TECHNICAL SPECIFICATIONS

for

RTE-IV

RTE-IV TECHNICAL SPECIFICATIONS

SECTIONS

I	DISPATCHER
II	RTIOC
III	EXECUTIVE/MEMORY ALLOCATION
IV	SCHEDULAR
V	PARITY ERROR
VI	SYSTEM LIBRARY
VII	EMA FIRMWARE
VIII	CONFIGURATOR (CURRENTLY UNAVAILABLE)
IX	GENERATOR
X	SWTCH (CURRENTLY UNAVAILABLE)
XI	LOADR (CURRENTLY UNAVAILABLE)
XII	MULTI-TERMINAL MONITOR
XIII	ASSEMBLER

TECHNICAL SPECIFICATIONS

FOR

DSP4 - RTE IV DISPATCHER

EJW
January 23, 1978
Project 1106

TABLE OF CONTENTS

1.0 General Overview of Operation

2.0 External Communication

2.1 System Tables Referenced

2.1.1 ID Segment
2.1.2 Memory Protect Fence Table
2.1.3 Memory Allocation Table

2.2 System Base Page Communication
2.3 Dispatcher Entry Points
2.4 Dispatcher's External Tables and Pointers

3.0 Detailed Technical Aspects of Operations

3.1 Initialization
3.2 Dispatching

3.2.1 Memory Resident Programs
3.2.2 Disc Resident Programs
3.2.3 Disc Resident Map Setup

3.3 Swapping
3.4 Mother Partition Usage
3.5 Clean Up

FIG. 1 RTE IV ID Segment
FIG. 2 RTE IV Memory Protect Fence Table
FIG. 3 RTE IV Memory Allocation Table
FIG. 4 RTE IV Memory Allocation Table Entry
FIG. 5 RTE IV Mother Partitions
FIG. 6 RTE IV Dispatcher's Allocated Partition Lists
FIG. 7 RTE IV User Base Page

1.0 GENERAL OVERVIEW OF OPERATION

The Dispatcher module's main function is to control program execution by switching CPU control to the highest priority program in the scheduled list if it is ready to run. The Dispatcher serves as the return point from the operating system back to the user. If there are no programs scheduled, the Dispatcher prepares for execution of the idle loop under the user map and waits for an interrupt to occur. An interrupt is generated by an I/O device, a user program request, or an error condition signal which requires entry into the operating system for a response. The Dispatcher turns control over for program execution until the next interrupt.

In the process of turning control to a user program, the Dispatcher needs to perform a number of smaller tasks. Once it is determined that a program should be executed, the Dispatcher needs to check whether or not the program is present in memory. If the program is not already in memory then it needs to be loaded from the disc. The Dispatcher is responsible for finding an empty partition of the proper type and size, if one was not previously specified. If there are no free partitions, it would be necessary to swap out a dormant program or a lower priority executing program to make the partition available for the load. Then finally the user map and memory protect fence are set up before executing the program.

Other functions of the Dispatcher are to set up the partition list headers at initialization of the operating system and to coordinate the cleaning up of a program's resources and system available memory when a program is aborted.

2.0 EXTERNAL COMMUNICATION

The Dispatcher communicates with the rest of the operating system through the system tables, base page communication area, and subroutine calls to/from the rest of the system. There are no direct paths of communication between the Dispatcher and any user program because there are no functions in the Dispatcher which would be useful to a calling program.

2.1 System Tables Referenced

2.1.1 ID Segment (Figure 1)

An ID segment is used for many things by the Dispatcher. Each ID segment is linked into the scheduled list by word 0. The priority, type, and status are checked to determine whether or not the program may execute. The memory protect fence table index (MPFI) and number of pages are set into word 21 by the system generator or by the relocating loader. The number of pages (which does not include a base page) is the number of pages actually occupied by a program and its largest segment (if any) or the override size specified by the user. The size of an EMA program includes the mapping segment size (still excludes the base page).

The program address bounds in words 22-25 are used by the Dispatcher for loading the program into memory. The program's disc address is in word 26. If the program is swapped out, the Dispatcher keeps the address of the swap tracks in word 27.

ID Segment word 28 is used by the Dispatcher to determine whether or not a program is using an extended memory area (EMA). If this word is zero, no EMA is used. If the word is non-zero, the ID Extension is checked to see if the default EMA size was chosen. If it was not defaulted then the program size (from word 21) minus the MSEG size is added to the EMA size plus base page to determine the partition size required. If the EMA size was defaulted, the largest Mother partition size, $MCHN, is used as the partition size required (see Section 3.2.2).

2.1.2 Memory Protect Fence Table (Figure 2)

The Memory Protect Fence table ($MPFT) a table of addresses used by the Dispatcher to set the correct value for the memory protect fence. This address is stored in the base page word FENCE (1775). Bits 7-9 of word 21 in the ID Segment contain the index into this table.

2.1.3 Memory Allocation Table (Figure 3)

Each partition defined by the user at generation time will have an entry in the Memory Allocation Table. The table starts at the entry point $MATA and extends upwards toward high memory. The word $MNP contains the total number of partitions (set by the generator). Each partition entry (see Figure 4) is 7 words long. The MAT Link Word contains -1 if the partition is undefined.

There are 3 different types of partitions:

1. Real Time Partitions headed by $RTFR at system start-up.
2. Background Partitions headed by $BGFR at system start-up.
3. Mother Partitions headed by $CFR at system start-up.

These 3 lists are set up by the Generator in order of increasing size. The primary purpose for having RT partitions and BG partitions is to keep the two classes of programs, RT and BG, from contending with each other for memory. There are no differences in the two classes of partitions. Mother Partitions are primarily for EMA programs.

Mother partitions are defined during generation by a "YES" answer to the prompt "SUBPARTITIONS?" 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 (see Figure 5) or linked into a BG or RT (either free or allocated) list. When the subpartitions are part of a Mother partition, the Mother partition's MAT entry word 6 (Subpartition Link Word - SLW) will point to the Link Word (word 0) of the first subpartition whose SLW will point to the link word of the second subpartition and so on. The SLW of the last subpartition will point to the Link Word of the Mother. If no subpartitions were actually defined but the prompt for subpartitions was answered "YES" then the SLW of the Mother partition will point to the Link Word of the same MAT entry.

When a Mother partition is in use, the entire chain of subpartitions is in use and this is indicated by the C bit being set. In this case, all partition status information is kept in the Mother partition MAT entry; i.e.; priority, ID segment address, and Read Completion flag. In addition, the chained partitions are treated as a single entity whil;e the C bit is set. Individual subpartitions are not swappable in this case - the whole Mother partition may be swapped if needed.

The Dispatcher checks for empty partition lists at start-up (see Section 3.1). If there are no Real Time partitions then the header of the RT partitions list will point to the Background partitions. If there are no BG partitions then the RT partitions are used in the BG list. If there are no Mother partitions BG partitions are used unless there are no BG partitions, in which case the RT partitions are used.

The sizes of the largest partition of each type are kept in 3 words:

1. $MRTP - size of the largest non-reserved RT partition.
2. $MBGP - size of the largest non-reserved BG partition.
3. &MCHN - size of the largest non-reserved Mother partition.

2.2 System Base Page Communication

```
* XIDEX 1645          ID EXT addr of current prog or 0
* XMATA 1646          MAT address of current program or 0
* XI    1647          Pointer to current program's X-Y save area
* SKEDD 1711          Head of scheduled list
* XEQT  1717-1733     Current program's ID Segment pointers
  SWAP  1736          Swap delay in bits 8-15
  BPA2  1743          Last word user base page (add 1 for BP fence)
  LBORG 1745          Logical address of Resident Library
* RTDRA 1750          Dynamically set address
* AVMEM 1751              bounds for partition
* BGDRA 1754              resident programs
* TATLG 1755          Dispatcher locks disc tracks for FMGR
  TATSD 1756          #Tracks system disc
  SECT2 1757          #Sectors LU2
  SECT3 1760          #Sectors LU3
* FENCE 1775          Memory Protect Fence value for user
* BGLWA 1777          Dynamically set bound
```

* Set or changed by the Dispatcher.

2.3 Dispatcher Entry Points

$ALDM Subroutine, used by SCHEDULER to unlink a partition's MAT entry from the allocated list into the dormant list.

The calling sequence:

<A-reg> has ID Segment Address
JSB $ALDM

$BRED Subroutine, used by SCHEDULER to read in program segments.

The calling sequence:

<B-reg> has ID Segment Address
JSB $BRED

$DMAL Subroutine, used by SCHEDULER to unlink a partition's MAT entry from the dormant list into the allocated list.

The calling sequence:

<A-reg> has ID Segment Address
JSB $DMAL

$MAXP Subroutine, called by the routine $PERR when a partition is undefined or by the SCHEDULER when a partition's "reserved" status is changed by the RS command. This subroutine searches the MAT to determine the values of $MBGP, $MRTP and $MCHN by scanning each MAT entry for the largest partition of the specified type and update the appropriate word. $MAXP may have to update more than one (and maybe all) if the size words if any of the partition lists were initially empty. This is necessary because the DISPATCHER would have changed the list pointers of the empty list(s) to point to a non-empty list.

The calling Sequence:

JSB $MAXP
<return>

$PRCN — Subroutine, called by DISPATCHER and SCHEDULER to relink a partition in the allocated list by priority.

Calling sequence:

<A-reg> has ID Segment Address
<B-reg> has new priority
JSB $PRCN

$RENT — Entry point, JMP there from the DISPATCHER and EXEC for setting up the Resident Library address in the memory protect fence.

$SMAP — Subroutine, called by DISPATCHER and RTIOC to set up the user map. Unused pages are write-protected.

Calling sequence:

<A-reg> has length of program to map in pages
<B-reg> has MAT entry address
JSB $SMAP

$UNPE — Subroutine, called by $PERR to unlink partition MATA entries and undefine the partition where a parity error is detected in it.

Calling sequence:

<B-reg> has MATA address of partition
JSB $UNPE

$XCQ — Entry point, actual entry for $XEQ where the main dispatching algorithm is performed.

$XDM — Subroutine, called by non-privileged drives to set up user map.

Calling sequence:

<A-reg> has ID Segment Addr
JSB $XDMP (Entry via Table Area II entry point)

$ZZZZ — Entry point, used by DISPATCHER and SCHEDULER as the head of the program abort list. This entry point is used as a subroutine during the start-up sequence.

2.4 Dispatcher's External Tables and Pointers

All of these entry points are located in Table Area 2 unless otherwise specified.

$BGFR Pointer, BG free list initialized by the Generator.

$CFR Pointer, free list header of Mother partitions, initialized by Generator.

$CMST Value, start page number of COMMON area, set up by the Generator.

$EMRP Address, last word of memory resident program area, set by the Generator.

$ENDS Value, number of pages occupied by the system, its base page. Both table areas, System Driver Area, driver partition area, Common, and the first 2K of System Available Memory. This word is initialized by the Generator.

$IDEX Pointer, ID Extension list.

3.0 DETAILED TECHNICAL ASPECTS OF OPERATION

This portion of the Technical Specifications is a detailed description of major portions of the DISPATCHER code as outlined in the general overview (Section 1.0). It is assumed that the reader has a good general understanding of the RTE operating system.

3.1 Initialization

$ZZZZ serves as the entry point of the initialization subroutine which is executed only once during system initialization. The routine first sets up the starting address of SSGA in $SGAF for the EXEC. Then $ZZZZ loads the user map with the memory resident map registers which were built by the Generator in $MRMP. Whenever a memory resident program is dispatched, $MRMP is used to set up the user map.

Next, the $ZZZZ routine will set the base page fence so that all addresses between 16XX and 1777 are not mapped. The fence value minus one is contained in base page location BP2 (1743) which is the address of the last available user link.

A number of system dependent address and sizes are calculated and saved so that they may be reused by the DISPATCHER without being recalculated each tim Some of these "constants" include the address of the memory resident library, the number of pages in the memory resident library, the starting register number and the number of registers to be used for mapping chunks of EMA to be swapped.

The partition free lists are also checked by the Dispatcher during the initialization of the system. If the BG free list header ($BGFR) is zero, meaning that it is empty, then the RT free list pointer ($RTFR) is stored into $BGFR. If there were no RT partitions then the RT free list pointer will be set equal $BGFR. The maximum unreserved partition size words, $MRTP and $MBGP, will be updated accordingly. If the Mother partition list is empty the $BGFR pointer will be used, assuming that the BG list is not empty since we've already done the check above. $MCHN will also be updated. However, if it turns out that the BG list is empty because the RT list was empty, then with all three lists empty the SCHEDULER will report an error on any scheduling attempts. This code is in subroutine LSTIN which is also called by $MAXP.

The last thing done during initialization is the scheduling of the File Manager program, FMGR. The Dispatcher first locks all of the disc tracks by saving the number of tracks word (TATLG) in the FMGR's first parameter word and then replacing TATLG with -1.

e DISPATCHER'S initialization code is overlayed by the disc
O triplets which are built for doing program loads and swaps.

3.2 Dispatching

$XCQ (user map entry point, $XEQ) is the entry point into the DISPATCHER code which performs the allocation of execution time to programs. The primary objective of $XCQ is to execute the highest priority program in the scheduled list, SKEDD, if possible.

First the DISPATCHER checks to see if there are any programs which were aborted (see Section 3.5). If no programs were aborted, then the DISPATCHER checks the scheduled list. If there are no programs in the scheduled list, or for some reason the programs in the list can't be executed at this time, the "idle loop" is executed instead of a user program. The base page point-of-suspension word, XSUSP, is set to the idle loop code address and the base page register save area pointers (XA, XB, XEO and XI) are set up to use a two-word dummy save area. The idle loop code ($IDLE) and dummy save area are located in Table Area I so the user map may be used. The base page word XEQT is cleared to indicate that no program is executing, the memory protect fence register is set to zero and stored in FENCE on base page. Then it exits the system via $IRT to enable memory protect, the interrupt system and the user map.

If there are programs to be scheduled in the SKEDD list, the DISPATCHER makes the decision to execute a program based on the program's priority, status, type, and address space needs.

If the currently executing program is of equal or higher priority then the program in the scheduled list, execution of the current program is resumed. If the program in the scheduled list is higher, a check is made to see if the program is in memory and if it can be executed. If it can be executed the user map registers are set up with the program's physical page numbers. The program's logical address bounds are set up in RTDRA, AVMEM, BKLRA and BKLWA. The ID Segment pointers are set up in base page at XEQT. The X and Y registers save area address is also set up at XI.

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. Then control is turned over to the program by exiting through $IRT which enables memory protect, enables the interrupt system, and enables the user map.

The general dispatching procedure described above is slightly different for different types of programs.

3.2.1 Memory Resident Programs

If the scheduled program is a Memory Resident Program, the memory protect fence may be set differently since these programs are the only type which may reference the Resident Library. The Dispatcher will clear the Write Protect bit from the Resident Library pages in the Memory Resident Map by clearing the sign bit from those words when the User Map is being set up. All the other registers would remain the same as in the Memory Resident Map Table ($MRMP) which is never changed. Then the memory protect fence is set at LBORG if the reentrant bit is set in the ID Segment. This code has an entry point of $RENT for access by the EXEC.

3.2.2 Disc Resident Programs

If the scheduled program is a disc resident program, it needs to be loaded from the disc into a partition which was allocated for it. If a partition was pre-assigned at relocation time, that MAT entry will be checked to see if it is available or if its resident program is swappable. If either case is true, the MAT entry will be set up for the new program and will be linked into the allocated list of the pertinent type. The MAT try will not be modified or relinked if its current resident the program which RTE is trying to dispatch.

If a partition was not specified at load time, the MAT entry for the partition in which the program was last resident will be checked to see if the program is still resident. The MAT entry is first checked to see if the partition still exists. The partition may be undefined if a parity error was detected in one of its pages since the program was last resident there. If the partition still exists and the program was the last occupant in there, the partition is set up to be used and the program is dispatched there after the user map is built (see Section 3.2.3).

If the program is no longer resident in the partition or the partition became undefined, a default partition list will be scanned for a free partition. The default partition types are:

a. RT program (Type 2) - RT partition ($RTFR)
b. BG program (Type 4) - BG partition ($BGFR)
c. Privileged program (Type 3) - BG partition ($BGFR)
d. EMA program - Mother partition ($CFR)

For RT, BG and privileged programs, the appropriate free list will be scanned for the smallest free partition in which the program can fit. EMA programs which have a specific EMA size declared will get the smallest free Mother partition. But, EMA programs which let the EMA size default will take on the maximum Mother partition size ($MCHN). This size minus the program code size is then put into word 29 of the IO segment to prevent the problem where a swapped out program may be reloaded into a smaller partition if $MCHN was changed because of a parity error on the partition or it became reserved.

If a free partition is not available, the appropriate dormant list (RTDM, BGDM or CDM) will be scanned for a partition with a swappable program. The dormant lists are a subset of the allocated lists (see Figure 6). The last entry in the dormant list points to the allocated list so if the dormant list is empty, the dormant list just points to the allocated list.

Upon finding a suitable partition, the occupant will be swapped out. The MAT entry will be reset and relinked, and prepared for loading of the disc resident program. If no dormant partition qualifies for the swap, the allocated list (RTPR, BGPR or CPR) will be scanned for the lowest priority program which can be swapped. The same procedure described for the dormant swap will be followed.

3.2.3 Disc Resident Map Set Up

Once a partition is allocated for a program, the user map is set up for the program. If the program is being scheduled initially (program's first dispatch) the User Map registers must be loaded by the DISPATCHER and a copy of it saved in the user's protected portion of base page (see Figure 7). If the program is being re-dispatched, 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 page value in word 3 of the MAT entry. The next registers are then loaded sequentially with numbers starting at one end 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.

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: Table Area 1, the Driver Partition Area, COMMON (including SSGA), write-protected System Driver Area and write-protected Table Area 2.

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 non-standard MSEG bit (bit 15 of word 0 in the ID Segment extension) is set if the program is an EMA program to force the EMA subroutines to remap.

e remaining registers in the user map will be read/write protected to insure that a program cannot access memory outside of its partition. This mapping is done in $SMAP which is the only code which loads the user map to describe a specific program. It is also called by RTIOC.

A copy of the user map is saved in the last 32 words of the user's physical base page (see Figure 7). 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 for dispatch.

3.3 Swapping

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 the ones in the dormant list. These programs are the ones which have terminated with either the save resources or serially reusable option. Otherwise, programs with the lowest priority will be checked for swappability.

Programs are not swappable if any of the following are true:

1. A memory lock is in effect.
2. If has a higher priority than the program to be scheduled.
3. It is dormant but is higher priority and is in the time list to be scheduled in less than the minimum permissible amount of time specified in SWAP.
4. It is I/O suspended with the buffer in the program area.

When a swap is required, the necessary number of tracks needed to swap the program out are computed and a request is made to $DREQ for the contiguous disc tracks. The number of tracks is computed by rounding up the number of sectors (to next even sector) needed for the base page and rounding up the number of sectors (to the next even sector) needed for the Main Code. The number of sectors is then converted to tracks and it is then rounded up to the end of a full track. If tracks are not currently available, swapping cannot take place and the DISPATCHER proceeds to check the next program in the scheduled list (if any). If tracks are available, the necessary $XSIO parameters are computed.

SETUP is the subroutine which creates the parameters for the $XSIO disc calls. SETUP guarantees that all calls to read or write disc tracks are broken down into groups of smaller I/O requests. Each one of these smaller groups of 3 words each (triplets) define an I/O request which will not cross a track boundary. The triplets have (1) starting memory address of the piece of data, (2) the number of words to transfer, and (3) the starting track and sector address. These triplets are built in memory overlaying the DISPATCHER'S initialization code (code following $ZZZZ). There may be up to seven triplets for a $XSIO call (enough for a 32K transfer with 6K words per track). The triplets are terminated by a zero.

