word is not zero, or the top 2 bits indicate a different library (system or user), then this is not considered a match. If the MLS word equals the current node number, any value stored in the value word of the symbol table entry is really an address to fix up. The address is the top DEF table entry for the symbol. This address must be fixed up and then the symbol's real value can be filled into its symbol table entry. If the MLS word is zero, normal processing occurs.

MLLDR uses a hashed system library search routine to search the system library for undefined externals which exist in a program. On boot up, file $SYENT is built on LU 2. MLLDR actually uses this hashed file to find entries in the system library. If, for some reason the file is purged, RU,LOADR,-1,-1 will rebuild it. Note that MLLDR cannot build this file. Assuming the file is good, the module M.SNP retrieves entries from the system library using M.HSH to hash the symbol being processed.

13.8 LOADING DISC RESIDENT NODES

Each node that has son nodes will have a DEF transfer table at its end. Each node that has a father (i.e., all nodes but the root) will have a DEF transfer table at its beginning.

The DEF transfer table at the end of a node has one entry per external reference satisfied in a son node. This entry is the address of the thunk routine associated with the son node which must be brought in from disc. All fixups to the external are changed to go indirect through the DEF table. The DEF table can immediately be filled in since all thunks will have been relocated already. The DEF table will be built in the same order the external references appear in the symbol table. By scanning the loader command file, the node which contains a particular entry point satisfying this node's external can be found and the sequence number of the son node can be placed in the symbol's symbol table entry. By incrementing the thunk pointer (which points to the first thunk in the current node), by the thunk length found in the first word of each thunk, the correct thunk address can be found. Note that while doing thunk loading, base page linking only is allowed. This is so incrementing the thunk pointer is simplified. After the thunks are loaded, current page linking is reset (if that is what was being used)

The DEF transfer table at the beginning of a node is built to overlay the DEF table at the end of its father when it is brought in from the disc.

The father's bottom DEF table is read in as a basis for each son's top DEF table. As symbols are found in the son, associated DEFs are changed. Each entry in the DEF table of the father caused the associated symbol in the symbol table to be marked with the node number of the son in which it occurs. Furthermore, the "value" of the symbol is set to the address of its DEF table entry. When a symbol is actually defined, the DEF table entry will be fixed to be a DEF to the symbol. This is done by finding the symbol in the symbol table, seeing that it belongs in the current node being processed and that it is currently undefined. The DEF table address is taken out of the value word in the symbol table and is fixed using the real value just defined. Then the real value is set into the value word, the symbol is marked as defined, and any further fixups will be done normally.

In order to access routines in son nodes, the loader appends a subroutine for each son node referenced in the father node. The appended subroutines, called THUNKS, bring the requested node into memory. A JSB to a routine in the son will be changed to a JSB indirect to the DEF table. The DEF will be to the appropriate THUNK which will bring in the required node and re-execute the JSB through the DEF table. Now the son node is in memory and the DEF table entry will be a DEF to the correct routine which is now in memory.

The thunks are relocated at the end of a node, after all program code and before the DEF table. By scanning the command file, the sons can be found and THUNKS will be relocated in pre-order. Also, the number of thunks can be found. For each son, the following thunk is built:

```
      EXT $LOD$
      ENT $DTHK
$DTHK OCT 10          length of thunk
      JSB $LOD$       bring in the disc resident node
.DTAB OCT STADR       start addr. of son node
                      FWA of DEF table at top
      OCT LWAPG       last word + 1 of son node
                      lwa of bottom def table
      OCT PDISC       relative sector # from start
                      for code of son node
      OCT BDISC       relative sector # from start
                      for base page of son node
.ORD  <NUMBER>        this node's leaf #
.NOD# <NUMBER>        this node's ordinal #
      END
```

The entries in .DTAB are fixed up and filled in as each son node is relocated.

.DTAB of a particular thunk cannot be filled in until the node which corresponds to that thunk is relocated. Consequently, MLLDR must keep track of the locations of all the .DTABs in all the thunks until each corresponding node is relocated. Since the number of thunks in a program varies, a dynamic scheme is used for recording the locations of the .DTABs. The scheme takes advantage of the fact that until each .DTAB is filled in, space is available there to store location information. Thus, the .DTAB locations are linked through the .DTABs themselves. The last .DTAB in a node is linked back to the next .DTAB in that node's father (i.e., the .DTAB in the thunk following this node's thunk in the father). Thus, whenever all .DTABs in a node are filled in, the next .DTAB to fill in can be found since it is simply the next .DTAB in the father.

As nodes are relocated in pre-order, if a node is not a leaf, it will also have thunks with .DTABs which will have to be fixed up. And, since nodes are relocated in pre-order, all of this node's thunks will be fixed up before the next thunk in the father will need to be fixed. Therefore, the loader needs to know the location of the next thunk in a father node when the last thunk in a son node has been fixed up. This is exactly what is accomplished by the link stored in the last .DTAB of a node. Otherwise, the next thunk to fix up is just the next one in the current node.

The following scheme is set up to simplify the .DTAB fixups which must be made. DTABN points to the .DTAB of the thunk of the next node to be loaded. This node can be found since nodes are loaded in pre-order. DTABP is the previous value of DTABN. That is, each time DTABN is upated, DTABP gets DTABN's value before the update. The .DTAB table in the last THUNK of a son points back to the next THUNK of the father (i.e., the one after this son's THUNK) DTABP + THUNK length. The last thunk can easily be found since the total number is known. Since the .DTAB is not filled in at this point, there is no problem storing other information there, as long as that information is used before the table is filled in. As .DTABs are filled in, if the .DTAB contained a pointer back to the father, DTABN is set to point back to the father so that information is not lost. The pointer consists of two words, the word offset into the sector and the relative sector # from the start.

Consider the following example:

```
                 A
                +---+
                | 1 |
                +---+
                  |
           -----------------
       B   |    C   |    D   |
         +---+    +---+    +---+
         | 2 |    | 6 |    | 7 |
         +---+    +---+    +---+
           |
   -----------------
 E   |    F   |    G   |
   +---+    +---+    +---+
   | 3 |    | 4 |    | 5 |
   +---+    +---+    +---+
```

```
RE,A
          +------------+
          | 1          |
          |            |        DTABP = NIL
          |------------|
DTABN ->  | THUNK 2    |
          |------------|
          | THUNK 6    |
          |------------|
          | THUNK 7    |
          |------------|
          | DEF TABLE  |
          +------------+
```

RE,B

```
                                     +-----------+
                                     | 1         |
                                     |-----------|
                           DTABP ->  | THUNK 2   |
                                     |-----------|
                                  -> | THUNK 6   |
                                  |  |-----------|
                                  |  | THUNK 7   |
                                  |  |-----------|
                                  |  | DEF TABLE |
                                  |  +-----------+
         +-----------+            |
         | 2         |            |
         |-----------|            |
DTABN -> | THUNK 3   |            |
         |-----------|            |
         | THUNK 4   |            |
         |-----------|            |
         | THUNK 5   |------------|
         |-----------|
         | DEF TABLE |
         +-----------+
```

Fill in THUNK 2s DTABL

```
RE,E                          +------------+
                              | 1          |
                              |------------|
                              | THUNK 2    |
                              |------------|
                           |->| THUNK 6    |
                           |  |------------|
                           |  | THUNK 7    |
                           |  |------------|
                           |  | DEF TABLE  |
                           |  +------------+
         +------------+    |
         | 2          |    |
         |------------|    |
DTABP -> | THUNK 3    |    |
         |------------|    |
DTABN -> | THUNK 4    |    |
         |------------|    |
         | THUNK 5    |-|
         |------------|
         | DEF TABLE  |
         +------------+
+--------+
| 3      |
+--------+
```

Fill in THUNK 3s DTABL

```
RE,F                      +-----------+
                          | 1         |
                          |-----------|
                          | THUNK 2   |
                          |-----------|
                       |->| THUNK 6   |
                       |  |-----------|
                       |  | THUNK 7   |
                       |  |-----------|
                       |  | DEF TABLE |
                       |  +-----------+
         +-----------+ |
         | 2         | |
         |-----------| |
         | THUNK 3   | |
         |-----------| |
DTABP -> | THUNK 4   | |
         |-----------| |
DTABN -> | THUNK 5   |-|
         |-----------|
         | DEF TABLE |
         +-----------+
+-------+            +-------+
| 3     |            | 4     |
+-------+            +-------+
```

Fill in THUNK 4s DTABL

```
RE,G                          +------------+
                              | 1          |
                              |------------|
                              | THUNK 2    |
                              |------------|
                  DTABN ->    | THUNK 6    |
                              |------------|
                              | THUNK 7    |
                              |------------|
                              | DEF TABLE  |
                              +------------+
               +------------+
               | 2          |
               |------------|
               | THUNK 3    |
               |------------|
               | THUNK 4    |
               |------------|
DTABP ->       | THUNK 5    |
               |------------|
               | DEF TABLE  |
               +------------+
+--------+     +--------+     +--------+
| 3      |     | 4      |     | 5      |
+--------+     +--------+     +--------+
```

Fill in THUNK 5s DTABL
(change DTABN before)

```
RE,C                           +-----------+
                               | 1         |
                               |-----------|
                               | THUNK 2   |
                               |-----------|
                  DTABP ->     | THUNK 6   |
                               |-----------|
                  DTABN ->     | THUNK 7   |
                               |-----------|
                               | DEF TABLE |
                               +-----------+
            +-----------+              +-------+
            | 2         |              | 6     |
            |-----------|              +-------+
            | THUNK 3   |
            |-----------|
            | THUNK 4   |
            |-----------|
            | THUNK 5   |
            |-----------|
            | DEF TABLE |
            +-----------+
+-------+      +-------+      +-------+
| 3     |      | 4     |      | 5     |
+-------+      +-------+      +-------+
```

Fill in THUNK 6s DTABL

```
RE,D                    +------------+
                        | 1          |    DTABN = NIL
                        |------------|
                        | THUNK 2    |
                        |------------|
                        | THUNK 6    |
                        |------------|
            DTABP ->    | THUNK 7    |
                        |------------|
                        | DEF TABLE  |
                        +------------+
           +------------+       +--------+        +--------+
           | 2          |       | 6      |        | 7      |
           |------------|       +--------+        +--------+
           | THUNK 3    |
           |------------|
           | THUNK 4    |
           |------------|
           | THUNK 5    |
           |------------|
           | DEF TABLE  |
           +------------+
  +--------+     +--------+     +--------+
  | 3      |     | 4      |     | 5      |
  +--------+     +--------+     +--------+
```

Fill in THUNK 7s DTABL

13.9 BUILDING THE THUNK

MLLDR must build one kind of THUNK for disc resident nodes and another kind for memory resident nodes. In addition, it must build modified thunks if DEBUG is to be appended and it must build differently modified thunks if the profile option was specified.

The library $MLSLB contains the six different node load routines required by the THUNKS for the six different kinds of node loads. Depending on if the node being brought in is disc resident or memory resident, and if the program is to have debug appended, the profile routine appended or neither appended, the appropriate routine is used in a THUNK. The six routines have different entry point names so the appropriate name is put into the loader's symbol table as an undefined external at the beginning of relocation. The system library (containing the routines) is searched and the correct routines are loaded with the root. The six routines are grouped into three modules, debug to be appended ($LOCD,$LODD), profile to be appended ($LOCP,$LODP), neither to be appended (LOC,LOD). Whichever are loaded, the names are changed to LOC and LOD at the end of root processing so that later THUNK processing only has to contend with two names.

When a THUNK is built the following template (to be filled in when the actual node is relocated) is output

```
MEMORY              DISK

OCT 10              OCT 10
JSB $LOC$           JSB $LOD$
NOP                 NOP
NOP                 NOP
NOP                 NOP
DEF .CNOD+0         NOP
NOP                 NOP
NOP                 NOP
```

For the JSB LOD and LOC a JSB to base page indirect (JSB BP,I) is built and a base page link is allocated once for the LOD and once for the LOC. The link is used for all THUNKS once one THUNK is set up. (Remember that LOC and LOD are really the appropriate routines with the name changed if necessary.)

```
ASMB,R,L,C,Q   ** $LOC$ -- RTE-6/VM LOAD ON CALL PREAMBLE
*
*
      HED $LOC$ -- RTE-6/VM
*     DATE:   11/19/80
*     NAME:   $LOC$
*     SOURCE: 92084-18415
*     RELOC:  PART OF 92084-12015
*      PGMR:   C.M.M.
*
*   ***************************************************************
*   * (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1980.  ALL RIGHTS     *
*   * RESERVED.  NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,      *
*   * REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT*
*   * THE PRIOR WRITTEN CONSENT OF HEWLETT-PACKARD COMPANY.       *
*   ***************************************************************
*
      NAM $LOC$,7 92084-1X415 REV.2121 810723
*
*
      ENT $LOC$,$LOD$
      EXT .SVRG,.RRGR,$LOC,.CNOD,EXEC
*
*
*
*  *     CALLING SEQUENCE :
*  *
*  $MTHK NOP           LENGTH OF THUNK
*        JSB $LOC$     GO TO MICRO CODE TO DO THE REMAP
*  .DTAB OCT 0         LOG PG# AT WHICH THIS NODE BEGINS (0-31)
*        OCT 0         REL PG# FRM BGN PTTN OF NODE STRT 0-1023
*        OCT 0         REL PG# FRM BGN PTTN OF BASE PAGE 0-1023
*        DEF .CNOD+0   ADDRESS OF CURRENT PATH WORD..
*  .ORD  NOP           THIS NODES LEAF NODE #  (IE PATH #)
*  .NOD# NOP           THIS NODES ORDINAL #
*
*
*
$LOC$ NOP
      JSB .SVRG     SAVE OUR REGISTERS
*
      LDB J$LOC      OK, SO LETS SEE
      CPB 0105X        IF WE HAVE MICRO CODE IN THIS BOX
      JMP EXIT1          YES !!!!!!!
*
      LDA $LOC$          GET THE SOURCE ADDRESS
      LDB DEST           AND THE DEST ADDR
      MVW D6             MOVE 6 WORDS
*
      JMP *+2
*
```

```
$THNK DEF BACK
*
J$LOC JSB $LOC       NO, SO DO THE SOFTWARE $LOC CALL
.DTAB NOP
      NOP
      NOP
CNODE DEF .CNOD+0
.ORD  NOP            PATH #
.NOD# NOP            THIS NODES ORD #
*
      LDB $LOC$      OK, SO CALCULATE OUR RETURN ADDRESS
BACK  ADB DM2          FROM THE ORGINAL CALL
      CCA
      ADA B,I          A = RETURN ADDRESS
      JMP EXIT2      SO RETURN ALREADY .
*
EXIT1 CCA            SO,LETS PATCH UP THE CALL
      ADA $LOC$
      STB A,I          SET THE MICRO OP INTO THE ORIGINAL CALL
*
EXIT2 STA $LOC$      FIX UP OUR RETURN ADDRESS
      JSB .RRGR      RESTORE THE REGISTERS
      JMP $LOC$,I      AND RETURN
*
*
A     EQU 0
B     EQU 1
DM2   DEC -2
D6    DEC 6
DEST  DEF .DTAB
0105X OCT 105241    OCT CODE FOR MICRO CODED $LOC
*
*
      HED  RTE-6/VM    **  DISC NODE LOAD CODE  **
*
*
*
* THIS SECTION CALLS THE EXEC TO PERFORM THE DISC NODE LOAD
* THE O.S. WILL CHECK FOR THIS NEW TYPE OF NODE LOAD CALL
* AND PERFORM THE APPROPRIATE CHECKS. IT RETURNS TO THE
* FIRST WORD AFTER THE EXEC CALL.
*
*
*     CALLING SEQUENCE :
*
*  $DTHK NOP            LENGTH OF THUNK
*        JSB $LOD$      BRING IN THE DISC RESIDENT NODE
*  .DTAB NOP            STRT ADDR SON NODE, FWA OF TOP DEF TABLE
*        NOP            LST WRD+1 SON NODE, LWA OF BTM DEF TABLE
*        NOP            REL SECT # FROM STRT FOR SON NODE'S CODE
*        NOP            REL SECT # FROM STRT FOR SON'S BASE PAGE
```

```
*  .ORD  NOP           THIS NODES LEAF #  (PATH #)
*  .NOD# NOP           THIS NODES ORDINAL #
*
*
 *
 *   EXEC WILL DO THE FOLLOWING :
 *
 *        1.  DO NODE LOAD FOR CODE AND BASE PAGE
 *        2.  UPDATE CURRENT PATH #
 *        3.  RETURN TO    CONTENTS (CONTENTS ( 4TH PRAM) ) -1
 *
 *
 *
 *
 $LOD$ NOP
       JSB EXEC       CALL EXEC TO DO THE NODE LOAD
       DEF *+5
       DEF D8         MAKE IT LOOK LIKE A SEG LOAD
       DEF EXEC+0      SECURITY CODE. THIS IS CHECKED IN EXEC
       DEF $LOD$,I    PASS IN THE .DTAB ADDRESS
       DEF $LOD$        PASS IN ADDRESS OF RETURN ADDRESS +1
 *
 *
 D8    DEC 8
 *
 *
       END
 *
```

```
 ASMB,R,L,C,Q    ** $LOCP -- RTE-6/VM LOAD ON CALL PROFILE ROUTINE
 *
 *
       HED $LOCP -- RTE-6/VM  LOAD ON CALL PROFILE ROUTINE
 *     DATE:    11/19/80
 *     NAME:    $LOCP
 *     SOURCE: 92084-18417
 *     RELOC:   PART OF 92084-12015
 *     PGMR:    C.M.M.
 *
 *   ***************************************************************
 *   * (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1980.  ALL RIGHTS     *
 *   * RESERVED.  NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,      *
 *   * REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT*
 *   * THE PRIOR WRITTEN CONSENT OF HEWLETT-PACKARD COMPANY.       *
 *   ***************************************************************
 *
       NAM $LOCP,7 92084-1X417 REV.2121 810723
 *
 *
       ENT $LOCP,$LODP
       EXT .SVRG,.RRGR,$LOC,.CNOD,EXEC
       EXT .PADR
 *
 *
 *
 *   *      CALLING SEQUENCE :
 *   *
 *   $MTHK NOP            LENGTH OF THUNK
 *         JSB $LOC$      GO TO MICRO CODE TO DO THE REMAP
 *   .DTAB OCT 0          LOG PG# AT WHICH THIS NODE BEGINS (0-31)
 *         OCT 0          REL PG# FRM BGN PTTN OF NODE STRT 0-1023
 *         OCT 0          REL PG# FRM BGN PTTN OF BASE PAGE 0-1023
 *         DEF .CNOD      ADDRESS OF CURRENT PATH WORD .
 *   .ORD  NOP            THIS NODES LEAF NODE #  (IE PATH #)
 *   .NOD# NOP            THIS NODES ORDINAL #
 *
 *
*
$LOCP NOP
      JSB .SVRG     SAVE OUR REGISTERS
*
      LDA $LOCP           GET THE SOURCE ADDRESS
      LDB DEST            AND THE DEST ADDR
      MVW D6              MOVE 6 WORDS
*
      JMP *+2
*
$THNK DEF BACK
*
J$LOC JSB $LOC       NO, SO DO THE SOFTWARE $LOC CALL
```

```
.DTAB NOP
      NOP
      NOP
CNODE DEF .CNOD+0
.ORD  NOP             PATH #
.NOD# NOP             THIS NODES ORD #
*
      LDB $LOCP       OK, SO CALCULATE OUR RETURN ADDRESS
BACK  ADB DM2           FROM THE ORGINAL CALL
      CCA
      ADA B,I           A = RETURN ADDRESS
      STA $LOCP       FIX UP OUR RETURN ADDRESS
*
      ADB D7          GET THIS NODES ORDINAL #
      LDA B,I
      ADA .PADR        ADD IN BASE ADDRESS OF FAULT COUNT TABLE
      ISZ A,I           AND INCREMENT THE COUNT
      NOP                'OPPS'  ????
*
      JSB .RRGR       RESTORE THE REGISTERS
      JMP $LOCP,I       AND RETURN
*
*
A     EQU 0
B     EQU 1
DM2   DEC -2
D6    DEC 6
D7    DEC 7
DEST  DEF .DTAB
*
*
      HED  RTE-6/VM    **  DISC NODE LOAD CODE  **
*
*
*
* THIS SECTION CALLS THE EXEC TO PERFORM THE DISC NODE LOAD
* THE O.S. WILL CHECK FOR THIS NEW TYPE OF NODE LOAD CALL
* AND PERFORM THE APPROPRIATE CHECKS. IT RETURNS TO THE
* FIRST WORD AFTER THE EXEC CALL.
*
*
*      CALLING SEQUENCE :
*
*  $DTHK NOP             LENGTH OF THUNK
*        JSB $LOD$       BRING IN THE DISC RESIDENT NODE
*  .DTAB NOP             STRT ADDR SON NODE, FWA OF TOP DEF TABLE
*        NOP             LST WRD+1 SON NODE, LWA OF BTM DEF TABLE
*        NOP             REL SECT # FROM STRT FOR SON NODE'S CODE
*        NOP             REL SECT # FROM STRT FOR SON'S BASE PAGE
*  .ORD  NOP             THIS NODES LEAF #  (PATH #)
*  .NOD# NOP             THIS NODES ORDINAL #
```

```
*
*
*   EXEC WILL DO THE FOLLOWING :
*
*        1.  DO NODE LOAD FOR CODE AND BASE PAGE
*        2.  UPDATE CURRENT PATH #
*        3.  RETURN TO    CONTENTS (CONTENTS ( 4TH PRAM) ) -1
*
*
*
*
$LODP NOP
      JSB .SVRG      SAVE OUR REGISTERS
*
      JSB EXEC       CALL EXEC TO DO THE NODE LOAD
      DEF *+5
      DEF D8         MAKE IT LOOK LIKE A SEG LOAD
      DEF EXEC+0
      DEF $LODP,I    PASS IN THE .DTAB ADDRESS
      DEF *+1        SET UP ADDRESS TO RETURN TO
      DEF *+2
*
      LDB $LODP     OK SO NOW RETURN
      JMP BACK       TO THE CALLER
*
D8    DEC 8
*
*
      END
*
```

```
ASMB,R,L,C,Q   ** $LOCD -- RTE-6/VM LOAD ON CALL DEBUG ROUTINE
*
*
      HED $LOCD -- RTE-6/VM  LOAD ON CALL DEBUG ROUTINE
*     DATE:   11/19/80
*     NAME:   $LOCD
*     SOURCE: 92084-18416
*     RELOC:  PART OF 92084-12015
*     PGMR:   C.M.M.
*
*  ***************************************************************
*  * (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1980.  ALL RIGHTS     *
*  * RESERVED.  NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,      *
*  * REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT*
*  * THE PRIOR WRITTEN CONSENT OF HEWLETT-PACKARD COMPANY.       *
*  ***************************************************************
*
      NAM $LOCD,7 92084-1X416 REV.2121 810723
*
*
      ENT $LOCD,$LODD
      EXT .SVRG,.RRGR,$LOC,.CNOD,EXEC
      EXT .DML1,.DML2
*
*
*
*  *     CALLING SEQUENCE :
*  *
*  $MTHK NOP             LENGTH OF THUNK
*        JSB $LOCD       GO TO MICRO CODE TO DO THE REMAP
*  .DTAB OCT 0           LOG PG# AT WHICH THIS NODE BEGINS (0-31)
*        OCT 0           REL PG# FRM BGN PTTN OF NODE STRT 0-1023
*        OCT 0           REL PG# FRM BGN PTTN OF BASE PAGE 0-1023
*        DEF .CNOD       ADDRESS OF CURRENT PATH.
*  .ORD  NOP             THIS NODES LEAF NODE #  (IE PATH #)
*  .NOD# NOP             THIS NODES ORDINAL #
*
*
*
$LOCD NOP
      JSB .SVRG     SAVE OUR REGISTERS
*
      JSB .DML1      OK, SO LETS TELL DBUGR THINGS ARE GOING TO CHANGE
      DEF $LOCD      AND PASS HIM AN ADDRESS TOO.
*
      LDA $LOCD         GET THE SOURCE ADDRESS
      LDB DEST          AND THE DEST ADDR
      MVW D6            MOVE 6 WORDS
*
      JMP *+2
*
```

```
$THNK DEF BACK
*
J$LOC JSB $LOC       NO, SO DO THE SOFTWARE $LOC CALL
.DTAB NOP
      NOP
      NOP
CNODE DEF .CNOD+0
.ORD  NOP            PATH #
.NOD# NOP            THIS NODES ORD #
*
      LDB $LOCD      OK, SO CALCULATE OUR RETURN ADDRESS
BACK  ADB DM2          FROM THE ORGINAL CALL
      CCA
      ADA B,I          A = RETURN ADDRESS
      STA $LOCD      FIX UP OUR RETURN ADDRESS
*
      JSB .DML2     TELL DBUGR WE'RE BACK
*
      JSB .RRGR      RESTORE THE REGISTERS
      JMP $LOCD,I      AND RETURN
*
*
A     EQU 0
B     EQU 1
DM2   DEC -2
D6    DEC 6
DEST  DEF .DTAB
*
*
      HED  RTE-6/VM    **  DISC NODE LOAD CODE  **
*
*
*
* THIS SECTION CALLS THE EXEC TO PERFORM THE DISC NODE LOAD
* THE O.S. WILL CHECK FOR THIS NEW TYPE OF NODE LOAD CALL
* AND PERFORM THE APPROPRIATE CHECKS. IT RETURNS TO THE
* FIRST WORD AFTER THE EXEC CALL.
*
*
*      CALLING SEQUENCE :
*
*  $DTHK NOP           LENGTH OF THUNK
*        JSB $LODD     BRING IN THE DISC RESIDENT NODE
*  .DTAB NOP           STRT ADDR SON NODE, FWA OF TOP DEF TABLE
*        NOP           LST WRD+1 SON NODE, LWA OF BTM DEF TABLE
*        NOP           REL SECT # FROM STRT FOR SON NODE'S CODE
*        NOP           REL SECT # FROM STRT FOR SON'S BASE PAGE
*  .ORD  NOP           THIS NODES LEAF #  (PATH #)
*  .NOD# NOP           THIS NODES ORDINAL #
*
*
```

```
*   EXEC WILL DO THE FOLLOWING :
*
*        1.  DO NODE LOAD FOR CODE AND BASE PAGE
*        2.  UPDATE CURRENT PATH #
*        3.  RETURN TO    CONTENTS (CONTENTS ( 4TH PRAM) ) -1
*
*
*
*
$LODD NOP
      JSB .SVRG     SAVE OUR REGISTERS
 *
       JSB .DML1     CALL DBUGR
       DEF $LODD
 *
 *
       JSB EXEC      CALL EXEC TO DO THE NODE LOAD
       DEF *+5
       DEF D8        MAKE IT LOOK LIKE A SEG LOAD
       DEF EXEC+0     SECURITY CODE
       DEF $LODD,I   PASS IN THE .DTAB ADDRESS
       DEF *+1        SET UP THE RETURN ADDRESS
       DEF *+2
 *
       LDB $LODD    OK SO NOW RETURN
       JMP BACK        TO THE CALLER
 *
 *
 D8    DEC 8
 *
       END
```

13.10 LOADING MEMORY RESIDENT NODES

Memory resident nodes for MLS-LOC programs contain one DEF table per node as opposed to two that disc resident nodes contain. The memory resident node DEF table is aligned to a page boundary and is located at the beginning of each son node. The values in the DEF table are calculated the same as for disc resident nodes. Since there is no bottom DEF table to use as a basis for the top DEF table, the image of the memory resident DEF table is saved in a temporary disc area of MLLDR.