There are five separate $XSIO calls, one for each type load/swap I/O so that each call can be started independently and overlap in time. Disc accesses for each type of partition can be completed at different times depending on their sizes. For each of these calls, there are triplets tables. The following table shows the names used by DISPATCHER.

Type of $XSIO CALL	Code Busy Flag	Triplets Area Terminator	Triplets Terminator Address	Triplets Area Pointer
RT	RTSWP	RTRIP	RTRPA	RTRP
BG	BGSWP	BTRIP	BTRPA	BTRP
Mother	CHSWP	CTRIP	CTRPA	CTRP
EMA	CHSW2	CTRIP	CTRPA	CTRP2
Segment	SGSWP	STRIP	STRPA	STRP

When a swap out is completed, the disc logical unit track address and number of tracks are stored in the ID Segment (word 27, SMAN).

When a program is loaded (or swapped back) into memory, $XSIO is called using parameters computed by SETUP. ID Segment value DMAN (word 26) is used for the disc address if the program is not swapped out; and SMAN if the program is swapped out. The program's dispatching status in word 5 of the MAT entry is cleared to indicate that a program read is in progress. The program's status is changed to I/O suspend via $LIST. When the read is complete, any swap tracks are released via $DREL and the program is scheduled via $LIST. A check is made to see if the read was correct. If not, the program is aborted via $ABRT which sets it dormant, releases its tracks, and removes it from the time list. If the read was correct, the dispatching status is set to one to indicate program is read in. The program is ready to execute.

When an EMA program needs to be swapped, the swap out to the disc is done in two parts. The program's code up to the page where the mapping segment starts (MSEG) and the program's base page are swapped out first using the CHSWP $XSIO call. The number of swap tracks needed is computed by adding the number of integral tracks needed for the program code and base page to the number of integral tracks needed for the EMA area. The program is swapped just like any other disc resident program. Note that in the ID Segment word 27, the number of tracks refer to just those used for the program code.

The EMA area is swapped out next, beginning at the next even sector boundary following the program code's swap tracks. EMA is swapped out in large chunks equal in size to the maximum logical address space in the user map (up to a maximum of 28K words). The User Map registers from $CMST to the end of the the map, inclusive, define the number of pages in each chunk. The chunk is mapped in the User Map, the triplets are built and then the chunk is swapped out using the CHSW2 $XSIO call. When the transfer is completed, the next chunk is mapped and swapped. This process repeats until all of the EMA is swapped. A similar process takes place when swapping into memory.

The computation for the number of swap tracks needed must allow an extra sector for each chunk. The number of tracks for the EMA area is saved in word 2 of the program's ID Segment Extension. The number of tracks is needed so that the correct number of tracks can be released when the program terminates or gets aborted.

When a program is swapped out, the program's map is used. When a program is mapped in, it is necessary for the map to be rebuilt according to the information in the ID segment rather than using a copy of the user map in the protected portion of base page (because it is swapped out on the disc). Programs which do their own mapping must lock itself in memory. Because these programs can't be swapped back with the altered map registers. EMA programs which have been swapped back into memory will have the MSEG registers recalculated and remapped because the program may have been swapped back into a different mother partition (and therefore different physical page numbers). This calculation is performed by determining the number of pages offset from the beginning of the EMA and using the same offset in the new EMA. The physical page number of the first page in the currently mapped MSEG is saved temporarily in the ID segment extension in place of the current MSEG number during the time the program is swapped out. When the program is swapped back into memory, the MSEG pages are remapped after calculating the offset into the new partition. The remaining pages remaining in the MSEG which are past the end of the EMA are read-write protected. If no MSEG was mapped at the time of the swap out, the MSEG pages will all be read-write protected when the program is swapped back into memory.

3.4 Mother Partition Usage

If a program (any type) is assigned to a Mother partition or an EMA program defaults to any Mother partition, there is more handling involved than is the case with RT or BG partitions. If a Mother partition is used when it is in the free list ($CFR), each subpartition must be checked. If a subpartition is either free or is occupied by a swappable program the C bit is set in word 4 of the MAT entry to prevent the subpartition from getting used while the Dispatcher continues to check each subpartition. If all of the subpartitions are either free or swappable, a second pass is made on all of the subpartitions to perform the necessary swaps. The subpartitions are unlinked from any lists they might be in. When all of the subpartitions are free, the Mother partition is unlinked from the free list ($CFR) and linked into the allocated list. The program can then be loaded into the Mother partition.

If any one of the subpartitions has a memory-locked program or a program which is doing I/O in it´s own program space the subpartition can´t be made available by swapping. All of the C bits must be cleared from each one of the previously scanned subpartitions and the dispatch is terminated. The next program in the scheduled list is examined.

When a program terminates and it was using a Mother partition, the Mother partition is relinked in the free list. All of the subparetitions are linked into the free list of the appropriate type (BG or RT).

When the C-bits are set on the subpartitions (set in chained mode), programs which are assigned to these subpartition will have to wait if the DISPATCHER is still in the process of swapping out any subpartitions. If a program is already in the Mother Partition the normal swappability checks apply.

In the case where a program of lower priority was in the process of loading a Mother Partition and a scheduled program is assigned to a subpartition the loading process is aborted. Then the subpartitions are released from chained mode and relinked into the proper free list. A special check is made (at SMABT) when a Mother Partition load needs to be aborted to free up a specific subpartition. If the partition type is BG and the BGSWP call is busy, the abort is not performed. If the type if RT and the RTSWP is busy, the abort also does not take place. This check prevents a deadlock which could keep the interrupt system off and the busy RT or BG call would not be able to complete. When it is necessary to clear out all the subpartitions for a Mother Partition the CHSWP call is used so that regular RT/BG partitions may continue to be used for other programs.

When a RT or BG program is scheduled and it is not assigned to a partition, a search is made for a partition of the same type which is large enough. If none can be found in the free list, none in the dormant list, and none can be found in the allocated list or it contains non-swappable programs, then the dormant Mother partition list will be searched for one which has a subpartition of the correct type and size. If a suitable subpartition can be found, the dormant program in the Mother partition will be swappe out.

3.5 Clear Up

Whenever a program is terminated, either by an EXEC call 6 or aborted by the system because of an error, the program is put into the dormant state and the list processor adds the program's ID Segment into a list headed by $ZZZZ. The linking is through word 8 of the ID Segment (the point of suspension save area) since the program will begin execution at the primary entry point if it is rescheduled. Everytime the system goes to $XEQ, $ZZZZ is checked. If it is non-zero the DISPATCHER performs five major clean up tasks.

First, if the program is disc resident, any swap tracks it may have are released. This may happen if a program is aborted while it is swapped out. When tracks are released by $DREL in the EXEC module it will also call the list processor at $LIST to reschedule any programs waiting for disc tracks. If the program was an EMA program, it will be necessary to call $DREL twice; once for the program swap tracks, and once for the EMA swap tracks.

Second, $ABRE in the $EXECUTIVE is called to return any reentrant memory the program may have. This may happen if a program terminates or is aborted while in a reentrant subroutine. If $ABRE returns any memory programs waiting for memory will be reshceduled.

Third, the DISPATCHER calls $WATR in the SCHEDULER to reschedule any programs which were waiting to schedule (EXEC 23,24) this program.

Fourth, the DISPATCHER calls $TRRN which calls $ULLU to unlock any LUs which may have been locked by the program. $TRRN also unlocks any locally locked RNs and deallocates any locally allocated RNs the program may have. Each of these processes may call $SCD3 to reschedule any programs waiting for these resources.

Fifth and last, if the program is a disc resident program and it is still resident in the partition, the partition is linked into the free list.

TECHNICAL SPECIFICATIONS FOR

RTE-IV RTIOC

EJW
1/12/78
Project #1106

1.0 GENERAL OVERVIEW OF OPERATION

The RTIOC module controls all aspects of the system's input and output operations. It serves as the centralized I/O interrupt handler which identifies the source of an interrupt and turns control over to the appropriate processor. All I/O requests are made to this module either by EXEC calls from user programs or by $XSIO calls from the other parts of the operating system. The I/O requests are passed directly to the appropriate device driver if the driver is available. The I/O module also queues I/O requests for busy drivers or for buffered requests. All the necessary mapping and base page communication area words needed to perform I/O will be done by RTIOC. Upon completion of I/O, the next request (if any) is started and control is returned to any waiting programs. RTIOC also detects and reports errors at various phases of the process and performs any necessary clean-up.

2.0 EXTERNAL COMMUNICATION

RTIOC communicates to the rest of the operating system through table structures, the system's base page communication area, and by subroutines available only to the system modules.

2.1 Tables Used by RTIOC

2.1.1 Equipment Table (Figure 1,2)

Each I/O controller and device controlled by the IOC/driver relationship is defined by static and dynamic information in the Equipment Table. Each EQT entry is composed of 15 words. If there is an EQT extension, the address of the extension is in EQT Word 13 and the size of the extension is in EQT Word 12. This table is built by the Generator.

2.1.2 Interrupt Table (Figure 3)

A table, ordered by hardware interrupt priority, designates the associated software processor and the procedure for initiating the processor. This table is constructed by the System Generator on information supplied by the user in configuring the system. The table consists of one entry per interrupt source--each entry contains only one word. The contents of each valid entry is the identifier of the processor. System processors are noted by positive values, user processors by negative values, and a zero denotes an unused entry.

2.1.3 Device Reference Table (Figures 4,5)

The Device Reference Table provides the user for logical addressing of physical units defined in the Equipment Table. "DRT" consists of two-word entries corresponding to the range of user-specified "logical units," 1 to n where n is less than or equal to 63 (decimal). All word 1's are in one table followed immediately by a second table containing all DRT word 2's. The contents of DRT word 1 for a given logical unit is the relative position of the EQT entry defining the assigned physical unit, in bits 0-5, and the subchannel of the EQT entry to be referenced by this logical unit number, in bits 11-15. The LU lock flag (the resource number being used for the lock) is in bits 6-10. An unassigned unit contains entry value of zero.

DRT word 2 contains a flag (bit 15) indicating whether a device (lu) is up or down (0/1). If a device is down, then all I/O associated with the device is stacked on the major LU (first LU for this device in the DRT) in bits 0-14 of DRT word 2. If the downed LU is not the major LU, then bits 0-14 of DRT word 2 will contain the LU number of the device's major LU.

Certain logical unit numbers are permanently assigned to facilitate system, user and system support I/O operations. These are:

0 - Bit Bucket
1 - System Teletypewriter
2 - System Disc
3 - Auxiliary Disc
4 - "Standard" Output Unit
5 - "Standard" Input Unit
6 - "Standard" List Unit
7
.
. Assigned by user
.
63

2.1.4 Track Assignment Table (Figure 6)

The TAT is a table describing the availability of each track on the System Disc and Auxiliary Disc (if included in the configuration). The TAT is ordered by track number and consists of a one-word entry per track. The value of an entry defines its availability.

0 - Available for assignment to user or system
100000 - Assigned to system (or not available)
077777 - Assigned globally
077776 - Assigned to file management package
<ID Segment Address> - The ID Segment address of the assigned user program.

2.1.5 LU Switch Table

RTIOC will scan a two entry table for each call made for I/O with the BATCH flag set. If the request LU does not refer to a disc and is found, the LU will be switched to the table defined LU. This table will give an LU to LU transform for BATCH programs only. The table format is:

```
       ENT $LULU
$LULU  DEC -N
       REP N
       OCT -1
```

Each active word is set with the address LU in the low 8-bits and the LU to be used in its place in the high 8- bits. There may be up to N entries in any order. This table will be generated by the Generator and maintained by the BASCH monitor.

2.1.6 Driver Partition Map Table (Figure 7)

Each EQT will have associated with it, a two-word driver map table entry which indicates whether the driver for that EQT is in the System Driver Area (SDA) or is in a driver partition and whether the driver (if it is in SDA) does its own mapping or not. If the driver is in a partition, the entry contains the starting page number of the partition. This page number is put into the appropriate system map or user map registers to map in the driver.

The second word of each entry is used when I/O is started on the corresponding driver. The sign bit of the second word indicates whether or not, I/O is being done for a memory resident program. The word is zero for system I/O. The low 10 bits contain the page number of the user's physical base page if it is a partition resident program. This word is used to save time on setting up the proper map on processing interrupts.

2.2 RTIOC Entry Points

$BITB Value - non zero if requests are queued on bit bucket
$BLLO Value - low buffer limit
$BLUP Value - high buffer limit
$CICO Entry point - jumps here from $CIC for interrupt processing
$CKLO Subroutine - check if below the buffer limit on the current EQT.

Calling Sequence:

JSB $CKLO

$CON1 Entry point - driver completion return
$CON2 Entry point - driver continuation return
The code to enter the driver's continuator section must be in all maps since drivers return via the address resulting from a subroutine call. RTIOC will do a JMP $UCON when ready to enter a driver's continuator section under the user map.
$CVEQ Subroutine - converts an EQT entry number into the actual EQT address and calls $ETEQ to set up the base page EQT pointers.
$CXC Entry point - jumps here from $YCIC to skip the CLF and LIA in RTIOC at $CIC.
$DEVT Entry point - jumps here from the system clock routine when a device times out.
$DLAY Subroutine - used to set up a timeout to delay initiation of an I/O operation on a timed-out EQT.

Calling Sequence:

```
LDA EQT1
JSB $ELAY
```

$DMEQ Value-address of the dummy EQT used for bit bucket operations.

$DMS Value-Dynamic Mapping System status is saved here when $CIC is entered. It is used by $PEFR.

$DRVM Subroutine - Sets up the map registers for entry to a driver after the base page EQT pointers are already built in base page. See Section 3.3.5 for details of $DRVM. Called by RTIOC and DVP43.

Calling Sequence:

```
(EQT1-EQT15 already set up)
   JSB $DRVM
   <returns> A-reg and B-reg same as on entry
    E=0 need to enter driver in System Map
    E=1 need to enter driver in User Map
```

$ETEQ Subroutine-sets up the base page EQT pointers EQT1-EQT15

$IODN Entry point - to set an LU or EQT down. This processes any operator DN request (from the scheduler's message processor) that a device (LU) or an I/O slot (EQT) be down. It first determines if an LU or EQT is being set down. If an EQT is being set down, it checks the validity of the EQT via the <IODNS> subroutine. It also determines if the EQT (I/O slot) to be set down is associated with the system console. If either error condition exists, it prints an "INPUT ERROR" message and returns to the Scheduler module's message processor. Otherwise it uses <XUPIO> to set all downed LU's on this EQT into the "up" state and uses <$UNLK> to down the EQT by setting its availability indicator (bits 14-15 of EQT 4) to 01. After suspending in the general wait list any programs queued making unbuffered I/O requests, it returns to the Dispatcher.

If an LU (device) is being set down, it first checks the validity of the LU and whether the user is trying to down LU 1 or an LU pointing to the bit bucket. If one of these error conditions exists, the message "INPUT ERROR" is issued and return is made to the Scheduler module's message processor. If the LU's EQT is down, then the LU is simply marked down (set bit 15 DRT word 2). If the EQT is up, then set the LU and all other LU's associated with the device down and relink any I/O on the device's major - LU (first LU for the device in the DRT). Return is made to the Dispatcher.

$IORQ Entrypoint - All EXEC calls for I/O related requests are processed here.

$IOUP Entrypoint - To make an EQT available again. This processes any operator request (from the Scheduler's message processor) to set a device "UP". It first checks the validity of the EQT number of the device to be set up via the <IODNS> subroutine. If valid, it next schedules all programs waiting on a downed EQT or LU. Next it uses <XUPIO> to set up all LU's associated with this EQT. <XUPIO> will use <$XXUP> to relink on to the EQT any I/O found on a downed LU. If the EQT was down or available, the "up" processor will reset the EQT "up" and return to a point in <IOCOM> to start the next request. Otherwise, return is made to the Dispatcher.

$IRT Entrypoint - Common exit point from the system back to the user. $IRT is a routine used in exiting from the system back to the user program. It does nothing except clear the memory protect flag in a non-privileged system and restores the registers. In a privileged system, $IRT clears control on the privileged interrupt card so that when the interrupt system is reenabled for all devices, all devices can interrupt. The exit from $IRT back to the user program is a "user map enable and jump" instruction. Note that $IRT always enables the user map. The map is loaded before $IRT gets control.

$SYMG Subroutine, system error message output. This routine provides for the output of system messages and error diagnostics on the system console. The routine maintains a "rotating" buffer area consisting of five 10-word blocks; i.e., the maximum length of a message is 18 characters (9 words) plus 1 word preceding the message which contains the character count.

$UNLK Subroutines, used to unlink I/O requests from the current EQT I/O request queue. This is called when an LU is set down and all of the I/O for that LU is moved to the LU's down queue.

$UP Entrypoint - jumped to by $UPIO from Table Area 1 via SJP This entry is used by drivers to automatically "UP" the EQT and is essentially the same code as $IOUP.

$$DIO Subroutine - called by the operating system modules to perform I/O.

$AXUP Subroutine - takes an I/O queue and positions the I/O requests (by calling the LINK subroutine) in the current EQT queue according to their priority. It returns a flag if an I/O operation should be initiated.

```
        Calling sequence:
A-reg is EQT address of old device
B-reg is address of first stacked I/O requests to
   be linked onto the current EQT
JSB $AXUP
B-reg is 0 on return
A-reg is the address of the head of the current queue
   with an I/O operation to be initiated.
```

2.5 Base Page Communication

XI
EQTA
EQT#
DRT
LUMAX
INTBA
INTLG
TAT
KEYWD
EQT1-EQT15
CHAN
TBG
SYSTY
RQCNT
RQRTN
RQP1-RQP7
XEQT etc
OPATN
DUMMY
TATLG
TATSD
SECT2
SECT3
LGOTK
LGOC
MPTFL

These are all located in the System Communication Area of Base Page.

If the user map is needed, entry to the driver's continuator section is entered by a JMP $UCON. If the system map is used, the simple JSB E,I will be done.

3.2 I/O Requests

All input/output operations are performed concurrently with program computation in the overall system. A program is I/O suspended until the transmission or operation is completed unless automatic buffering (output only) was specified for the device or the request was a class I/O request. In these two cases the buffer is moved to system available memory and the user program is not suspended.

If a program is I/O suspended with the buffer in the user program area the program is not swappable. If the buffer is in common or system available memory, the program is swappable. A user may call REIO to move his buffer to system available memory and make the I/O call.

The user program making the request is scheduled immediately if return code 4 is used by a driver. The 5 return is made by a driver if it needs DMA to do the current request but the DMA bit is not set in the EQT.

3.2.1 User I/O Requests

All user I/O requests are channeled to $IORQ after initial request processing by "EXEC". $IORQ performs validity checks on the request parameters and sets the addresses of the referenced EQT entry. (Error conditions and diagnostics are described later.) The buffer address and length is examined for legality for input requests to insure that protected memory is not altered during the transfer. The last page of I/O buffers in the User Map are checked for read/write protect status to insure valid memory accesses.

Disc I/O Requests

If the referenced I/O device is a disc unit, the request is checked to insure that parameters are supplied. If the disc LU number is either 2 or 3, the request is additionally checked to insure that the disc track and sector numbers are legal and that the transfer does not exceed a track boundary. If the request is output, a referenced track on LU 2 or 3 must belong to the user (i.e., the TAT entry address must equal the ID Segment Address of the user) unless the track is a load-and-go track or a global track.