The thunk code for memory resident nodes looks like the thunk code for disc resident nodes except for LOD and .DTAB. LOD does disc resident node loads. For memory resident nodes, $LOC, a microcoded routine, is called from LOC, to simply accomplish a map switch. The memory resident node thunk is set up as follows:

```
      EXT $LOC$,.CNOD
      ENT $MTHK
*
$MTHK OCT 10         *THUNK LENGTH
      JSB $LOC$      *call to microcoded macro to do mapping
*
.DTAB OCT LGPG#     *logical pg # at which this node begins (0-31)
      OCT RELPG      *relative pg # from beginning of partition
                     *where node starts (0-1023)
      OCT RELBP      *relative pg # from start of partition
                     *where base page resides (0-1023)
      DEF .CNOD+0    *address of current path word
.ORD  <number>      *this node's leaf node number
.NOD# <number>      *this node's ordinal number
```

LGPG# can be filled in at the end of relocating the current node. RELPG and RELBP can be filled in after the specified node is relocated. DEF .CNOD can be filled in immediately since .CNOD is in the program preamble and has been relocated already.

13.11 MEMORY ALLOCATION

During loading, programs are relocated to start at the beginning of the disc-resident program area of logical memory. The logical address of the program always begins at a page boundary. The first two words of the program location(0,1) are allocated for saving the contents of the X and Y registers whenever the program is suspended. The next word (2) is a DEF to EMA if EMA is being used or a DEF to VMA if VMA is being used. The next four words(3-6) must be set to 0,1,1,0. Word 7 is 0 if no profiling is to be done, or is the address of .STAR if the profile option was specified. Word 8 is the EMA start page if EMA is being used, or is 0. The next word is .CNOD, which contains the leaf node number of the last node that any thunk code brought in. A node's leaf node number is the node number of the leaf at the end of the left most path containing the node. The next word (10) will be the first word of the root node. Following the code of the root node will be the root node's thunk routines and finally it's DEF table if there are no other memory resident nodes. The DEF table immediately follows the thunks.

If there are disc resident nodes only initially, only the root is loaded. Upon a fault, a son node will be loaded. The son's top DEF table will be located at the same logical address as the father's DEF table, followed by the son's code and if he in turn has sons, thunks and another DEF table. Note: this means that the son's top DEF table overlaps the father's table.

If there are memory resident nodes only, all nodes will be in memory (in a partition) but upon initial dispatch only the leftmost path of a tree will be mapped into logical memory. Following the root node, the relocation address is aligned to a page. Remember, there is no bottom DEF table in this case. The first son's top DEF table is relocated followed by his code and thunks if he in turn has sons. The relocation address is again aligned to a page and the DEF table code for the next node in preorder is relocated.

If there are disc and memory resident nodes, all memory resident nodes are in physical memory as in the all memory resident node case. And, the DEF tables are aligned to page boundaries as before. Disc resident nodes who are sons of the root must have their top DEF tables aligned to page boundaries to appear the same as the memory resident nodes. (Note that in the all disc resident case, only even sector boundary alignment was required.) All other disc resident nodes, further down the tree, are relocated as before. First will be the top DEF table, followed by the node's code, followed by its thunks and finally the bottom DEF table. In this case, the root does not have a bottom DEF table as in the all disc resident case. All other disc resident nodes (except the leaves) do have the bottom DEF table.

The physical memory requirements depend in part on the type (memory resident or disc resident) of nodes being relocated. An all memory resident program would require a partition large enough for all memory resident nodes and all copies of the base page required for rotating base page. An all disc resident program would require a partition large enough for the root (the root is always memory resident) and the longest disc path to a leaf. A mixed memory and disc resident program would require a partition large enough for all memory resident nodes and all copies of the base page required by the memory resident nodes for rotating base page, plus the longest disc resident path to a leaf.

Memory Resident Nodes Only

```
               +--------+                 +----------------+
               |   M    |                 |                | End of
               +--------+                 |                | Partition
                   |                      +- - - - - - - - +
             ---------------              |                |
             |             |              |  Current MSEG  |
        +--------+     +--------+         |                |
        |  M.1   |     |  M.2   |         +- - - - - - - - +
        +--------+     +--------+         |                |
            |                             |                |
     ---------------                      |                |
     |             |                      +----------------+
 +--------+     +--------+                |                |
 | M.1.1  |     | M.1.2  |                |    Dynamic     |
 +--------+     +--------+                |     Buffer     |
                  .                       |      Area      |
                  .                       |                |
                  .                       |                |
          +----------------+              +----------------+
          |                |              |                |
          |  Current MSEG  |              +- - - - - - - - +
          |                |              |                |
          +----------------+              |    M.2 Code    |
          |                |              +- - - - - - - - +
          |    Dynamic     |              |   DEF Table    |
          |  Buffer Area   |              +----------------+
          |                |              |                |
          +----------------+              |    Rotating    |
          |                |              |   Base Page    |
          |                |              |                |
          +- - - - - - - - +              +----------------+
          |                |              |                |
          |   M.1.1 Code   |              +- - - - - - - - +
          +- - - - - - - - +              |                |
          |   DEF Table    |              |   M.1.2 Code   |
          +----------------+              +- - - - - - - - +
          |                |              |   DEF Table    |
          +- - - - - - - - +              +----------------+
          |     Thunks     |              |                |
          +- - - - - - - - +              +- - - - - - - - +
          |                |              |                |
          |    M.1 Code    |              |   M.1.1 Code   |
          +- - - - - - - - +              +- - - - - - - - +
          |   DEF Table    |              |   DEF Table    |
          +----------------+              +----------------+
          |                |              |                |
          +- - - - - - - - +              +- - - - - - - - +
          |     Thunks     |              |     Thunks     |
          +- - - - - - - - +              +- - - - - - - - +
```

```
         |               |              |               |
         | Root (M) Code |              |   M.1 Code    |
         |               |              +- - - - - - - -+
         +---------------+              |   Def Table   |
         |               |              +---------------+
         |   Op. Sys.    |              |               |
         | Communication |              +- - - - - - - -+
         |               |              |    Thunks     |
         +---------------+              +- - - - - - - -+
         |               |              |               |
         |   Base Page   |              | Root (M) Code |
         |               |              |               |
         |               |              +---------------+
         |               |              |               |
         |               |              |   Base Page   |
  PAGE 0 |               |              |               |
         +---------------+              +---------------+
             Logical                        Physical
                           Initial Load
```

Disc Resident Nodes Only

```
              +--------+
              |   M    |
              +--------+
                  |
           ---------------
           |             |
      +--------+     +--------+
      |  D.1   |     |  D.2   |
      +--------+     +--------+
           |
    ---------------
    |             |
+--------+     +--------+                          +---------------+ End of
| D.1.1  |     | D.1.2  |                          |               | Partition
+--------+     +--------+                          |               |
                   .                               |               |
                   .                               |               |
                   .                               |               |
            +---------------+                      +- - - - - - - -+
            |               |                      |               |
            | Current MSEG  |                      | Current MSEG  |
            |               |                      |               |
            +---------------+                      |               |
            |               |                      +- - - - - - - -+
            |    Dynamic    |                      |               |
            |  Buffer Area  |                      |               |
            |               |                      |      EMA      |
            |               |                      |               |
            |               |                      |               |
            |               |                      +---------------+
            +- - - - - - - -+                      |               |
            |               |                      |               |
            |     D.1.1     |                      |    Dynamic    |
            +- - - - - - - -+                      |  Buffer Area  |
Before      |   DEF Table   |                      |               |
D.1.1 is    +- - - - - - - -+                      |               |
loaded it   |    Thunks     |                      |               |
will be     +- - - - - - - -+                      +- - - - - - - -+
the DEF     |               |                      |               |} Dynamic
Table of    |               |                      +- - - - - - - -+ Buffer Area
D.1         |               |                      |               | starts past
            |      D.1      |                      |               | lowest path
            +- - - - - - - -+                      |     D.1.1     | possible
Initially   |   DEF Table   |                      |               |
will be     +- - - - - - - -+                      +- - - - - - - -+
DEF Table   |    Thunks     |                      |   DEF Table   |
of Mopt     +- - - - - - - -+                      +- - - - - - - -+
```

```
|                 |              |    Thunks       |
|                 |              +- - - - - - - - -+
|                 |              |                 |
| Root (M) Code   |              |                 |
|                 |              |                 |
+-----------------+              |      D.1        |
|                 |              +- - - - - - - - -+
|    Op. Sys.     |              |   DEF Table     |
| Communication   |              +- - - - - - - - -+
|                 |              |    Thunks       |
+-----------------+              +- - - - - - - - -+
|                 |              |                 |
|   Base Page     |              | Root (M) Code   |
|                 |              |                 |
|                 |              +- - - - - - - - -+
|                 |              |                 |
|                 |              |   Base Page     |
|                 |              |                 |
+-----------------+              +-----------------+
      Logical                          Physical
                    Leftmost Path
```

Memory and Disc Resident Nodes

```
                 +--------+                  +---------------+ End of
                 |   M    |                  | Current MSEG  | Partition
                 +--------+                  |               |
                     |                       +- - - - - - - -+
           -------------------               |               |
           |                 |               |      EMA      |
       +--------+        +--------+          |               |
       |  M.1   |        |  D.2   |          +---------------+
       +--------+        +--------+          |               |
           |                 |               | Dynamic Buffer|
     ------------        ------------        |     Area      |
     |          |        |          |        +---------------+
+--------+ +--------+ +--------+ +--------+  |               |
| M.1.1  | | M.1.2  | | D.2.1  | | D.2.2  |  +- - - - - - - -+
+--------+ +--------+ +--------+ +--------+  |               |
                                             |     D.2.2     |
                                             +- - - - - - - -+
                                             |   DEF Table   |
           +---------------+                 +- - - - - - - -+
           |               |                 |    Thunks     |
           | Current MSEG  |                 +- - - - - - - -+
           |               |                 |               |
           +---------------+                 |      D.2      |
           |               |                 +- - - - - - - -+
           |    Dynamic    |                 |               |
           |  Buffer Area  |                 |   DEF Table   |
           |               |                 +- - - - - - - -+
           +---------------+                 |               |
           |               |                 +- - - - - - - -+
           |               |                 |    Thunks     |
           +- - - - - - - -+                 +- - - - - - - -+
           |               |                 |     M.1.2     |
           |     D.2.2     |                 +- - - - - - - -+
           +- - - - - - - -+                 |               |
           |   DEF Table   |                 |   DEF Table   |
           +- - - - - - - -+                 +- - - - - - - -+
           |    Thunks     |                 |               |
           +- - - - - - - -+                 +- - - - - - - -+
           |               |                 |    Thunks     |
           |               |                 +- - - - - - - -+
           |               |                 |               |
           |      D.2      |                 |     M.1.1     |
           +- - - - - - - -+                 +- - - - - - - -+
           |   DEF Table   |                 |   DEF Table   |
           +---------------+                 +---------------+
           |               |                 |               |
           +- - - - - - - -+                 +- - - - - - - -+
           |    Thunks     |                 |    Thunks     |
```

```
+- - - - - - - - -+              +- - - - - - - - -+
|                 |              |                 |
| Root (M) Code   |              |       M.1       |
|                 |              +- - - - - - - - -+
+-----------------+              |    Def Table    |
|                 |              +-----------------+
|     Op. Sys.    |              |                 |
|  Communication  |              +- - - - - - - - -+
|                 |              |     Thunks      |
+-----------------+              +-----------------+
|                 |              |                 |
|    Base Page    |              | Root (M) Code   |
|                 |              |                 |
|                 |              +-----------------+
|                 |              |                 |
|                 |              |    Base Page    |
|                 |              |                 |
+-----------------+              +-----------------+
      Logical                          Physical
                     Rightmost Path
```

13.12 DISC FORMAT

The disc format for MLS-LOC programs differs slightly between disc resident nodes and memory resident nodes. Disc resident nodes are aligned to the next even sector boundary. Memory resident nodes are aligned to the next page boundary with respect to memory (which will also be an even sector boundary).

One of the advantages of this format is that for memory resident nodes the program on disc is in memory image format and can be loaded directly into memory. If disc resident nodes are mixed with memory resident nodes, the disc resident nodes must all reside in the right hand side of the tree with no memory resident nodes further to the right. Thus, the entire memory resident part can be loaded into low physical memory first and the disc resident nodes can be loaded later as called.

An additional restriction which results in optimal disc format and run time mapping is that disc resident nodes must start at the first node below the root and no other memory resident nodes can occur after that point.

When disc and memory resident nodes exist, the DEF table at the top of all first disc resident nodes after the root must look like memory resident nodes. That is, they must be aligned to the next page boundary with the same relative start address as the memory resident nodes right after the root. All other disc resident nodes will be in the regular disc resident format.

13.13 ROTATING BASE PAGE

Rotating base page is a scheme whereby base pages in the leaves of the program tree are shared. This scheme helps conserve disc and memory space. Under a more typical base page scheme, a separate base page would be required for each leaf of the tree.

In the ideal case, the algorithm for determining when to start a new base page and when to continue using the current one is simple.

1. Relocate nodes in pre-order.

2. Mark the current base page address before relocating the next node (CBP.L -> BPS.M).

3. Relocate the node CBP.L -> BP1.M.

4. Mark the base page address after relocating the node (CBP.L).

5. If the node we relocated is not a leaf, go to step 2.

6. Check adjacent brother node. If the brother is a leaf, go to step 7. If the brother is not a leaf, go to step 8. If there is no adjacent brother, go to step 9.

7. Relocate the brother, using the same base page starting at the next available location (the location marked in step 4). After relocating and marking the last base page location (CBP.L) go to 6.

8. Start relocating node and allocate a new base page. Start at the base page address marked in step 2 (BPS.M -> BP1.M). After relocating, go to 4.

9. Determine the next node to relocate. Set the base page pointers as they were set at the end of relocating this node's father. Go to step 2.

In the ideal case, the above algorithm works fine. However, there is one case which it does not cover. If a brother node continues using the same base page but runs out of base page links in the middle of the relocation, some other steps are necessary. This should not be a base page overflow condition since there should be room available for this node (namely those base page locations used by the first brother). The rotating concept applies here.

The following pointers will be necessary.

BPR.L Base Page Beginning
Starting base page location for the 1st node.

BKGBL Base Page End
Ending base page location of RTE.

BP1.M Base Page Node #1
Starting base page location for this node.

CBP.L Base Page Node #N (last+1)
Ending base page location for this node.

BPS.M Base Page Save
Saved base page location where new base page will start.

BPL.M Last available base page location (limit) for this node.

DBTBL Base Page Next
Pointer to the disc to where the next base page should go.
(2 words - 1st word available, relative sector from start)

DTB.M Start Node
Pointer to the disc of the start of the current node being relocated.
(3 words - 1st word available, relative track and sector.)

BPR.L and BKGBL, once set, will not be changed. BP1.M, CBP.L, and BPS.M will be updated as indicated in the optimal case. To begin with, DBTBL will point to the beginning of the disc area for this program, DTB.M will be 16 sectors (1 page) farther down, and BPL.M will equal BKGBL.

Consider the following example:

```
         +---+
         | M |
         +---+
           |
     -------------
     |     |     |
   +---+ +---+ +---+
   |M.1| |M.2| |M.3|
   +---+ +---+ +---+
     |
   -------------------------------------------------
   |       |       |       |       |       |       |
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+
|M.1.1| |M.1.2| |M.1.3| |M.1.4| |M.1.5| |M.1.6| |M.1.7|
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+
                                                   |
                                              -----------
                                              |         |
                                          +-------+ +-------+
                                          |M.1.7.1| |M.1.7.2|
                                          +-------+ +-------+
```

After relocating M, suppose the following base page has been set up.

```
      BPR.L
BPS.M BP1.M -> 2
      CBP.L -> 777
                .
                .
                .
      BPL.M -> 1644
      BKGBL
```

Suppose all links up to M.1.2 fit on the page. Then we would have:

```
      BPR.L ->    2   start of links for M
                777   end of links for M
               1000   start of links for M.1
               1200   end of links for M.1
      BPS.M -> 1201   start of links for M.1.1
               1350   end of links for M.1.1
      BP1.M -> 1351   start of links for M.1.2
      CBP.L -> 1600   end of links for M.1.2
                .
                .
                .
BPL.M BKGBL -> 1644
```

Now in trying to relocate M.1.3, we will run off the end of base page (1644). All we really need on base page are those links on the path up from M.1.3 to the root. That is, for M.1.3, we do not need the links for M.1.1 or M.1.2. Therefore, we record this base page on the disc (how the disc location is calculated will be explained later), reset the next link to allocate to 1201 (the value of BPS.M) and start a new base page. All values from BPR.L to BPS.M and from BP1.M to BKGBL are still valid. 1601 is now the upper limit we must not overflow. BPL.M is set to 1601. After relocating M.1.3, the base page will look as follows:

```
      BPR.L ->    2   start of links for M
                777   end of links for M
               1000   start of links for M.1
               1200   end of links for M.1
      BPS.M -> 1201   links for M.1.3
      CBP.L -> 1500   end of links for M.1.3
                  .
                  .
                  .
BPL.M BP1.M    1601   start of links for M.1.3
      BKGBL    1644   links for M.1.3
```

Suppose M.1.4 can be relocated without overflowing this base page. We would then have:

```
      BPR.L ->    2   start of links for M
                777   end of links for M
               1000   start of links for M.1
               1200   end of links for M.1
      BPS.M -> 1201   links for M.1.3
               1500   end of links for M.1.3
      BP1.M -> 1501   start of links for M.1.4
      CBP.L -> 1555   end of links for M.1.4
                  .
                  .
                  .
      BPL.M -> 1601   start of links for M.1.3
      BKGBL -> 1644   links for M.1.3
```

Remember that the new upper bound we are checking against is 1601. Now we try to relocate M.1.5. It is a leaf so we try to use the same base page. Suppose the links for M.1.5 go beyond location 1601. Then, we must record this base page on the disc, reset the next link to allocate to 1601 and start a new base page. All values from BPR.L to BPS.M and from BP1.M to CBP.L will be valid. We now have:

```
      BPR.L ->    2   start of links for M
                777   end of links for M
               1000   start of links for M.1
               1200   end of links for M.1
      BPS.M -> 1201
```

```
                  .
                  .
                  .
      BP1.M -> 1556   start of links for M.1.5
      CBP.L -> 1640   end of links for M.1.5
                  .
                  .
                  .
BPL.M BKGBL -> 1644
```

At this point, note that there are two free areas on base page, from 1641 to 1644 and from 1201 to 1555. If M.1.5 or any nodes on the path to its leaf node if it hadn't been a leaf had needed more base page links, these two areas would have been available. M.1.6 is also a leaf (and a brother). Base page pointers are set up so that locations 1641 to 1644 are available for M.1.6 links (ie use the same page). Suppose while relocating M.1.6, locations 1641 through 1644 are used. Now another base page location is needed so the current base page is written out to the disc and a new one is started. The links which have been allocated so far for M.1.6 are still valid and 1201 through 1640 are now available. (Note: 1201 to 1555 were available to M.1.5) We now have:

```
      BPR.L ->    2   start of links for M
                777   end of links for M
               1000   start of links for M.1
               1200   end of links for M.1
      BPS.M -> 1201          links for M.1.6
      CBP.L -> 1600   end of links for M.1.6
                  .
                  .
                  .
BPL.M BP1.M -> 1641   start of links for M.1.6
      BKGBL -> 1644             links for M.1.6
```

M.1.7 is not a leaf, so we would allocate a new base page starting with 1201.

```
      BPR.L ->    2   start of links for M
                777   end of links for M
               1000   start of links for M.1
               1200   end of links for M.1
      BP1.M -> 1201   start of links for M.1.7
      CBP.L -> 1444   end of links for M.1.7
      BPS.M -> 1445
BPL.M BKGBL -> 1644
```

If the links for M.1.7.1 and M.1.7.2 do not fit in locations 1445 through 1644, two base pages would be allocated, one with M.1.7.1's links beginning at location 1445 and one with M.1.7.2's links beginning where M.1.7.1's links end and continuing at 1445. But, if all links fit on the same base page, we would have something like the following:

```
      BPR.L ->    2   start of links for M
                777   end of links for M
               1000   start of links for M.1
               1200   end of links for M.1
               1201   start of links for M.1.7
               1444   end of links for M.1.7
      BPS.M -> 1445   start of links for M.1.7.1
               1553   end of links for M.1.7.1
      BP1.M -> 1554   start of links for M.1.7.2
      CBP.L -> 1572   end of links for M.1.7.2
BPL.M BKGBL -> 1644
```

Now we are ready to relocate M.2. We must reset pointers as if we just finished relocating M. We cannot share a base page with M.1 since M.1 was not a leaf. We now have:

```
      BPR.L ->    2   start of links for M
                777   end of links for M
      BPS.M -> 1000   start of links for M.2
      CBP.L -> 1335   end of links for M.2
BPL.M BKGBL -> 1644
```

Since M.3 is a leaf and a brother, we try to share the base page with M.2 If this were not possible, M.3 would have a new base page with links from 1336 to 1644 and from 1000 to the end of links for M.3 (which must be somewhere before 1336). Suppose M.3 can share a base page with M.2, we would then have:

```
      BPR.L ->    2   start of links for M
                777   end of links for M
      BPS.M -> 1000   start of links for M.2
               1335   end of links for M.2
      BP1.M -> 1336   start of links for M.3
      CBP.L -> 1511   end of links for M.3
BPL.M BKGBL -> 1644
```

There are a few points to note in this scheme. Most of the time, BPL.M is the same as BKGBL. But, if BKGBL is allocated and more links are needed, then BPL.M is set to the current value of BP1.M, a new base page is allocated saving BPR.L to BPS.M and BP1.M to BKGBL, and locations BPS.M to BPL.M are available.

As the base page gets chopped into pieces, we have to keep track of which are valid links and which are available for allocation. Most of the time all links from BPR.L to BPS.M and from BP1.M to CBP.L are valid. Those from CBP.L to BPL.M are available. All others are subject to rotation in which case they must be saved first. If BKGBL was allocated and a new base page started, BPR.L to BPS.M, BPS.M to CBP.L, and BP1.M to BKGBL are valid. CBP.L to BPL.M (currently the same as BP1.M) are available. If relocation causes a new base page to be created because BPL.M was reached and BPL.M does not equal BKGBL, then BPR.L to BPS.M, and BP1.M to CBP.L are valid links. BPS.M to BP1.M and CBP.L to BKGBL are available. A flag is set to indicate that there are two separate pieces available.

In the algorithm, there is a step which requires all base page pointers to be reset as they were set at the end of relocating the current node's father. This occurs when a node is to be relocated after a brother and his sons, etc., have been relocated. This can be done since BPR.L and BKGBL will be set already. BPL.M will be the same as BKGBL. The last base page location of the father can be found since it will have been marked in a previous step 2 and saved in the node path kept in memory. This value will be used to set BP1.M, BPS.M, and initially CBP.L.

The image of the program which the loader places on the disc includes the rotating base page. Each base page image preceeds the code for the nodes which use that base page. When the load begins, DBTBL is set to point to the first disc location available to the program. DTB.M is set to point 2000B words further down the disc. This will be the starting disc location for node M. As each next node is relocated, DTB.M will be updated to point to the starting disc location of the current node. When it is discovered that a new base page should be allocated, the old base page must be recorded on the disc. The old image of the base page will be stored on the disc beginning at location DBTBL. DBTBL will then be updated to the current value of DTB.M ([trk * #sct/trk]+ sect). All code from DTB.M to the last word relocated must be moved 2000B words down on the disc to make room for the new base page to be stored, when it is completed. DTB.M is updated to DTB.M+2000B ({[trk*#sct/trk]+sect+16}/ #sct/trk) gives new track and sector, 16 sectors= 2000B words). Each time a new base page is required, the old one is stored at DBTBL. DBTBL is updated to DTB.M, all code from DTB.M to the last word relocated is moved 2000B words down the disc, and DTB.M is updated to DTB.M+2000B. After the final node is loaded, the last base page is recorded at the current DBTBL location.

The 2000B word move down the disc may cause an overflow of the current track allocation. If this is the case, a larger track allocation must be made before the 2000B word move is made. MLLDR checks to see if the next consecutive track is free. If so, it assigns it to itself and reissues the request to move code 2000B words down the disc. If that track is not free, MLLDR makes a request for n+1 tracks, where n is the number of tracks it currently has. It moves the sectors it has output so far to the new tracks, releases the old tracks, and updates the base track and sector of the program for the ID segment. Now the request to move code 2000B words down the disc is reissued. If n+1 tracks are not available, the loader suspends with a "WAITING FOR DISC SPACE" message.

13.14 SETBP

SETBP is called to set up the rotating base page pointers when a node command is encountered and a new node is about to be relocated. Four cases can occur:

1. The previous node was not a leaf.
2. The previous node was a leaf. This node is a brother and a leaf.
3. The previous node was a leaf. This node is a brother, but not a leaf.
4. The previous node was a leaf. This node is not a brother.

CASE I

If the previous node just relocated was not a leaf, the same base page must be used, and the BPS.M value updated.

```
LDA CBP.L
STA BPS.M
STA BP1.M
```

The previous node was not a leaf if this node is a son.

CASE II

If the previous node was a leaf and this node is a brother and a leaf, we try to share the base page. We keep the same BPS.M and use the same base page.

```
LDA CBP.L
STA BP1.M
```

The previous node was a leaf if this node is on the same level or farther up the tree. This node is a brother if it is on the same level. It is also a leaf if the next node in preorder is not a son.

CASE III

If the previous node was a leaf and this node is a brother, but not a leaf, we need a new base page. A call is made to M.OBP to output the old base page and a new one is set up keeping the current BPS.M and using it as the first location.

```
JSB M.OBP
LDA BPS.M
STA BP1.M
STA CBP.L
LDA BKGBL
STA BPL.M
```

CASE IV

If the previous node was a leaf, but this node is not a brother, it is some node farther up the tree, BPS.M must be reset to be the value at the end of relocating this node's father and used as the first location. This BPS.M can be found the current path kept in memory, in the father node's entry.

```
JSB M.OBP
LDA father's last+1 base page (from saved path information)
STA BPS.M
STA BP1.M
STA CBP.L
LDA BKGBL
STA BPL.M
```

This node is not a brother if it is at a level farther up the tree than the previous node.

13.15 M.OTB,M.ABT,M.OBP - ABSOLUTE OUTPUT ROUTINES

The M.OTB routine is called to output a word to the disc. It in turn calls the M.ABT routine, which does the actual output of the absolute program word if M.OTB finds nothing wrong with the request.

For each node on the path from the root node to the current node being processed, seven words of relocation information are kept in memory. These words include sector offset location so that fixups to an earlier node can be made.

Fixups to an earlier node will be limited to filling in the .DTAB entry in the thunk which corresponds to this node and is located in the father of this node. There should be no fixup table entries when processing of a new node begins. Any forward references should have been changed to go indirect through the node's DEF TABLE which would go to the appropriate THUNK or routine. The top DEF table of the current node will need to be fixed, but these locations will not be in the fixup table. Each will be in the appropriate symbol's symbol table entry and the entry will be marked undefined. Thus, when the symbol is defined, the DEF table location to fix is taken out of the symbol table entry and fixed and then the symbol's value is put into the symbol table and the symbol is marked defined.

For the current node, a three word table, DTBL, is set up. It contains DEF's to three values, the base memory address for the node, the base track offset, and the base sector offset.

When a call is made to M.OTB two parameters are also passed, the memory address of the word to output and its value. The address is checked against the current node's base address. If it is above the base address, a check is made against 2000B to see if the address is on base page. If so, the value is put out on the memory resident dummy base page. That is, a disc write is not done. If the address falls between the base page and the current node, the current DTBL is saved and DTBL is set up to point to values for a thunk in the current node's father. This can be easily done since the information is saved in DTABP. Now we check that the given address is above the father's base. If not, an error occurs since calls can only be down one level. If the address is O.K., M.ABT is called and upon return, the original DTBL values are reset.

M.ABT is called from M.OTB to output an absolute program word. A check is made to see if we have overflowed the allocated tracks. If not, we see if the current track and sector in core is the one required. If so, the word is written. If not, the sector is written out to disc and the desired one is read in. Then the word is written and if necessary the upper bound pointer, TH2.L is updated. If more space is required on the disc, we see if the next track is free. If so, we allocate it and go through the same step as if no overflow occurred. Otherwise, we write out the current sector in memory and make a request for more tracks. If the larger number of tracks is not available, MLLDR suspends and issues a message "WAITING FOR DISC SPACE". If a larger area was found, all information stored so far on disc is moved to the new tracks, the old tracks are released. All path information stored per node (disc location and base page location) should not need to be updated to reflect the new disc location of the program since they are stored relative to the beginning disc location of the program. The ID segment being built is updated to reflect the new disc location.

The routine M.OBP is used to rotate base page. When base page must be rotated, the old base page must be written out, and space for the new one must be allocated. Allocating space means moving all code relocated so far for the current module 2000B words (a page) down the disc. This will leave space for the new base page (when it is complete) immediately preceeding the code for which that base page is valid.

To output the base page currently in the dummy base page area, a flag is set so that the M.ABT routine will skip certain checks and pointer updating. DTBL is set up to be the address, track, and sector for this base page (these are calculated using values in DBTBL). M.ABT is called in a loop to output one word at a time until the last word of base page is output. DTBL is restored, and the next base page location on disc must be set up.

The current module's disc base location is the new base page disc base location. The new disc base location for the module is the old location+2000B words or 16 sectors. This new disc base location for the module is where all code for the module outputted so far must be moved. To calculate how many sectors need to be moved, the difference between the last address relocated for this module and the first is divided by 64 (the number of words per sector). If there is a remainder, this value is incremented by one.

Now a check is made to see if moving the calculated number of sectors 16 sectors farther down the disc will overflow the current track allocation. If not, each old sector is read into a buffer and written out to the new location until the calculated number of sectors have been written. Note that the last sector is moved first so that we do not write over a sector before we are sure it is moved.

If an overflow occurs and only one more track or less is needed, a check is made to see if the next track is free. If so, it is taken and sectors are moved as in the no overflow case.

If more than one track is needed or one is enough but the next one isn't free, the current track allocation variables are saved, the current sector is written out, and a new track request (for more tracks) is made. If the tracks requested are received, all information from the old tracks are moved to the new ones, the old tracks are released, and all necessary pointers are updated. Now, the current node can be moved down the disc as before.

If the requested tracks are not available, the loader suspends with a "WAITING FOR DISC SPACE" message.

```
+--------------------------------------------------+------------------+
|                                                  |                  |
|   DVP43 Power-Fail Auto-Restart Driver           |   APPENDIX  A    |
|                                                  |                  |
+--------------------------------------------------+------------------+
```

The power-fail auto-restart driver must reside in the System Driver Area because it is entered directly in the System Map from the trap cell when power-fail or power-up occurs. The entry point for the interrupt is $POWR.

On power failure, DVP43 stops DMA transfers on both ports, saves all the programmable registers and S-registers, and saves all of the map registers. DVP43 also saves the location of the last memory protection violation if the system was in the process of registers. With all this done, the driver does a JMP* to wait for power to fade out completely.

When power returns, $POWR is entered and control is transferred to the UP section of code. A switch is set so that if another power failure occurs, while DVP43 is in the process of restoring the system, none of the DOWN code is executed. This preserves the saved information from the first power failure in case it could not be restored before a subsequent failure occurred.

In the UP code, all of the map registers are restored and the base page fence is set up again. Then a search is made for the DVP43 EQT entry. If it is not found, a halt will be simulated. The S-register will contain 103004 octal while a JMP* is executed. Once the power-fail driver EQT is found, the entry in the Driver Mapping Table is modified to indicate that the driver does its own mapping. The time-out handling bit is set in the power-fail driver EQT and the EQT is set up to time-out on the next clock tick.

The EQT count is set up for scanning all of the EQTs later. If this is another restart attempt (after being interrupted from the UP process by another power failure), the current EQT being processed for restart is set busy so another restart try will be done during the scan of all EQTs.

The current time-of-day is saved, if it was already saved on a previous restart attempt. This preserves the time of the first power failure if there happens to be a number of successive failures. Then the clock is restarted by a call to $SCLK.

The privileged I/O terminator card (if present) is set up before the registers are restored, and return is made to the point of interrupt. The switch is reset to allow another power-fail to be processed.

Each time-out entry into DVP43 causes an EQT to be checked for an I/O request in progress. If an EQT is busy and it has the power-fail handling bit set, the driver map is set up by a call to $DRVM, and the driver is entered at the initiator entry point. If the driver is busy but does not do its own power-fail recovery, the driver is set down and $UPIO is called to restart the last I/O request.

When all the EQTs have been checked, AUTOR is scheduled. AUTOR is aborted before it is scheduled because it may still be scheduled from a previous power failure. Finally, the time-out counter is cleared in the DVP43 EQT and control is returned to the system via $XEQ.

Reentrant List Structure — APPENDIX B

The first word, TDB, is used by the system as follows:

0 - subroutine is available
nonzero - points at 4 word block describing current "owner"; (program currently executing in subroutine)

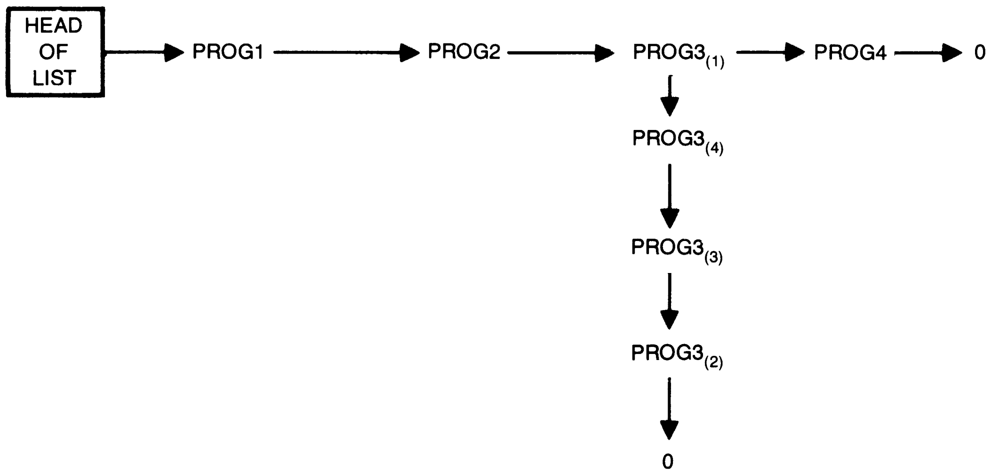
When the TDB is moved to system memory, the first word is changed to point to the location the TDB must be moved back to.

The sign bit of the third word of the block indicates if the block was moved or not. The sign bit of the ID-address indicates if the fourth word block is four words (0) or five words (1) long. (This is caused by a one word imprecision in memory allocation.)

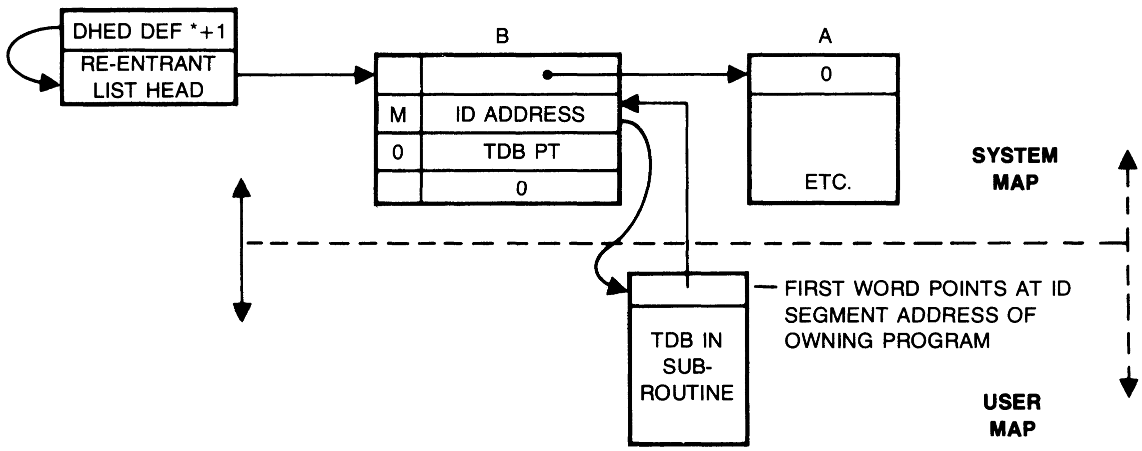
The ID Extension List is a two-dimensional, one-way linked list. The HEAD of the list points to all programs processing reentrant subroutines. They are added to the head of the list as each JSB $LIBR is processed. The other dimension is a list of all reentrant subroutines being processed by one program; that is, on reentrant subroutine calling another. The general reentrant list structure is illustrated in the example given in Figure B-1. Expansions of the structure are given in Figure B-2 through B-5 respectively. Figure B-6 provides a detailed illustration of the total process.

Reentrant List Structure

In Figure B-1, subscripts of PROG3 refer to the order in which the reentrant subroutines were called. PROG4 was the first to enter a reentrant routine; PROG1 was the last.



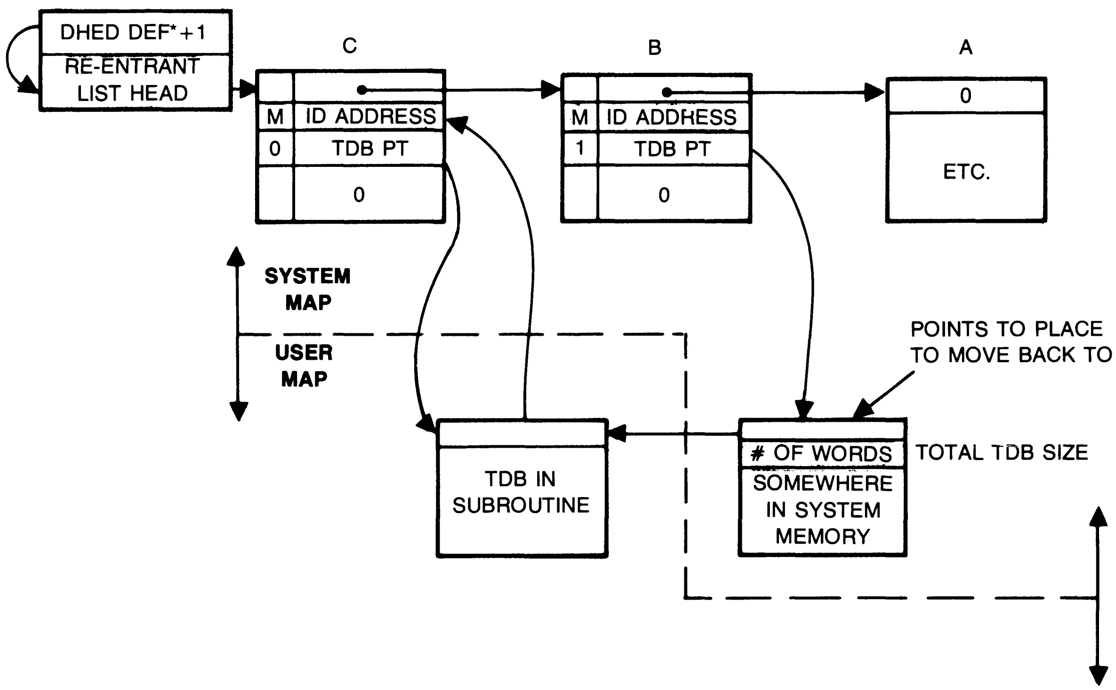
In Figure B-2, one four-word block is created each time a reentrant routine is entered. Programs A and B are both in reentrant subroutines. A entered its routine first.



Reentrant List Structure

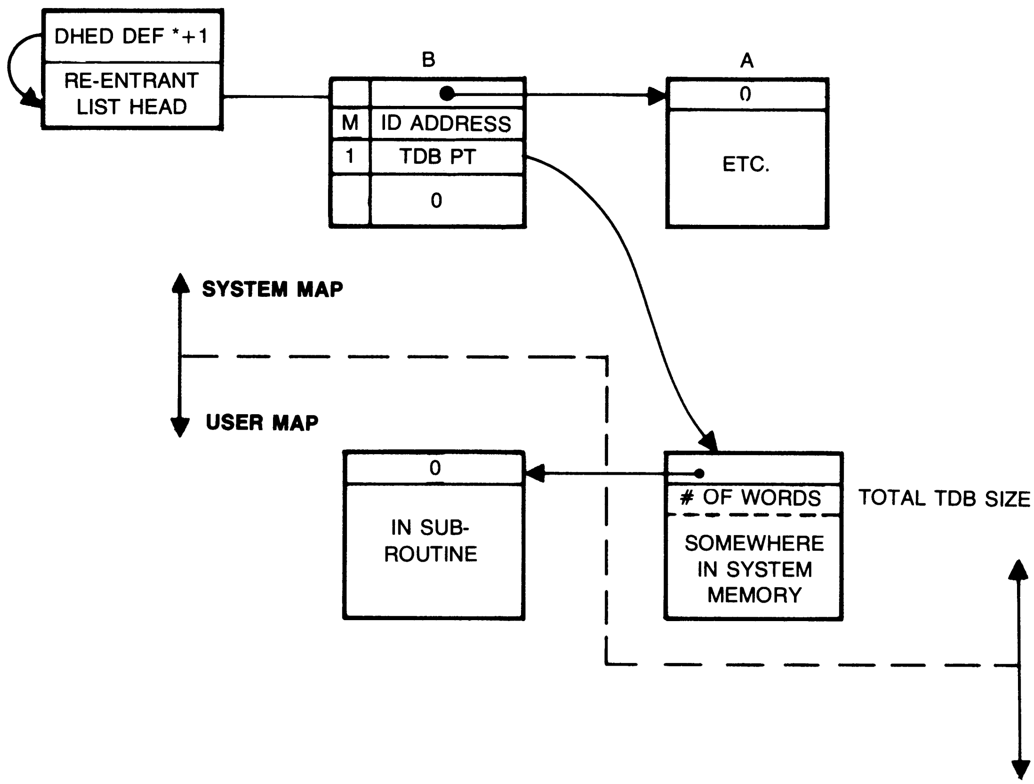
In Figure B-3, program "B" is suspended -- program "C" reenters "B's" subroutine.

NOTE: The moved status is indicated by "B's" TDB pointer not pointing in turn to B's ID segment address.

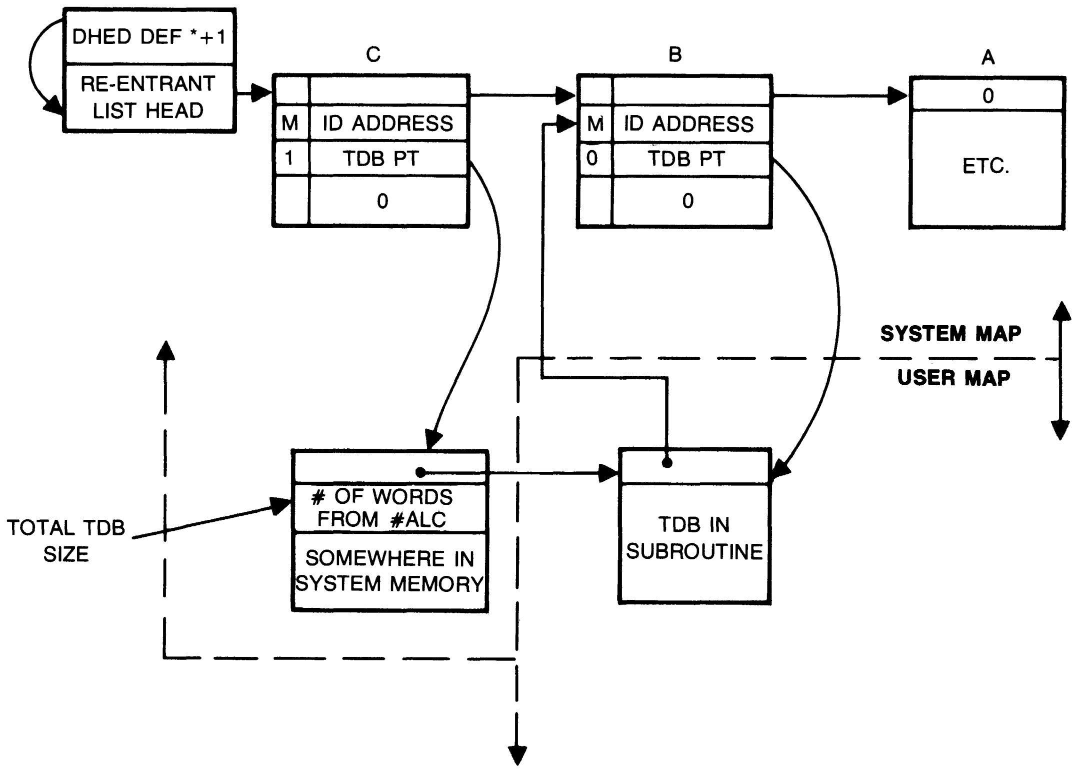


In Figure B-4, program "C" exits the routine.

The routine is available - B's memory will be moved back when the dispatcher is committed to run it.



Assume that in Figure B-3, program "B" was to be executed prior to "C's" exit from the reentrant routine. Then "C's" memory must be saved and "B's" moved back in as illustrated in Figure B-5.



Suppose starting at Figure B-3, routine "C" now calls another reentrant subroutine -- the list structure is now as illustrated in Figure B-6.

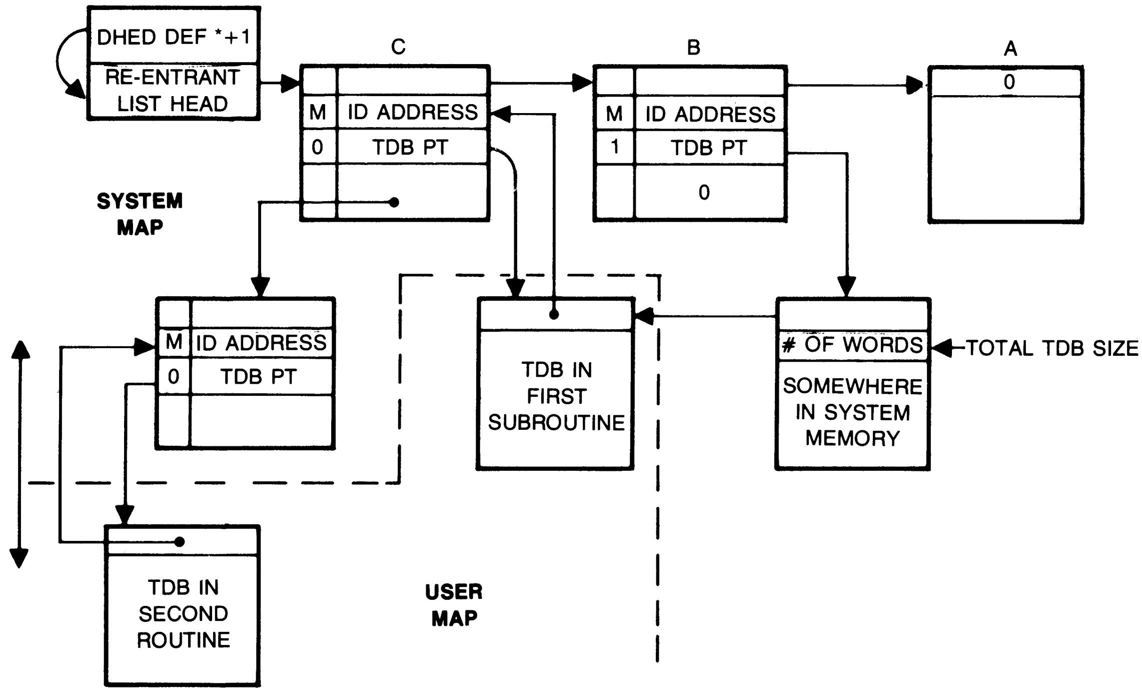
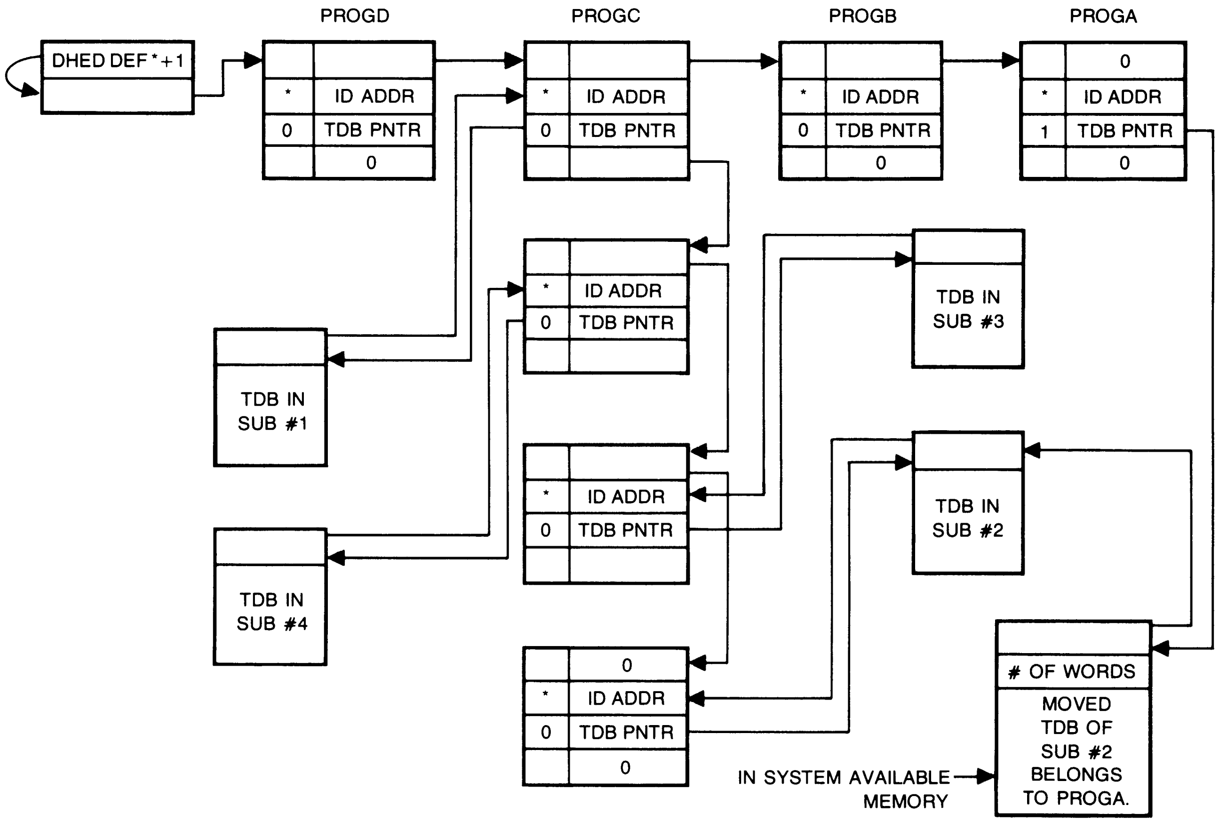


Figure B-7 shows programs PROGD, PROGC, PROGB, and PROGA, all of which are executing reentrant subroutines. PROGC is the program currently running.



Note:
Downward list structure for PROGC. The 4th word of the table is used only for the entry at the head of the list. After the head of the list, the downward list is a push-down stack.

.ZPRV/.ZRNT Calling Sequences — APPENDIX C

The externals .ZPRV and .ZRNT are treated as special entry points in the RTE-6/VM Operating System. The RTE Generator modifies the code that is loaded for subroutines that reference these externals. The changes made depend on whether or not the code is loaded into the core resident library (and hence may be shareable) or if the code is loaded with the program (not sharable), in the latter case the externals are satisfied by replacing the calls to .ZPRV or .ZRNT with an RSS (i.e., .ZPRV,RP,2001). These RPs are passed to the on-line loader in the same manner as an operator RP command at RTGEN time, thus, the on-line loader can perform the same functions as the RTE Generator with respect to the externals .ZPRV, .ZRNT, $LIBR, and $LIBX. The following examples should help to illustrate how an assembled subroutine is modified.

```
  AS ASSEMBLED              WHEN "SUB" IN          WHEN "SUB" NOT
                            CORE RESIDENT          IN CORE RESIDENT
                              LIBRARY                 LIBRARY
 ------------------------------------------------------------------
 ------------------------------------------------------------------

        N O R M A L   P R I V I L E G E D   R O U T I N E
        -------------------------------------------------
 SUB   NOP                 SUB  NOP               SUB  NOP
       JSB .ZPRV                JSB $LIBR              RSS
       DEF LIBX                 NOP                    DEF LIBX
      ...                      ...                    ...
      ...                      ...                    ...
LIBX  JMP SUB,I           LIBX JSB $LIBX         LIBX JMP SUB,I
      DEF SUB                  DEF SUB                DEF SUB
```

.ZPRV/.ZRNT Calling Sequences

P R I V I L E G E D   W I T H   " . E N T R "

```
PARM1 NOP                        PARM1 NOP                   PARM1 NOP
PARM2 NOP                        PARM2 NOP                   PARM2 NOP
SUB   NOP                        SUB   NOP                   SUB   NOP
      JSB .ZPRV                        JSB $LIBR                   RSS
      DEF LIBX                         NOP                         DEF LIBX
      JSB .ENTP                        JSB .ENTP                   JSB .ENTP
      DEF PARM1                        DEF PRAM1                   DEF PRAM1
      ...                              ...                         ...
      ...                              ...                         ...
LIBX  JMP SUB,I                  LIBX  JSB $LIBX             LIBX JMP SUB,I
      DEF SUB                          DEF SUB                     DEF SUB
```

N O R M A L   R E - E N T R A N T   R O U T I N E

```
SUB   NOP                        SUB   NOP                   SUB   NOP
      JSB .ZRNT                        JSB $LIBR                   RSS
      DEF LIBX                         DEF TDB                     DEF LIBX
      ...                              ...                         ...
       ...                              ...                         ...
      ISZ SUB                          ISZ SUB                     ISZ SUB
      ISZ TDB+2                        ISZ TDB+2                   ISZ TDB+2
      NOP                              NOP                         NOP
      ...                              ...                         ...
LIBX  JMP SUB,I                  LIBX  JSB $LIBX             LIBX  JMP SUB,I
      DEF TDB                          DEF TDB                     DEF TDB
      DEC 0                            DEC 0                       DEC 0
```

R E - E N T R A N T   W I T H   " . E N T R "

```
PRAM1 NOP                        PRAM1 NOP                PRAM1 NOP
PRAM2 NOP                        PRAM2 NOP                PRAM2 NOP
SUB   NOP                        SUB   NOP                SUB   NOP
      JSB .ZRNT                        JSB $LIBR                RSS
      DEF LIBX                         DEF TDB                  DEF LIBX
      JSB .ENTP                        JSB .ENTP                JSB .ENTP
      DEF PRAM1                        DEF PRAM1                DEF PRAM1
      STA TDB+2                        STA TDB+2                STA TDB+2
      ...                              ...                      ...
      ...                              ...                      ...
LIBX  JMP TDB+2,I                LIBX  JSB $LIBX          LIBX  JMP TDB+2,I
      DEF TDB                          DEF TDB                  DEF TDB
      DEC 0                            DEC 0                    DEC 0
```

```
+-----------------------------------------------------+------------------+
|                                                     |                  |
|   ID Segments                                       |   APPENDIX  D    |
|                                                     |                  |
+-----------------------------------------------------+------------------+
```

Once the loader successfully relocates a program, the operating system must be told about it. Program information is kept in what is called an ID segment, created for each program loaded with the loader (or loaded at generation time). The loader must appropriately fill in the ID segment for the successfully relocated program.