Requests Involving Buffered Output and Class I/O

If a Write or Control request references a device for which the user designated automatic buffering, a block equal to the buffer size, plus control information (5 words), is requested for allocation in the system available memory area. (Call to $ALC.) A Class I/O request is also moved into system available memory.

If the block cannot be allocated, the user program is suspended and linked into the memory suspension list. (The memory processor ($RTN) will cause the user program to be scheduled as soon as a block is released.)

After a block is allocated, the control information (CONWD and buffer length), priority and buffer (if a Write request) are moved into the block. The first word of the block is used for linking into the device list. (See Section 3.3.1.)

Normal User Requests and REIO Requests

The parameters of a user request (which is not buffered as above) are moved into the 5-word temporary area in the ID Segment of the program. Word 1 of the ID Segment is used for linking into the device list. (See Section 3.3.1.)

The user program is suspended with a suspension code of 2 (I/O suspension). This is also done for an REIO call. The only difference is that the buffer address will point to system available memory instead of the user area. The sign bit of the buffer address in the temporary words of the ID segment is set if the buffer is moved as the result of an $REIO call. This is to tell the system that the driver must process that I/O request under the system map.

Error Conditions and Diagnostics

Detection of the following error conditions causes a dignostic identifying the error type, a program name and location of the request to be printed on the system teletype. The program is then aborted ($ABRT in EXEC).

	Code	Meaning
IO	01	Insufficient # of request parameters.
	02	Illegal logical unit #, or less than 5 parameters with A-bit set.
	03	Illegal EQT reference, select code = 0.
	04	User buffer violates system (or Real-Time) boundary.
	05	Illegal disc track or sector # in disc request.
	06	Write reference to protected track.
	07	Driver rejected the request as illegal for the device (unbuffered requests only).
	08	Disc transfer exceeds track boundary.
	09	Overflow of load-and-go area.
	10	Class GET occurred and one GET call outstanding on this class.
	11	Illegal user map request in System Driver Area.

[illegible] System I/O Request Processor <$XSIO>

A privileged entry is provided at $XSIO to allow modules of Real-Time Executive to call for I/O operations without incurring the overhead and procedures involved with user I/O requests. No error checking is performed, the request is linked into the appropriate I/O list at a priority of zero (highest priority) except that disc request may specify a priority, and control is immediately returned to the first word following the request.

Request Format: A system I/O request differs from the user I/O request in format and power. Specifically, a system disc call can specify a series of transfers to be performed before the next operation is initiated.

A completion address can be specified for operation of an open subroutine at the end of the operation. This facility is only available to system routines and is useful for resetting flags, etc., because an I/O operation is always buffered to the system. A zero completion address indicates absence of a completion routine.

```
Word
            EXT    $XSIO
             .
             .
             .
 1          JSB    $XSIO
 2          OCT    <logical unit number>
 3          DEF    <completion routine address> or 0
 4          NOP    <list pointer word-set by "LINK">
 5          OCT    <control information/request code>
 6          DEF    buffer address, or location of disc control>
 7          DEC    buffer length or <disc request priority>
 8          OCT    map word
```

Word 5 is in the same field format as the control word in a user request except that the request code replaces the logical unit.

Word 8 is set to zero if the request is to be processed under the system map. If the user map is required, the word 8 must contain the ID segment address of the program to be described. Word 8 is 100000 (octal) if the request is to be processed under the User Map as it is currently-without change. Word 8 is set to an ID segment address with the sign bit set if a modified user map is used (e.g., when the Dispatcher is swapping a portion of LBA).

Also, the $XSIO call uses the same routine, DRIVR, to set up for and then enter the driver.

Disc Version of Request: Word 6 points to an array containing "n" sets of triplets designating the buffer control, for each transfer. The array of triplets is open-ended and terminated by a zero word:

```
Word
-----
 1          DEF    <buffer address>
 2          DEC    <buffer length>
 3          OCT    <track/sector#>>
 .
 .
 .
 n          OCT 0
```

Word 7 in this case is used as the priority of the request.

3.3 Initiation of I/O Requests

3.3.1 I/O Requests

The control word is set up as follows:

```
|       |            |                        |        |       |
|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
|<T>   |<SUB 2> |<CONTROL INFO>|      <SUB 1> <RQ >
|       |            |  | DR |            |        |       |
|       |            |                        |        |       |
```

RQ - User request (1=READ, 2=WRITE, 3=CONTROL)
SUB1 - High 2 bits of subchannel
SUB2 - Low 3 bits of subchannel
DR - Set only for a disc request. Indicates if disc is system auxiliary, or peripheral

T - Request type identifier
- 00 = User (Normal Operation)
- 01 = User (Automatic Buffering)
- 10 = System
- 11 = Class I/O

After the necessary legality checks are made, the request is linked into the queue for the referenced I/O device. If the request is a normal user request, the parameters are set in the temporary storage area of the ID segment. If the request is class I/O or the device has automatic buffering (output or control only), the request parameters are moved into system available memory.

I/O requests are linked in a list for each device according to priority. The requests are user (normal), user (automatic output buffering), class I/O, or system. Identification of the request type is the code in bits 15-14 of the control word in each request format. This field, the "T" field, identifies the request as:

- 00 User (normal operation)
- 01 User (automatic buffering)
- 10 System
- 11 Class I/O

1) User (normal operation)

The parameters from the request are stored in the temporary area of the program ID Segment. The link word of the segment is used to contain the linkage for the I/O list.

Word	Contents
1	linkage word
2	T,control information,request code
3	buffer address or control parameter
4	buffer length
5	disc track # or optional parameter or zero
6	disc starting sector # or optional parameter or zero
7	program priority
.	remainder
.	of
.	ID Segment

2) User (Automatic Output Buffering)

Requests of this type are constructed in the system available memory area.

Word	Contents
1	linkage word
2	T,control information,request code
3	priority of requesting program
4	total block length in words
5	user buffer length
6	word 1 of user buffer
.	
.	
.	
n+5	word n of user buffer

If the device or controller is "down" or "busy", no action is taken and return is made to the caller. If a DMA channel is required but no channel is available, the "AV" field in the EQT is set to 3 (waiting for DMA), one is added to "DMACF" for the number of devices waiting for DMA, and return is made to the caller.

If the device is available (and a DMA channel is assigned if required), the device time-out clock is set from the clock reset value (in EQT14), the subchannel is set into EQT4 (from EQT6), and preparations are made for calling the driver. DRVMP is called to do the driver map set up (see Section 3.3.4). If a DMA channel is being used, DRIVR also sets up the DMA map by copying either the System Map or the User Map into the correct port map. Note that the DMA map does not need to be reloaded until it is reallocated. If the driver needs the user map, that map must be reloaded before each entry into the driver.

3.3.4 DRVMP

This code is called to set up and enable the necessary maps for the driver. The preparation for setting up the maps for the driver call includes checking word 1 and setting up word 2 of the Driver Mapping Table (Figure 7). The first word in the table is found by using the EQT number to index from $DVMP. The second word is found by adding EQT# to the address of the entry's first word.

If the driver is in the System Driver Area and does its own mapping, the driver is always entered under the System Map. NOTE: $XSIO calls must have zero in the eighth parameter, no checks are made.

If the driver is in the System Driver Area but does not do its own mapping, the T-field of the request is examined. If the T-field is zero (normal user) the program must be privileged (type 3) in order to use the unmodified user map. The second word of the driver's Mapping Table entry is set to the physical page number of the program's base page (first page of partition). Any other types of I/O requests or types of programs requesting I/O for a driver which doesn't do mapping and is in the System Driver Area will cause the request to be rejected (a program will be aborted with an IO11 error message). NOTE $XSIO calls may not be used to call a driver in the System Driver Area if the driver does not do its own mapping! The IO11 error message will also be issued in this case, the request will be rejected and returns control to the XS10 caller.

If the driver is in a driver partition, the T-field is checked. If the T-field is 2 (system request), the eighth word is checked to see if it is either a disc program load request or a special disc I/O request. If the eighth word of the $XSIO request is 100000 (octal), it is a special request which specifies that the current User Map is used. The special request is used by the DISPATCHER and by the reconfigurator. If the eighth word is a positive value, it is the ID Segment address of a program to be loaded. The program's map is built by $SMAP. If the eighth word is negative, it is an ID segment address with the sign bit set. This form is used by the Dispatcher for swapping channels of EMA. A special user map is kept in the user's protected portion of base page just below the normal copy of the user's map. The physical page number of the base page is set into the second word of the Driver Mapping Table entry is set to zero.

All buffered user request and class I/O requests use the modified system map. If the T-field indicates that it is an unbuffered user request, the second word of the Driver Mapping Table (see Figure 7) is checked to see if the request was made by a Memory Resident program. If it was, the MR bit of the second word of the Driver Mapping Table Entry is set and the modified Memory Resident map is used. If it was not a Memory Resident program request, the first page number of the program's partition is entered into the second word of the Driver Mapping Table entry and the modified User Map is used.

When a driver is in a driver partition, the map under which the driver is entered must be changed to address the physical pages of the partition. The modified map is saved if it was a user map (other than when I/O is being done by a Memory Resident program) that was modified. The purpose of this is to save set up time on each continuation interrupt. The page number in the second word of the Driver Mapping Table entry is loaded into the System Map's driver partition register ($DVPT) to map in the user's physical base page (see Figure 8). The copy of the modified User's Map is then stored in the top portion of the physical base page via a cross-map store through the driver partition register in the system map.

3.3.5 $DRVM Subroutine

When a driver needs to be called as a result of an interrupt (continuation) $DRVM is called to check the Driver Mapping Table entry. The first word of the entry determines whether or not a driver partition's pages need to be set up in a map. The second word indicates which map to use.

If the second word is zero, the System Map is used. If the driver is in a partition (Word 1 has the partition's starting page number) the System Map is modified to address this partition. If the driver is in the System Driver Area, no modification is necessary. $DRVM returns with an indication that the System Map is to be used (E=0).

If the second word has the sign bit set the Memory Resident Map is used. The current user map is saved in a buffer (SVUSR). Then the User Map is set up with the Memory Resident Map and the driver partition registers are set up according to word 1 of the mapping table. $DRVM returns with an indication that the User Map is needed (E=1).

If the second word has a page number, the necessary user map is already set up and is stored in the last 32 words of the indicated page. $DRVM saves the current user map in SVUSR. Regardless of the driver's area of residence, the User Map is specified (E=1).

Note that the user map is saved and set up to the required map only if it is not already mapped in the user map. This saves time in setting up duplicate maps.

3.3.6 $RSM Subroutine

This routine is called on every return from a driver. It checks the flag DVMPS to see if the user map was changed. If it was, RSTUS reloads the user map with its original contents (saved by DRVMP or $DRVM) and clears flag DVMPS.

3.3.7 I/O Driver Initiation Return

Upon return from the driver $RSM is called to restore the user map (if it was changed) to its status prior to driver entry. The driver returns a code to DRIVR indicating whether the operation was accepted or rejected and the cause of the reject. This code is in A on return:

0 - operation successfully initiated
1 - Read or Write request illegal for device
2 - Control request illegal or not defined
3 - equipment malfunction or not ready
4 - operation successful-immediate completion
5 - driver requires a DMA channel for this operation
6-59 - Reserved for HP RTE System modules and system drivers
60-99 - Reserved for user drivers.

If the code is 5 a DMA allocation is attempted and if successful, the driver is reentered witht he request.

If the operation was otherwise rejected, DRIVR returns to P+2 of the call with the reject code in A.

If the code in A is 3, the device was found to be unavailable for I/O (not ready). The device availability indicator is set to 01. If a DMA channel was allocated, it is released. The "NR" diagnostic is printed and IOCM is exited either back to $XEQ in the Scheduler or to the completion routine specified in a system request. If the code in A is 1, 2, 4, 6 or greater, control is transferred to subroutine <ILLCD>. If a zero is returned, I/O was initiated successfully, with subsequent device interrupt expected, and control is transferred to $XEQ in the Scheduler module to switch to the next lower priority which requires execution time.

3.3.8 DMA Channel Allocation

The two DMA channels are dynamically allocated to the high-speed and synchronous devices identified to RTIOC (bit 15-1 of word 4 in the EQT entry). The assignment process consists of setting the EQT address of the device in the DMA channel entry in the Interrupt Table and setting the channel number in the word "CHAN" in the Communications Area.

A driver with its EQT DMA bit not set may also request a DMA channel by setting A=5 and returning to the system at initialization of the I/O request.

If more than one device is waiting for a channel, the order of priority for assignment is the order of the positions in the Equipment Table. There are two exceptions to this scheme:

1) If the first entry in the EQT is waiting for a DMA, the channel is assigned to that device, which is assumed to be the system disc.

2) If the first entry encountered (other than entry #1) just released a DMA channel, then the next lower priority device waiting for DMA is used. This allows for a "switching" operation in the allocation of a DMA channel.

Special processing is required by any I/O driver which uses the interrupt on a DMA channel to perform data transmission with the device. A software flag must be set after a DMA channel is initiated to indicate that the channel is active and that a completion interrupt is expected. The setting of this flag is to set Bit 15=1 in the Interrupt Table word corresponding to the DMA channel:

INTBL(1) - channel #1 (location 6)
INTBL(2) - channel #2 (location 7)

The address of INTBL is contained in the word "INTBA" in the Base Page Communication Area. When Bit 15 is set, the rest of the word must not be altered. This operation must be done only if DUMMY is non-zero. When a system has privileged drivers, i.e., DUMMY = 0, control is cleared on both DMA channels everytime an interrupt is processed through CIC - in order to let the privileged interrupts be the only ones "on". Thus if a driver needs that DMA interrupt, it must set bit 15 in the appropriate word. $IRQ checks these words and if the bit is set, it reenables the DMA interrupt.

3.4 Completion of I/O

The return point by an I/O driver (from a call by CIC) indicates the continuation or completion of the I/O operation. In RTE IV, the user map is restored if it was modified for driver entry. RSTUS is the routine called to do this.

1. Return at (P+1): Completion of the operation. CIC transfers directly to the IOC completion section at "IOCOM."

2. Return at (P+2): Continuation of the Operation. CIC restores all registers and returns to the point of interruption, with the exception of special processing which must be done for operator attention: If the flag is Base Page Communication Area "OPATN" is set = 0, control is transferred to $TYPE in SCHED: "OPATN" is set = 0.

3.4.1 IOCOM

This section is responsible for the initiation of stacked I/O operations, placing a program back in the scheduled state when its I/O operation is completed, dynamic allocation of the two DMA channels among synchronous devices, and calling for operator notification of equipment errors or malfunction.

<IOCOM> is entered directly from <CIC> when an IO operation is terminated and all error recovery procedures have been attempted. On entry to this section, (B) contains the number of words (or characters) transferred. ((B)=track # on which error occurred if disc.)

The addresses of the Equipment Table entry are in EQT1 to EQT15 in the Communication Area in Base Page from the CIC pre-processing. The device time-out clock is cleared.

After completing the processing for the completed successful operations, IOCOM checks a stacked request for the device. If none, IOCM transfers to "IOCX." The user program for the completed operation has already been rescheduled.

If a request is stacked, the subroutine DRIVR is called to initiate the operation.

The IOCOM exit section "IOCX" transfers control to:

1) <Completion Routine> if the system I/O request specified.

2) L.136 if the bit bucket has I/O stacked on it which must be completed.

3) $TYPE (in SCHED) if the operator attention flag is set (the flag is also cleared by "IOCX").

3.4.3 ILLCL Subroutine

This subroutine is entered primarily if an illegal request is detected by an I/O driver. The reason is a Read or Write operation is illegal for the device or a control request is meaningless for the device. An additional reason for transfer to this section is an "Immediate Completion" (Code 4) return from the driver; it is processed as a control reject.

Additional error messages may be defined by HP system or user drivers as follows:

6-59	Reserved for HP RTE system modules and system drivers.
20-29	Reserved for the Spool Monitor (see Spool IMA).
60-99	Reserved for user drivers.

Error procedure is:

1. If the request is processed as buffered output, the temporary block is released to available memory.
2. The reject is ignored if a system program generated the request --however, a completion routine, if specified in the request, is operated. (NOTE: this philosophy is based on the assumption that this condition should never occur.)
3. A user control request (A=2 or 4, refer to pg. 22) which is rejected is treated as if it was performed. The program is linked back into the schedule list.
4. A unbuffered user read or write request reject (A=1) causes a diagnostic to be issued ("IO 07") and the program aborted.
5. Other reject codes (A>5) for unbuffered user read or write requests will be mapped into an IOxx error message where "xx" is the error code and the program will be aborted.

3.5 Miscellaneous Routines

<$IOCL> Subroutine

The function of this routine is to remove a program from an I/O hang-up condition resulting from an input request not being completed by the device.

This "clearing" procedure is initiated by the operator in using the I/O Abort version of the "OF,XXXXX,1" command. The "OF" statement processor in "SCHED" calls this section if the referenced program is suspended for an I/O input request.

The list of each EQT entry and down DRT entry is searched to find the queued request corresponding to the ID Segment of the referenced program. The entry is removed from the list and the list is appropriately linked to reflect the change. If the entry was the first one in an EQT list (i.e., an active request) and the EQT is not down or in DMA wait then a clear request (100003B) is forced into the initiator. This can be done only after the Driver Mapping Table entry is checked and the driver is mapped in if needed. If the request is accepted then an interrupt is expected and the device set busy with an arbitrary timeout of 1 second. The sign bit of EQT word 1 is set indicating device clearing. The timeout will be trapped in <$DEVT> and routed to <IOCOM>. <IOCOM> recognizes special interrupts on timeouts associated with device clearing by checking the sign bit of EQT word 1. If the request is not accepted, then the timeout is cleared and control is given to <IOCOM> for initiating the next request.

<IODNS> Subroutine

This subroutine checks the legality of an EQT number. If it is valid, it returns to the caller; otherwise, it sets up to print out the diagnostic "INPUT ERROR" and goes to the Scheduler module's message processor.

3.6 Class I/O Requests

Class I/O refers to no-wait I/O in which the user directs the completion information to a "class" by number. The user requests I/O on a class. The RTIOC requests buffer memory for the request, moves the request to the buffer memory, queues the request on the specified EQT, and enter in the class queue that a request is pending. On completion, the completed request is queued in the class queue, and any program waiting for the class is restarted.

The class table is defined at generation time and is located at $CLAS. The table consists of a length word defining the number of classes, followed by one word for each class.

Class I/O "Queue format and its use

The class queue can be in four different states.

```
 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
-------------------------------------------------
| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|
-------------------------------------------------
```

State 1: Class deallocated, available

```
 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
-------------------------------------------------
| 0| ADDRESS OF FIRST ENTRY                      |
-------------------------------------------------
```

State 2: Pointer to first entry in class queue

```
 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
-------------------------------------------------
| 1| 0| X|SECURITY CODE | NUMBER OF PENDING RES |
-------------------------------------------------
```

State 3: Class allocated, no one waiting on class. Number of pending requests counter may be 0-255

```
 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
-------------------------------------------------
| 1| 1 |X| SECURITY CODE| NUMBER OF PENDING REQS|
-------------------------------------------------
```

State 4: Class allocated, someone waiting (suspended). Number of pending requests counter may be 0-255.

Actions to be taken when handling a class I/O or get request depend on the current state of the class queue head.

Get Requests:

State 1. Abort the program IO00, no class
State 2. Return the data from class buffer
State 3. Set the some one waiting bit (bit 14), suspend program
State 4. Abort the Program IO00, only one program may be suspended per class.

Class I/O Requests:

State 1. State 3 is set up, security code is low 5 bits of program ID Number, counter is set to 1.