RTE-6/VM ID SEGMENT:

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| List Linkage                                  | Word  0
|-----------------------------------------------|
| TEMP 1                                        |       1
| TEMP 2                                        |       2
| TEMP 3                                        |       3
| TEMP 4                                        |       4
| TEMP 5                                        |       5
|-----------------------------------------------|
| Priority                                      |       6
| Primary Entry Point                           |    *  7
|-----------------------------------------------|
| Point of Suspension                           |       8
| A-Register                                    |       9
| B-Register                                    |      10
| EO-Registers                                  |      11
|-----------------------+-----------------------|
| Name 1                | Name 2                |    * 12
| Name 3                | Name 4                |    * 13
|                       |--+--+--+--+-----------|
| Name 5                |TM|ML|TS|SS| Type      |    * 14
|--+--+--+--+--+--+--+--+--+--+--+--+-----------|
|NA|NS|NP| W| A|FS| O|LP| R| D|/////| Status    |      15
|--+--+--+--+--+--+--+--+--+--+-----+-----------|
| Time List Linkage                             |      16
|--------+--+-----------------------------------|
| RES    | T| Multiple                          |      17
|                                               |
```

```
|                                              |
|--------+--+----------------------------------|
| Low Order 16 Bits of Time                    |      18
|----------------------------------------------|
| High Order 16 Bits of Time                   |      19
|--+--+--+--+--+--+--+--+----------------------|
|BA|FW| M|AT|RM|RE|PW|RN| Father ID Segment No. |     20
|--+--+--+--+--+--+--+--+--+--+----------------|
|RP|#pgs. (no BP) | MPFI   |DE| Partition No. -1|     21
|--+--------------+--------+--+----------------|
| Low Main Address                             |    * 22
|----------------------------------------------|
| High Main Address + 1 (root)                 |    * 23
|----------------------------------------------|
| Low Base Page Address (Non-MLS Program) OR   |
| # of 128 word sectors                        |    * 24
|----------------------------------------------|
| High Base Page Address + 1 OR                |    * 25
| 1645B for MLS Program                        |
|--+-------------------------------------------|
|//| Program Track #                           |    * 26
|--+-------------------------------------------|
|LU| Swap Track #                              |      27
|--+--------------+----------------------------|
|ID Extension No. | EMA Size                   |      28
|-----------------+----------------------------|
| High Address + 1 of Largest Segment or node  |      29
|----------------------------------------------|
| Time Slice Word                              |      30
|-----------+--+--+--+--+----------------------|
| Open Flg  |SH|DC|CP|DS| Session ID           |      31
|-----------+--+--+--+--+----------------------|
| SCB Pointer                                  |      32
|--+--------------+----------------------------|
|MS|  # pages disc| # pages mem res            |      33
|  |        res   |                            |
|--+--------------+--+--+----------------------|
|MP| # pages dyn  |E |DB| # of swap tracks     |      34
|  |  buf area    |  |  |                      |
|--+--------------+--+--+----------------------|
| start sector address  | LU # of program      |    * 35
|         of prog       |                      |
+-----------------------+----------------------+
```

* = words used in short ID segment

where:

Word 0 is the linkage word for the program. Whenever the program is put into a state (scheduled operator suspend, etc.) the program is put into a linked list threaded through word 0. This word is also used to queue the program up on EQT's for I/O processing.

Words 1-5, called XTEMP, are used dynamically in the ID segment for operating system information regarding the program. Initially at program schedule, the scheduler places the schedule parameters into this 5 word area. For example, a RU,PROGX,1,2,3 would cause words 1-5 in the ID segment to contain 1,2,3,0 and 0 respectively. The scheduler also takes the address of Word 1 and places this into word 10 of the ID segment, the B-Register, at suspension word. When the program starts executing, the system library subroutine RMPAR can be called; it uses word 10 to pick up the run parameters. The words are also used for unbuffered I/O.

Word 1 of the ID segment is also used to specify why a program is in the general wait state. A program can get into the general wait state in eight ways. The reason for being in a state is specified in ID segment word 1 by the following rules:

REASON	CONTENTS OF ID WORD 1
Waiting for Resource # allocation	Address of $RNTB
LU# locked	Bits 6-10 of DRT for LU reference = RN#
Resource # locked	Address of referenced RN #
Waiting for class # allocation	Address of $CLAS
Waiting for Class Get Competition	Address of $CLAS entry referenced
Device (LU or EQT) down	4
Waiting for Son to complete	Son's ID address
Buffer Limited	EQT address

Word 1 is also used by the $ALC routine anytime a program needs more system available memory than is currently available, assuming enough memory can ever be available. In this case $ALC places the number of words requested in Word 1 of the requesting program ID segment. Every time memory is returned through $RTN, Word 1 of the highest priority memory suspended program is checked to see if the memory suspended program can be rescheduled. No lower priority memory suspended programs are suspended until the highest priority memory suspended program is rescheduled.

Word 6 is the priority word. This has the priority of the program. Priorities range from 1 to 32767 for user programs. Occasionally, systems programs give themselves a priority of 0. FMGR does this at Boot up. This allows the program to run at the highest possible priority.

Word 7 is the primary entry point of the program or program segment. It is the relocated address of the first instruction in the program to be executed.

Word 8 is the point of suspension. Each time a program is suspended or interrupted, Word 8 is the address within the program to start the continuation of that program when it is rescheduled. Whenever a program terminates, this word is set to zero. However, if the program terminates saving resources (NOT SERIALLY REUSABLE) word 8 is not reset because to terminate saving resources is to save the point of suspension. Then whenever the program is rescheduled, execution will begin at the address specified in word 8.

This word does have one other use which is not generally known. The word can also be used as a debug tool for the systems level programmer. Since the word always defines the point of suspension, it always defines the area of a program which is in an infinite loop. This is especially useful for the assembly language programmer because the infinite loop location can be quickly pinpointed.

Words 9,10,11 contain the A,B and E/O registers at suspension. Words 12,13 and the upper byte of word 14 contain the five ASCII characters of the program name.

The lower byte of word 14 contains the TM,CL,AM and SS bits plus the type field. Word 15 contains the NA,NP,W,A,O,R and D bits plus the status field. These bits and fields are used as follows:

Word 14:

TM This bit is set if the program is temporary. That is, there is no permanent copy of the ID segment in the system area of the disc. If the bit is clear the program is a permanent one.

ML Memory lock bit. This bit is set by the EXEC 22 request if the user wishes to lock the program into memory and thus prevent swapping.

TS Transportability bit; set if program is transportable.

SS Short segment bit. If set, then the ID segment is a 9 word ID segment used for segments in a segmented program. This is set up by the generator and never changed.

Type This field of word 14 specifies the program type Memory Resident = 1, Real Time =2, Background =3, Large Background =4, Segment =5 (refer to summary of types in user manual).

Word 15:

NA No abort bit. This bit is set if the sign bit of the current EXEC call is set. It informs the system that certain errors are in this request to be handled by the program itself and should not cause the program to be aborted. (MP,RQ,RE,PE,DP, and DM errors will abort the program regardless). Note that setting the sign bit of the EXEC request also increments the normal return address of the EXEC request by one.

NS No suspend bit. This bit will be set by EXEC exactly like the NA bit is set. A CALL EXEC(X+100000B,LU,BUF,LEN) sets the NA bit. EXEC will set the NS bit on a CALL EXEC(X+140000B,LU,BUF,LEN) or CALL EXEC(X+40000B,LU,BUF,LEN) This will be the means for programs to specify no suspension on I/O errors.

NP No parameters allowed on reschedule. This bit is set if no parameters should be passed to the program on reschedule. This bit is set if the program is operator suspended or if a father suspends a son.

W The wait (W) bit is set whenever a program (father) has scheduled another program (son) with wait (EXEC 9 or 23). The son's ID address will be found in word 1 of the father's ID segment.

A This is the abort bit. This bit is set when a program is to be aborted. If the A bit is ever set on a LIST processor entry, no matter what the request, the program is immediately put dormant. The A bit is set by the system on detection of certain errors.

FS The file system bit. FS is set when a program opens a file and is cleared when the program terminates normally or is aborted. D.RTR uses this bit to determine the validity of open flags or files.

O The operator suspend (O) bit. This bit is set when an operator suspension is attempted at a time when it is not feasible to do it directly. The bit indicates that the system should do it at some later time. This is what is meant by deferred action. The system tried to do something, found that it was not feasible, so it wrote a note to itself (set a bit) to remind it to do the requested action as soon as it is feasible. Uncompleted DISC I/O would be one reason for deferred action.

LP Load-in-progress bit, is set while program is being brought into memory from the disc. This inhibits the dispatch of the program until the load is complete. This permits a program to stay in the scheduled list and maintains its timeslice position with programs of the same level.

R R bit, for the most part, is also a deferred action bit in that it indicates how a program is to be set dormant when it is set dormant. (This bit has nothing to do with a serially reusable program termination.) When the program is set dormant the bit is cleared. Word 8 = 0 is the flag by which the system knows that the program terminated saving resources.

D The dormant (D) bit is a deferred action bit which is set if a program cannot be set dormant on request. It indicates that the program is to be set dormant as soon as feasible.

Status - The current state of the program. States are 0, 1, 2, 3, 4, 5, 6 - dormant, scheduled, I/O suspend, general wait, memory suspend, disc suspend, and I/O suspend respectively.

Words 16,17,18 and 19 contain time scheduling information about the program. The four words are used in the operator command *ST,PROGX to give time information about the program.

Word 16 is the time list linkage word. All programs in the time list will be linked together through this word.

RES (bits 15-13) in word 17 contains the resolution code. Multiple (bits 11-0) contain the multiplier for the resolution. The T bit is set if the program is in the time list. Words 18 and 19 contain the system time in 10's of milliseconds of when the program is to execute next. The two words give a 10 millisecond resolution for a 24 hour period. Word 19 contains the high order bits of the time.

Word 20 of the ID segment contains the BA,FW,M,AT,RM,RE,PW and RN bits plus the father ID segment number field. They are used as follows:

BA Batch bit. This bit is set if the program is running under batch Program JOB or the FMGR :JO command set this bit. The batch bit is propagated from father to son. That is, if the father is under batch and schedules a son the son's BA bit will be set.

FW Father waiting bit. If the father is scheduled with wait (EXEC 9 or 25), the son's FW bit is set. If the father is scheduled W/O wait, the bit is clear.

M Multi-Terminal Monitor Bit. This bit is set if the program is operating under the multi-terminal monitor mode. Like the BA bit, this bit is propagated from father to son.

AT Attention bit also called the break bit. This bit is set by the BR operator command and cleared by the IFBRK system library routine and program termination.

RM Reentrant memory moved bit. This bit is set if the program has information (Temporary Data Block) in System Available Memory that must be moved into the program area before the program can continue to execute.

RE Reentrant routine in control now. This bit is set anytime a reentrant subroutine of this program is executing.

PW Program wait bit. This bit is set when some program wishes to schedule this program with wait (EXEC 9 or 25) but this program is currently active. The perspective father will be in the general wait state with the prospective son's ID address in word 1 of the prospective fathers ID.

RN This bit is set when a resource number is either owned or locked by this program.

Father The field is used if this program is a son. The field will have the ordinal number of the father's ID segment. The number will be there regardless of the type of schedule; i.e., EXEC 9,10,23 or 24. The least significant bit is also set if the program terminates serially reusable. This bit is a flag for the dispatcher to avoid certain program clean up procedures. The bit is cleared later.

Word 21 contains the RP bit, the # of pages field, memory protect fence index (MPFI) field, the deferred end bit (DE) and the partition number field.

RP Reserved partition bit. This bit is set if the program is assigned to a partition. The partition number will be in the partition number field. The numbers start counting from 0.

# of Pages This field contains the number of pages the program takes up not counting base page. For segmented programs it is the # of pages of the main, subroutines and largest segment. For EMA programs the size includes main, subroutines, largest segment and the MSEG size.

For non EMA Programs
1 <= # of pages <= MAXIMUM LOGICAL ADDRESS SPACE (in pages)

For EMA Programs
2 <= # of pages <= MAXIMUM LOGICAL ADDRESS SPACE + MSEG SIZE

MPFI This is the memory protect fence index field. This field contains an index (0-5) which, when added to the start of the memory protect fence table, gives a location containing the proper memory protect address for this program. This is set by the LOADR or generator and does not change.

DE Defer EXEC 6 command. The loader sets this bit when the profile option is set (i.e., when the command PF,<lu#> was entered or profile <lu#> was specified in the RU,MLLDR runstring). The scheduler will check this bit against the NS bit and NA bit to determine if this is a deferred termination or a real one. If the NS bit and NA bit are set, the normal termination is performed.

Partition No. This field contains the partition number that the program last executed in. (Counts from 0.)

Word 22 contains the low main address of this program.

Word 23 is the High Main Address + 1. For multi-level programs, this is the address of the first word after the root. This value will be the value of the loader library flag TH2.L after relocation of the root and all processing required before any next node. If there are any memory resident nodes, word 23 is the first word address of the next page (which is where the memory resident node would be relocated). This can be calculated as follows.

```
      LDA TH2.L        get the high word so far
      AND M1777        mask off the page
      SZA,RSS          exactly one page boundary?
      JMP DONE         yes
      LDA TH2.L        no, get high word again
      IOR M1777        calc. end of the page
      INA,RSS          inc. to next page
DONE  LDA TH2.L        TH2.L is correct as is
      STA WRD23        this is it.
```

where:

M1777 OCT 1777

If there are disc resident nodes only (besides the root) TH2.L is the correct value.

Actually, the steps described are just those steps required to update TH1.L (the loader library routine flag for base address of current module) after relocation of the root and to be ready for the next node. Another way of looking at this word is that it is the value of TH1.L after loading the root and being updated.

Word 24 contains the low base page address of the program or the number of sectors, if an MLS program.

Word 25 is the High Base Page Address + 1. For multi-level programs this word is set to 1645B, the last available link address + 1 for RTE-6/VM. If loaded by MLLDR, then the program is multi-level and this word is set to 1645B.

Word 26 contains the disc track # of the virgin copy of the program on the disc.

Word 27 is formatted similar to word 26, but is the swap track # in the track pool for the program. If the program is on LU 2, bit 15 is clear; if the program is on LU 3, bit 15 is set.

Word 28 is used only for EMA programs and is zero for non EMA programs. Bits 15-10 contain the original number of the 3 word ID extension associated with this program. Bits 9-0 contain the EMA size of this program. The value here will be 1 if a default EMA size is taken and the program has not yet run. Else the value will be a minimum of 2 to the maximum size of the largest partition minus the program size.

Word 29 is the High address + 1 of largest segment or node. For MLS-LOC programs this is the longest path in the tree + 1. The M.ABT routine which puts out the absolute program word, updates this word each time a higher address is output. At the end of the load, it will automatically be the highest address of the entire load. If any memory resident nodes (besides the root) exist, this address is bumped up to the next page boundary.

```
     LDA WRD29 - get the high word
     AND M1777 - mask off the page
     SZA,RSS   - exactly on page boundary?
     JMP DONE  - yes
     LDA WRD29 - no, get the word again
     IOR M1777 - calc. end of the page
     INA       - inc. to next page
     STA WRD29 - store as new high
DONE
```

Word 30 is the Time Slice word. This word defines the time slice of the program and has the following states:

=1: The program has just been placed into the scheduled list and has not been dispatched (or redispatched) or the program is not being time sliced.

=0: The program is not scheduled or has used a full time slice.

>0: The program is currently running under time slice control or was bumped from execution by a higher priority program. This word represents the remaining time slice for this program (in 10's of milliseconds).

Word 31. The Termination Sequence counter (SEQCNT) is incremented each time a program completes or aborts (except Save Resources termination). This value is used by the FMP in the definition of a valid open file. The counter is the property of the ID segment, not the program currently resident in the ID. Therefore, these bits are not altered when LOADR or FMGR builds an ID for a new program.

The SH bit is the sharable EMA flag.

The DC bit indicates that the program may not be duplicated. Since permanent programs are now copied, (D.RTR, SMP, GASP), this is needed as certain programs must not be copied. The generator sets the DC bit whenever the primary type code is expanded by adding 128 to it. The LOADR also provides an op code (NC) to provide for the setting of the DON'T COPY bit.

The CP bit indicates that this program is a copy of another program. It is set by the FMP routine IDDUP and is used in the killing of ID segments.

LOADR checks the DC and CP bits during purge operations. If the CP bit is set, the ID is just killed. If the DC bit is set, the standard purge operation is performed. Otherwise, (i.e., the program is not a copy, but can be copied) LOADR checks to see that no copies exist of the program. If copies exist, LOADR rejects the purge and an error is issued.

The DS bit is reserved for DS/1000 usage.

The session ID is used by program LGOFF to give back ID segments at session termination. It identifies the creator of this ID segment. All ID segments created by the session are killed at log off. The one exception here is that if the program occupying the ID segment is currently active and was run by another session, then the ID segment is given to that session. The session identifier in the ID segment is changed to that of the other session. Note that programs created under the MANAGER.SYS account or programs not created under session will have an ID = 0. Therefore, the LGOFF program will not remove programs thus created.

Word 32. In session systems word 32 is the SST address of the session this program is operating under.

In MTM systems word 32 is the -LU of the terminal associated with the program.

When operating in the non-session environments, word 32 is set to 0.

Note that this word is passed down from father to son programs.

Word 33 Bit 15, multilevel segmentation bit. This bit is set by the MLS-LOC loader for every program that MLLDR loads.

Word 33 Bits 14-10, # of pages of the longest disc resident node path minus the the root. If there are no disc resident nodes, this field is set to zero (i.e., no D commands occur). Otherwise, the value is calculated as follows:

When the first D command is encountered, the current address is saved (this is the end of the main + 1 or the first word of the disc resident node) in LDK.M. The M.ABT routine which puts out the absolute program word also keeps track of the high address + 1 for disc resident nodes, HDK.M. This word is updated each time a higher address is outputted for a disc resident node. At the end of the load, the HDK.M and LDK.M addresses are used to calculate the longest disc path (minus the root) in pages. Note that if no D commands occur, HDK.M and LDK.M will be equal.

```
      LDA LDK.M - get the low disc address
      CMA,INA   - make it minus
      ADA HDK.M - high disc addr - low
      ADA M1777 - bump to next page
      AND M0760 - get the page bits
      IOR WRD33 - merge into word 33
      STA WRD33 - and store it

M1777 OCT 1777
M0760 OCT 76000
```

If there are memory resident nodes besides the root, one more page must be added for disc resident base page.

Word 33 Bits 9-0, # of pages of root and memory resident nodes. If no memory resident nodes besides the root are specified, this field is set to zero. Otherwise, this field will have the total number of pages in the root and all memory resident nodes. When the first disc resident node is encountered, the high track and sector written to so far is saved in MTK.M and MST.M. The number of sectors per track is saved in MS#.M.

When final ID segment processing is done, a check is made to see if the root was the only memory resident node. If so, this field is set to zero. Then a check is made to see if there were any disc resident nodes. If there were, MTK.M, MST.M and MS#.M are set already. Otherwise, they must be set. If not set, they are set to the current high track and sector bumped up to the next even sector. MS#.M is set to the current disc's number of sectors per track. Now the following calculation takes place:

```
    LDA MTK.M       high track
    CLB             multiply to get
    MPY MS#.M       # of sectors
    ADA MST.M       add left over
    CLB             now get # of pages
    DIV P16         do not count
    ADA N1          1st base page
```

Word 34 Bit 15, save maps bit. This bit is set by the code segment mapping microcode every time that code is called. It is used to inform the operating system that the user map has been modified and must be saved on a context switch. This bit could be set by single level programs to get a change in the maps saved.

Word 34 Bits 14-10, number of pages of dynamic buffer area. This is set for multi-level programs. It is the # of pages between the last page of program code and the start of EMA (or the end of the allocated program area). ID segment word 29, high address + 1 of largest segment or node, is the last page of program code, LSTPG.

If a size was specified, LSTPG is subtracted from the size given and the result is the # of pages of dynamic buffer area.

If no size was specified, the # of pages of dynamic buffer area is set to zero. If an incremental size was given, the # of pages of dynamic buffer area is equal to the increment.

Word 34 Bits 7-0, # of swap tracks used to swap the code area and EMA of the program.

Word 35 Bits 15-8, start sector address of the program.

Word 35 Bits 7-0, LU # of the program. The LU # of the disc where the program image resides. For SP'd programs, this can be other than LU 2 or 3.

ID EXTENSION

For programs using EMA, not only does an ID segment have to be set up, but an ID extension must be set up also. Word 28 bits 15-10 in the ID segment give the ID extension number associated with this program.

Words 0 and 1 of the ID extension contain the NS bit, DE bit, the current MSEG field, the MSEG Size field, starting prog page MSEG field, and start page EMA field.

ID extension word 2 is now only filled in for programs using sharable EMA. ID extension words 3 and 4 are filled in for programs using VMA.

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+----------------------------+--------------+
|NS| Current MSEG #             | # Pgs mseg   |  Word 0
|--+-----------+--+-------------+--------------|
| MSEG start   |DE| (Physical) EMA start page  |  Word 1
| PG (logical) |  |                            |
|--+--+--+-----+--+----------------------------|
| C|  |S |        |                            |  Word 2
| O|  |W |        |         INDEX #            |
| M|  |  |        |                            |
|--+--+--+--------+----------------------------|
| LAST PAGE OF VIRTUAL MEMORY                  |  Word 3
|----------------------------------------------|
| RESERVED                                     |  Word 4
+----------------------------------------------+
```

NOTE: Word 0 is set up by the loader, but is not currently used by the operating system.

Word 0, Bit 15 NS = 0 if the MSEG is pointing to a standard segment of the EMA (set up by .EMAP)

= 1 if the MSEG is pointing to a non-standard segment (set up by .EMIO or .EMAP)

Word 1, Bit 10 DE = 0 if the EMA size was specified by the user
Bit 10

= 1 if the EMA size is allowed to default to the maximum size available to the system

Word 2, Bit 15 COM — The COM bit is set to indicate that this is a shared EMA program instead of a local EMA program. Thus, this bit can be set once a loader SH command is encountered.

Word 2, Bits 7-0 — Index number. This is the index into the $EMTB table for the label specified in the SH,<label> command. When the $EMTB table is searched for the existence of <label>, the index is updated as as the table is searched. Thus, when the label is found, the current index saved is the correct index number to place in the ID extension. The first $EMTB table entry has an index of zero.

Word 3 — This word is set equal to the last page of VMA.

Word 4 — (Not currently used.) This word is reserved for use by the VMA microcode.

Dispatcher Interface to List Processor — APPENDIX E

As described in Chapter 1, $LIST pushes programs that terminate into a stack through word 8 of the ID segment headed at $ZZZZ in the dispatcher. The dispatcher uses this stack for program clean up. Every time the system has nothing else to do, it jumps to $XEQ in the dispatcher. Every time the system goes to $XEQ it first checks $ZZZZ. If it is non-zero, it does the following to clean up every program:

First it sets the program's point of suspension (ID word 8) to 0 in case the program is to be run later. Next, if the program is disc resident, any swap tracks the program has are released. This can happen if a swapped-out program is aborted. A father may also abort his son, causing this condition. The tracks are released by $DREL in the OS1EX. $DREL also makes a call to $SDSK, which calls $LISR in the scheduler to reschedule any programs that have been waiting for disc space.

The dispatcher then calls $ABRE in EXEC to return any reentrant memory the program has. This can happen if a program terminates or is aborted while in a reentrant subroutine. If the $ABRE routine returns any memory via the $RTN subroutine in $ALC, programs waiting for memory can be rescheduled by calls to the list processor.

Next a call is made to the $RTST routine in the scheduler. This routine returns any string memory the program owns. If any memory is returned, programs waiting for memory can be scheduled by a call to the list processor. The system then calls $WATR in the scheduler to schedule any programs that made SCHEDULE WITH QUEUE requests (EXEC 23,24) for the program. $WATR calls $SCD3, which calls $LIST for any such programs. $SCD3 scans the general wait list (major state=3) looking for entries that have Word 1 of their ID segment equal to the ID segment address of the terminating program. Programs in the general wait list will have Word 1 of their ID segment set as follows:

REASONS	CONTENTS OF ID(1)
WAIT TO SCHEDULE A PROGRAM	Program's ID segment address
WAIT FOR COMPLETION OF A SON	Son's ID segment address
RN ALLOCATE WAIT	Address of RN table
RN LOCK WAIT	Address of RN number
LU LOCK WAIT	Address of RN number associated with LU LOCK
DOWN DEVICE	4

BUFFER LIMIT EXCEEDED	Address of EQT on which limitation was exceeded

NOTE: This call also handles programs that schedule with QUEUE.

After $WATR returns, the system calls $TRRN which in turn calls $ULLU to unlock any lock LUs the program has. $TRRN also unlocks any local RN locks and deallocates any local RN allocates the program has. Each of these processes may call $SCD3 to pick up and schedule waiting programs. If there are any such programs, $SCD3 will call $LIST.

Next, if any EQTs are locked, a call to $EQCL is made to unlock the EQTs. Now the program is examined to see if it is a sharable EMA program. If it is, the number of active users of that partition is decremented. If the program is the last user, the partition is freed by a call to $ECLR. A call is then made to $F.CL to determined if any class I/O must be cleaned up.

Lastly, if the program is a disc resident program and the program still owns the partition but did not terminate serially reusable, that partition is released and made available to other programs.

$LIST Calls used in DVR00, DVR05, DVR37

DVR00 and DVR05 both schedule the system program PRMPT with a $LIST call. In addition, the B register at suspension (word 11) is set to point to EQT word 4 of the EQT of the interrupting device. Both of these functions are performed by one $LIST call. The calling sequence is:

```
JSB $LIST
OCT 601
OCT IDADR <ID ADDRESS>
OCT BVAL  <THIS VALUE PLACED INTO ID WORD 4>
```

The HP-IB driver (DVR37) will schedule a program on interrupt and pass three words into the temporary area of that programs ID segment. The $LIST calling sequence is:

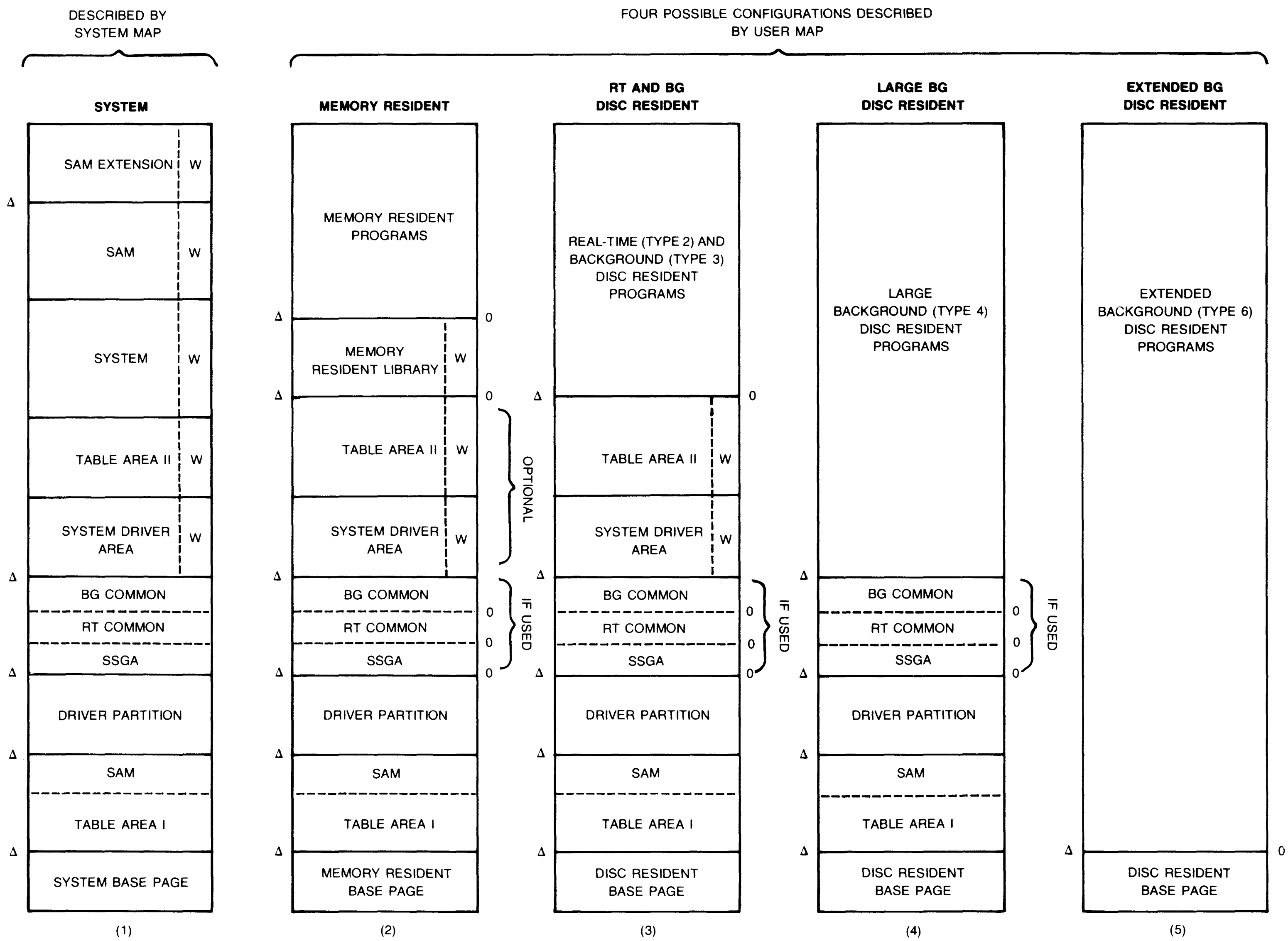
```
JSB $LIST
OCT 1
DEF RTN
DEF IDADR
DEF P1
DEF P2
DEF P3

RTN
```

Memory Addressing Spaces

APPENDIX F

Figure F-1 shows Memory Addressing Spaces:

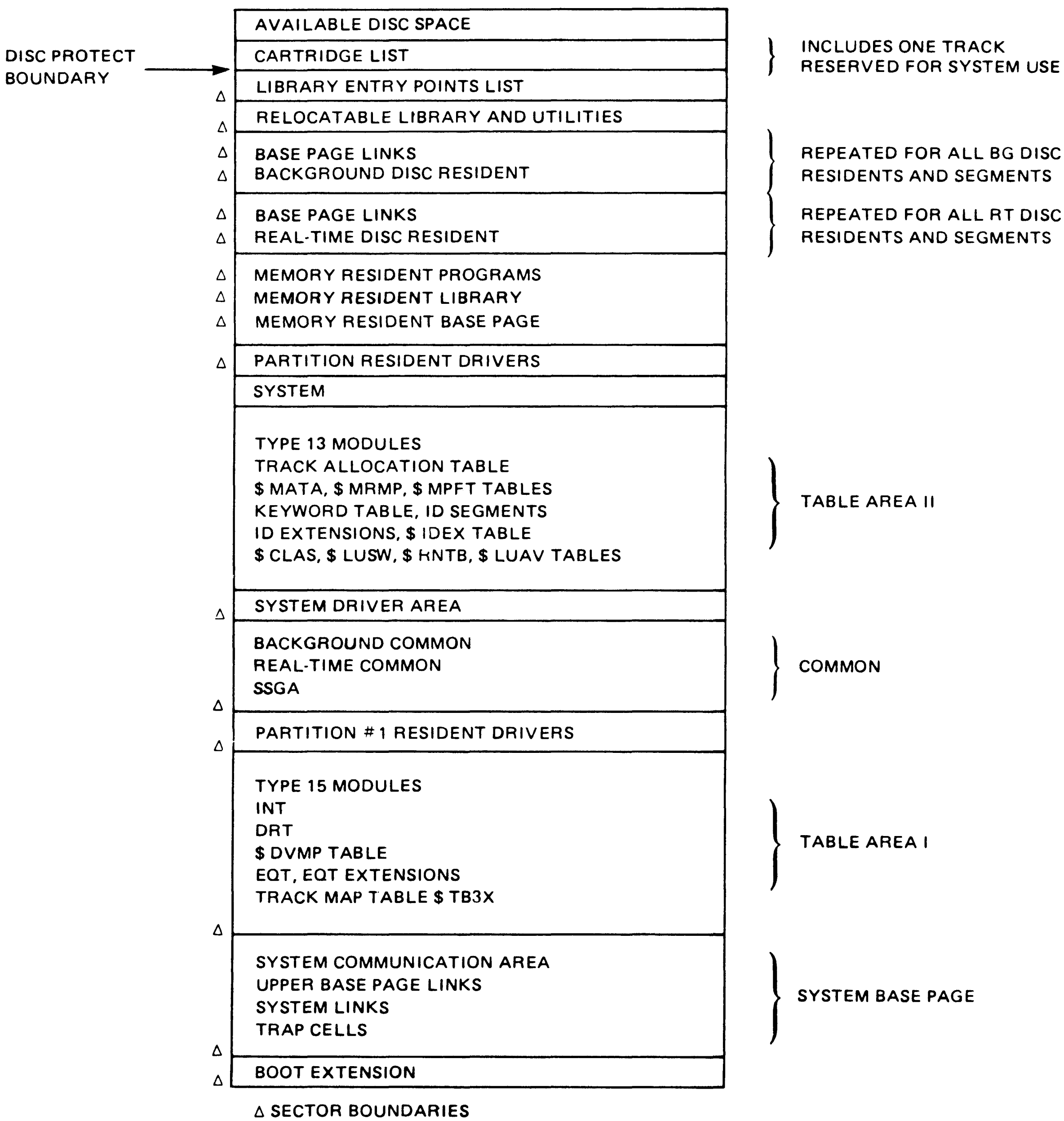


Δ = PAGE BOUNDARIES
W = WRITE PROTECT
0 = MEMORY PROTECT FENCE SETTINGS

8100-31

System Disc Layout	APPENDIX G

Figure G-1 shows the RTE-6/VM System Disc Layout.



Entry Point Layout on Disc

APPENDIX H

The entry-point layout on disc is shown below:

```
                DSCLB (1761)
                |
                |                                  END OF SYSTEM
DSCUT (BP 1763  |<--SYSLN-->|<-------DSCLN--------->|
|               |   (1764)  |        (1762)         |
|               |           |                       |
V               V           V                       V
----------------------- ---------------------\\-----------------\\
|               |           |           |           |
|DISC-          |SYSTEM     |USER-      |USER       |     SYSTEM
|RESIDENT       |ONLY       |AVAILABLE  |ENTRY      |     AVAILABLE
|LIBRARY        |ENTRY      |SYSTEM     |POINTS     |     TRACKS
|ROUTINES       |POINTS     |ENTRY      |           |
|               |           |POINTS     |           |
---------------------------------------------\\-----------------\\
^               ^          ^            ^          |
|               |          |            |          |
|               |          |            |         TRACK BOUNDARY
TYPE 6,7,14     |          |            |
SUBROUTINES     |          |           TYPE 6,7,14
               TYPE 0,16   |           MODULES
               MODULES     |
                           |
                          TYPE 13,15,30
                          MODULES.
                          ALL ABS, RP,
                          COMMON
```

NOTE: SYSLN AND DSCLN CONTAIN THE NUMBER OF 4-WORD ENTRIES

Entry Point Layout on Disc

Library entry points list format:

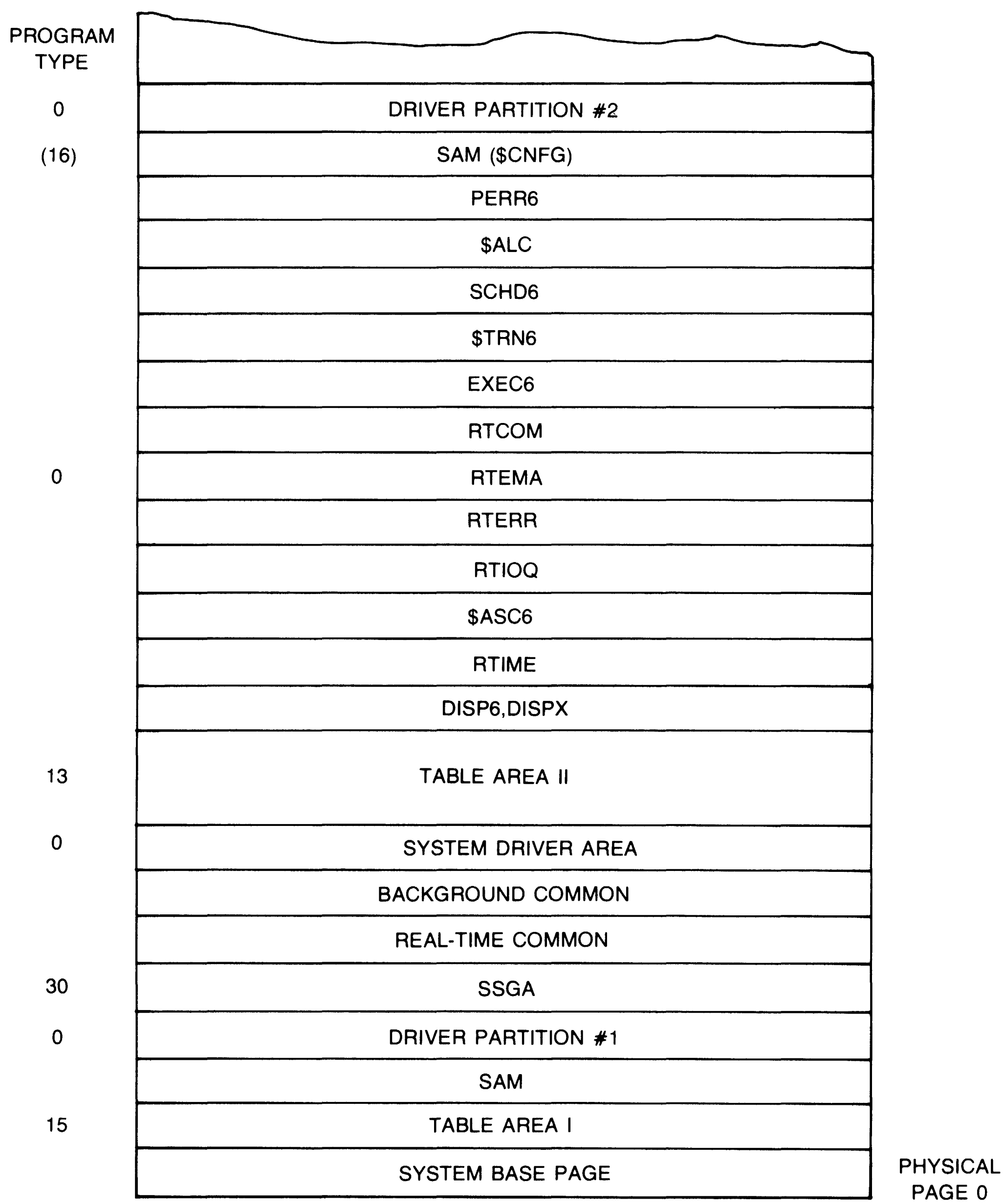
```
Word 1 - Name 1,2
Word 2 - Name 3,4
Word 3 - Name 5, flag bits
Word 4 - Value: memory address,
                microcode instruction
                absolute value
                track (bits 15-7) and (bits 6-0) sector address
```

Flag bits:

```
000 - Memory resident entry point
001 - Disc-resident subroutine
010 - Common entry point
011 - Absolute
100 - Replace
```

Physical Memory Allocation — APPENDIX I

Figure I-1 shows RTE-6/VM Physical Memory Allocation.



8100-46

TIMESLICE QUANTUM DEFAULT

The default timeslice quantum is defined as 1.5 seconds. The major factor considered in determining the default slice was the percentage of time spent swapping the timeslice programs as compared to the execution slice allowed.

To arrive at a slice value (x), the following assumptions were made:

```
program size = 16K (swap time (y) = 300 ms).
y/x          = .20 (swap time in relation to execution
                slice).
```

Plug in the swap time to get .300/x = .20 or 1500 ms (1.5 sec).

The swap overhead is defined as:

	7925	7920	7905
Worst case seek time:	48.5ms	45ms	45ms
Average latency	11.1ms	8.33ms	8.33ms
Total access time =	59.6ms	53.33ms	53.33ms

Using an average access time of 55ms for the above discs, the time to swap a program (16K) would be:

```
   90ms of data transfer
  +55ms of access time
 -----
 =145ms
  x 2
  ---
  290ms = worst case swap of a 16K program.
```

Timeslice Quantum Definition

The following section is intended for those familiar with the RTE-III and RTE-IV Dispatcher, Scheduler and RTIME modules.

TIMESLICING AND THE DISPATCHER

The following routines make special checks to provide for the the timeslicing function.

Switching section (X0010) - If the program from the scheduled list has equal priority and current program (XEQT) have used a full timeslice, then attempt execution switching.

The partition search routine (FNDSG) - When searching the allocated list, if the resident is of equal priority (with program in scheduled list) and has used a full timeslice (indicated by ID word 30 containing a zero value), allow the allocation of the partition to the new program.

The swap check routine (SWPCK) - If the residents priority is equal to the contender and the resident is flagged as having used a full slice allow the swap.

Program execution setup (X0042) - Note that this section is entered only when a new program is to be dispatched. If the current program (XEQT) is to be reentered, set-up is at $RENT.

If priority of new program is <timeslice limit (default is 41 but this value may be changed via the "QU" operator command), setup a dummy timeslice location and continue with the dispatch.

If the priority of new program is > timeslice limit, set the address of ID 30 as the timeslice location ($LICE). If the contents of ID word 30 is negative (remaining timeslice count), continue with the dispatch.

If the remaining count is > 0 (from ID word 30), calculate a full timeslice value using the following equation:

Program slice value = SYS Slice * Z + SYS Slice

where: SYS Slice = 1500ms default. This value (1.5 seconds) may be increased or decreased via the "QU" operator command.

If the new program is under session control, set the CPU usage word ($CPU) to point at the programs session control block word 10.

If not under session, set the CPU usage pointer to point at a dummy system location.

TIMESLICING AND RTIME

Before checking for time scheduled programs (CL010), RTIME performs the following functions:

1. Increment the slice counter (indirect through $LICE). If not zero, continue. Else, if the currently executing program's priority is > timeslice limit and another program of the same priority is scheduled, relink (VIA $RLNK) the currently executing program and force a new dispatch (i.e., put it behind all other programs of the same priority. Refer to the section on Timeslicing and the Dispatcher for details on the dispatch of the new program.

   Note that a zero value in ID word 30 indicates the usage of a full timeslice.

2. Increment the double word CPU usage word (indirect through $CPU).

TIMESLICING AND SCHEDULING

A new subroutine was created to provide the RTIME module with a very fast method for relinking a program in the scheduled list (within its own priority level). All it does is call the link processor to remove the XEQT programs from the scheduled list and then insert it into the scheduled list.

```
+--------------------------------------------------------+-------------------+
|                                                        |                   |
|   Session Monitor Tables                               |   APPENDIX  K     |
|                                                        |                   |
+--------------------------------------------------------+-------------------+
```

This appendix contains information on the following:

* SESSION CONTROL BLOCK (SCB)

* SESSION SWITCH TABLE (SST) AND CONFIGURATION TABLE

* SESSION TABLE RELATIONSHIP

SESSION CONTROL BLOCK (SCB)

A Session Control Block (SCB) is established for each user who has successfully logged-on to the system. The SCB contains the information necessary to identify the user to the system and describe his capabilities in terms of command processing and I/O addressing space The format of the SCB is shown below.

```
WORD
             +-------------------+
0  $SHED  >|   List Linkage    | > 0
             +-------------------+
1            |   SCB Length      |
             +-------------------+
2            |   Reserved        |
             +-------------------+
3            |   Identifier      |
             +-------------------+
4            |   Directory #     |
             +-------------------+
5            |   Capability      |
             +-------------------+
6            |                   |
             |                   |
7            |   Error           |
             |   Mnemonic        |
8            |                   |
             |                   |
9            |                   |
             +-------------------+
```

```
10         |                  |
           |   CPU Usage      |
11         |                  |
           +------------------+
12         |   User ID        |
           +------------------+
13         |   Group ID       |
           +------------------+
14         |   Disc Limit     |
           +------------------+
15         |   -SST Length    |  ID Segment Session Word
           +------------------+
           | Sys LU |  Ses LU |
           |        |         |
           |   .    |    .    |
           |   .    |    .    |
           |        |         |  P = Added SSt entry for this disc
           |        |         |  G = This is a group cartridge
           +------------------+  I = This Cartridge is idle
           | -Disc Limit Ctr  |
           +------------------+
           |P|G|I|    LU      |
           +------------------+
```

SESSION SWITCH TABLE (SST) AND CONFIGURATION TABLE

When operating in the session environment, every I/O request is routed to the appropriate I/O device via the Session Switch Table (SST). Each SST entry describes a session LU, which the user addresses, and associated system LU where the I/O request will actually be directed. The SST describes the session user's I/O addressing capabilities by defining the system LUs the user has access to and the associated session LUs by which the user accesses them.

When the user makes an I/O request, the SST is searched for the specified session LU. If the requested LU is found, it is switched to the associated system LU as specified in the SST entry and the I/O request is processed. If the requested LU is not found, an error is returned (IO12-LU not defined for this session).

The Session Switch Table is maintained in memory as part of the Session Control Block (SCB). The format of the SST is shown below.

```
+---------------------+
|                     |
|   SCB      .        |
|            .        |
|            .        |
|                     |
+---------------------+
| Negative SST length | Session word (word 33 of
+---------------------+  program ID segment)
|  Sys LU  | Ses LU   |
+---------------------+
|     .    |    .     |
+---------------------+  SST
|     .    |    .     |
+---------------------+
|     .    |    .     |
+---------------------+
|     .    |    .     |
+---------------------+
|                     |
|   SCB      .        |
|            .        |
~                    ~
```

System LUs can be integer numbers between 1 and 254. Session LUs can be integer numbers between 1 and 63. Session LUs are assigned:

* at log-on, via user and group account file entries, or

* at log-on, via Configuration Table entries, or

* On-line using the SL command (refer to the RTE-6/VM Terminal User's Reference Manual for a description of SL).

The Configuration Table describes the default logical units to be used for specific device logical units. Each station (terminal) logical unit defined in the Configuration Table has associated with it a set of device logical units which are assigned default logical units to be used when a user logs on at this station (terminal). The default logical unit associated with the station itself is always 1.

at log-on, these default values are written from the Configuration Table in the account file into the user's session Control Block (SCB), unless overridden by entries in this particular user's SST. The format of the Configuration Table is shown below.

```
+------------------------------+
|            LENGTH            |
|------------------------------|
| STATION LU  |       1        |
|------------------------------|
| SYSTEM LU   |  DEFAULT LU    |
|------------------------------|
|            LENGTH            |
|------------------------------|
| STATION LU  |       1        |
|------------------------------|
| SYSTEM LU   |  DEFAULT LU    |
|------------------------------|
| SYSTEM LU   |  DEFAULT LU    |
|------------------------------|
| SYSTEM LU   |  DEFAULT LU    |
|------------------------------|
|              .               |
|              .               |
|              .               |
|------------------------------|
|              0               |
+------------------------------+
```

The account file structure is shown below.

ACCOUNT FILE STRUCTURE

```
RECORD   ----------------------------------
         |                                |
  1      |      ACCOUNT FILE HEADER       |
         |                                |
         |--------------------------------|
         |                                |
 2-N     |      ACTIVE SESSION TABLE      |
         |                                |
         |--------------------------------|
         |                                |
         |      CONFIGURATION TABLE       |
         |                                |
         |--------------------------------|
         |                                |
         |      DISC ALLOCATION POOL      |
         |                                |
         |--------------------------------|
         |                                |
         |       USER-GROUP ID MAP        |
         |                                |
         |--------------------------------|
         |                                |
         |          DIRECTORY             |
         |                                |
         |--------------------------------|
         |                                |
         |        USER AND GROUP          |
         |        ACCOUNT ENTRIES         |
         |                                |
         |               .                |
         |               .                |
         /               .                \

         /                                \
         |                                |
         ----------------------------------
```

ACCOUNT FILE HEADER

WORD		
1	LOCATION OF ACTIVE SESSN TABLE #	
2	LOCATION OF CONFIGURATION TBL	
3	LOCATION OF DISC POOL	
4	LOCATION OF USER/GROUP ID MAP	
5	LOCATION OF DIRECTORY	
6	LOCATION OF 1ST ACCOUNT ENTRY	
7-9	SYSTEM MESSAGE FILE	
10	SECURITY CODE	
11	CARTRIDGE	
12	# OF CHARS IN PROMPT STRING	<--0 if using def prompt
13-22	PROMPT STRING	
23	LOWEST PRIVATE ID USED	
24	HIGHEST GROUP ID USED	
25	RESOURCE NO.	
26	LU # OF MSG. FILES	
27	I MEMORY ALLOCATION SIZE (WDS)	If bit 15=1, use monitor memory prompt
28	- SESSION LIMIT	
29	NUMBER OF ACTIVE SESSIONS	
30	SHUT DOWN FLAG	
31	COPY OF SESSION LIMIT	
32	CLASS NUMBER	
33	LENGTH OF CONFIG TABLE	
34	IRN2	
35	DISC POOL LENGTH	

ACTIVE SESSION TABLE

```
WORD      +--------------------------------+
 1        | LOGICAL UNIT (0 IF FREE ASB)   | <------
          |--------------------------------|       |
 2        |                                |       |
 3        |        LOG-ON TIME**           |active session block
          |                                |     (ASB)
          |--------------------------------|       |
 4        |  DIRECTORY ENTRY NUMBER        |       |
          |--------------------------------| <------
          |                                |
          |               .                |
          |               .                |
          |               .                |
          +--------------------------------+
```

DISC ALLOCATION POOL

```
           15 14        8 7            0
WORD      +----------------------------+
 1        |  |  *       | LOGICAL UNIT |
 2        |  |  *       | LOGICAL UNIT |
 3        |  |  *       | LOGICAL UNIT | bit 15 = 1 if disc
 .        |  |  .       |      .       | has been allocated
 .        |  |  .       |      .       |
 .        |  |  .       |      .       |
          |  |          |      .       |
          |  |          |      .       |
          |  |          |      .       |
128       |  |          |      .       |
          +----------------------------+
```

* RESERVED FOR FUTURE USE

** LOG-ON TIME FORMAT

```
WD1  15-13        12-7      6-0
WD1  -----        ----      ---
WD1  year offset  min       sec
     from 1978

     15-14        13-5      4-0
     -----        ----      ---
WD2  reserved     day       hour
```

USER/GROUP ID MAP

```
        15                              0
WORD    ---------------------------------
 1      | | | | | | | | | | | | | | | | | bit 15 = 1 if ID is
        --------------------------------- assigned to an account
 2      | | | | | | | | | | | | | | | | |
        ---------------------------------
 3      | | | | | | | | | | | | | | | | |
        ---------------------------------
 .      |               .               |
 .      |               .               |
 .      /               .               /

        /                               /
        |                               |
        ---------------------------------
256     | | | | | | | | | | | | | | | | |
        ---------------------------------
```

ACCOUNT FILE DIRECTORY

```
WORD   +---------------------------------+
 1     |  # CHARS      | # CHARS GROUP   |  0 = end of directory
       |---------------------------------| -1 = free entry
 2     |                                 | -2 = extension
 3     |              USER               |
 4     |              NAME               |
 5     |                                 |
 6     |                                 |
       |---------------------------------|
 7     |                                 |
 8     |              GROUP              |
 9     |              NAME               |
10     |                                 |
11     |                                 |
       |---------------------------------|
12     |             USER ID             |(0 if entry is for
       |---------------------------------|a group account)
13     |            GROUP ID             |
       |---------------------------------|
14     |       GROUP ACCT RECORD #       |if bit 15 = 1, account
       |---------------------------------|is in second 64 words
15     |       USER ACCT RECORD #        |(0 if entry is for a
       |---------------------------------|group account)
16     |                *                |
       +---------------------------------+
```

* RESERVED FOR FUTURE USE

USER ACCOUNT ENTRY

```
         15            6 7              0
WORD     +-------------------------------+ if bit 15 = 1,
  1      | I      *    | CHARS IN PASSWD | account extends
         |-------------------------------| to second block
 2-6     |           PASSWORD            |
         |-------------------------------|
 7-9     |        USER HELLO FILE        |
         |-------------------------------|
 10      |         SECURITY FILE         |
         |-------------------------------|
 11      |           CARTRIDGE           |
         |-------------------------------|
12-16    |               *               |
         |-------------------------------|
17-19    |       USER MESSAGE FILE       |
         |-------------------------------|
20-21    |                               |
         |-------------------------------|
 2P      |          CAPABILITY           |
         |-------------------------------|
23-24    |       LAST LOG-OFF TIME       | (same format as ASB)
         |-------------------------------|
25-26    |   CUMULATIVE TIME (MINUTES)   | 2 words
         |-------------------------------|
27-28    |     CPU USAGE (SECONDS)       | 2 words
         |-------------------------------|
 29      |            USER ID            |
         |-------------------------------|
 30      |           GROUP ID            |
         |-------------------------------|
 31      |          DISC LIMIT           |
         |-------------------------------|
 32      | GRP.SST LENGTH | #SST SPARES  |
         |-------------------------------|
 33      | USER/GROUP SST LENGTH (TOTAL) |
         |-------------------------------|
  .      |  SYSTEM LU     |  SESSION LU  |
  .      |     "          |      "       | user SST
  .      |     "          |      "       |
         |-------------------------------|
         |  SYSTEM LU     |  SESSION LU  |
         |     "          |      "       | group SST
         |     "          |      "       |
         |-------------------------------| if bit 15 of word
 64      |                               | 1 = 1, this word is
         +-------------------------------+ record number of 2nd
                                           block of account
```

GROUP ACCOUNT ENTRY

```
          15                            0
WORD      +-----------------------------+
  1       |          GROUP ID           | bit 15 = 1 indicates
          |-----------------------------| account extends to
  2       |      CUMULATIVE TIME        | second half of block
  3       |                             |
          |-----------------------------|
  4       |    CUMULATIVE CPU USAGE     |
  5       |                             |
          |-----------------------------|
  6       |     - GROUP SST LENGTH      |
          |-----------------------------|
  .       |  SYSTEM LU    |  SESSION LU |
  .       |       .       |      .      |
  .       |       .       |      .      |
          |       .       |      .      |
          +-----------------------------+
```

COMMAND TABLE

A listing of the operating system command capability table, $CMND, appears on the following pages. Each command is defined by a two-word entry of the form:

```
 15 14             8 6              0
+-----------------------------------+
|        CHAR 1    |      CHAR 2    |
+-----------------------------------+
|  P| R|               | NUM        |
+-----------------------------------+
```

Where: CHAR1 and CHAR2 = the two character ASCII command

P = 0 If any number of parameters allowed.

= 1 If limitation placed on number of parameters allowed.

NUM = Maximum number of parameters allowed with command (specified when P=1).

R = 0 No reference check required.

= 1 Program specified for first parameter of command must be attached to session (program ID segment word 33 must equal caller word 33 of caller) or program must be non-session (word 33 equals zero).

The command capability level associated with a command will be determined by the position of the command entry relative to level pointers located at the head of the table. Refer to the listing for details.

If you wish to substitute your own command table for the HP supplied table, it must be specified AFTER the operating system relocatables during generation. The capability level assigned to commands depends on their position within the table relative to table pointers located at the front of the command table. Each command is defined by a two-word entry. To change the capability level of a command, relocate the two-word entry to the appropriate table section for the desired capability level. Do not modify the two-word entry.

Then reassemble the modified capability table and relocate it after the file manager modules (i.e. %BMPG1,...) during generation. (You can ignore GEN05 and GEN08 errors here).

NOTE: Hewlett-Packard does not support modified command capability tables.

Calling Sequences to D.RTR — APPENDIX L

D.RTR is the central manager of the RTE file management system. It owns the directory and performs all writes on it.

Programs wishing to access the directory must schedule this program with wait. Default access rights are:

System Manager: Read/write access to any mounted cartridge

Non-Session User: Read/write access to system and non-session discs

Session User: Read/write access to all cartridges in SESSION cartridge list; read-only access to cartridges on LU 2 and LU 3.

Requests are communicated to D.RTR via parameters passed in the schedule call, (optionally) through string passage, and (optionally) on system tracks.

The open request is first separated from all others by testing the sign bit of the first (schedule call) parameter: sign bit set indicates open call; sign bit must be clear for all other requests. If the request is NOT open, a request code is passed in the low-order six bits of the second parameter

Session monitor override bits are passed in bits 15, 14, and 13 of P2:

bit: 15 set to allow access to all discs in system cartridge list

14 set to allow read/write access to LU 2 and LU 3 (normally read-only access)

13 restrict access to system discs.