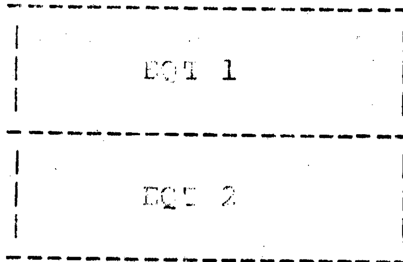
State 2. The counter at end of queue is incremented by 1
State 3. The counter is incremented by 1
State 4. The counter is incremented by 1

On Completion of Class I/O Requests:

State 1. Illegal--should never happen--buffer is returned and the completion is ignored
State 2. The new data is added at the end of the list (FIFO) and the counter is decremented by 1
State 3. The new data is added at the end of the list (FIFO) and the counter is decremented by 1
State 4. The waiting program is scheduled and the counter is decremented by 1 and the someone waiting bit (Bit 14) is cleared.

RTE IV Equipment Table

EQTA



(EQT#*15) words

EQT#=N

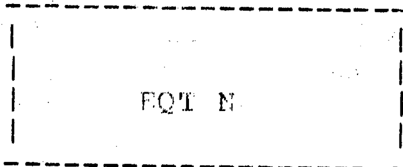


FIGURE 1

RSX-IV EQUIPMENT TABLE ENTRY

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	Driver "Initiation" Section Address
3	Driver "Completion" Section Address
4	D | B | P | S | T | Unit # | Channel #
5	AV | IO TYPE CODE | STATUS
6	CONWD (Current I/O Request Word)
7	Request Buffer Address
8	Request Buffer Length
9	Temporary Storage for Optional Parameter
10	Temporary Storage for Optional Parameter
11	Temporary Storage for Driver
12	Temporary Storage for Driver
13	Temporary Storage for Driver
14	Device Time-Out Reset Value
15	Device Time-Out Clock

FIGURE 2

FIGURE 2 con't.

Where:

```
     D =     1 if DMA required
     B =     1 if automatic output buffering used
     P =     1 if driver is to process power fail
     S =     1 if driver is to process time-out
     T =     1 if device timed out (system sets to zero before each
                    I/O request)
     Unit = Last sub-channel addressed
     Channel= I/O select code for the I/O controller (lower number if
                a multi-board interface)
     AV =    I/O controller availability indicator:
             0=      available for use
             1=      disabled (down)
             2=      busy(currently in operation)
             3=      waiting for an available DMA channel

STATUS =     the actual physical status or simulated status at the end
             of each operation.  For paper tape devices, two status
             conditions are simulated:  Bit 5 = 1 means end-of-tape
             on input, or tape supply low on output.

EQ     =     type of device.  When this octal number is
TYPE         linked with "DVCx," it identifies the device's
CODE         software driver routine

CONWD  =     user control word supplied in the I/O EXEC call
             (see Section III).
```

RTE-IV INTERRUPT TABLE

```
INTBA        -----------------
             |SELECT CODE 6   |
             -----------------
             |SELECT CODE 7   |
             -----------------
             |SELECT CODE 10  |
             -----------------
             |SELECT CODE 11  |
             -----------------                INTLG

                    .
                    .
                    .

             -----------------
             |SELECT CODE     |
             |INTLG+4         |
             -----------------
             |SELECT CODE     |
             |INTLG+5         |
             -----------------
```

FIGURE 3

RTE-IV DEVICE REFERENCE TABLE

```
DRT     -----------------------
        |  LU 1               |
        -----------------------
        |  LU 2               |
        -----------------------
        |  LU 3               |
        -----------------------      LUMAX WORDS
                   .                 FIRST WORDS ONLY
                   .
                   .
        -----------------------
        |  LU LUMAX           |
        -----------------------
        |  LU 1               |
        -----------------------
        |  LU 2               |
        -----------------------
                                     LUMAX WORDS
                   .
                   .
                   .                 SECOND WORDS ONLY

        -----------------------
        |  LU LUMAX           |
        -----------------------
```

FIGURE 4

RTE-IV DEVICE REFERENCE TABLE ENTRY

```
DRT WORD 1    -------------------------------------------------------------
              |15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
              -------------------------------------------------------------
              |<SUBCHANNEL>  |<LU LOCK>      | <EQT #>                    |

DRT WORD 2    -------------------------------------------------------------
              |15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
              -------------------------------------------------------------
              |  |       Major LU or Down I/O Queue                        |
              Up/Down
```

FIGURE 5

RTE-IV TRACK ASSIGNMENT TABLE

```
TAT    ---------------
       | TRACK 0     |
       ---------------
       | TRACK 1     |
       ---------------
       | TRACK 2     |          TATSD
       ---------------
              .
              .
              .
       ---------------
       | TRACK n     |                      -TATLG
       ---------------
       | TRACK LU3.0 |
       ---------------
       | TRACK LU3.1 |
       ---------------
              .
              .
              .
       ---------------
       |TRACK LU3.n  |
       ---------------
```

FIGURE 6

RTE-IV DRIVER MAPPING TABLE

```
                          15  14 13 12 11 10| 9 8 7 6 5 4 3 2 1 0
     $DVMP                ----------------------------------------
                          |SE|              |                    |
                  EQT 1   |  |              |         N          |
                          |  |              |                    |
Entry                     ----------------------------------------
First                     |SE|              |                    |
Word              EQT 2   |  |              |         N          |
                          |  |              |                    |
EQT#                      ----------------------------------------
Words                     |SE|              |                    |
                  EQT 3   |  |              |                    |
(static                   |  |              |         N          |
info)                     ----------------------------------------
                    .                           .
                    .                           .
                    .                           .
                          ----------------------------------------
                  EQT  n  |SE|              |                    |
                          |  |              |         N          |
                          |  |              |                    |
                          ----------------------------------------
                  EQT 1   |MR|              |                    |
                          |  |              |         N          |
                          |  |              |                    |
                          ----------------------------------------
                  EQT 2   |MR|              |                    |
                          |  |              |         N          |
                          |  |              |                    |
Entry                     ----------------------------------------
Second                                      .
Word                                        .
                                            .
                          ----------------------------------------
                   EQT n  |MR|              |                    |
EQT #                     |  |              |         N          |
words                     ----------------------------------------

(dynamic
info)
```

FIGURE 7

EXEC & $ALC

Mike Manley
January 19, 1978
Project #1106

TABLE OF CONTENTS

Introduction

System Request Analyzer

Resident Library Execution Control

Privileged & Reentrant Subroutine Processors

Disc Track Allocation & Release Processors

Error Messages

System Available Memory Processor

Appendices

A-Reentrant List Structure
B-Special Entry Points .ZRENT, .ZPRIV

1. Introduction

This part of the technical specs manual deals with the EXEC and system available memory portion of the RTE-IV 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 EXEC modules contains 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. He may code something like:

```
      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 op 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 op system into a module called RTIOC.

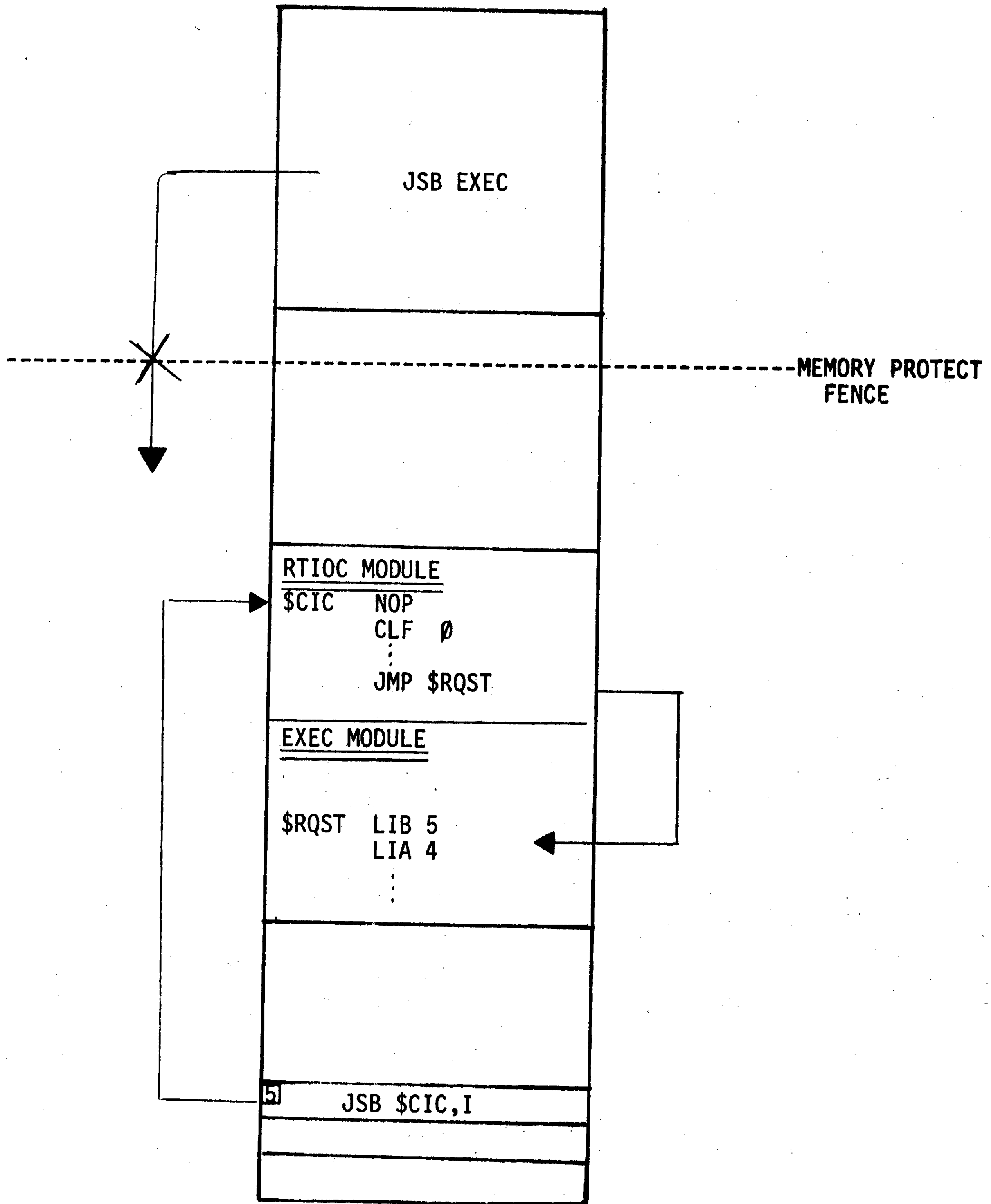
RTIOC is obliged to find out where the interrupt came from and what kind of interrupt it was. By executing a LIA 4 RTIOC will receive the interrupt code # of last interrupt. If the interrupt code corresponds to the Time Base Generator RTIOC jumps to $CLCK in the RTIME module. If the interrupt code is 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 1 shows a graphic representation of a JSB EXEC.

We now know how the system enters the EXEC.

FIGURE 1



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.

(1) 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, re-entrant, 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 the violation 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

The TDB (Temporary Data Block) and return adjustment is only for re-entrant format. The return adjustment for re-entrant 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 priviledged subroutine.

Re-entrant programs may call other re-entrant and privileged programs. However, privileged programs may only call privileged programs.

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

PRIVILEGED & REENTRANT PROCESSING
Priveleged or reentrant processing starts whenever the initial memory protect or DMA vilation for that service is detected. This can happen in two ways.

Consider the two cases below:

```
    CASE 1      ANY PROGRAM
    .
    .
    .
    JSB     SUB
    .
    .
    .                                all in the same program
    .
    .


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 re-entrant subroutine. The 1st word of the TDB is checked. If it is zero, then the subroutine is not being reentered.

The 1st word is then set up to point to the 2nd 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 A at this time should help in understanding the discussion below.)

If the link word is non-zero, the subroutine is bieng re-entered (i.e., two memory resident programs want the same subroutine) and $ALC is called by the EXEC MTDB routine to allocate a block in available memory equal to the lengt 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 1st word of the moved TDB in the system map is chnged to point to the 1st word of the original TDB in the user map.

The address of the original program call is set in word 3 of the program TDB as the return address. The re-entrant program must not modify the first thre 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 subrorutines 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 the 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 re-entrant program is executing. In the latter case, EXEC transfers control to $LIBX indirectly after the initial protect violatic processing.

If the executing program is privileged -- i.e., ($PVCN>0) one is subtrancted 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 tht return address is calculated by adding the contents of the 3rd word of the TDB which contains the P+1 of the original JSB $LIB 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.

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 1st dimension is a stack and is one entry per program. The 2nd dimension is for programs that make nested reentrant calls and is a push down stack after the 1st 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 A.

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 LU 2 or LU 3) are not managed through the track assignment table.

Figure 3 shows the structure of the system disc (LU 2). 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 approximate track 20 to say 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, # of tracks).

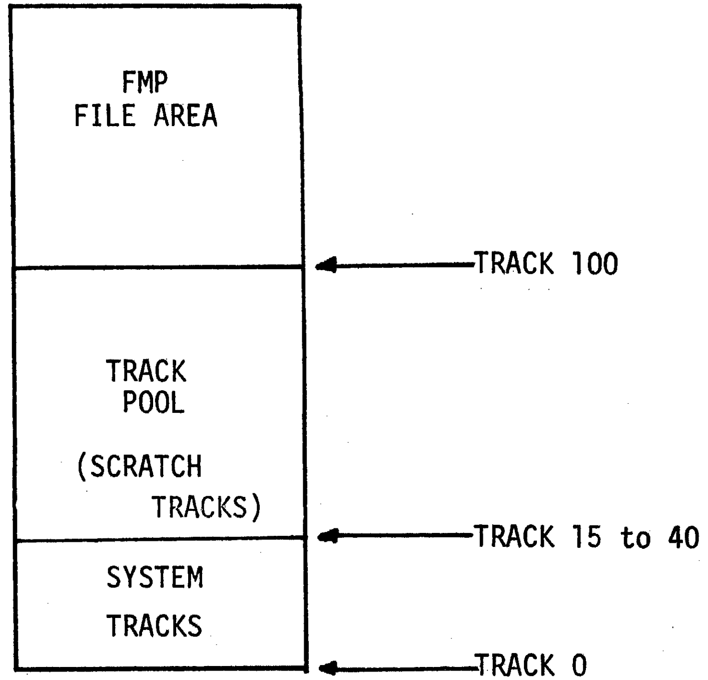
The Track Pool is used by the system for swapping, text editing, loading permanent program additions, etc. There must be a minimum of 8 track pool tracks on LU 2, however, a minimum of 70 track pool tracks is recommended.

If the Extended memory Feature of RTE IV 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.

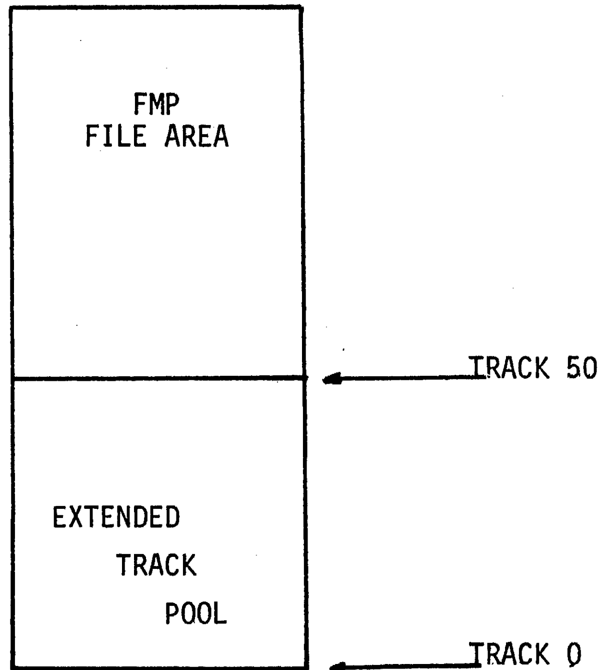
An auxiliary disc (LU 3), Figure 4, can be used with RTE to extend the size of the track pool if desired.

Figure 3



LU 2

FIGURE 4



LU 3

Figure 3

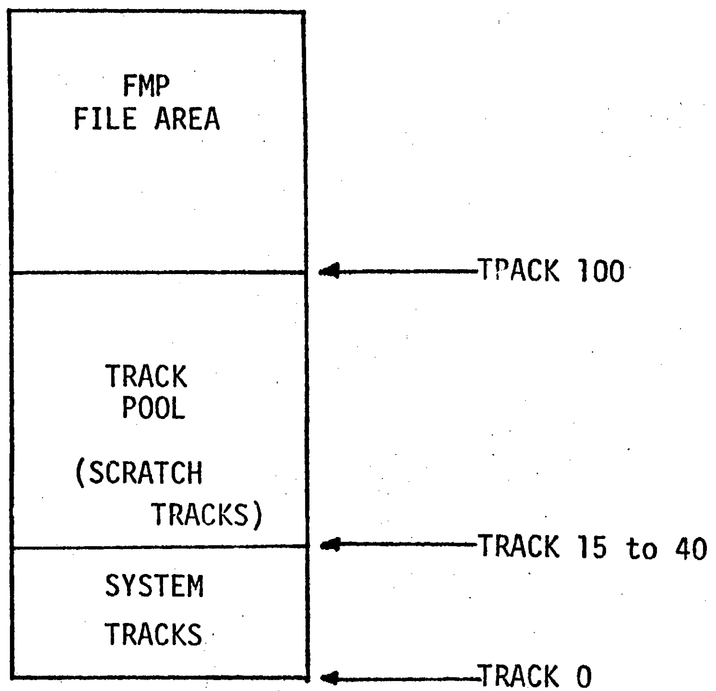
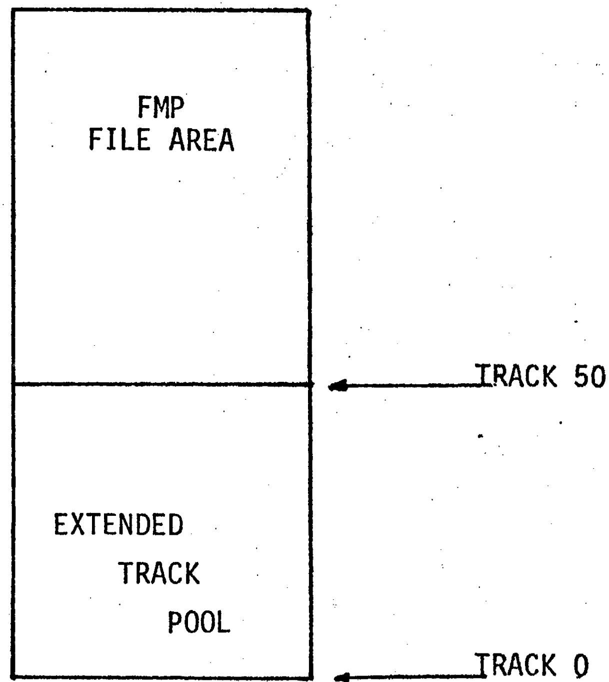


FIGURE 4



Track Assignment Table (TAT)

The TAT is a variable length table describing the availability of each disc track on the system and auxiliary discs. The TAT is constructed by "RT4GN" based on user parameters declaring the size of the system disc and the availability and size of an auxiliary disc. Each track is represented by a one-word entry. The first words of the table correspond to the "n" tracks of the system disc. The word "TATSD" in the Base Page Communication Area contains the size of the system disc as a positive integer. If an auxiliary disc is included, the rest of the TAT contains one-word entries to describe the tracks on that disc.

"RT4GN" initializes the protected tracks of the system disc to be assigned to the system (permanently unavailable).

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

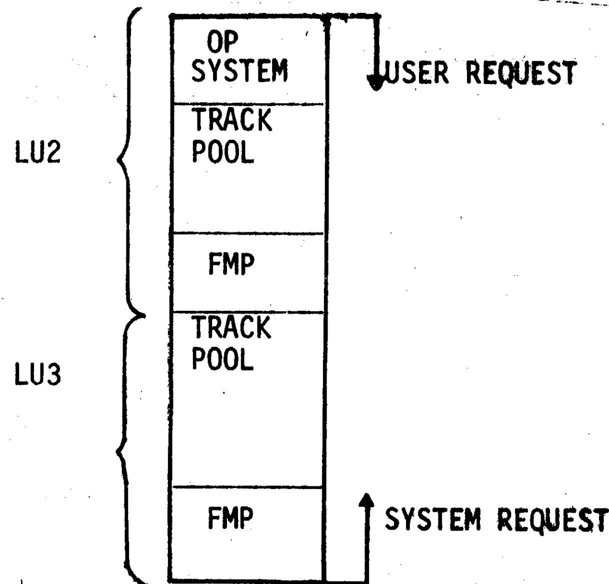
Contents of Track Assignment Table

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 (LU 2)
1760	SECT3	# of Sectors/Track on Aux Disc (LU 3)

Graphically the TAT is searched as shown below:



From the diagram above, the user can see how to optimize system search time for free tracks. No FMP area (or a very small area) on LU 3, 8 tracks of track pool on LU 2 (minimum required) will optimize system search time for system tracks. The user can also improve system performance physically by putting LU 2 and LU 3 on separate physical discs.

LU 2 and LU 3 are both limited to a maximum of 256 tracks.

5) 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:

MEMORY PROTECT

IN RTE 4 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 INTERUPT. THE OPERATING SYSTEM INTERCEPTS THE INTERUPT 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           XXXXX = OFFENDING OCTAL INSTRUCTION CODE
ABE PPPPPP QQQQQQ R        CONTENTS OF A,B & E REGISTERS AT ABORT
XYO PPPPPP QQQQQQ R        CONTENST OF X,Y & O REGISTERS AT ABORT
MP YYYYY   ZZZZZ           YYYYY = PROGRAM NAME, ZZZZZ = VIOLATION ADDRESS
YYYYY ABORTED
```

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 VIOLATION REGISTER
DM INST = XXXXXX
ABE PPPPPP QQQQQQ R
XYO PPPPPP QQQQQQ R
DM YYYYY  ZZZZZ
YYYYY ABORTED
```

EX ERRORS

IT IS POSSIBLE TO EXECUTE IN THE PRIVLEDGED MODE (IE INTERUPT 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 INTERUPT SYSTEM OFF WILL CAUSE THE CALLING PROGRAM TO BE ABORTED AND THE FOLLOWING MESSAGE PRINTED :

```
EX   YYYYY  ZZZZZ
X    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.

UNEXPECTED DM AND MP ERRORS

THE OPERATING SYSTEM HANDLES ALL MP AND DM VIOLAIIONS. CERTAIN OF THESE VIOLAITONS 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 VILATION. 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 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 OCCURED. THIS IS A SERIOUS VIOLATION OF OP SYSTEM INTEGRITY. MOST TIMES IT MEANS USER WRITTEN SOFTWARE (DRIVER, PRIVLEDGED SUBROUTINE) HAS DAMAGED THE OPERATING SYSTEM INTEGRETY OR INADAQUATELY 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 (IE FINISH UP EDITING, ETC) AND REBOOT THE SYSTEM. IF ONLY H-P SYSTEM MODULES ARE PRESENT IN THE OPERATING SYSTEM, CPU FAILURE IS HIGHLY SUSPECTED AND CPU DIAGNOSTICS SHOULD BE RUN.

SYSTEM AVAILABLE MEMORY

Allocation of the System Available Memory for the Reentrant ID List and Moved IDB's

The reentrant processing of reentrant subroutines and automatic buffering to low-speed I/O devices requires the temporary use of blocks of memory. A section of the computer memory must be designated as the System Available Memory for this purpose. The size of this area is designated by the user when the system is generated.

The management of the available memory area is performed by the routines $ALC and $RTN. These routines maintain a chained linkage in a circular fashion of the available blocks in the area.

If a block size requested is not available, $ALC returns a reject indication to the caller. $RTN checks the list of programs suspended waiting for memory each time a block is released. $RTN calls $LIST to schedule all waiting programs (list type #4).

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)
```