NON-OPEN

The non-open requests use the schedule call parameters in one of the following two formats:

P1: ID segment address of caller
P2: Override bits + EVEN function code
P3-P4: Directory address:

```
     +------+------------+-----------+
     |sector|            |lu # of file|
P3   |offset|  sector #  |  directory |
     +------+------------+-----------+
      15  13 12         6 5          0

      +-------------------------------+
      |                               |
P4    |            track  #           |
      +-------------------------------+
       15                             0
```

P5, string, and tracks are function-dependent.

Alternate form:

P1: ID segment address of caller
P2: Override bits + ODD function code
P3: Cartridge selection:

 0 = Search/use any cartridge
>0 = Cartridge reference number
<0 = Negative LU number

P5, string, and tracks are function-dependent.

OPEN

P1. 1,ID (bit 15 set)

P2. override bits

P3. -LU, +CRN, 0

P4. security code

STRING: 1. E,NAME(1,2)
2. S,NAME(3,4)
3. NAME(5,6)

E (bit 15) = 1 if exclusive open
S (bit 15) = 1 if scratch file purge

CLOSE

```
P1.  ID

P2.  0

P3.  -\
       > Directory address
P4.  -/

STRING:  1. -\  Doubleword truncate option:
         2. -/
                 0 = no truncation
                >0 = truncate extents
                <0 = negative number of sectors
                     to truncate
```

CREAT

```
P1.  ID

P2.  1 + override bits

P3.  -LU, +CRN, 0

STRING:  1.  NAME(1,2)
         2.  NAME(3,4)
         3.  NAME(5,6)
         4.  Type
         5.
         6\  Doubleword file size:
         7/
             >0 = positive number of sectors
             <0 = negative number of 128-block chunks

             double word -1 = allocate rest of disc
                              (<=32767 chunks)

             single word -1 = allocate rest of disc
                              (<=16383 blocks)

          8.  Record length
         9.  Security code
```

Calling Sequences to D.RTR

CHANGE NAME

```
P1.  ID

 .  2
P3.  -\
       > Directory address
P4.  -/

STRING:  1.  -\  6-character
         2.   ->   new
         3.  -/    name
```

SET, CLEAR LOCK

```
P1.  ID

P2.  3 for set, 5 for clear, + override bits

P3.  -LU, +CRN       0 not legal
```

FAULT-TOLERANT BACKUP OPEN

```
P1.  ID

P2.  4

P3.  -\
       > Directory address
P4.  -/

P5.  contents of open flag for primary program
     (bits 0-14 only, bit 15 = zero)
```

EXTENSION OPEN

```
P1.  ID

P2.  6 for read, 8 for write

P3.  -\
       > Directory address of main file
P4.  -/

P5.  Extent number
```

GENERATE, PACK, UPDATE

```
P1.  ID

P2.  7 + override bits

P3.  -LU, +CRN      0 not legal

P4.  S, #sectors/track
     S (bit 15) = 1 if cartridge list update

STRING:  1.  -\  Data track
         2.  -/    address:


              +----------------+----------------+
              |                |      lu #      |
     WORD 1   +----------------+----------------+
              15              8 7               0

              +---------------------------------+
              |             track  #            |
     WORD 2   +---------------------------------+
              15                                0
```

PACK

```
P1.  ID

P2.  9 + override bits

P3.  -LU, +CRN      0 not legal

P4.  Relative directory sector

STRING:  -\    128-word
          -\   directory
          -/   sector to
        -/    be written
```

REMOVE CARTRIDGE

P1. ID

P2. 11 + override bits

P3. -LU 0 not legal

P4. SCB address, if different from caller

Return parameter R3 will be zero if the cartridge is no longer mounted by any computer (i.e. pool entry may be set non-busy). If any other computer still has the the cartridge mounted, R3 will be non-zero.

MOUNT CARTRIDGE

P1. ID

P2. 13 + override bits

P3. -LU 0 not legal

P4. SCB Address if mounting to session other than the one you are operating under

P5. ID to which disc is to be mounted.
bit 15 = 1 initialize directory

STRING: First nine words of cartridge specification entry. String is passed only if disc file directory to be initialized. If no string is passed, adding CL entry to the directory is all that is done.

ALTER CL ENTRY

P1. ID

P2. 15 + override bits

P3. -LU, +CRN   0 not legal

```
STRING:  -\    4 word
          -\   cartridge
          -/   directory
         -/    entry
```

RECOVER RESOURCES FROM FAILED COMPUTER

P1. ID segment address

P2. 17 + override bits
Resource recovery attempted only on cartridges to which the caller has write access

P3. -LU, +CRN, or 0

P4. failed computer number

STRING: List of LU, last track pairs for every disc LU in system (including dismounted discs)

REPORT AND/OR CHANGE GENERAL-PURPOSE FLAGS

P1. ID segment address

P2. 19 + override bits
(must have write access to cartridge
to change flag)

P3. -LU, +CRN (0 not allowed)

P4. 0 = report flag only
1 = set flag
2 = clear flag

P5. Flag number

Flags 0-15 = unconditionally cleared when
unmounted cartridge is first mounted.

Flags 16-31 = preserved through all mount/dismount
sequences.

For any P4 code, return parameter R3 will always contain
the previous state of the selected flag (this allows the
flags to be used as semaphores):

0 = flag clearal
1 = flag set

RETURN PARAMETERS

R1. Error code OR 0

```
R2.  -\
       > Directory address
R3.  -/
```

R4. Starting track # of file
LU # if type = 0

R5. #sectors/track (bits 8-15) Starting sector (bits 0-7)

Data Control Block, File Directory Formats — APPENDIX M

DATA CONTROL BLOCK FORMAT

```
    5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
   +------+---------------+----------+
   |sector|  sector # of  |dir OR disc|\
 0 |offset|file directory | file LU # | \  File Dir
   |------+---------------+----------|  /  Address
 1 |     track # of file directory   |/
   |---------------------------------|
 2 | file type - may be overridden at |
   |       open, unless type 0        |
   |---------------------------------|
   | track address of file (type >0)  |
   |              OR                  |
 3 |      LU # of file (type 0)       |
   |---------------------------------|
   | sector address of file (type >0) |
   |              OR                  |
 4 |    end-of-file code (type 0)     |
   |---------------------------------|
   |file size: -chunks,+sec. (type >0)|
   |              OR                  |
 5 |      spacing code (type 0)       |
   |---------------------------------|
   |     record length (type >0)      |
   |              OR                  |
 6 |     read/write code (type 0)     |
   |-+-----------------+-+-+-+-+-+-+-|
   |S|   # of blocks   |/|E|S|O|I|E|W|
 7 |C|  in DCB buffer  |/|X|Y|M|B|F|R|
   |-+-----------------+-+-+-+-+-+-+-|
 8 |  # sectors/track   (type >0)     |
   |---------------------------------|
 9 |      open/close Indicator        |
   |                                  |
```

Data Control Block, File Directory Formats

```
       |                                   |
       |-----------------------------------|
       |       track # of current          |\
    10 |    file position  (type >0)       | \
       |-----------------------------------|  \   Current
       |        sector # of current        |   \  position
    11 |     file position (type >0)       |   /  in
       |-----------------------------------|  /   file
       |  location of next word in file    | /
    12 |   (buffer pointer) (type >0)      |/
       |-----------------------------------|
    13 |  record number of current file    |
       |-----------------------------------|
    14 |  position (double word integer)   |
       |-----------------------------------|
    15 |          extent number            |
       |-----------------------------------|
    16 |         DCB buffer area           |
       +-----------------------------------+
```

WORD		CONTENT
0	File Directory Addesss	bit 6-12 = Physical sector # (block) of file directory bit 13-15 = Entry offset from beginning of block (origin 0)
4	End-of File Code, type 0 file:	011u = EOF on Magnetic Tape 101u = EOF on Paper Tape 111u = EOF on Line Printer
5	Spacing Code, type 0 file:	bit 15 = 1 - backspace legal bit 0 = 1 - forwardspace legal
6	Read/Write Code, type 0 file:	bit 15 = 1 - input legal bit 0 = 1 - output legal

7 Security Code Check/Open Mode/Buffer Size/In Buffer/To be Written/EOF Read Flag; all file types

(SC) Security Code Check:	bit 15 = 1 - codes agree = 0 - codes do not agree
DCB Buffer:	bits 14-7 = # blocks in DCB buffer
(SY) System Disc	bit 4 = 1 file on system disc = 0 not on system disc
(EX) Extendability	bit 5 = 1 file not extendable = 0 file is extendable
(OM) Open Mode:	bit 3 = 1 - update open = 0 - standard open
(IB) In Buffer Flag:	bit 2 = 1 - data in DCB buffer = 0 - data not in buffer
EOF Read Flag:	bit 1 = 1 - EOF read = 0 - EOF not read
(WR) To Be Written Flag:	bit 0 = 1 - data in DCB buffer to be written = 0 - data in DCB buffer not to be written

9 Open/Close Indicator: If open, contains ID segment location of program performing open. If closed, set to zero.

The file directory starts in sector 0 of the last track on all disc LUs. The first entry in each File Directory is the specification entry for the cartridge itself:

```
     15                       0
    +--+-----------------------+
    |  |                       |\    Bit 15 set to
  0 |  |                       | \   distinguish cart.
    |--+-----------------------|  \  specification entry
    |                          |   \ from file entry.
  1 |                          |   / 6-character
    |--------------------------|  /  cartridge label.
    |                          | /
  2 |                          |/
    |--------------------------|
    |                          |
  3 |  cartridge reference label |
    |--------------------------|
    |                          |
  4 | first available track for FMP |
    |----------+---------------|
    |          |               |
  5 | reserved |next available sector|
    |----------+---------------|
    |                          |
  6 |  number of sectors per track |
    |--------------------------|
    |    lowest directory track |
  7 |    (last file track + 1) |
    |--------------------------|
    |    # tracks in directory |
  8 |      (negative value)    |
    |--------------------------|
  9 |   next available FMP track |
    |--------------------------|
 10 |      first bad track     |
    |                          |
    |--------------------------|
    |            .             |
    |            .             |
    |            .             |
    |            .             |
    |--------------------------|
 15 |      sixth bad track     |
    |                          |
    +--------------------------+
```

The 16-word cartridge entry is followed by an entry for each file:

```
      15             8|7             0
     +---------------+---------------+
   0 |                               |
     |          6-character          |
   1 |           file name           |
     |                               |
   2 |                               |
     |-------------------------------|
   3 |   file type (1 thru 32767)    |
     |-------------------------------|
   4 |   starting track              |
     |---------------+---------------|
   5 |   extent #    | starting sector|
     |---------------+---------------|
   6 |   size in +sectors or -chunks |
     |-------------------------------|
   7 |   record length  (type 2 only) |
     |-------------------------------|
   8 |   security code               |
     |-------------------------------|
   9 |          Open Flags           |
     |                               |
  10 |   15    = 1 exclusive open    |
     |   14-12 = reserved            |
  11 |    11-8 = sequence counter    |
     |     7-0 = keyword offset of   |
  12 |           opening program     |
     |           ID segment          |
  13 |                               |
     |                               |
  14 |                               |
     |                               |
  15 |                               |
     |                               |
     +-------------------------------+

    Word 0 =  0 if last entry in directory
           = -1 if file is purged
```

Cartridge Directory and File Record Formats — APPENDIX N

The cartridge directory is two blocks long and is located in the system area on LU 2. The track and sector address are defined in entry points $CL1 and $CL2.

```
     5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
    +---------------+---------------+
    |               |               |
0   |     lock      |      LU       |   lock = 0 if not locked, else
    |---------------+---------------|   keyword table offset of ID
    |                               |   segment address of locking
1   |          last track           |   program.
    |-------------------------------|   Locked discs are available
    |                               |   only to the locker.
2   |   Cartridge Reference Label   |
    |-------+-----------------------|
    |       |                       |   ID identifies disc opener:
3   |       |          ID           |
    |-------+-----------------------|   ID = 0000 non-session
    |                               |   ID = 7777 system cart.
    |                               |   0<ID<7777 session monitor
    |  Up to 32 4-word entries in   |        group or private cart.
    |     first block of CL.        |
    |  Up to 31 4-word entries in   |
    |     second block.             |
    |                               |
    /                               /   NOTE:  Words 124, 125, 126
                                        127 are unique only in SECOND
    /                               /   block of CL.  FIRST block
    |                               |   will hold 32 entries in words
    |                               |   0 through 127.
    |-------------------------------|
252 |              0                |
    |-------------------------------|   Sum of contents of base page
    |                               |   words 1650 thru 1657, 1742
253 |  initialization code word     |   thru 1747, 1755 thru 1764.
    |-------------------------------|
    |                               |   Set when system cartridge
254 |     master security code      |   initialized.
    |-------------------------------|
255 |   reserved for future use     |
    +-------------------------------+
```

DISC FILE RECORD FORMATS

Fixed Length Formats   (Types 1 and 2)

```
+-------------------------------------------/  /---------------+---+
|          |            |            |  /  /   |             |   |
|  Block 1 |  Block 2   |  Block 3   | <  <    |   Block N   |EOF|
|          |            |            |  \  \   |             |   |
---------------------------------------------\  \-------------+---+
\         /                                      \           /  |
 \128 words/                                      \127 words/   |
                                                                |
                                                         1 word-+
```

Type 1 Record length = Block length = 128 words.

Type 2 Record length is user-defined; may cross block boundaries but not past EOF.

Variable Length Formats   (Types 3 and Above)

```
 / Record 1 \  /Record 2\  /3\  /Record 4\          /Record N\
/            \/          \/   \/          \        /          \
-----------------------------------------------/  /-----------------
| |          | | |         | | | | |          | |    /  /   | |          | |E|
|L|   Data   |L|L|  Data   |L|0|0|L|   Data   |L|   <  <    |L|   Data   |L|O|
| |          | | |         | | | | |          | |    \  \   | |          | |F|
-----------------------------------------------\  \-----------------
 ^          ^              \ /                                          ^
 |          |               |                                           |
 |          |               |---0-length             EOF = -1 in |
 |----------|--length            records             first length
               words         (sub-file mark)         word of next
                                                     record
```

Type 6 File Format. Files created by the SP command as memory image program files are always accessed as Type 1 files.

Word	Content	
0	-1	<- EOF unless forced to Type 1
1-5	Not used	
6	Priority	
7	Primary entry point	
8-11	Not used	
12-13	Original program name	
14	Program type	
15-16	Not used	
17-19	Time parameters	
20	Substatus 1: word 20 of ID segment	
21	Substatus 2: word 21 of ID segment	
22	Low main address	
23	High main address +1	
24	Low base page address	
25	High base page address +1	
26	Program track	
27	Swap track	
28	ID extension #/EMA size	
29	High address +1 of largest segment	
30	Not used	
31	Open flag word	
32	Not used	

```
         |                                     |
         +-------------------------------------+
      33 | MLS word 1                          |
         +-------------------------------------+
      34 | MLS word 2                          |
         +-------------------------------------+
      35 | Sector/LU of program                |
         +-------------------------------------+
      36 | Checksum of words 0-32              |
         +-------------------------------------+
      37 | Setup code word                     |
         +-------------------------------------+
      38 + I/O extension word 0                |
         +-------------------------------------+
      39 | I/O extension word 1                |
         +-------------------------------------+
      40 | I/O extension word 2                |
         +-------------------------------------+
      41 | I/O extension word 3                |
         +-------------------------------------+
      42 | I/O extension word 4                |
         +-------------------------------------+
   43-45 | Shared EMA name                     |
         +-------------------------------------+
      46 | Owner ID                            |
         +-------------------------------------+
      47 | Owner group ID                      |
         +-------------------------------------+
      48 | Capability level required           |
         +-------------------------------------+
  49-112 | Not used                            |
         +-------------------------------------+
 113-123 | Type 6 file created                 |
         +-------------------------------------+
 124-127 | Not used                            |
         +-------------------------------------+
```

Words 0-35 and 38-42 contain the program's ID segment information.

Word 37 represents the sum of the contents of words 1650 through 1657, words 1742 through 1747, and words 1755 through 1764 in base page.

If the sign bit is set in word 46, the program file is protected to this user ID. If the sign bit is set in word 47, the program file is protected to this group ID.

Word 48 represents the minimum capability level required to RU or RP this program.

The remainder of the file is an exact copy of program being saved.

A

E

F

G

R

S

U

V

READER COMMENT SHEET

RTE-6/VM
Technical Specifications

92084-90015 **April 1983**

Update No. ________
(If Applicable)

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

FROM:

Name ____

Company ____

Address ____

____

Phone No. ______________________________ **Ext.** __________

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 141 CUPERTINO, CA.

— POSTAGE WILL BE PAID BY —

Hewlett-Packard Company
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014
ATTN: Technical Publications

FOLD FOLD

Product Line Sales/Support Key

Key	Product Line
A	Analytical
CM	Components
C	Computer Systems Sales only
CH	Computer Systems Hardware Sales and Services
CS	Computer Systems Software Sales and Services
E	Electronic Instruments & Measurement Systems
M	Medical Products
MP	Medical Products Primary SRO
MS	Medical Products Secondary SRO
P	Personal Computation Products
*	Sales only for specific product line
**	Support only for specific product line

IMPORTANT: These symbols designate general product line capability. They do not insure sales or support availability for all products within a line, at all locations. Contact your local sales office for information regarding locations where HP support is available for specific products.

HP distributors are printed in italics.

ANGOLA
Telectra
Empresa Técnica de Equipamentos Eléctricos, S.A.R.L.
R. Barbosa Rodrigues, 41-I DT.
Caixa Postal 6487
LUANDA
Tel: 35515,35516
E,M,P

ARGENTINA
Hewlett-Packard Argentina S.A.
Avenida Santa Fe 2035
Martinez 1640 **BUENOS AIRES**
Tel: 798-5735, 792-1293
Telex: 17595 BIONAR
Cable: HEWPACKARG
A,E,CH,CS,P

Biotron S.A.C.I.M. e I.
Av Paseo Colon 221, Piso 9
*1399 **BUENOS AIRES,***
Tel: 30-4846, 30-1851
Telex: 17595 BIONAR
M

Fate S.A. I.C.I.Electronica
Venezuela 1326
*1095 **BUENOS AIRES***
Tel: 37-9020, 37-9026/9
Telex: 9234 FATEN AR
P

AUSTRALIA
Adelaide, South Australia Office
Hewlett-Packard Australia Ltd.
153 Greenhill Road
PARKSIDE, S.A. 5063
Tel: 272-5911
Telex: 82536
Cable: HEWPARD Adelaide
A*,CH,CM,,E,MS,P

Brisbane, Queensland Office
Hewlett-Packard Australia Ltd.
49 Park Road
MILTON, Queensland 4064
Tel: 229-1544
Telex: 42133
Cable: HEWPARD Brisbane
A,CH,CM,E,M,P
Effective November 1, 1982:
10 Payne Road
THE GAP, Queensland 4061
Tel: 30-4133
Telex: 42133

Canberra, Australia Capital Territory Office
Hewlett-Packard Australia Ltd.
121 Wollongong Street
FYSHWICK, A.C.T. 2609
Tel: 80 4244
Telex: 62650
Cable: HEWPARD Canberra
CH,CM,E,P

Melbourne, Victoria Office
Hewlett-Packard Australia Ltd.
31-41 Joseph Street
BLACKBURN, Victoria 3130
Tel: 877 7777
Telex: 31-024
Cable: HEWPARD Melbourne
A,CH,CM,CS,E,MS,P

Perth, Western Australia Office
Hewlett-Packard Australia Ltd.
261 Stirling Highway
CLAREMONT, W.A. 6010
Tel: 383-2188
Telex: 93859
Cable: HEWPARD Perth
A,CH,CM,,E,MS,P

Sydney, New South Wales Office
Hewlett-Packard Australia Ltd.
17-23 Talavera Road
P.O. Box 308
NORTH RYDE, N.S.W. 2113
Tel: 887-1611
Telex: 21561
Cable: HEWPARD Sydney
A,CH,CM,CS,E,MS,P

AUSTRIA
Hewlett-Packard Ges.m.b.h.
Grottenhofstrasse 94
Verkaufsburo Graz
A-8052 **GRAZ**
Tel: 291-5-66
Telex: 32375
CH,E*

Hewlett-Packard Ges.m.b.h.
Stanglhofweg 5
A-4020 **LINZ**
Tel: 0732 51585
CH

Hewlett-Packard Ges.m.b.h.
Lieblgasse 1
P.O. Box 72
A-1222 **VIENNA**
Tel: (0222) 23-65-11-0
Telex: 134425 HEPA A
A,CH,CM,CS,E,MS,P

BAHRAIN
Green Salon
P.O. Box 557
BAHRAIN
Tel: 255503-255950
Telex: 84419
P

Wael Pharmacy
P.O. Box 648
BAHRAIN
Tel: 256123
Telex: 8550 WAEL BN
M, E

BELGIUM
Hewlett-Packard Belgium S.A./N.V.
Blvd de la Woluwe, 100
Woluwedal
B-1200 **BRUSSELS**
Tel: (02) 762-32-00
Telex: 23-494 paloben bru
A,CH,CM,CS,E,MP,P

BRAZIL
Hewlett-Packard do Brasil I.e.C. Ltda.
Alameda Rio Negro, 750
Alphaville 06400 **BARUERI SP**
Tel: (11) 421-1311
Telex: 01 133872 HPBR-BR
Cable: HEWPACK Sao Paulo
A,CH,CM,CS,E,M,P

Hewlett-Packard do Brasil I.e.C. Ltda.
Avenida Epitacio Pessoa, 4664
22471 **RIO DE JANEIRO-RJ**
Tel: (21) 286-0237
Telex: 021-21905 HPBR-BR
Cable: HEWPACK Rio de Janeiro
A,CH,CM,E,MS,P*

CANADA
Alberta
Hewlett-Packard (Canada) Ltd.
210, 7220 Fisher Street S.E.
CALGARY, Alberta T2H 2H8
Tel: (403) 253-2713
A,CH,CM,E*,MS,P*

Hewlett-Packard (Canada) Ltd.
11620A-168th Street
EDMONTON, Alberta T5M 3T9
Tel: (403) 452-3670
A,CH,CM,CS,E,MS,P*

British Columbia
Hewlett-Packard (Canada) Ltd.
10691 Shellbridge Way
RICHMOND,
British Columbia V6X 2W7
Tel: (604) 270-2277
Telex: 610-922-5059
A,CH,CM,CS,E*,MS,P*

Manitoba
Hewlett-Packard (Canada) Ltd.
380-550 Century Street
WINNIPEG, Manitoba R3H 0Y1
Tel: (204) 786-6701
A,CH,CM,E,MS,P*

New Brunswick
Hewlett-Packard (Canada) Ltd.
37 Sheadiac Road
MONCTON, New Brunswick E2B 2VQ
Tel: (506) 855-2841
CH**

Nova Scotia
Hewlett-Packard (Canada) Ltd.
P.O. Box 931
900 Windmill Road
DARTMOUTH, Nova Scotia B2Y 3Z6
Tel: (902) 469-7820
CH,CM,CS,E*,MS,P*

Ontario
Hewlett-Packard (Canada) Ltd.
552 Newbold Street
LONDON, Ontario N6E 2S5
Tel: (519) 686-9181
A,CH,CM,E*,MS,P*

Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive
MISSISSAUGA, Ontario L4V 1M8
Tel: (416) 678-9430
A,CH,CM,CS,E,MP,P

Hewlett-Packard (Canada) Ltd.
2670 Queensview Dr.
OTTAWA, Ontario K2B 8K1
Tel: (613) 820-6483
A,CH,CM,CS,E*,MS,P*

Hewlett-Packard (Canada) Ltd.
220 Yorkland Blvd., Unit #11
WILLOWDALE, Ontario M2J 1R5
Tel: (416) 499-9333
CH

Quebec
Hewlett-Packard (Canada) Ltd.
17500 South Service Road
Trans-Canada Highway
KIRKLAND, Quebec H9J 2M5
Tel: (514) 697-4232
A,CH,CM,CS,E,MP,P*

Hewlett-Packard (Canada) Ltd.
Les Galeries du Vallon
2323 Du Versont Nord
STE. FOY, Quebec G1N 4C2
Tel: (418) 687-4570
CH

CHILE
Jorge Calcagni y Cia. Ltda.
Arturo Burhle 065
Casilla 16475
***SANTIAGO** 9*
Tel: 222-0222
Telex: Public Booth 440001
A,CM,E,M

Olympia (Chile) Ltda.
Av. Rodrigo de Araya 1045
Casilla 256-V
***SANTIAGO** 21*
Tel: 2-25-50-44
Telex: 340-892 OLYMP CK
Cable: Olympiachile Santiagochile
CH,CS,P

CHINA, People's Republic of
China Hewlett-Packard Rep. Office
P.O. Box 418
1A Lane 2, Luchang St.
Beiwei Rd., Xuanwu District
BEIJING
Tel: 33-1947, 33-7426
Telex: 22601 CTSHP CN
Cable: 1920
A,CH,CM,CS,E,P

COLOMBIA
Instrumentación
H. A. Langebaek & Kier S.A.
Carrera 7 No. 48-75
Apartado Aereo 6287
***BOGOTA** 1, D.E.*
Tel: 287-8877
Telex: 44400 INST CO
Cable: AARIS Bogota
A,CM,E,M,PS,P

COSTA RICA
Cientifica Costarricense S.A.
Avenida 2, Calle 5
San Pedro de Montes de Oca
Apartado 10159
SAN JOSÉ
Tel: 24-38-20, 24-08-19
Telex: 2367 GALGUR CR
CM,E,MS,P

CYPRUS
Telerexa Ltd.
P.O. Box 4809
14C Stassinos Avenue
NICOSIA
Tel: 62698
Telex: 2894 LEVIDO CY
E,M,P

DENMARK
Hewlett-Packard A/S
Datavej 52
DK-3460 **Birkerod**
Tel: (02) 81-66-40
Telex: 37409 hpas dk
A,CH,CM,CS,E,MS,P

Hewlett-Packard A/S
Navervej 1
DK-8600 **SILKEBORG**
Tel: (06) 82-71-66
Telex: 37409 hpas dk
CH,E

ECUADOR
CYEDE Cia. Ltda.
Avenida Eloy Alfaro 1749
Casilla 6423 CCI
QUITO
Tel: 450-975, 243-052
Telex: 2548 CYEDE ED
A,CM,E,P

Hospitalar S.A.
Robles 625
Casilla 3590
QUITO
Tel: 545-250, 545-122
Telex: 2485 HOSPTL ED
Cable: HOSPITALAR-Quito
M

EGYPT
International Engineering Associates
24 Hussein Hegazi Street
Kasr-el-Aini
CAIRO
Tel: 23829, 21641
Telex: IEA UN 93830
CH,CS,E,M

Informatic For Systems
22 Talaat Harb Street
CAIRO
Tel: 759006
Telex: 93938 FRANK UN
CH,CS,P

Egyptian International Office for Foreign Trade
P.O.Box 2558
CAIRO
Tel: 650021
Telex: 93337 EGPOR
P

EL SALVADOR
IPESA de El Salvador S.A.
29 Avenida Norte 1216
SAN SALVADOR
Tel: 26-6858, 26-6868
Telex: Public Booth 20107
A,CH,CM,CS,E,P

FINLAND
Hewlett-Packard Oy
Revontulentie 7
SF-02100 **ESPOO** 10
Tel: (90) 455-0211
Telex: 121563 hewpa sf
A,CH,CM,CS,E,MS,P

Hewlett-Packard Oy
Aatoksenkatv 10-C

SALES & SUPPORT OFFICES
Arranged Alphabetically by Country

SF-40720-72 **JYVASKYLA**
Tel: (941) 216318
CH

Hewlett-Packard Oy
Kainvuntie 1-C
SF-90140-14 **OULU**
Tel: (981) 338785
CH

FRANCE

Hewlett-Packard France
Z.I. Mercure B
Rue Berthelot
F-13763 Les Milles Cedex
AIX-EN-PROVENCE
Tel: (42) 59-41-02
Telex: 410770F
A,CH,E,MS,P*

Hewlett-Packard France
Boite Postale No. 503
F-25026 **BESANCON**
28 Rue de la Republique
F-25000 **BESANCON**
Tel: (81) 83-16-22
CH,M

Hewlett-Packard France
Bureau de Vente de Lyon
Chemin des Mouilles
Boite Postale 162
F-69130 **ECULLY** Cédex
Tel: (7) 833-81-25
Telex: 310617F
A,CH,CS,E,MP