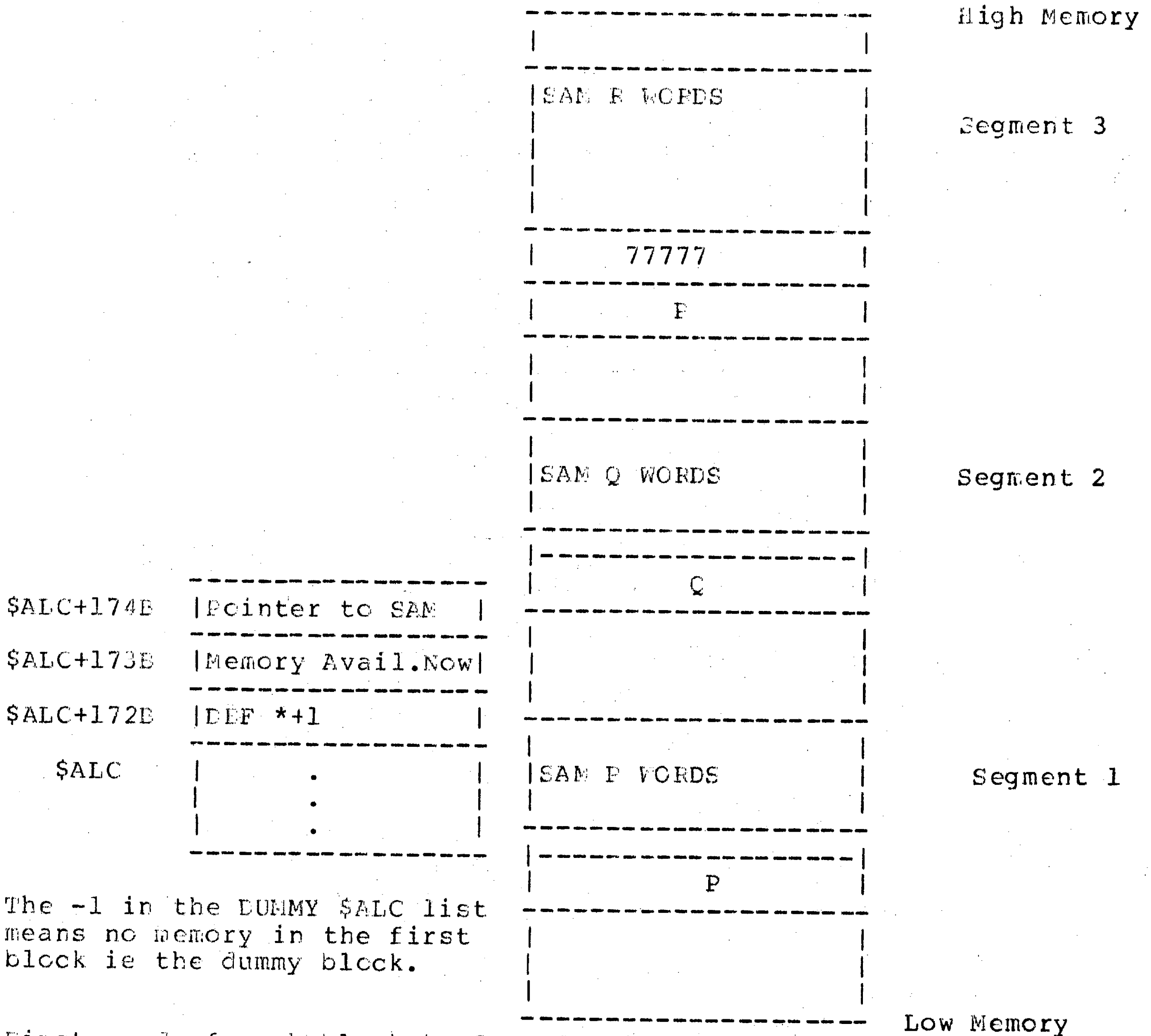
If no block is large enough to allocate the requested length, (A)=0 on return.

```
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.

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. 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 A also shows how this one extra word is carried along if the need arises.

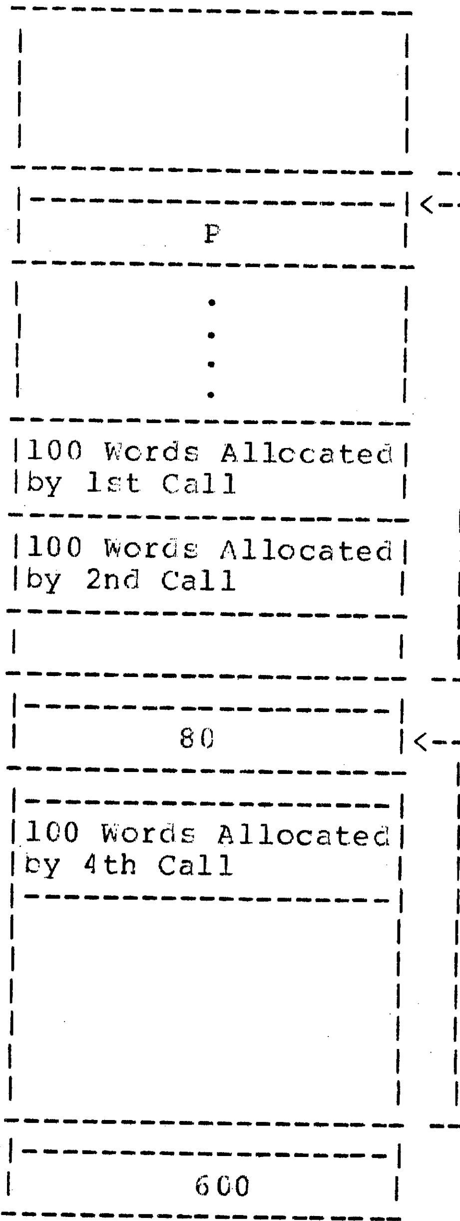
Memory is allocated in contiguous chunks; however, $ALC is written so that SAM need not be contiguous memory. The disconnected blocks of memory are linked through the first two words of each block. A drawing of the linkage for RTL-II is shown below so the reader will understand how the routine will work in the general case.



First word of each block=# of words fin the block (P,Q,R)
Second word of each block=Pointer to 1st word of next block or 77777 if no more blocks.

FIGURE A

Now suppose the user returns C-[illegible] ([illegible] words)
SAM now looks like:



Also;
Returned memory is always concatenated if that memory is found to be contiguous to a free block above or below the returned memory. This insures continual maximum block size and eliminates need for garbage collection.

Asking $ALC for more contiguous memory than is currently available (assuming that that much will ever be available), will force the requesting program into the unavailable memory suspend state (state 4). $ALC performs the necessary $LIST call and places the # of words requested (but unsatisfied) into word 2 of the requesting program's ID segment. Thereafter, everytime memory is returned to the system $RTN checks to see if the suspended program can be given enough memory and rescheduled. Until that program can be rescheduled no more memory is given away to programs of lower priority. $RTN checks only the 1st program in the unavailable memory suspend list. This insures that the highest priority program gets the memory first. (Recall that the unavailable memory suspend list is ordered by program priority.)

APPENDIX A

APPENDIX A

Re-entrant List Structures

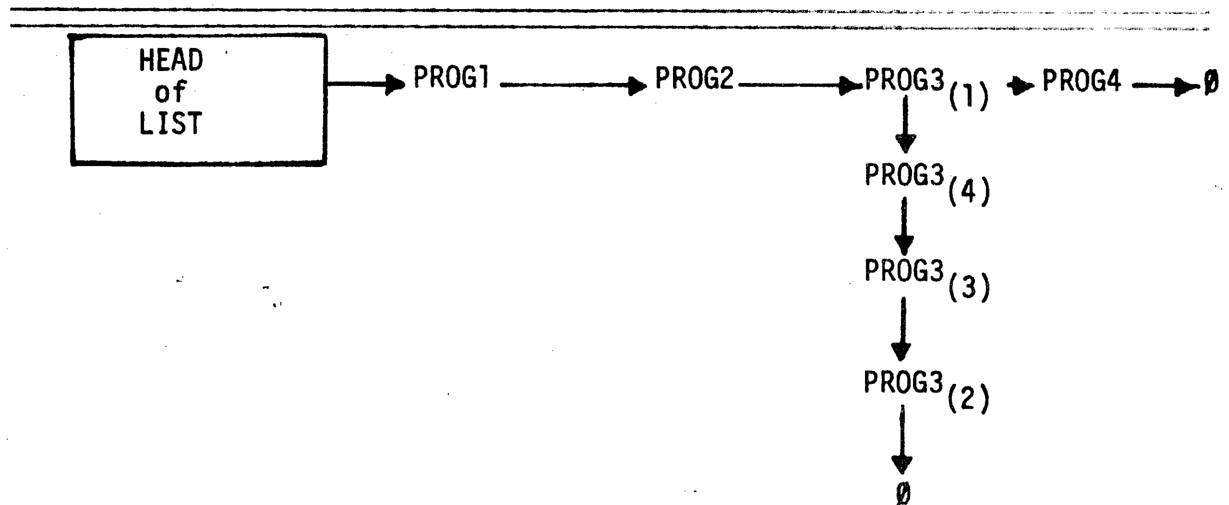
The first word, TDB, will be used by the system as follows:

0 - subroutine is available
=0 - points at 4 word block describing current "owner";
i.e. the program currently executing in the subroutine.

When the TDB is moved to system memory, the 1st word is changed to point to the location the TDB must be moved back to.

The sign bit of the 3rd word of the block indicates if the block was moved or not. The sign bit of the ID-address indicates if the 4 word block is 4 words (0) or 5 words (1) long. (This is caused by a one word inprecision 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.

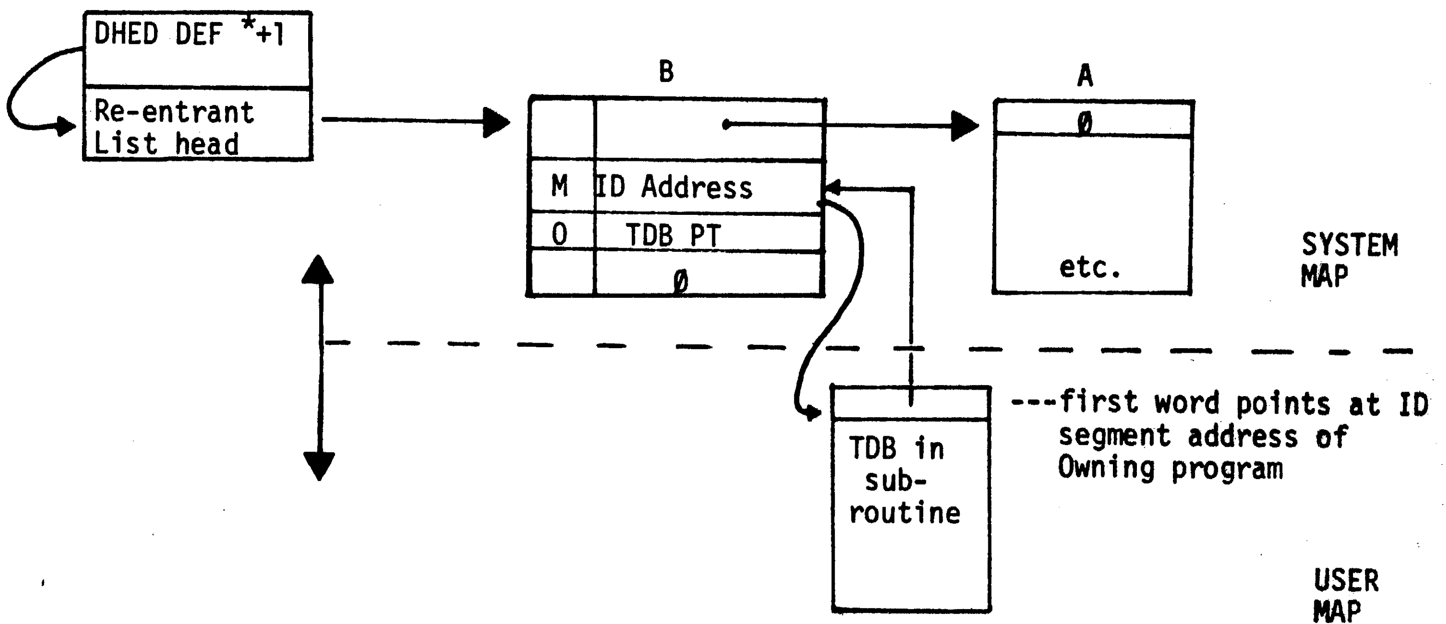


Subscripts of PROG3 refer to order in which the PROG3 reentrant subroutines were called. PROG4 was the first to enter a reentrant routine; PROG1 was the last.

APPENDIX A

Figure 1

One 4-word block is created each time a reentrant routine is entered. Program A and B are both in reentrant subroutines. A entered its routine first.

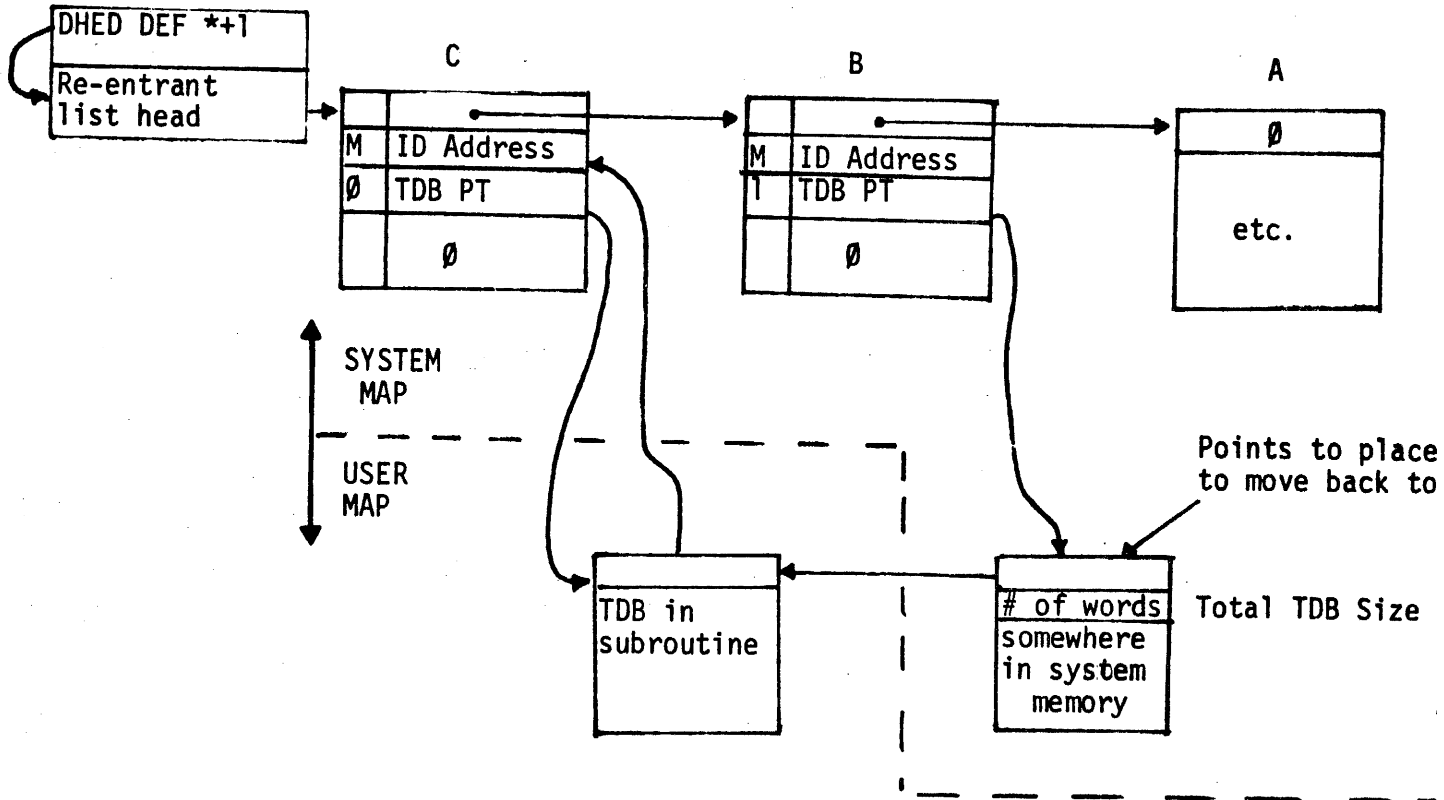


APPENDIX A

Figure 2

"B" is suspended -- program "C" reenters "B's" subroutine.

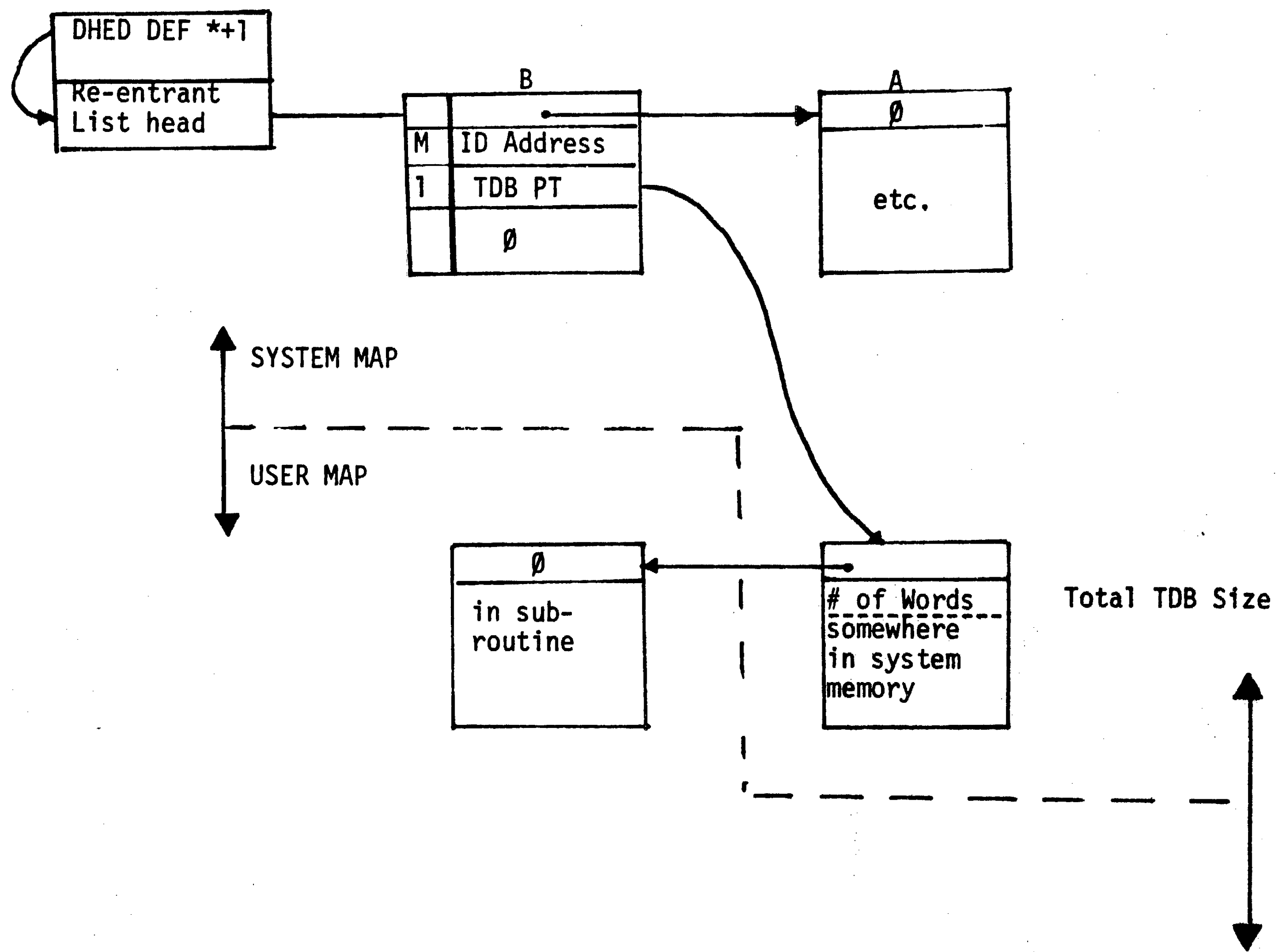
NOTE: The moved status is indicated by "B'S" TBD pointer not pointing in turn t B's ID segment address.



APPENDIX A

Figure 3

Program "C" exists the routine.

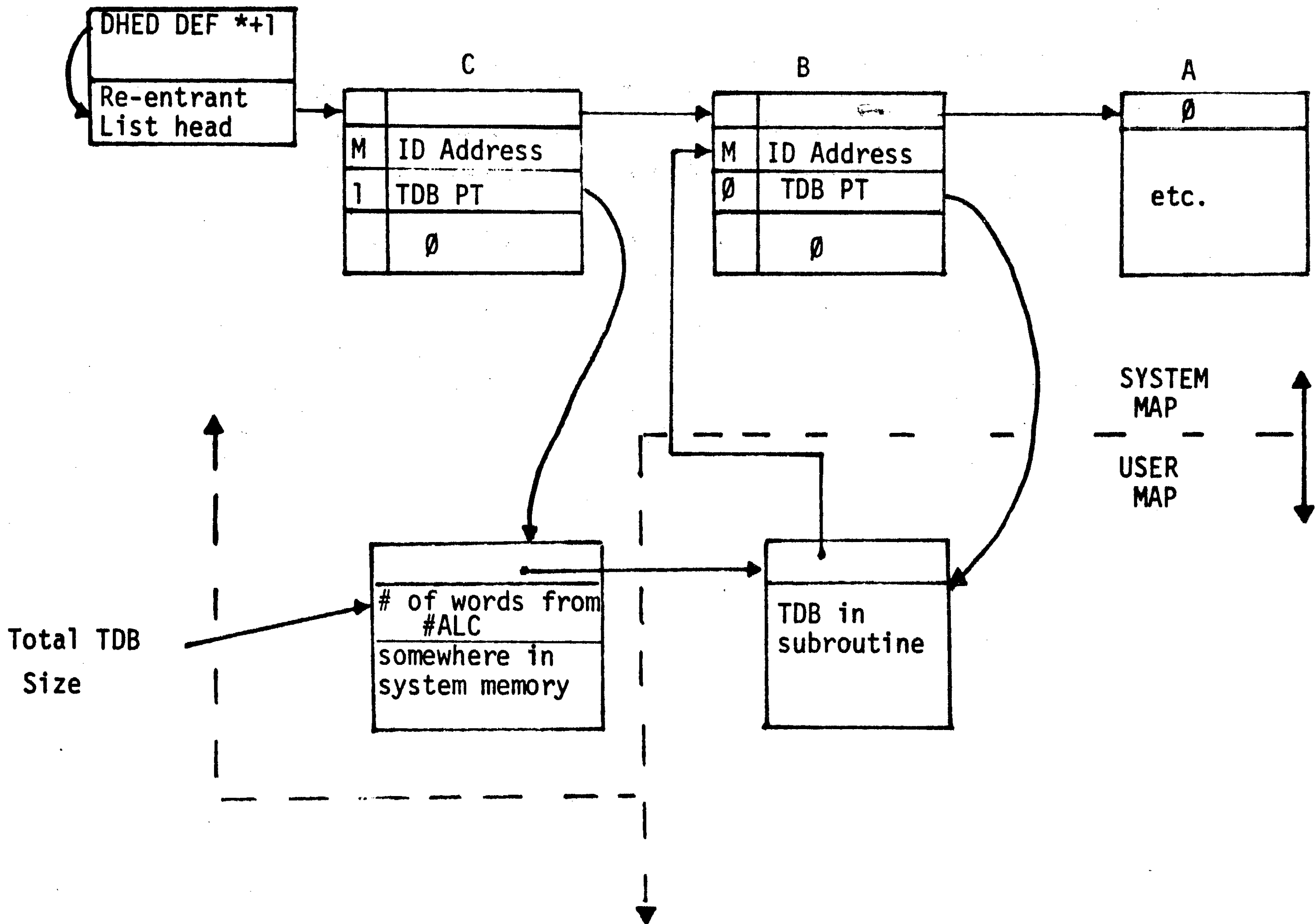


The routine is available - Bs memory will be moved back when the dispatcher is committed to run it.

APPENDIX A

Figure 4

Suppose in Figure 2, program "B" was to be executed prior to "C's" exit from the reentrant routine. Then "C's" core must be saved and "B's" moved back in.



APPENDIX A

Figure 5

Suppose starting at Figure 2, routine "C" now calls another re-entrant subroutine -- the list structure is now:

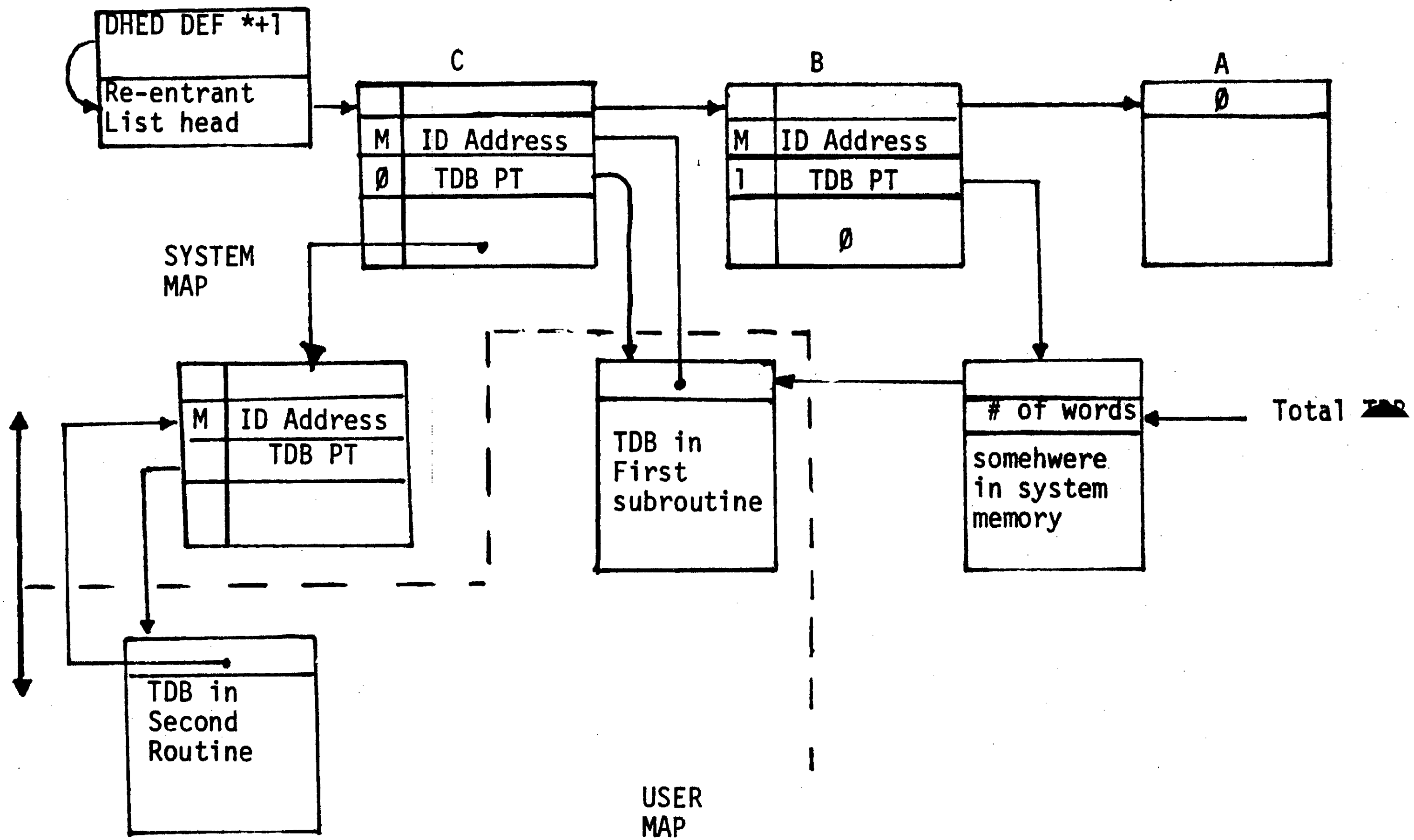
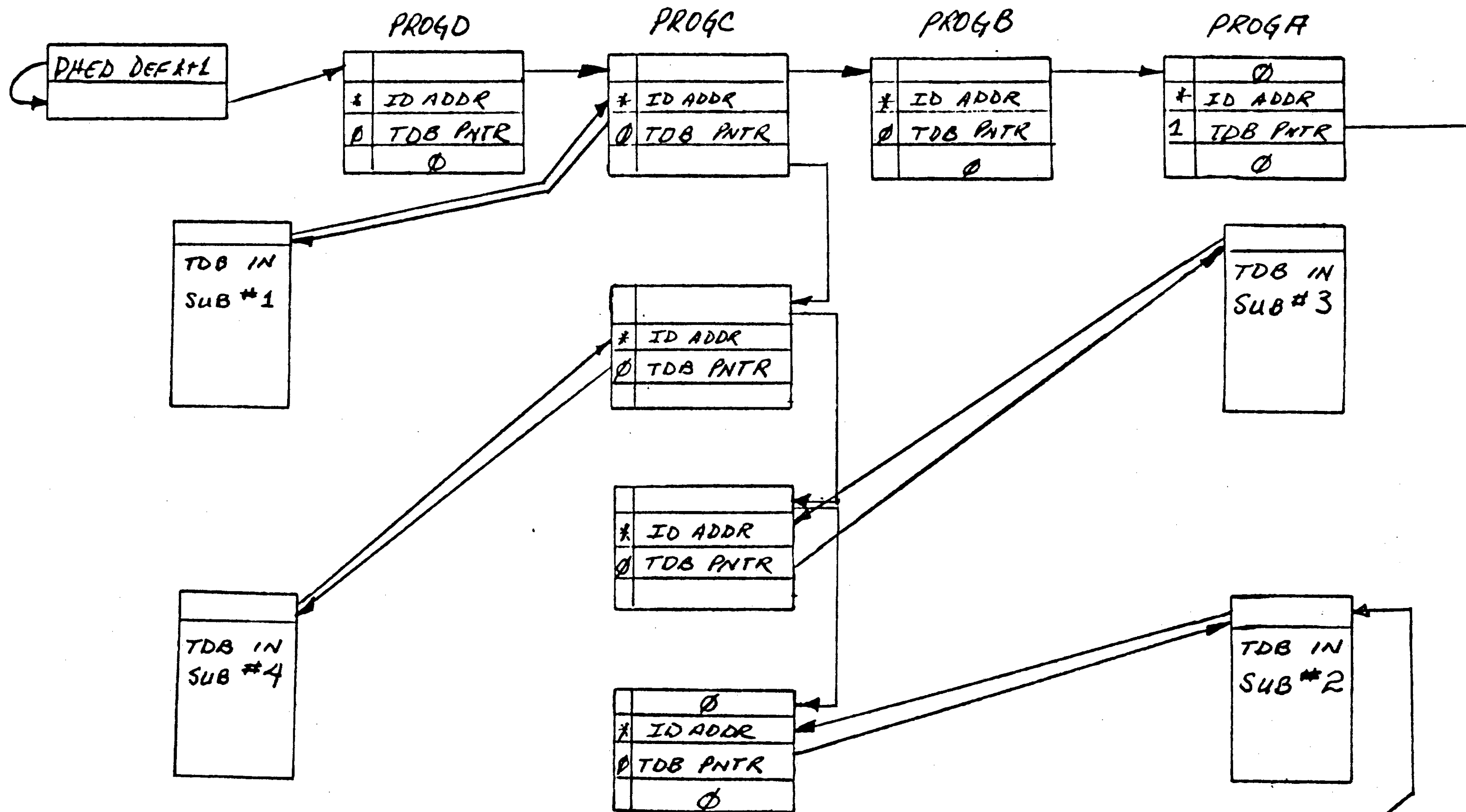


FIGURE 6



IN SYS AVAIL MEMORY

OF WORDS

MOVED TDB OF SUB #2 BELONGS TO PROGA

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.

APPENDIX B

The externals .ZPRV and .ZRNT are treated as "special" entry points in the RTE Disc-Based Operating Systems, in RTE-II, RTE-III and RTE-IV. 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 hencce may be shareable) or if the code is loaded with the program (not shareable), 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 RP's are passed to the on-line loader in the same manner as an operators 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.

***NOTE - The capability of handling calls to REIO must also be added for compatibility reasons since the new library references this routine.

The code of .ENTP and .ENTR is included.