Hewlett-Packard France
Immeuble France Evry
Tour Lorraine
Boulevard de France
F-91035 **EVRY** Cédex
Tel: (6) 077-96-60
Telex: 692315F
E

Hewlett-Packard France
5th Avenue Raymond Chanas
F-38320 **EYBENS**
Tel: (76) 25-81-41
Telex: 980124 HP GRENOB EYBE
CH

Hewlett-Packard France
Centre d'Affaire Paris-Nord
Bâtiment Ampère 5 étage
Rue de la Commune de Paris
Boite Postale 300
F-93153 **LE BLANC MESNIL**
Tel: (01) 865-44-52
Telex: 211032F
CH,CS,E,MS

Hewlett-Packard France
Parc d'Activites Cadera
Quartier Jean Mermoz
Avenue du President JF Kennedy
F-33700 **MERIGNAC**
Tel: (56) 34-00-84
Telex: 550105F
CH,E,MS

Hewlett-Packard France
32 Rue Lothaire
F-57000 **METZ**
Tel: (8) 765-53-50
CH

Hewlett-Packard France
Immueble Les 3 B
Nouveau Chemin de la Garde
Z.A.C. de Bois Briand
F-44085 **NANTES** Cedex
Tel: (40) 50-32-22
CH**

Hewlett-Packard France
Zone Industrielle de Courtaboeuf
Avenue des Tropiques
F-91947 Les Ulis Cédex **ORSAY**
Tel: (6) 907-78-25
Telex: 600048F
A,CH,CM,CS,E,MP,P

Hewlett-Packard France
Paris Porte-Maillot
15, Avenue De L'Amiral Bruix
F-75782 **PARIS** 16
Tel: (1) 502-12-20
Telex: 613663F
CH,MS,P

Hewlett-Packard France
2 Allee de la Bourgonette
F-35100 **RENNES**
Tel: (99) 51-42-44
Telex: 740912F
CH,CM,E,MS,P*

Hewlett-Packard France
98 Avenue de Bretagne
F-76100 **ROUEN**
Tel: (35) 63-57-66 CH**,CS

Hewlett-Packard France
4 Rue Thomas Mann
Boite Postale 56
F-67200 **STRASBOURG**
Tel: (88) 28-56-46
Telex: 890141F
CH,E,MS,P*

Hewlett-Packard France
Pericentre de la Cépière
F-31081 **TOULOUSE** Cedex
Tel: (61) 40-11-12
Telex: 531639F
A,CH,CS,E,P*

Hewlett-Packard France
Immeuble Péricentre
F-59658 **VILLENEUVE D'ASCQ** Cedex
Tel: (20) 91-41-25
Telex: 160124F
CH,E,MS,P*

GERMAN FEDERAL REPUBLIC

Hewlett-Packard GmbH
Technisches Büro Berlin
Keithstrasse 2-4
D-1000 **BERLIN** 30
Tel: (030) 24-90-86
Telex: 018 3405 hpbln d
A,CH,E,M,P

Hewlett-Packard GmbH
Technisches Büro Böblingen
Herrenberger Strasse 110
D-7030 **BÖBLINGEN**
Tel: (07031) 667-1
Telex: bbn or
A,CH,CM,CS,E,MP,P

Hewlett-Packard GmbH
Technisches Büro Dusseldorf
Emanuel-Leutze-Strasse 1
D-4000 **DUSSELDORF**
Tel: (0211) 5971-1
Telex: 085/86 533 hpdd d
A,CH,CS,E,MS,P

Hewlett-Packard GmbH
Vertriebszentrale Frankfurt
Berner Strasse 117
Postfach 560 140
D-6000 **FRANKFURT** 56
Tel: (0611) 50-04-1
Telex: 04 13249 hpffm d
A,CH,CM,CS,E,MP,P

Hewlett-Packard GmbH
Technisches Büro Hamburg
Kapstadtring 5
D-2000 **HAMBURG** 60
Tel: (040) 63804-1
Telex: 021 63 032 hphh d
A,CH,CS,E,MS,P

Hewlett-Packard GmbH
Technisches Büro Hannover
Am Grossmarkt 6
D-3000 **HANNOVER** 91
Tel: (0511) 46-60-01
Telex: 092 3259
A,CH,CM,E,MS,P

Hewlett-Packard GmbH
Technisches Büro Mannheim
Rosslauer Weg 2-4
D-6800 **MANNHEIM**
Tel: (0621) 70050
Telex: 0462105
A,C,E

Hewlett-Packard GmbH
Technisches Büro Neu Ulm
Messerschmittstrasse 7
D-7910 **NEU ULM**
Tel: 0731-70241
Telex: 0712816 HP ULM-D
A,C,E*

Hewlett-Packard GmbH
Technisches Büro Nürnberg
Neumeyerstrasse 90
D-8500 **NÜRNBERG**
Tel: (0911) 52 20 83-87
Telex: 0623 860
CH,CM,E,MS,P

Hewlett-Packard GmbH
Technisches Büro München
Eschenstrasse 5
D-8028 **TAUFKIRCHEN**
Tel: (089) 6117-1
Telex: 0524985
A,CH,CM,E,MS,P

GREAT BRITAIN

Hewlett-Packard Ltd.
Trafalgar House
Navigation Road
ALTRINCHAM
Chesire WA14 1NU
Tel: (061) 928-6422
Telex: 668068
A,CH,CS,E,M

Hewlett-Packard Ltd.
Oakfield House, Oakfield Grove
Clifton
BRISTOL BS8 2BN, Avon
Tel: (027) 38606
Telex: 444302
CH,M,P

Hewlett-Packard Ltd.
(Pinewood)
Nine Mile Ride
EASTHAMPSTEAD
Wokingham
Berkshire, 3RG11 3LL
Tel: 3446 3100
Telex: 84-88-05
CH,CS,E

Hewlett-Packard Ltd.
Fourier House
257-263 High Street
LONDON COLNEY
Herts., AL2 1HA, St. Albans
Tel: (0727) 24400
Telex: 1-8952716
CH,CS,E

Hewlett-Packard Ltd
Tradax House, St. Mary's Walk
MAIDENHEAD
Berkshire, SL6 1ST
Tel: (0628) 39151
CH,CS,E,P

Hewlett-Packard Ltd.
Quadrangle
106-118 Station Road
REDHILL, Surrey
Tel: (0737) 68655
Telex: 947234 CH,CS,E

Hewlett-Packard Ltd.
Avon House
435 Stratford Road
SHIRLEY, Solihull
West Midlands B90 4BL
Tel: (021) 745 8800
Telex: 339105
CH

Hewlett-Packard Ltd.
West End House 41
High Street, West End
SOUTHAMPTON
Hampshire S03 3DQ
Tel: (703) 886767
Telex: 477138
CH

Hewlett-Packard Ltd.
King Street Lane
WINNERSH, Wokingham
Berkshire RG11 5AR
Tel: (0734) 784774
Telex: 847178
A,CH,E,M

GREECE

Kostas Karaynnis S.A.
8 Omirou Street
***ATHENS** 133*
Tel: 32 30 303, 32 37 371
Telex: 215962 RKAR GR
A,CH,CM,CS,E,M,P

PLAISIO S.A.
G. Gerardos
24 Stournara Street
ATHENS
Tel: 36-11-160
Telex: 221871
P

GUATEMALA

IPESA
Avenida Reforma 3-48, Zona 9
GUATEMALA CITY
Tel: 316627, 314786
Telex: 4192 TELTRO GU
A,CH,CM,CS,E,M,P

HONG KONG

Hewlett-Packard Hong Kong, Ltd.
G.P.O. Box 795
5th Floor, Sun Hung Kai Centre
30 Harbour Road
HONG KONG
Tel: 5-8323211
Telex: 66678 HEWPA HX
Cable: HEWPACK HONG KONG
E,CH,CS,P

CET Ltd.
1402 Tung Way Mansion
199-203 Hennessy Rd.
*Wanchia, **HONG KONG***
Tel: 5-729376
Telex: 85148 CET HX
CM

Schmidt & Co. (Hong Kong) Ltd.
Wing On Centre, 28th Floor
Connaught Road, C.
HONG KONG
Tel: 5-455644
Telex: 74766 SCHMX HX
A,M

ICELAND

Elding Trading Company Inc.
Hafnarnvoli-Tryggvagotu
P.O. Box 895
IS-REYKJAVIK
Tel: 1-58-20, 1-63-03
M

INDIA

Blue Star Ltd.
Sabri Complex II Floor
24 Residency Rd.
***BANGALORE** 560 025*
Tel: 55660
Telex: 0845-430
Cable: BLUESTAR
A,CH,CM,CS,E

Blue Star Ltd.
Band Box House
Prabhadevi
***BOMBAY** 400 025*
Tel: 422-3101
Telex: 011-3751
Cable: BLUESTAR
A,M

Blue Star Ltd.
Sahas
414/2 Vir Savarkar Marg
Prabhadevi
***BOMBAY** 400 025*
Tel: 422-6155
Telex: 011-4093
Cable: FROSTBLUE
A,CH,CM,CS,E,M

Blue Star Ltd.
Kalyan, 19 Vishwas Colony
*Alkapuri, **BORODA**, 390 005*
Tel: 65235
Cable: BLUE STAR
A

Blue Star Ltd.
7 Hare Street
***CALCUTTA** 700 001*
Tel: 12-01-31
Telex: 021-7655
Cable: BLUESTAR
A,M

Blue Star Ltd.
133 Kodambakkam High Road
***MADRAS** 600 034*
Tel: 82057
Telex: 041-379
Cable: BLUESTAR
A,M

Blue Star Ltd.
Bhandari House, 7th/8th Floors
91 Nehru Place
***NEW DELHI** 110 024*
Tel: 682547
Telex: 031-2463
Cable: BLUESTAR
A,CH,CM,CS,E,M

Blue Star Ltd.
15/16:C Wellesley Rd.
***PUNE** 411 011*
Tel: 22775
Cable: BLUE STAR
A

Blue Star Ltd.
2-2-47/1108 Bolarum Rd.
***SECUNDERABAD** 500 003*
Tel: 72057
Telex: 0155-459
Cable: BLUEFROST
A,E

Blue Star Ltd.
T.C. 7/603 Poornima
Maruthankuzhi
***TRIVANDRUM** 695 013*
Tel: 65799
Telex: 0884-259
Cable: BLUESTAR
E

INDONESIA

BERCA Indonesia P.T.
P.O.Box 496/JKT.
Jl. Abdul Muis 62
JAKARTA
Tel: 373009
Telex: 46748 BERSAL IA
Cable: BERSAL JAKARTA
P

BERCA Indonesia P.T.
Wisma Antara Bldg., 17th floor
JAKARTA
A,CS,E,M

BERCA Indonesia P.T.
P.O. Box 174/SBY.
Jl. Kutei No. 11
SURABAYA
Tel: 68172
Telex: 31146 BERSAL SB
Cable: BERSAL-SURABAYA
A,E,M,P*

IRAQ
Hewlett-Packard Trading S.A.
Service Operation
Al Mansoor City 9B/3/7
BAGHDAD
Tel: 551-49-73
Telex: 212-455 HEPAIRAQ IK
CH,CS

IRELAND
Hewlett-Packard Ireland Ltd.
82/83 Lower Leeson St.
DUBLIN 2
Tel: (1) 60 88 00
Telex: 30439
A,CH,CM,CS,E,M,P

Cardiac Services Ltd.
Kilmore Road
Artane
***DUBLIN** 5*
Tel: (01) 351820
Telex: 30439
M

ISRAEL
Eldan Electronic Instrument Ltd.
P.O. Box 1270
JERUSALEM 91000
16, Ohaliav St.
JERUSALEM 94467
Tel: 533 221, 553 242
Telex: 25231 AB/PAKRD IL
A

Electronics Engineering Division
Motorola Israel Ltd.
16 Kremenetski Street
P.O. Box 25016
TEL-AVIV 67899
Tel: 3-338973
Telex: 33569 Motil IL
Cable: BASTEL Tel-Aviv
CH,CM,CS,E,M,P

ITALY
Hewlett-Packard Italiana S.p.A.
Traversa 99C
Via Giulio Petroni, 19
I-70124 **BARI**
Tel: (080) 41-07-44
M

Hewlett-Packard Italiana S.p.A.
Via Martin Luther King, 38/111
I-40132 **BOLOGNA**
Tel: (051) 402394
Telex: 511630
CH,E,MS

Hewlett-Packard Italiana S.p.A.
Via Principe Nicola 43G/C
I-95126 **CATANIA**
Tel: (095) 37-10-87
Telex: 970291
C,P

Hewlett-Packard Italiana S.p.A.
Via G. Di Vittorio 9
I-20063 **CERNUSCO SUL NAVIGLIO**
Tel: (2) 903691
Telex: 334632
A,CH,CM,CS,E,MP,P

Hewlett-Packard Italiana S.p.A.
Via Nuova San Rocco a
Capodimonte, 62/A
I-80131 **NAPLES**
Tel: (081) 7413544
Telex: 710698
A,CH,E

Hewlett-Packard Italiana S.p.A.
Viale G. Modugno 33
I-16156 **GENOVA PEGLI**
Tel: (010) 68-37-07
Telex: 215238
E,C

Hewlett-Packard Italiana S.p.A.
Via Turazza 14
I-35100 **PADOVA**
Tel: (049) 664888
Telex: 430315
A,CH,E,MS

Hewlett-Packard Italiana S.p.A.
Viale C. Pavese 340
I-00144 **ROMA**
Tel: (06) 54831
Telex: 610514
A,CH,CM,CS,E,MS,P*

Hewlett-Packard Italiana S.p.A.
Corso Svizzera, 184
I-10149 **TORINO**
Tel: (011) 74 4044
Telex: 221079
CH,E

JAPAN
Yokogawa-Hewlett-Packard Ltd.
Inoue Building
1-21-8, Asahi-cho
ATSUGI, Kanagawa 243
Tel: (0462) 28-0451
CM,C*,E

Yokogawa-Hewlett-Packard Ltd.
Towa Building
2-2-3, Kaigandori, Chuo-ku
KOBE, 650, Hyogo
Tel: (078) 392-4791
C,E

Yokogawa-Hewlett-Packard Ltd.
Kumagaya Asahi Yasoji Bldg 4F
3-4 Chome Tsukuba
KUMAGAYA, Saitama 360
Tel: (0485) 24-6563
CH,CM,E

Yokogawa-Hewlett-Packard Ltd.
Asahi Shinbun Dai-ichi Seimei Bldg., 2F
4-7 Hanabata-cho
KUMAMOTO-SHI,860
Tel: (0963) 54-7311
CH,E

Yokogawa-Hewlett-Packard Ltd.
Shin Kyoto Center Bldg. 5F
614 Siokoji-cho
Nishiiruhigashi, Karasuma
Siokoji-dori, Shimogyo-ku
KYOTO 600
Tel: 075-343-0921
CH,E

Yokogawa-Hewlett-Packard Ltd.
Mito Mitsui Building
1-4-73, San-no-maru
MITO, Ibaragi 310
Tel: (0292) 25-7470
CH,CM,E

Yokogawa-Hewlett-Packard Ltd.
Sumitomo Seimei Nagoya Bldg.
2-14-19, Meieki-Minami,
Nakamura-ku
NAGOYA, 450 Aichi
Tel: (052) 571-5171
CH,CM,CS,E,MS

Yokogawa-Hewlett-Packard Ltd.
Chuo Bldg., 4th Floor
5-4-20 Nishinakajima,
Yodogawa-ku
OSAKA, 532
Tel: (06) 304-6021
Telex: YHPOSA 523-3624
A,CH,CM,CS,E,MP,P*

Yokogawa-Hewlett-Packard Ltd.
1-27-15, Yabe,
SAGAMIHARA Kanagawa, 229
Tel: 0427 59-1311

Yokogawa-Hewlett-Packard Ltd.
Shinjuku Dai-ichi Seimei 6F
2-7-1, Nishi Shinjuku
Shinjuku-ku, **TOKYO** 160
Tel: 03-348-4611-5
CH,E

Yokogawa-Hewlett-Packard Ltd.
3-29-21 Takaido-Higashi
Suginami-ku **TOKYO** 168
Tel: (03) 331-6111
Telex: 232-2024 YHPTOK
A,CH,CM,CS,E,MP,P*

Yokogawa-Hewlett-Packard Ltd.
Daiichi Asano Building 4F
5-2-8, Oodori,
UTSUNOMIYA, 320
Tochigi
Tel: (0286) 25-7155
CH, CS, E

Yokogawa-Hewlett-Packard Ltd.
Yasudaseimei Yokohama
Nishiguchi Bldg.
3-30-4 Tsuruya-cho
Kanagawa-ku
YOKOHAMA, Kanagawa, 221
Tel: (045) 312-1252
CH,CM,E

JORDAN
Mouasher Cousins Company
P.O. Box 1387
AMMAN
Tel: 24907, 39907
Telex: 21456 SABCO JO
CH,E,M,P

KENYA
ADCOM Ltd., Inc., Kenya
P.O.Box 30070
NAIROBI
Tel: 331955
Telex: 22639
E,M

KOREA
Samsung Electronics Computer Division
76-561 Yeoksam-Dong
Kangnam-Ku
C.P.O. Box 2775
SEOUL
Tel: 555-7555, 555-5447
Telex: K27364 SAMSAN
A,CH,CM,CS,E,M,P

KUWAIT
Al-Khaldiya Trading & Contracting
P.O. Box 830 Safat
KUWAIT
Tel: 42-4910, 41-1726
Telex: 22481 Areeg kt
CH,E,M

Photo & Cine Equipment
P.O. Box 270 Safat
KUWAIT
Tel: 42-2846, 42-3801
Telex: 22247 Matin-KT
P

LEBANON
G.M. Dolmadjian
Achrafieh
P.O. Box 165.167
BEIRUT
Tel: 290293
*MP***

LUXEMBOURG
Hewlett-Packard Belgium S.A./N.V.
Blvd de la Woluwe, 100
Woluwedal
B-1200 **BRUSSELS**
Tel: (02) 762-32-00
Telex: 23-494 paloben bru
A,CH,CM,CS,E,MP,P

MALAYSIA
Hewlett-Packard Sales (Malaysia)
Sdn. Bhd.
1st Floor, Bangunan British
American
Jalan Semantan, Damansara Heights
KUALA LUMPUR 23-03
Tel: 943022
Telex: MA31011
A,CH,E,M,P*

Protel Engineering
Lot 319, Satok Road
P.O.Box 1917
*Kuching, **SARAWAK***
Tel: 53544
Telex: MA 70904 PROMAL
Cable: PROTELENG
A,E,M

MALTA
Philip Toledo Ltd.
Notabile Rd.
MRIEHEL
Tel: 447 47, 455 66
Telex: 649 Media MW
P

MEXICO
Hewlett-Packard Mexicana, S.A. de C.V.
Av. Periferico Sur No. 6501
Tepepan, Xochimilco
MEXICO D.F. 16020
Tel: 676-4600
Telex: 17-74-507 HEWPACK MEX
A,CH,CS,E,MS,P

Effective November 1, 1982:
Hewlett-Packard Mexicana, S.A. de C.V.
Ejercito Nacional #570
Colonia Granada
11560 **MEXICO**, D.F.
CH**

Hewlett-Packard Mexicana, S.A. de C.V.
Rio Volga 600
Pte. Colonia del Valle
MONTERREY, N.L.
Tel: 78-42-93, 78-42-40, 78-42-41
Telex: 038-2410 HPMTY ME
CH
Effective Nov. 1, 1982
Ave. Colonia del Valle #409
Col. del Valle
Municinio de garza garcia
MONTERREY, N.V.

ECISA
Taihe 229, Piso 10
*Polanco **MEXICO D.F.** 11570*
Tel: 250-5391
Telex: 17-72755 ECE ME
M

MOROCCO
Dolbeau
81 rue Karatchi
CASABLANCA
Tel: 3041-82, 3068-38
Telex: 23051, 22822
E

Gerep
2 rue d'Agadir
Boite Postale 156
CASABLANCA
Tel: 272093, 272095
Telex: 23 739
P

NETHERLANDS
Hewlett-Packard Nederland B.V.
Van Heuven Goedhartlaan 121
NL 1181KK **AMSTELVEEN**
P.O. Box 667
NL1180 AR **AMSTELVEEN**
Tel: (20) 47-20-21
Telex: 13 216
A,CH,CM,CS,E,MP,P

Hewlett-Packard Nederland B.V.
Bongerd 2
NL 2906VK **CAPPELLE**, A/D Ijessel
P.O. Box 41
NL2900 AA **CAPELLE**, Ijssel
Tel: (10) 51-64-44
Telex: 21261 HEPAC NL
A,CH,CS

NEW ZEALAND
Hewlett-Packard (N.Z.) Ltd.
169 Manukau Road
P.O. Box 26-189
Epsom, **AUCKLAND**
Tel: 687-159
Cable: HEWPACK Auckland
CH,CM,E,P*

Hewlett-Packard (N.Z.) Ltd.
4-12 Cruickshank Street
Kilbirnie, **WELLINGTON 3**
P.O. Box 9443
Courtenay Place, **WELLINGTON 3**
Tel: 877-199
Cable: HEWPACK Wellington
CH,CM,E,P

Northrop Instruments & Systems Ltd.
369 Khyber Pass Road
P.O. Box 8602
AUCKLAND
Tel: 794-091
Telex: 60605
A,M

Northrop Instruments & Systems Ltd.
110 Mandeville St.
P.O. Box 8388
CHRISTCHURCH
Tel: 486-928
Telex: 4203
A,M

Northrop Instruments & Systems Ltd.
Sturdee House
85-87 Ghuznee Street
P.O. Box 2406
WELLINGTON
Tel: 850-091
Telex: NZ 3380
A,M

NORTHERN IRELAND
Cardiac Services Company
95A Finaghy Road South
***BELFAST** BT 10 OBY*
Tel: (0232) 625-566
Telex: 747626
M

NORWAY
Hewlett-Packard Norge A/S
Folke Bernadottes vei 50
P.O. Box 3558
N-5033 **FYLLINGSDALEN** (Bergen)
Tel: (05) 16-55-40
Telex: 16621 hpnas n
CH,CS,E,MS

Hewlett-Packard Norge A/S
Österndalen 18
P.O. Box 34
N-1345 **ÖSTERºAS**
Tel: (02) 17-11-80
Telex: 16621 hpnas n
A,CH,CM,CS,E,M,P

OMAN
Khimjil Ramdas
P.O. Box 19
MUSCAT
Tel: 722225, 745601
Telex: 3289 BROKER MB MUSCAT
P

SALES & SUPPORT OFFICES
Arranged Alphabetically by Country

Suhail & Saud Bahwan
P.O. Box 169
MUSCAT
Tel: 734 201-3
Telex: 3274 BAHWAN MB

PAKISTAN
Mushko & Company Ltd.
1-B, Street 43
Sector F-8/1
ISLAMABAD
Tel: 26875
Cable: FEMUS Rawalpindi
A,E,M

Mushko & Company Ltd.
Oosman Chambers
Abdullah Haroon Road
***KARACHI** 0302*
Tel: 511027, 512927
Telex: 2894 MUSKO PK
Cable: COOPERATOR Karachi
*A,E,M,P**

PANAMA
Electrónico Balboa, S.A.
Calle Samuel Lewis, Ed. Alfa
Apartado 4929
PANAMA 5
Tel: 64-2700
Telex: 3483 ELECTRON PG
A,CM,E,M,P

Foto Internacional, S.A.
Colon Free Zone
Apartado 2068
COLON 3
Tel: 45-2333
Telex: 8626 IMPORT PG
P

PERU
Cía Electro Médica S.A.
Los Flamencos 145, San Isidro
Casilla 1030
LIMA 1
Tel: 41-4325, 41-3703
Telex: Pub. Booth 25306
A,CM,E,M,P

PHILIPPINES
The Online Advanced Systems Corporation
Rico House, Amorsolo Cor. Herrera Street
Legaspi Village, Makati
P.O. Box 1510
*Metro **MANILA***
Tel: 85-35-81, 85-34-91, 85-32-21
Telex: 3274 ONLINE
A,CH,CS,E,M

Electronic Specialists and Proponents Inc.
690-B Epifanio de los Santos Avenue
*Cubao, **QUEZON CITY***
P.O. Box 2649 Manila
Tel: 98-96-81, 98-96-82, 98-96-83
Telex: 40018, 42000 ITT GLOBE MACKAY BOOTH
P

PORTUGAL
Mundinter
Intercambio Mundial de Comércio S.a.r.l
P.O. Box 2761
Av. Antonio Augusto de Aguiar 138
P-LISBON
Tel: (19) 53-21-31, 53-21-37
Telex: 16691 munter p
M

Soquimica
Av. da Liberdade, 220-2
*1298 **LISBON** Codex*
Tel: 56 21 81/2/3
Telex: 13316 SABASA P

Telectra-Empresa Técnica de Equipmentos Eléctricos S.a.r.l.
Rua Rodrigo da Fonseca 103
P.O. Box 2531
***P-LISBON** 1*
Tel: (19) 68-60-72
Telex: 12598
CH,CS,E,P

PUERTO RICO
Hewlett-Packard Puerto Rico
P.O. Box 4407
CAROLINA, Puerto Rico 00628
Calle 272 Edificio 203
Urb. Country Club
RIO PIEDRAS, Puerto Rico 00924
Tel: (809) 762-7255
A,CH,CS

QATAR
Nasser Trading & Contracting
P.O. Box 1563
DOHA
Tel: 22170, 23539
Telex: 4439 NASSER DH
M

Computearbia
P.O. Box 2750
DOHA
Tel: 883555
Telex: 4806 CHPARB
P

Eastern Technical Services
P.O. Box 4747
DOHA
Tel: 329 993
Telex: 4156 EASTEC DH

SAUDI ARABIA
Modern Electronic Establishment
Hewlett-Packard Division
P.O. Box 281
Thuobah
AL-KHOBAR
Tel: 864-46 78
Telex: 671 106 HPMEEK SJ
Cable: ELECTA AL-KHOBAR
CH,CS,E,M,P

Modern Electronic Establishment
Hewlett-Packard Division
P.O. Box 1228
Redec Plaza, 6th Floor
JEDDAH
Tel: 644 38 48
Telex: 402712 FARNAS SJ
Cable: ELECTA JEDDAH
CH,CS,E,M,P

Modern Electronic Establishment
Hewlett Packard Division
P.O. Box 2728
RIYADH
Tel: 491-97 15, 491-63 87
Telex: 202049 MEERYD SJ
CH,CS,E,M,P

SCOTLAND
Hewlett-Packard Ltd.
Royal Bank Buildings
Swan Street
BRECHIN, Angus, Scotland
Tel: (03562) 3101-2
CH

Hewlett-Packard Ltd.
SOUTH QUEENSFERRY
West Lothian, EH30 9GT
GB-Scotland
Tel: (031) 3311188
Telex: 72682
A,CH,CM,CS,E,M

SINGAPORE
Hewlett-Packard Singapore (Pty.) Ltd.
P.O. Box 58 Alexandra Post Office
SINGAPORE, 9115
6th Floor, Inchcape House
450-452 Alexandra Road
SINGAPORE 0511
Tel: 631788
Telex: HPSGSO RS 34209
Cable: HEWPACK, Singapore
A,CH,CS,E,MS,P

Dynamar International Ltd.
Unit 05-11 Block 6
Kolam Ayer Industrial Estate
***SINGAPORE** 1334*
Tel: 747-6188
Telex: RS 26283
CM

SOUTH AFRICA
Hewlett-Packard So Africa (Pty.) Ltd.
P.O. Box 120
Howard Place
Pine Park Center, Forest Drive, Pinelands
CAPE PROVINCE 7405
Tel: 53-7954
Telex: 57-20006
A,CH,CM,E,MS,P

Hewlett-Packard So Africa (Pty.) Ltd.
P.O. Box 37099
92 Overport Drive
DURBAN 4067
Tel: 28-4178, 28-4179, 28-4110
Telex: 6-22954
CH,CM

Hewlett-Packard So Africa (Pty.) Ltd.
6 Linton Arcade
511 Cape Road
Linton Grange
PORT ELIZABETH 6001
Tel: 041-302148
CH

Hewlett-Packard So Africa (Pty.) Ltd.
P.O. Box 33345
Glenstantia 0010 **TRANSVAAL**
1st Floor East
Constantia Park Ridge Shopping Centre
Constantia Park
PRETORIA
Tel: 982043
Telex: 32163
CH,E

Hewlett-Packard So Africa (Pty.) Ltd.
Private Bag Wendywood
SANDTON 2144
Tel: 802-5111, 802-5125
Telex: 4-20877
Cable: HEWPACK Johannesburg
A,CH,CM,CS,E,MS,P

SPAIN
Hewlett-Packard Española S.A.
c/Entenza, 321
E-BARCELONA 29
Tel: (3) 322-24-51, 321-73-54
Telex: 52603 hpbee
A,CH,CS,E,MS,P

Hewlett-Packard Española S.A.
c/San Vicente S/N
Edificio Albia II,7 B
E-BILBAO 1
Tel: (4) 23-8306, (4) 23-8206
A,CH,E,MS

Hewlett-Packard Española S.A.
Calle Jerez 3
E-MADRID 16
Tel: (1) 458-2600
Telex: 23515 hpe
A,CM,E

Hewlett-Packard Española S.A.
c/o Costa Brava 13
Colonia Mirasierra
E-MADRID 34
Tel: (1) 734-8061, (1) 734-1162
CH,CS,M

Hewlett-Packard Española S.A.
Av Ramón y Cajal 1-9
Edificio Sevilla 1,
E-SEVILLA 5
Tel: 64-44-54, 64-44-58
Telex: 72933
A,CS,MS,P

Hewlett-Packard Española S.A.
C/Ramon Gordillo, 1 (Entlo.3)
E-VALENCIA 10
Tel: 361-1354, 361-1358
CH,P

SWEDEN
Hewlett-Packard Sverige AB
Sunnanvagen 14K
S-22226 **LUND**
Tel: (046) 13-69-79
Telex: (854) 17886 (via SPÅNGA office)
CH

Hewlett-Packard Sverige AB
Vastra Vintergatan 9
S-70344 **OREBRO**
Tel: (19) 10-48-80
Telex: (854) 17886 (via SPÅNGA office)
CH

Hewlett-Packard Sverige AB
Skalholtsgatan 9, Kista
Box 19
S-16393 **SPÅNGA**
Tel: (08) 750-2000
Telex: (854) 17886
A,CH,CM,CS,E,MS,P

Hewlett-Packard Sverige AB
Frötallisgatan 30
S-42132 **VÄSTRA-FRÖLUNDA**
Tel: (031) 49-09-50
Telex: (854) 17886 (via SPÅNGA office)
CH,E,P

SWITZERLAND
Hewlett-Packard (Schweiz) AG
Clarastrasse 12
CH-4058 **BASLE**
Tel: (61) 33-59-20
A

Hewlett-Packard (Schweiz) AG
Bahnhoheweg 44
CH-3018 **BERN**
Tel: (031) 56-24-22
CH

Hewlett-Packard (Schweiz) AG
47 Avenue Blanc
CH-1202 **GENEVA**
Tel: (022) 32-48-00
CH,CM,CS

Hewlett-Packard (Schweiz) AG
19 Chemin Château Bloc
CH-1219 **LE LIGNON**-Geneva
Tel: (022) 96-03-22
Telex: 27333 hpag ch
Cable: HEWPACKAG Geneva
A,E,MS,P

Hewlett-Packard (Schweiz) AG
Allmend 2
CH-8967 **WIDEN**
Tel: (57) 31 21 11
Telex: 53933 hpag ch
Cable: HPAG CH
A,CH,CM,CS,E,MS,P

SYRIA
General Electronic Inc.
Nuri Basha
P.O. Box 5781
DAMASCUS
Tel: 33-24-87
Telex: 11216 ITIKAL SY
Cable: ELECTROBOR DAMASCUS
E

Middle East Electronics
Place Azmé
Boite Postale 2308
DAMASCUS
Tel: 334592
Telex: 11304 SATACO SY
M,P

TAIWAN
Hewlett-Packard Far East Ltd.
Kaohsiung Office
2/F 68-2, Chung Cheng 3rd Road
KAOHSIUNG
Tel: 241-2318, 261-3253
CH,CS,E

Hewlett-Packard Far East Ltd.
Taiwan Branch
5th Floor
205 Tun Hwa North Road
TAIPEI
Tel:(02) 751-0404
Cable:HEWPACK Taipei
A,CH,CM,CS,E,M,P

Ing Lih Trading Co.
3rd Floor, 7 Jen-Ai Road, Sec. 2
TAIPEI 100
Tel: (02) 3948191
Cable: INGLIH TAIPEI
A

THAILAND
Unimesa
30 Patpong Ave., Suriwong
***BANGKOK** 5*
Tel: 234 091, 234 092
Telex: 84439 Simonco TH
Cable: UNIMESA Bangkok
A,CH,CS,E,M

Bangkok Business Equipment Ltd.
5/5-6 Dejo Road
BANGKOK
Tel: 234-8670, 234-8671
Telex: 87669-BEQUIPT TH
Cable: BUSIQUIPT Bangkok
P

TRINIDAD & TOBAGO
Caribbean Telecoms Ltd.
50/A Jerningham Avenue
P.O. Box 732
PORT-OF-SPAIN
Tel: 62-44213, 62-44214
Telex: 235,272 HUGCO WG
A,CM,E,M,P

TUNISIA
Tunisie Electronique
31 Avenue de la Liberte
TUNIS
Tel: 280-144
E,P

Corema
1 ter. Av. de Carthage
TUNIS
Tel: 253-821
Telex: 12319 CABAM TN
M

TURKEY
Teknim Company Ltd.
Iran Caddesi No. 7
*Kavaklidere, **ANKARA***
Tel: 275800
Telex: 42155 TKNM TR
E

E.M.A.
Medina Eldem Sokak No.41/6
Yuksel Caddesi
ANKARA
Tel: 175 622
M

UNITED ARAB EMIRATES
Emitac Ltd.
P.O. Box 1641
SHARJAH
Tel: 354121, 354123
Telex: 68136 Emitac Sh
CH,CS,E,M,P

UNITED KINGDOM

see: GREAT BRITAIN

NORTHERN IRELAND

SCOTLAND

UNITED STATES

Alabama

Hewlett-Packard Co.
700 Century Park South
Suite 128
BIRMINGHAM, AL 35226
Tel: (205) 822-6802
CH,MP

Hewlett-Packard Co.
P.O. Box 4207
8290 Whitesburg Drive, S.E.
HUNTSVILLE, AL 35802
Tel: (205) 881-4591
CH,CM,CS,E,M*

Alaska

Hewlett-Packard Co.
1577 "C" Street, Suite 252
ANCHORAGE, AK 99501
Tel: (907) 276-5709
CH*

Arizona

Hewlett-Packard Co.
2336 East Magnolia Street
PHOENIX, AZ 85034
Tel: (602) 273-8000
A,CH,CM,CS,E,MS

Hewlett-Packard Co.
2424 East Aragon Road
TUCSON, AZ 85706
Tel: (602) 889-4631
CH,E,MS**

Arkansas

Hewlett-Packard Co.
P.O. Box 5646
Brady Station
LITTLE ROCK, AR 72215
111 N. Filmore
LITTLE ROCK, AR 72205
Tel: (501) 664-8773, 376-1844
MS

California

Hewlett-Packard Co.
99 South Hill Dr.
BRISBANE, CA 94005
Tel: (415) 330-2500
CH,CS

Hewlett-Packard Co.
7621 Canoga Avenue
CANOGA PARK, CA 91304
Tel: (213) 702-8300
A,CH,CS,E,P

Hewlett-Packard Co.
5060 Clinton Avenue
FRESNO, CA 93727
Tel: (209) 252-9652
MS

Hewlett-Packard Co.
P.O. Box 4230
1430 East Orangethorpe
FULLERTON, CA 92631
Tel: (714) 870-1000
CH,CM,CS,E,MP

Hewlett-Packard Co.
320 S. Kellogg, Suite B
GOLETA, CA 93117
Tel: (805) 967-3405
CH

Hewlett-Packard Co.
5400 W. Rosecrans Boulevard
LAWNDALE, CA 90260
P.O. Box 92105
LOS ANGELES, CA 90009
Tel: (213) 970-7500
Telex: 910-325-6608
CH,CM,CS,MP

Hewlett-Packard Co.
3200 Hillview Avenue
PALO ALTO, CA 94304
Tel: (415) 857-8000
CH,CS,E

Hewlett-Packard Co.
P.O. Box 15976 (95813)
4244 So. Market Court, Suite A
SACRAMENTO, CA 95834
Tel: (916) 929-7222
A*,CH,CS,E,MS

Hewlett-Packard Co.
9606 Aero Drive
P.O. Box 23333
SAN DIEGO, CA 92123
Tel: (714) 279-3200
CH,CM,CS,E,MP

Hewlett-Packard Co.
2305 Camino Ramon "C"
SAN RAMON, CA 94583
Tel: (415) 838-5900
CH,CS

Hewlett-Packard Co.
P.O. Box 4230
Fullerton, CA 92631
363 Brookhollow Drive
SANTA ANA, CA 92705
Tel: (714) 641-0977
A,CH,CM,CS,MP

Hewlett-Packard Co.
Suite A
5553 Hollister
SANTA BARBARA, CA 93111
Tel: (805) 964-3390

Hewlett-Packard Co.
3003 Scott Boulevard
SANTA CLARA, CA 95050
Tel: (408) 988-7000
A,CH,CM,CS,E,MP

Hewlett-Packard Co.
5703 Corsa Avenue
WESTLAKE VILLAGE, CA 91362
Tel: (213) 706-6800
E*,CH*,CS*

Colorado

Hewlett-Packard Co.
24 Inverness Place, East
ENGLEWOOD, CO 80112
Tel: (303) 771-3455
Telex: 910-935-0785
A,CH,CM,CS,E,MS

Connecticut

Hewlett-Packard Co.
47 Barnes Industrial Road South
P.O. Box 5007
WALLINGFORD, CT 06492
Tel: (203) 265-7801
A,CH,CM,CS,E,MS

Florida

Hewlett-Packard Co.
P.O. Box 24210 (33307)
2901 N.W. 62nd Street
FORT LAUDERDALE, FL 33307
Tel: (305) 973-2600
CH,CS,E,MP

Hewlett-Packard Co.
4080 Woodcock Drive, #132
Brownett Building
JACKSONVILLE, FL 32207
Tel: (904) 398-0663
C*,E*,MS**

Hewlett-Packard Co.
1101 W. Hibiscus Ave., Suite E210
MELBOURNE, FL 32901
Tel: (305) 729-0704
E*

Hewlett-Packard Co.
P.O. Box 13910 (32859)
6177 Lake Ellenor Drive
ORLANDO, FL 32809
Tel: (305) 859-2900
A,CH,CM,CS,E,MS

Hewlett-Packard Co.
6425 N. Pensacola Blvd.
Suite 4, Building 1
P.O. Box 12826
PENSACOLA, FL 32575
Tel: (904) 476-8422
A,MS

Hewlett-Packard Co.
5750B N. Hoover Blvd., Suite 123
TAMPA, FL 33614
Tel: (813) 884-3282
A*,CH,CM,CS,E*,M*

Georgia

Hewlett-Packard Co.
P.O. Box 105005
ATLANTA, GA 30348
2000 South Park Place
ATLANTA, GA 30339
Tel: (404) 955-1500
Telex: 810-766-4890
A,CH,CM,CS,E,MP

Hewlett-Packard Co.
P.O. Box 816 (80903)
2531 Center West Parkway
Suite 110
AUGUSTA, GA 30904
Tel: (404) 736-0592
MS

Hewlett-Packard Co.
200-E Montgomery Cross Rds.
SAVANNAH, GA 31401
Tel:(912) 925-5358
CH**

Hewlett-Packard Co.
P.O. Box 2103
WARNER ROBINS, GA 31099
1172 N. Davis Drive
WARNER ROBINS, GA 31093
Tel: (912) 923-8831
E

Hawaii

Hewlett-Packard Co.
Kawaiahao Plaza, Suite 190
567 South King Street
HONOLULU, HI 96813
Tel: (808) 526-1555
A,CH,E,MS

Illinois

Hewlett-Packard Co.
211 Prospect Road, Suite C
BLOOMINGTON, IL 61701
Tel: (309) 662-9411
CH,MS**

Hewlett-Packard Co.
1100 31st Street, Suite 100
DOWNERS GROVE, IL 60515
Tel: (312) 960-5760
CH,CS

Hewlett-Packard Co.
5201 Tollview Drive
ROLLING MEADOWS, IL 60008
Tel: (312) 255-9800
A,CH,CM,CS,E,MP

Indiana

Hewlett-Packard Co.
P.O. Box 50807
7301 No. Shadeland Avenue
INDIANAPOLIS, IN 46250
Tel: (317) 842-1000
A,CH,CM,CS,E,MS

Iowa

Hewlett-Packard Co.
1776 22nd Street, Suite 1
WEST DES MOINES, IA 50265
Tel: (515) 224-1435
CH,MS**

Hewlett-Packard Co.
2415 Heinz Road
IOWA CITY, IA 52240
Tel: (319) 351-1020
CH,E*,MS

Kansas

Hewlett-Packard Co.
1644 S. Rock Road
WICHITA, KA 67207
Tel: (316) 684-8491
CH

Kentucky

Hewlett-Packard Co.
10300 Linn Station Road
Suite 100
LOUISVILLE, KY 40223
Tel: (502) 426-0100
A,CH,CS,MS

Louisiana

Hewlett-Packard Co.
8126 Calais Bldg.
BATON ROUGE, LA 70806
Tel: (504) 467-4100
A**,CH**

Hewlett-Packard Co.
P.O. Box 1449
KENNER, LA 70062
160 James Drive East
DESTAHAN, LA 70047
Tel: (504) 467-4100
A,CH,CS,E,MS

Maryland

Hewlett-Packard Co.
7121 Standard Drive
HANOVER, MD 21076
Tel: (301) 796-7700
Telex: 710-862-1943
Eff. Dec. 1, 1982
3701 Koppers St.
BALTIMORE, MD 21227
Tel: (301) 644-5800
A,CH,CM,CS,E,MS

Hewlett-Packard Co.
2 Choke Cherry Road
ROCKVILLE, MD 20850
Tel: (301) 948-6370
A,CH,CM,CS,E,MP

Massachusetts

Hewlett-Packard Co.
32 Hartwell Avenue
LEXINGTON, MA 02173
Tel: (617) 861-8960
A,CH,CM,CS,E,MP

Michigan

Hewlett-Packard Co.
23855 Research Drive
FARMINGTON HILLS, MI 48024
Tel: (313) 476-6400
A,CH,CM,CS,E,MP

Hewlett-Packard Co.
4326 Cascade Road S.E.
GRAND RAPIDS, MI 49506
Tel: (616) 957-1970
CH,CS,MS

Hewlett-Packard Co.
1771 W. Big Beaver Road
TROY, MI 48084
Tel: (313) 643-6474
CH,CS

Minnesota

Hewlett-Packard Co.
2025 W. Larpenteur Ave.
ST. PAUL, MN 55113
Tel: (612) 644-1100
A,CH,CM,CS,E,MP

Mississippi

Hewlett-Packard Co.
P.O. Box 5028
1675 Lakeland Drive
JACKSON, MS 39216
Tel: (601) 982-9363
MS

Missouri

Hewlett-Packard Co.
11131 Colorado Avenue
KANSAS CITY, MO 64137
Tel: (816) 763-8000
A,CH,CM,CS,E,MS

Hewlett-Packard Co.
P.O. Box 27307
1024 Executive Parkway
ST. LOUIS, MO 63141
Tel: (314) 878-0200
A,CH,CS,E,MP
Effective September 1982:
13001 Hollenberg Drive
BRIDGETON, MO 63044

Nebraska

Hewlett-Packard
7101 Mercy Road
Suite 101, IBX Building
OMAHA, NE 68106
Tel: (402) 392-0948
CM,MS

Nevada

Hewlett-Packard Co.
Suite D-130
5030 Paradise Blvd.
LAS VEGAS, NV 89119
Tel: (702) 736-6610
MS**

New Jersey

Hewlett-Packard Co.
W120 Century Road
PARAMUS, NJ 07652
Tel: (201) 265-5000
A,CH,CM,CS,E,MP

Hewlett-Packard Co.
60 New England Av. West
PISCATAWAY, NJ 08854
Tel: (201) 981-1199
A,CH,CM,CS,E

New Mexico

Hewlett-Packard Co.
P.O. Box 11634
ALBUQUERQUE, NM 87112
11300 Lomas Blvd.,N.E.
ALBUQUERQUE, NM 87123
Tel: (505) 292-1330
Telex: 910-989-1185
CH,CS,E,MS

New York

Hewlett-Packard Co.
5 Computer Drive South
ALBANY, NY 12205
Tel: (518) 458-1550
Telex: 710-444-4691
A,CH,E,MS

Hewlett-Packard Co.
P.O. Box 297
9600 Main Street
CLARENCE, NY 14031
Tel: (716) 759-8621
Telex: 710-523-1893
CH

Hewlett-Packard Co.
200 Cross Keys Office
FAIRPORT, NY 14450
Tel: (716) 223-9950
Telex: 510-253-0092
CH,CM,CS,E,MS

Hewlett-Packard Co.
7641 Henry Clay Blvd.
LIVERPOOL, NY 13088
Tel: (315) 451-1820
A,CH,CM,E,MS

Hewlett-Packard Co.
No. 1 Pennsylvania Plaza
55th Floor
34th Street & 8th Avenue
NEW YORK, NY 10119
Tel: (212) 971-0800
CH,CS,E*,M*

SALES & SUPPORT OFFICES
Arranged Alphabetically by Country

Hewlett-Packard Co.
250 Westchester Avenue
WHITE PLAINS, NY 10604
CM,CH,CS,E

Hewlett-Packard Co.
3 Crossways Park West
WOODBURY, NY 11797
Tel: (516) 921-0300
Telex: 510-221-2183
A,CH,CM,CS,E,MS

North Carolina
Hewlett-Packard Co.
4915 Water's Edge Drive
Suite 160
RALEIGH, NC 27606
Tel: (919) 851-3021
C,M

Hewlett-Packard Co.
P.O. Box 26500
5605 Roanne Way
GREENSBORO, NC 27450
Tel: (919) 852-1800
A,CH,CM,CS,E,MS

Ohio
Hewlett-Packard Co.
9920 Carver Road
CINCINNATI, OH 45242
Tel: (513) 891-9870
CH,CS,MS

Hewlett-Packard Co.
16500 Sprague Road
CLEVELAND, OH 44130
Tel: (216) 243-7300
Telex: 810-423-9430
A,CH,CM,CS,E,MS

Hewlett-Packard Co.
962 Crupper Ave.
COLUMBUS, OH 43229
Tel: (614) 436-1041
CH,CM,CS,E*

Hewlett-Packard Co.
P.O. Box 280
330 Progress Rd.
DAYTON, OH 45449
Tel: (513) 859-8202
A,CH,CM,E*,MS

Oklahoma
Hewlett-Packard Co.
P.O. Box 32008
Oklahoma City, OK 73123
1503 W. Gore Blvd., Suite #2
LAWTON, OK 73505
Tel: (405) 248-4248
C

Hewlett-Packard Co.
P.O. Box 32008
OKLAHOMA CITY, OK 73123
304 N. Meridian Avenue, Suite A
OKLAHOMA CITY, OK 73107
Tel: (405) 946-9499
A*,CH,E*,MS

Hewlett-Packard Co.
Suite 121
9920 E. 42nd Street
TULSA, OK 74145
Tel: (918) 665-3300
A**,CH,CS,M*

Oregon
Hewlett-Packard Co.
1500 Valley River Drive
Suite 330
EUGENE, OR 97401
Tel: (503) 683-8075
C

Hewlett-Packard Co.
9255 S. W. Pioneer Court
WILSONVILLE, OR 97070
Tel: (503) 682-8000
A,CH,CS,E*,MS

Pennsylvania
Hewlett-Packard Co.
1021 8th Avenue
King of Prussia Industrial Park
KING OF PRUSSIA, PA 19406
Tel: (215) 265-7000
Telex: 510-660-2670
A,CH,CM,CS,E,MP

Hewlett-Packard Co.
111 Zeta Drive
PITTSBURGH, PA 15238
Tel: (412) 782-0400
A,CH,CS,E,MP

South Carolina
Hewlett-Packard Co.
P.O. Box 21708
Brookside Park, Suite 122
1 Harbison Way
COLUMBIA, SC 29210
Tel: (803) 732-0400
CH,E,MS

Hewlett-Packard Co.
Koger Executive Center
Chesterfield Bldg., Suite 124
GREENVILLE, SC 29615
Tel: (803) 748-5601
C

Tennessee
Hewlett-Packard Co.
P.O. Box 22490
224 Peters Road
Suite 102
KNOXVILLE, TN 37922
Tel: (615) 691-2371
A*,CH,MS

Hewlett-Packard Co.
3070 Directors Row
MEMPHIS, TN 38131
Tel: (901) 346-8370
A,CH,MS

Hewlett-Packard Co.
230 Great Circle Road
Suite 216
NASHVILLE, TN 32228
Tel: (615) 255-1271
MS**

Texas
Hewlett-Packard Co.
Suite 310W
7800 Shoalcreek Blvd.
AUSTIN, TX 78757
Tel: (512) 459-3143
E

Hewlett-Packard Co.
Suite C-110
4171 North Mesa
EL PASO, TX 79902
Tel: (915) 533-3555, 533-4489
CH,E*,MS**

Hewlett-Packard Co.
5020 Mark IV Parkway
FORT WORTH, TX 76106
Tel: (817) 625-6361
CH,CS*

Hewlett-Packard Co.
P.O. Box 42816
HOUSTON, TX 77042
10535 Harwin Street
HOUSTON, TX 77036
Tel: (713) 776-6400
A,CH,CM,CS,E,MP

Hewlett-Packard Co.
3309 67th Street
Suite 24
LUBBOCK, TX 79413
Tel: (806) 799-4472
M

Hewlett-Packard Co.
417 Nolana Gardens, Suite C
P.O. Box 2256
McALLEN, TX 78501
Tel: (512) 781-3226
CH,CS

Hewlett-Packard Co.
P.O. Box 1270
RICHARDSON, TX 75080
930 E. Campbell Rd.
RICHARDSON, TX 75081
Tel: (214) 231-6101
A,CH,CM,CS,E,MP

Hewlett-Packard Co.
P.O. Box 32993
SAN ANTONIO, TX 78216
1020 Central Parkway South
SAN ANTONIO, TX 78232
Tel: (512) 494-9336
CH,CS,E,MS

Utah
Hewlett-Packard Co.
P.O. Box 26626
3530 W. 2100 South
SALT LAKE CITY, UT 84119
Tel: (801) 974-1700
A,CH,CS,E,MS

Virginia
Hewlett-Packard Co.
P.O. Box 9669
2914 Hungary Spring Road
RICHMOND, VA 23228
Tel: (804) 285-3431
A,CH,CS,E,MS

Hewlett-Packard Co.
3106 Peters Creek Road, N.W.
ROANOKE, VA 24019
Tel: (703) 563-2205
CH,E**

Hewlett-Packard Co.
5700 Thurston Avenue
Suite 111
VIRGINIA BEACH, VA 23455
Tel: (804) 460-2471
CH,MS

Washington
Hewlett-Packard Co.
15815 S.E. 37th Street
BELLEVUE, WA 98006
Tel: (206) 643-4000
A,CH,CM,CS,E,MP

Hewlett-Packard Co.
Suite A
708 North Argonne Road
SPOKANE, WA 99206
Tel: (509) 922-7000
CH,CS

West Virginia
Hewlett-Packard Co.
4604 MacCorkle Ave., S.E.
CHARLESTON, WV 25304-4297
Tel: (304) 925-0492
A,MS

Wisconsin
Hewlett-Packard Co.
150 S. Sunny Slope Road
BROOKFIELD, WI 53005
Tel: (414) 784-8800
A,CH,CS,E*,MP

URUGUAY
Pablo Ferrando S.A.C. e L.
Avenida Italia 2877
Casilla de Correo 370
MONTEVIDEO
Tel: 80-2586
Telex: Public Booth 901
A,CM,E,M

Guillermo Kraft del Uruguay S.A.
Av. Lib. Brig. Gral. Lavalleja 2083
MONTEVIDEO
Tel: 234588, 234808, 208830
Telex: 22030 ACTOUR UY
P

VENEZUELA
Hewlett-Packard de Venezuela C.A.
3A Transversal Los Ruices Norte
Edificio Segre
Apartado 50933
CARACAS 1071
Tel: 239-4133
Telex: 25146 HEWPACK
A,CH,CS,E,MS,P

Colimodio S.A.
Este 2 - Sur 21 No. 148
Apartado 1053
CARACAS *1010*
Tel: 571-3511
Telex: 21529 COLMODIO
M

ZIMBABWE
Field Technical Sales
45 Kelvin Road, North
P.B. 3458
SALISBURY
Tel: 705 231
Telex: 4-122 RH
C,E,M,P

Headquarters offices
If there is no sales office listed for your area, contact one of these headquarters offices.

NORTH/CENTRAL AFRICA
Hewlett-Packard S.A.
7 Rue du Bois-du-Lan
CH-1217 **MEYRIN 2**, Switzerland
Tel: (022) 98-96-51
Telex: 27835 hpse
Cable: HEWPACKSA Geneve

ASIA
Hewlett-Packard Asia Ltd.
6th Floor, Sun Hung Kai Center
30 Harbor Rd.
G.P.O. Box 795
HONG KONG
Tel: 5-832 3211
Telex: 66678 HEWPA HX
Cable: HEWPACK HONG KONG

CANADA
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive
MISSISSAUGA, Ontario L4V 1M8
Tel: (416) 678-9430
Telex: 610-492-4246

EASTERN EUROPE
Hewlett-Packard Ges.m.b.h.
Lieblgasse 1
P.O.Box 72
A-1222 **VIENNA**, Austria
Tel: (222) 2365110
Telex: 1 3 4425 HEPA A

NORTHERN EUROPE
Hewlett-Packard S.A.
Uilenstede 475
NL-1183 AG **AMSTELVEEN**
The Netherlands
P.O.Box 999
NL-1180 AZ **AMSTELVEEN**
The Netherlands
Tel: 20 437771

OTHER EUROPE
Hewlett-Packard S.A.
7 Rue du Bois-du-Lan
CH-1217 **MEYRIN 2**, Switzerland
Tel: (022) 98-96-51
Telex: 27835 hpse
Cable: HEWPACKSA Geneve
(Offices in the World Trade Center)

MEDITERRANEAN AND MIDDLE EAST
Hewlett-Packard S.A.
Mediterranean and Middle East Operations
Atrina Centre
32 Kifissias Ave.
Maroussi, **ATHENS**, Greece
Tel: 682 88 11
Telex: 21-6588 HPAT GR
Cable: HEWPACKSA Athens

EASTERN USA
Hewlett-Packard Co.
4 Choke Cherry Road
Rockville, MD 20850
Tel: (301) 258-2000

MIDWESTERN USA
Hewlett-Packard Co.
5201 Tollview Drive
ROLLING MEADOWS, IL 60008
Tel: (312) 255-9800

SOUTHERN USA
Hewlett-Packard Co.
P.O. Box 105005
450 Interstate N. Parkway
ATLANTA, GA 30339
Tel: (404) 955-1500

WESTERN USA
Hewlett-Packard Co.
3939 Lankersim Blvd.
LOS ANGELES, CA 91604
Tel: (213) 877-1282

OTHER INTERNATIONAL AREAS
Hewlett-Packard Co.
Intercontinental Headquarters
3495 Deer Creek Road
PALO ALTO, CA 94304
Tel: (415) 857-1501
Telex: 034-8300
Cable: HEWPACK

15 Aug 1982 5952-6900

HEWLETT PACKARD

MANUAL PART NO. 92084-90015
Printed in U.S.A. April 1983
E0483

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014