```
                                     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

            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
```

RE-ENTRANT WITH ".ENTR"

```
PRAM1 NOP
PRAM2 NOP
SUB   NOP
      JSB .ZRNT
      DEF LIBX
      JSB .ENTP
      DEF PRAM1
      STA TDB+2
      ...
      ...
LIBX  JMP TDB+2,I
      DEF TDB
      DEC 0
```

```
PRAM1 NOP
PRAM2 NOP
SUB   NOP
      JSB $LIBR
      DEF TDB
      JSB .ENTP
      DEF PRAM1
      STA TDB+2
      ...
      ...
LIBX  JSB $LIBX
      DEF TDB
      DEC 0
```

```
PRAM1 NOP
PRAM2 NOP
SUB   NOP
      RSS
      DEF LIBX
      JSB .ENTP
      DEF PRAM1
      STA TDB+2
      ...
      ...
LIBX  JMP TDB+2,I
      DEF TDB
      DEC 0
```

SCHEDULER TECHNICAL SPECS

MIKE MANLEY
1/26/78

Project #1106

TABLE OF CONTENTS

INTRODUCTION

GENERAL OVERVIEW

List Processor
Link Processor
Mess Processor
System Start Up
Exec Request Handlers

MAJOR FUNCTIONS

Program Termination
Program Schedule
String Passing
Scheduler Interface with Dispatcher

APPENDIX A

ID Segment
ID Extension
Keyword Block
ID Extension Keyword Block

APPENDIX B

Dispatcher Interface To List Processor

APPENDIX C

$LIST calls available to drivers

INTRODUCTION

The scheduler is the RTE IV module which oversees program state transititions, responds to operator input commands, begins system start up at boot up, and satisfies or vectors to other processes 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 are found in Table Area 1 which is 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 33 word RTE-IV ID segment and 3 word ID extension. For those who are not, Appendix A at the end of this manual contains a complete description of every word, bit and field.

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 IV a program is always said to be in a state. 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 schduler 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.

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 prog 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 seg 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
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 oct 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 pram is value to put
                                    into B Reg @ susp)
           7 = ASCII prog anme (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 $WORK
OCT IDADR  DEF PNAME   OCT BVAL
DEF PRAM1  DEF PRAM1
DEF PRAM2  DEF PRAM2
DEF PRAM3  DEF PRAM3        (NO INDIRECT DEFS !!)
DEF PRAM4  DEF PRAM4
DEF PRAM5  DEF PRAM5


---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.

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

Dormant Request

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 B 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 $PUCN, 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 partameters bit, and call link processor.

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:

A1. If dormant bit set jump to dormant request processor.
A2. 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.
A3. If not 1 or 2 above set entire status word = 1 this clears out all other bits; then
A4. Call link processor to schedule program.

If the current status is 6 and

B1. Dormant bit set too, then set status to 0, clear R&D bits, and call link processor to make dormant.
B2. Wait bit set too, then change status to 3 (general wait) and call link processor to put program into general wait state.
B3. 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,

C1. 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.
C2. If D bit set jump to dormant request processor.

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

D1. Perform C1 and C2 in case the program was in the time list and C on SS command set the O bit.

D2. Check to see if the program is disc resident. If so, then check the proper $MATA table entry to see if the program terminated saving resources or serially reusable and is still i the partition. If no go do A1 - A4.

D3. If stil in partition then call the dispatcher routine $DMAL to set the partition up to be reused. Then go do A1 - A4.

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.

Operator Suspend Request

E1. 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.
E2. If current status field = 2, I/O suspend, then set O bit.
E3. 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.
E4. If not 1,2 or 3 above set status to 6 and call link processor.

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

LINK PROCESSOR

The REAL TIME EXECUTIVE "LINK PROCESSOR" function is to remove 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 piror. than any program in list, then last linkage is set to point to the program to be added and the program linkage is cleared.

3. 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 an * (asterisk) is output to system teletype via $XSIO and a request for teletype input is made to the system teletype via $XSIO with the completion address TYP10. 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.

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 $MSGP
```

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 adjusted 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.

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 char's if type = 2
3	used for ASCII only = 2nd two ASCII characters
4	used for ASII 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. SYSTEM START UP

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, to begin with, 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. $RTN, the system available memory return routine is then called at least twice to fill up SAM with the now contigious memory.

When this calculation is complete the system map is reset. Typically it would look as shown below.

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.

$CMST — Starting page of common.
Note that 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 generator

$COML — Number of pages of common
$COML - bits 14-10 shifted down of [MPFI (3) + BGCOM-$DLP] and then add 1 to this result.
Where: BGCOM = base page 1753 length of backround common
MPFI(3) = fourth entry in memory protect fence table. Start of backround 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 2.
$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]

After initializing these values $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 backround, 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.

The last thing $STRT does is pick up the ID address of FMGR, D.RTR, and SMP. These addresses are used later by the system for various types of error checking. $STRT then jumps to the EXEC to finish the system start up.

EXEC also saves D.RTR's address for error checking so that its disc tracks are not released improperly by the user. EXEC t.en jumps to $CGRN in the $TRRN module to set up the resource number table. 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.

5. 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 Backround 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

* The processing of these requests is shared with the system module RTIME.

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 (refer to the EXEC technical specs) and if all is well transfers control to the apropriate 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 request are briefly discussed here. The other requests are discussed in conjunction with other scheduler functions.

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.

SEGMENT LOAD REQUEST

This is an EXEC 8 request. The processor first 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 on 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 transfered to $XEQ.

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 2. Each bit corresponds to 10 MSEC with the most significant bits in the upper byte of the second word.

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

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 $TIMR in the RTIME module.

Program Termination

in RTE IV there are 9 ways a user may terminate his 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 PROGRAMATIC COMPLETION
   a. CALL EXEC (6,0,0)

We shall discuss each of these types in the order of increased system processing requirements.

The type 6, normal completion request, requires the least processing and is by far the most common of program terminations. It is mostly done in the scheduler TERM subroutine.

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 "RN","RE",

"PW and "RN" bits in words 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 B for a description of this process).

The last thing to happen in the normal termination is that 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.

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 RTIME), 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 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. Furhtermore, $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 $XEQ.

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 $ZZZZ. (Refer to Appendix B) ($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 there is the serially reusable completion. A check is made to make sure a father is not trying to terminate a son as serially reusable. If this is detected a normal termination results. If the program is terminating itself then 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 (refer to Appendix B) that the programs partition is not to be put in the free list. $ALDM is then called to take care of the partition. Lastly any optional parameters supplied are placed in the ID segment temporary area.

PROGRAM SCHEDULING

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

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 transfered 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 in RTIME 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 IDCHK 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. Makes sure the program will find a partition large enough to execute in, else generates an SC09 or SC008 error.
4. Places perspective son's NP bit and bits 0-3 of status field into the perspective father's A-Register at suspension word.
5. Calls the string passing routines if necessary. (i.e., if RQP9 = 0 no string passing.)
6. 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 SIZIT 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.

For non EMA programs the check uses the # of pages field, word 22, of the ID segment and compares this against:

# of pages ≤ $MBGP if the program is background

# of pages ≤ $MRTP if the program is real time

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 $MBGP or $MRTP is larger than a 32K address space. In this case the check # of pages ≤ MAX ADDRESS SPACE is used.

If the program is an EMA program, the following check is used.

OF PAGES - MSEG + EMA SIZE ≤ $MCHN OR ASSIGNED PARTITION SIZE

where MSEG is in word 1 of the ID extension, EMA size is in word 29 of the ID segment, and $MCHAN is the size of the largest Mother Partition.

If the check fails on SC09 or SC08 error (SIZE ERROR) will result. However, if the DE bit (EMA default) is set then the EMA size is reset to 1 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 or $MRTP (or the destination partition size) is used as a check to see if the program can fit into a partition. If $MCHN, $MRTP or $MBGP = 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.

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 1). 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 $\frac{N+1}{2}+3$ 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 parameter 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.

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 completition 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.

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 resour(termination, operator suspension does not release the partition.) The partition # field of word 22 is used as on index into the $MATA table and the $MATA residency word is checked to make sure the referenced program no longer owns the partition. If any of these conditions are not met the "ILLEGAL STATUS MESSAGE" is output.

Some other error checking is performed for the AS command. The Partition must exist and the size of the program is checked against the size of the referenced partition. For non EMA programs the # of pages field is compared against and must not be greater than the partitions $MATA entry. For EMA programs the formula used is:

# of Pages - MSEG SIZE + EMA SIZE $\leq$ MOTHER PARTITION SIZE

where: MSEG SIZE is in word 1 of the ID extension, EMA SIZE is in word 29 of the ID segment, and MOTHER PARTITION SIZE is in the $MATA table.

EMA programs may be assigned to regular partitions in addition to chained ones. The size check formula used in this case is

OF PAGES - MSEG+EMA SIZE $\leq$ PARTITION SIZE

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 or word 24 for non segmented programs is used as the lower limit of the error check. The upper limit is defined by the program type as follows:

new SIZE-1< $MBGP for backround programs
new SIZE-1< $MRTP for real time programs

If the program is assigned to a partition

new SIZE-1 ≤ ASSIGNED PARTITION SIZE

(The minus one is because $MBGP & $MRTP does not include Base Page.)

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, 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. Recall that a set DE bit means default EMA (not necessarily MSEG) was taken.

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

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 and $MRTP. 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.

APPENDIX A

The RTE IV ID Segment Table

The RTE IV ID segment for disc resident programs is 33 words long. In addition, all EMA Type programs have a three word ID extension. Memory resident programs have a 25 word ID segment and program segments have a 9 word ID segment. The format for the ID segment and ID extension is shown on the next page. A description of the various words, fields and bits follows.

Word 1 is the linkage word for the program. Whenever the program is put into a state (scheduled op suspend, etc.) the program is put into a linked list threaded through word 1. This word is also used to queue the program up on EQT's for I/O processing.

Words 2-6, 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 2-6 in the ID segment to contain 1,2,3,0 and 0 respectively. The scheduler also takes the address of Word 2 and places this into word 11 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 11 to pick up the run parameters. The words are also used for unbuffered I/O.

Word 2 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 2 by the following rules:

REASON	CONTENTS OF ID WORD 2
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 2 is also used by the $ALC routine anytime a program needs more system available memory than is currently available, assuming that that much memory can ever be available. In this case $ALC places the number of words requested in Word 2 of the requesting programs ID segment. Every time memory is returned through $RTN, word 2 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 7 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 8 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 9 is the point of suspension. Everytime a program is suspended or interrupted Word 9 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 9 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 9.

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 10,11,12 contain the A,B and E/O registers at suspension. Words 13,14 and the upper byte of word 15 contain the 5 ASCII characters of the program name.

The lower byte of word 15 contains the TM,CL,AM and SS bits plus the type field. Word 16 contains the NA,NP,W,A,O,R and D bits plus the status field. These bits and fields are used as follows:

TM This bit is set if the program is temporary. That is, there is no permenant copy of the ID segment in the system area of the disc. If the bit is clear the program is a permanent one.

CL Memory lock (core 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.

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 FIELD. This field of word 15 specifies the program type
Memory Res = 1, Real Time = 2, Backround = 3, Large Backround = 4, Segment = 5 (refer to summary of types in user manual.

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, 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.

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 2 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.

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 out, for one reason or another, that it wasn't 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.

R — The save resources (R) bit is set to indicate that the program would like to save its resources when it goes dormant. The 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 9 = 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 FIELD. This is 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 17,18,19 and 20 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 17 is the time list linkage word. All programs in the time list will be linked together through this word.

Resolution (bits 15-13) in word 18 contains the resolution code. Multiplier (bits 11-0) contain the multiplier for the resolution. The T bit is set if the program is in the time list. Words 19 and 20 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 20 contains the high order bits of the time.

Word 21 of the ID segment contains the BA,FW,MTN,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, scheduled with wait. (EXEC 9 or 25) The son's FW bit is set. If the father scheduled W/O wait the bit is clear.

MTN Multi Terminal Monnitor Bit. This bit is set if the program is operating under the session 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 2 of the prospective fathers ID.

RN This bit is set when a resource number is either owned or locked by this program.

Father FIELD. 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 resuable. This bit is a flag for the dispatcher to avoid certain program clean up procedures. The bit is cleared later.

Word 22 contains the RP bit, the # of pages field, memory protect fence index field, 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 FIELD. This field contains the number of pages the program takes up not counting base page. For segmented programs its 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 FIELD. 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.

Partition # field. This field contains the partition number that
the program last executed in. [Counts from 0.]

Words 23 and 24 contains the high main +1 and low main address respectively of this program. The high +1 address does not include the high main +1 of and program segments.

Words 25 and 26 contain the low and high base page address of the program. high +1 does not include link address for any program segments.

Word 27 contains the disc address of the virgin copy of the program on the disc. The program may only reside on LU2 or LU3. If on LU2 then bit 15 is clear, if on LU3 bit 15 is set. Bits 14-7 contain the track # (0-255) and bits 0-6 contain the sector number. Word 28 is formatted as word 27 but is the swap address in the track pool for the program.

Word 29 is used only for EMA programs and is zero for non EMA programs. Bits 15-10 contain the ordinal 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 30 is used only for segmented programs and is the High Main +1 address of the program, subroutines, and largest segment. If the program is not segmented it is 0.

Words 31, 32 and 33 are session monitor words.

ID EXTENSION

Word 1 and 2 of the ID extension contains the I/O bit, DE bit, the current MSEG field, the MSEG Size field, starting prog page MSEG field, and start page EMA field.

I/O This bit is set whenever the MSEG is modified in order to do I/O which crosses an MSEG boundary. It indicates the current MSEG number is not valid.

DE This bit is set if the default EMA size is taken the bit is not effected by MSEG.

MSEG This is the size in pages of the current mapping segment. It can be declared by the user program or defaulted. For default MSEG = Maximum Logical Address Space -[PROGRAM SIZE +LARGEST SEGMENT +SUBROUTINES]

Current MSEG. Number of the currently mapped MSEG.

START PROG PAGE MSEG. This is the logical starting page of the mapping segment.

START PAGE DMA. Physical starting page # of the EMA-i.e., first mapping segment directly behind the program.

The last word of the ID extension is the swap address of the EMA area of the program. It is the number of tracks of EMA array swapped to the disc. The swap location of the EMA array will begin on the first track following the swapped program.

KEYWORD BLOCK

Because ID segment sizes vary (memory resident size = 28, disc resident size 33, program segment size = 9), some method of indexing to the first word or the name word of an ID segments is necessary. The keyword block is used for this purpose.

Location 1657B on base page specifies the first entry in the keyword block. The keyword block in turn contains 1-word entries, each pointing to an ID segment, Last entry=0. Keyword Block entries are ordered at generation by program type; memory resident, real time disc resident, backround disc resident, available ID segments, and the program segment ID segments.

The keyword block entry (ID address) + 12 always points to the name-word. Thus, keyword entries for short ID's don't point to the first word of the ID.

NOTE: Keyword +12 always points to name.

ID EXTENSION KEYWORD BLOCK

Like the keyword block for ID segments, the ID EXTENSIONS also have a table of pointers also terminated by a 0. Word $IDEX points to the ID extension keyword block. Each word in that block points to an ID EXTENSION.

APPENDIX B

DISPATCHER INTERFACE TO THE LIST PROCESSOR

As mentioned earlier $LIST pushes programs that terminate into a stack through word 9 of the ID segment headed at $ZZZZ in the dispatcher. The dispatcher uses this stack to do 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 9) to 0 in case the program is to be run later. Next if the program is disc resident, any swap tracks it may have are released. This may happen if a program that is swapped out is aborted. A father may also abort his son causing this condition. The tracks are released by $DREL in the EXEC. $DREL also makes a call to $LIST in the scheduler to reschedule any programs that may have been waiting for disc space.

The dispatcher then calls $ABRE in EXEC to return any reentrant memory the program may have. This may 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 may 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 may own. If any memory is returned, programs waiting for memory may 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 which have word 2 of their ID segment equal to the ID segment address of the terminating program. Programs in the general wait list will have word 2 of their ID segment set as follows:

REASONS	CONTENTS OF ID(2)
WAIT TO SCHEDULE A PROGRAM	The programs ID segment address
WAIT FOR COMPLETION OF A "SON"	The "sons" ID segment address
RN ALLOCATE WAIT	Address of the RN table
RN LOCK WAIT	Address of the RN number
LU LOCK WAIT	Address of the RN number associated with the LU LOCK
DOWN DEVICE	4
BUFFER LIMIT EXCEEDED	Address of the EQT on which the limitation was exceeded

NOTE that this call also handles the programs that scheduled with QUEUE.

After $WATR returns, the system calls $TRRN which calls $ULLU to unlock any lock LU's the program has. $TRRN also unlocks any local RN locks and deallocates any local RN allocates the program may have. Each of these processes may call $SCD3 to pick up and schedule waiting programs. If there are any such programs $SCD3 will call $LIST.

Lastly, if the program is a disc resident program and the program still owns the partition but did not make a serial reusable termination, then that partition is released and made available to other programs.

The following page is a quick summary of all this activity.

$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 the EQT word 4 of the EQT of the interupting device. Both of these functions were seperate but now will be condensed into 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 HPIB driver will schedule a program on interrupt and pass three words into the temporary area of that programs ID segment. The $LIST calling sequence to do this is:

```
JSB $LIST
OCT 1
DEF RTN
DEF IDADR
DEF P1
DEF P2
DEF P3

RTN
```

It is up to the HPIB subroutine library to correctly find ID address of the program to be scheduled and pass this to the driver.

TECHNICAL SPECIFICATIONS

FOR

PERR4 - RTE-IV PARITY ERROR MODULE

E.J.W.
2/11/78
Project #1106

TABLE OF CONTENTS

1.0 General Overview of Operation

2.0 Extend Communication

- 2.1 System Tables Referenced
- 2.2 System Base Page Communicator
- 2.3 External Subroutine Called

3.0 Detailed Technical Aspects of Operations

- 3.1 Parity Error Detection
- 3.2 Parity Error verification
- 3.3 Parity Error Recovery Philosophy
- 3.4 Soft Parity Error
- 3.5 System Parity Error
- 3.6 User Program Parity Error
- 3.7 DCPC Parity Error

1.0 GENERAL OVERVIEW OF OPERATON

The Parity Error module's main task is to report parity errors detected by the hardware and to continue operation of the RTE-IV system if possible. PERR4 also tries to reproduce parity errors to identify and warn system users of soft parity errors: errors which may be intermitten or may be generated erroneously.

2.0 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.

2.1 System Tables Referenced

The System tables used by PERR4 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.

2.2 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.

2.3 External Subroutines Called

$ABRX	-	set up ABE,XYO regiter reporting messages and print them on system console
$CNV1	-	convert number to ASCII (one word)
$CNV3	-	convert number to ASCII (three words)
$ERMG	-	used by PERR4 to print "PE" error message and abort user program
$MAXP	-	reestablish maximum size words of unreserved partitions
$SYMG	-	print message on system console
$UNPE	-	unlink a partition entry from the proper list and undefine the partition.

2.4 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.

3.0 DETAILED TECHNICAL ASPECTS OF OPERATIONS

This portion of the Technical Specifications is a detailed description of the major portions of the Parity Error module, PERR4. It is assumed the reader is familiar with the detailed operations of the Dispatcher (DISP4) and the I/O module (RTIO4).

3.1 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't 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. PERR4 is entered only when LIA 4 = 5 and LIA 5 = 1xxxxx.

PERR4 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 PERR4 routine.

3.2 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 's 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 21MX/21XE.

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 RT104).

3.3 Parity Error Recovery Philosophy

While it is possible to always detect the occurance 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.

1. 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.

2. 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 RT1OC 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.

3. It's an inside job.
A parity error detected within the operating system itself may cause errorenous 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.

3.4 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.

3.5 System Parity Error

Parity errors in memory locations in the system itself cannot be recovered as described in section 3.3. 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.

3.6 User Program Parity Error

Parity errors within the memory resident area will cause the program to be aborted. The physical page number, ABEXYO register contents 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 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 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.

3.7 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 oprating 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 as in section 3.6.

SYSTEM LIBRARY

Shaila Kapoor
February 8, 1978
Project #1106

TABLE OF CONTENTS

1. Summary of System Library Changes

2. Technical Details of EMA Routines

   2a. .EMAP
   2b. .EMIO
   2c. MMAP
   2d. EMAST
   2e. Microcoded Routines

1. Summary of System Library Changes

Changes had to be made in several system library routines to make them compatible with RTE IV. The system entry points necessary for $ALRN, RNRQ, LURQ, COR.A, EQLU, IFBRK, PRTN and MESSS routines are included in Table Area 1. The only changes made to these routines were to do crossmap loads and stores to entries in the system. Routines KCVT, TMVAL, INPRS, CNUMO, CNUMD and PARSE need to use routines in the system whose entry points are not available to the user. The code for these routines ($CVT1, $CVT3, $TIMV and $PARS) is duplicated in the system library routines that call them.

Five new routines COR.B, .EMAP, MMAP, .EMIO and EMA were added to the system library. COR.B routine returns in the B register, the first word of free available memory of the program if there are no segments. If the program is segmented then COR.B returns the high address + 1 of the largest segment. If the id segment address passed to COR.B is that of a short id segment, COR.B makes an error return with a -1 in the A register. .EMAP resolves array addressing for normal arrays and for EMA's. MMAP maps mapping segments for EMA's, .EMIO handles EMA addressing and mapping for special cases to insure the entire buffer needed is mapped into the logical address space and EMA returns information on EMA. These new routines are described in the next section.

.EMAP, MMAP and .EMIO can be RP'ed in an RTE IV generation for use on a 21MX E-Series computer with instructions which will link to microcoded versions of these routines. .EMAP and .EMIO and MMAP are interruptible. The opcodes for the EMA microcode are:

```
.EMAP     105257
.EMIO     105240
MMAP      105241
```

2. Technical Details for EMA Routines

2a. .EMAP:

This routine is used to resolve addressing of an element in an n-dimensional array. The algorithm used to calculate displacement for an array element $(A_1, A_2, A_3, \ldots A_{n-1}, A_n)$ is:

Displacement=

$$(((\ldots((A_n - L_n)*d_{n-1} + (A_{n-1} - L_{n-1}))*A_{n-2} + \ldots$$

$$(A_3 - L_3))*D_2 + (A_2 - L_2))*D_1 + (A_1 - L_1))* \text{ \#words/element}$$

where $A_1, \ldots, A_n$ are subscript values defining an element in an n-dimensional array, $L_1, \ldots, L_n$ are the lower bounds of the dimensions, $D_1, \ldots, D_{n-1}$ are the magnitudes of subscript declarators ($Di=Ui-Li+1$, where V_i is the upper bound of the ith dimension) for dimensions 1 thru n-1, # words/elements is the number of words per element in the array (for e.g. 2 for real constants, 3 for double precision constants, etc.). The leftmost dimension (A_1 is the subscript value) is varied the fastest to calculate the displacement.

The user (compiler in the case of the higher level languages) must build a table containing the number of dimensions in the array, the negative of the lower bounds for every dimension, the magnitude of subscript declarators for dimensions 1 thru n-1, and the number of words per element. and two offset words if the array is in EMA. The format of the table is:

TABLE	
# dimensions	
$-L_n$	
D_{n-1}	
$-L_{n-1}$	
D_{n-2}	
.	
.	
$-L_2$	
D_1	
$-L_1$	
# of words/element	
offset word 1(low 16 bits)	used only for EMA
offset word 2(high 16 bits)	

Where Li is the lower bound and Di is the magnitude of the ith dimension.

The calling sequence for .EMAP is:

```
     JSB .EMAP
     DEF RTN          Return address
     DEF ARRAY        start of the array
     DEF TABLE        Table containing the array parameters
     DEF A            Subscript value for the nth dimension
          n
     DEF A            Subscript value for the (n-1)st dimension
          n-1
     .
     .
     .
     DEF A            subscrit value for the 2nd dimension
          2
     DEF A            subscript value for the 1st dimension
          1
RTN error return  A Reg = 15 (Ascii), B Reg = EM (Ascii)
RTN+1 normal return B Reg = address of the element
```

If the XIDEX, base page location in the communication area containing the currently executing program's id segment extension address is not zero and the start address of the array is greater than or equal to the starting logical page of MSEG for this program, then the array for which addressing has to be resolved is an EMA. The procedure followed by .EMAP to resolve element addresses for EMA and non-EMA arrays is the same except for two-word calculations being performed for the EMA. Following steps describe address evaluation for an element in EMA.

1. Initialize pointer PTABL to point to the first entry in TABLE and set the two summation words SUM1 and SUM2 to 0.

2. Get # of dimensions, if negative then error, if 0 then skip to Step 10.

3. Add -Li to Ai and add it to the current sum of previous terms (SUM1).

4. If bit 15 of the summation word (SUM1) is set, then increment SUM2 by 1. Clear bit 15 in SUM1.

5. Multiply SUM1 by D_{i-1}. Save A & B registers in SUM1 and SUM3 respectively.

6. Shift bit 15 of SUM1 into the bit 0 position of SUM3. Clear bit 15 of SUM1.

7. If SUM2 is not equal to 0 multiply it by D_{i-1}.

8. Add SUM3 to SUM2.

9. All dimensions done? If not, go to step 2 to evaluate displacement for the next dimension.

10. Get MSEG size and the logical start page of EMA from the first and second words of the id segment extension of the program.

11. Get the two offset words from the table. Adjust them so that the first word contains low 15 bits (bits 0-14) and the second word contains bits 15-31 of the double word offset. Error if second offse word was negative.

12. Add offset word 1 to SUM1 and offset word 2 to SUM2. Error if overflow occurs either into the sign bit or out of the sign bit of SUM2.

13. The array is an EMA. To get the # of pages in the displacement move bits 10-14 of SUM1 into the least significant bits of SUM2. Then SUM2 is the # of pages in the displacement. The remainder in SUM1 is the offset into the last page for the element address.

14. Get EMA size from the id segment of the program. If # of pages in the displacement from start of EMA +1 >EMA size then error.

15. Divide SUM2 by MSEG size. MSEG# = quotient. The number of words dis placement into the mapping segment = Remainder $*2000_8$ + SUM1. The address of the element in the mapping segment = displacement + base address.

16. The number of pages displacement from start of the EMA upto the start of the MSEG to be mapped JPGS=(number of pages displacement from the start of the EMA upto the page containing the element -#of pages displacement from the start of the MSEG to be mapped upto the page containing the element).

17. If the mapping segment number in the id segment extension = MSEG# and bit 15 of word 0 of the id segment extension is not set, then go to step 24.

18. To determine the highest possible MSEG#, divide EMA size by MSEG size and if the remainder is equal to 0 subtract 1 from the quotient.

19. The highest possible MSEG# is HMSEG = quotient.

20. If the remainder is not 0, the mapping segment size of HMSEG = remainder -1 otherwise it is the standard MSEG size.

21. If the mapping segment to be mapped is HMSEG then its size is that of HMSEG and jump to step 23.

22. The mapping segment to be mapped is not the highest mapping segment therefore the number of pages to be mapped is equivalent to the standard mapping segment size.

23. Call MMAP to map the mapping segment with IPGS, mapping segment size and mapping segment # as parameters.

24. Return with B register = the address of the element calculated in step 15 at location RTN+1.

2b. .EMIO

This routine is used when the user wants to access a certain portion of the external memory area and wants to insure that the entire portion will be mapped in the mapping segment. .EMIO checks if the requested length of buffer is contained within the current mapping segment or within any standard MSEG. If not, .EMIO maps the pages that contain this buffer into the MSEG logical memory.

The calling sequence for .EMIO routine is:

```
     JSB .EMIO
     DEF RTN          Return address
     DEF BUFL         # of words in the buffer
     DEF TABLE        Table containing the array parameters.
     DEF A            subscript value for the nth dimension
          n
     DEF A            subscript value for the (n-1)st dimension
          n-1
      .   .
      .   .
      .   .
     DEF A            subscript value for the 2nd dimension
          2
     DEF A            subscript value for the 1st dimension
          1
RTN error return A reg = 16 (Ascii), B reg = EM (Ascii)
RTN+1 normal return B reg = address of the element
```

Where the element i located at $(A_1, A_2, \ldots, A_{n-1}, A_n)$ in the array. TABLE is the table containing array parameters as described in .EMAP routine and BUFL is the length of the buffer to be accessed. The leftmost dimension (A_1 is the subscript value) is varied the fastest to calculate the displacement.

.EMIO routine follows these steps:

1. Check IDEX location on base page. If contents are 0 then not an EMA program, return with error.

2. Get the buffer length and set up the pointer to the table of array parameters.

3. Calculate element address following steps 1-16 described under .EMAP.

4. Add buffer length and number of words displacement from start of MSEG upto the start of the buffer. Convert to pages.

5. If this number of pages is greater than the standard mapping segment size, go to Step 7 to map a non-standard mapping segment for this buffer.

6. Add the number of pages displacement from the start of the EMA upto start of the MSEG containing the element to the number of pages calculated in Step 4. If this resultant number of pages is greater tha EMA size, then the buffer desired overflows beyond the EMA, make an error return. Otherwise the buffer fits inside the standard mapping segment, follow steps 17-23 described in .EMAP to map this standard mapping segment. Make a normal return to RTN+1 location with the address of the element in the B register.

7. The buffer does not fit within a standard mapping segment, therefore a special mapping segment must be mapped to include the entire buffer. The number of pages to be mapped is calculated by converting the number of words between the start of the page containing the element and the buffer length into pages. The number of pages offset will be the # of pages from the start of EMA upto the page containing the element.

8. Set MSEG # to -1. Call MMAP to map the non-standard mapping segment. MMAP will make an error return if the number of pages asked to map is greater than the standard MSEG size or it overflows the EMA.

9. Return to the calling program with the start address of the buffer in the B register.

2C. MMAP

This routine is used to map a sequence of physical pages into the mapping segment area within the logical address space of the program. The calling sequence is:

```
     JSB MMAP
     DEF RTN                   return address
     DEF IPGS                  displacement in # of pages from start
                               of EMA upto the start of the mapping
                               segment to be mapped.
     DEF NPGS                  # of pages to be mapped.
RTN
```

Returns: A reg = 0 if normal return
=-1 if error return

MMAP routine has three entry points:

MMAP is the main entry point into the routine, ..MP is used by .EMIO routine to map the non-standard mapping segment, .MMAP is used by .EMAP and .EMIO to map the standard mapping segment.

MMAP routine follows these steps:

1. If IPGS and NPGS are negative values then error return.

2. If EMA is not defined in the calling program, error return is made. Get EMA size from word 28 of the id segment, and MSEG size from the first word of the id segment extension of the program.

3. Divide IPGS by the mapping segment size. If the remainder is 0, the pages to be mapped start at a standard MSEG boundary and the quotient is the MSEG number to be mapped. If the remainder is not 0, a non-standard mapping segment has to be mapped.

4. If IPGS+NPGS is greater than EMA size or the number of pages to be mapped (NPGS) is greater than MSEG size, then an error return.

5. Next, MMAP adjusts the number of pages to be mapped (NPGS) to the standard MSEG size +1 (for the overflow page). If IPGS+MSEG size +1 is greater than EMA size, then NPGS is adjusted to EMA size-IPGS i.e., number of pages upto the end of EMA.

6. MMAP goes privileged at this point. Get the physical start page number of EMA from the second word of the program's id segment extension. Add the IPGS to this value to get the physical start page number of the mapping segment.

7. The user map residing in the upper 32 locations of the unmapped portion of the user base page has to be changed to point to the new mapping segment. The DMS base page fence is set such that these upper locations cannot be accessed wsith the normal user map setting. To work around this, MMAP reads DMS register 40,

which is the first DMS register for the user map and contains

the user base page number.

8. The system entry point $DVPT contains the logical start page number for the driver partition in the user map. MMAP changes the DMS register in the user map corresponding to $DVPT to point to the user base page.

9. The user map in the upper 32 locations of the user base page can now be addressed through $DVPT. The logical address at which MMAP starts changing MSEG page numbers is:

   MLOC=$DVPT*$2000_8$ +1740_8 +start logical page of MSEG.

10. The user map is changed for NPGS starting at physical start page of the segment to be mapped. If NPGS is less than the MSEG size+1, the rest of the pages in MSEG size+1 are read-write protected.

11. Transfer MSEG size+1 locations starting at MLOC into the DMS register starting at (40_8 +logical start page of MSEG).

12. If MSEG number is -1, set bit 15 of the first word of the id segment extension to indicate a non-standard mapping segment. Otherwise clear bit 15 and set up bits 5-14 of the first word of the id segment extension to reflect the MSEG # mapped. Return.

2d. EMAST

This routine returns the total EMA size, MSEG size and the starting logical page # for MSEG. All of this information is picked up from word 29 of the ID segment and words 0 and 1 of the ID segment extension of the currently executing program.

The calling sequence is:

```
JSB EMAST                 Return address
DEF RTN
DEF NEMA (returned)       Total EMA size
DEF NMSEG (returned)      Total MSEG size
DEF IMSEG (returned)      Starting logical page MSEG
```

RTN

Upon return A register = 0 if normal return and -1 if error return. An error return is made if an EMA does not exist i.e., XIDEX location on base page communication area is 0.

2e. Microcoded Routines

The major time savings realized by microprogramming the .EMAP, .EMIO and MMAP routines derives from two areas. The first is handling map register modifications from the microcode without the overhead of going priveleged in order to execute the appropriate DMS instruction. This will eliminate approximately 300 microseconds. The sec nd area is common to both .EMAP and .EMIO and envolves the arithmetic calculations necessary to determine element address. Every subscript past one will cause an additional iteration through this calculation loop adding an approximate 40 microseconds software (10 microseconds microexecution). The following table summarizes timing differences for the software versus microcoded routines:

	Software	Firmware
MMAP		49 + 1.2 x (# pages)
.EMAP no remapping		39 + 10 x (# dim)
.EMAP remapping		88 + 10 x (# dim) + 1.2 x (# pages)
.EMIO no remapping		40 + 10 x (# dim)
.EMIO remapping		92 + 10 x (# dim) + 1.2 (# pages)

All times are approximate and worst case in microseconds.

The approximate manufacturing cost based on using six 1K PROM's is 50$.

In order to modify maps from the microcode without DMS generating a memory protect violation, the microprogrammed map routines disable memory protect and re-enable it upon completion. This is accomplished by specifying IOG in the special field creating a memory protect violation. IAK is then specified in the special field of the next microinstruction executed (occurs at T6). The IAK is received by memory protect and it is disabled (assuming a non-I/O instruction is present in the instruction register IR). Memory Protect is re-enabled by building a STC 5 in the IR and jumping to the I/O group routine in the base set. Because of this, it is not possible to call the firmware EMA routines from privileged subroutines, although the software versions can be called from a privileged subroutine.

The operation of the firmware is essentially the same as the software with a few exceptions. Steps 4 through 6 of .EMAP (Section 2a) which do double precision calculations are somewhat different. The low order word is treated as a 16 bit quantity rather than 15 bits as in the software. The flag COUT is used for carries into the high order word. This is possible because the firmware does the required multiply directly rather than calling the multiply routine which is used by the software. Sign correction is not done, so the result is a 16 bit unsigned multiply, which is what is required.

The other difference is in the way base page is updated when a map change is required. The firmware uses user map register one (reg 41B) to map in the user base page. The map reg contents are referenced with addressed 37408 to 3777B. The previous contents of map register 41B are restored before the firmware completes.

Two procedures are available to verify the proper installation and operation of the firmware. A fourth opcode is provided which, when executed in single step mode, loads 102077 into the S register.

The procedure to run this test follows:

1. Set P to 0
2. Load A with 105242
3. Push Preset
4. Push Single Step

The display indicator should be S and the display register should be 102077. Any other result is an error. The firmware simply builds a 102077, puts it on the display register, sets the display indicator to S, and returns. It does test for single step so the instructions 105242 is a NOP if executed while running.

The second procedure is to run the EMA verifier (names #EMA) in RTE-IV. This does a complete test of all the operations of the firmware.

TECHNICAL SPECS FOR EMA MICROCODE

Don Venhaus
2/3/78

Project #1106

Microcoded Routines

The major time savings realized by microprogramming the .EMAP, .EMIO and MMAP routines derives from two areas. The first is handling map register modifications from the microcode without the overhead of going priveleged in order to execute the appropriate DMS instruction. This will eliminate approximately 300 microseconds. The second area is common to both .EMAP and .EMIO and envolves the arithmetic calculations necessary to determine element address. Every subscript past one will cause an additional iteration through this calculation loop adding an approximate 40 microseconds software (10 microseconds microexecution). The following table summarizes timing differences for the software versus microcoded routines:

	Software	Firmware
MMAP		49 + 1.2 x (# pages)
.EMAP no remapping		35 + 10 x (# dim)
.EMAP remapping		(firmware .EMAP always remaps)
.EMIO no remapping		43 + 10 x (# dim)
.EMIO remapping		90 + 10 x (# dim) + 1.2 (# pages)

All times are approximate and worst case in microseconds.

The approximate manufacturing cost based on using six 1K PROM's is 50$.

In order to modify maps from the microcode without DMS generating a memory protect violation, the microprogrammed map routines disable memory protect and re-enable it upon completion. This is accomplished by specifying IOG in the special field creating a memory protect violation. IAK is then specified in the special field of the next microinstruction executed (occurs at T6). The IAK is received by memory protect and it is disabled (assuming a non-I/O instruction is present in the instruction register IR). Memory Protect is re-enabled by building a STC 5 in the IR and jumping to the I/O group routine in the base set. Because of this, it is not possible to call the firmware EMA routines from privileged subroutines, although the software versions can be called from a privileged subroutine.

It should be noted that the operation of the firmware version of .EMAP is not identical to the software version. The calling sequence is the same and the result is the same (that is, the address of the element is returned in B). However, the firmware version of .EMAP does not use standard MSEGs. Instead, it maps in only the page containing the element and the following page, if possible.

Two procedures are available to verify the proper installation and operation of the firmware. A fourth opcode is provided which, when executed in single step mode, loads 102077 into the S register.

The procedure to run this test follows:

1. Set P to 0
2. Load A with 105242
3. Push Preset
4. Push Single Step

The display indicator should be S and the display register should be 102077. Any other result is an error. The firmware simply builds a 102077, puts it on the display register, sets the display indicator to S, and returns. It does test for single step so the instructions 105242 is a NOP if executed while running.

The second procedure is to run the EMA verifier (named #EMA) in RTE-IV. This does a complete test of all the operations of the firmware.

The opcodes for the EMA microcode are:

```
.EMAP  105257
.EMIO  105240
MMAP   105241
```

The following discussion assumes that the EMA microcode at hand and that the XE microprogramming manual is available. Address references are to micro control store addresses.

22000-22017

These are the 16 available entry points into this module. EMAP is the last one so that a jump is not required. There is no reason for the entry points not being sequential.

22017

A read is started to fetch the array address. P is incremented to point to the table pointer. The flag is cleared to indicated to MMAP (if it is called) that is was called from a microroutine rather than with a 105241 opcode. Note that the return address is not fetched here. It is fetched in the exit code.

22020

Since M has the address of the array address, M-1 is the address of the return point. This is saved in Y for returning and in case of an interrupt, Y will have the location of the calling opcode +1.

22021-22034

GETPARM is a subroutine which fetches a parameter. The address is left in M. This section of code saves the array address and the table address. The address of the address of the ID extension (1645) is built and put into S3, which is never used for anything else. The contents of 1645 is fetched and if zero, the non-EMA address calculations are used.

22035-22043

Word 1 of the ID.EXT, is fetched and masked to give the logical start of the MSEG. If the start of the array is below the log start MSEG, then it is not an EMA array.

22044

The RESOLVE subroutine calculates a 31-bit address offset into EMA from the subscripts provided.

22045-22051

This section of code saves the address within the page and converts the 31 bit address returned by RESOLVE to a physical page offset into EMA. If bits were shifted out then there was an error.

22051-22061

GETAD28 fetches word 28 of the ID segment. Next we put 1777B in L to do some masking. The LSB part of the 32 bit address is masked to give the address within the page of the desired element. S4 (which was loaded at 22040) is masked to give phys start page EMA. Finally, the EMA size is put into L and subtracted from the required offset. If the answer is positive then OFFSET> EMASIZE which is an error. If the result is -1, then the second page must be protected. This is signaled by setting the flag.

22062-22063

Add the physical start EMA to the offset to give the required physical page.

22064-22066

Form the logical start page of the MSEG in B.

22067

Check interrupts before we crash MP.

22070-22076

The IOG here forces a Memory Protect interrupt. The IAK four lines down acknowledges the interrupt and thus disables MP. On the next line CIR is checked to see who received the IAK. It can only be MP(SC5) or Powerfail(SC4). It is possible that a powerfail occurred between the JMP CNDX HOI and the IOG. If it did, we immediately bail out. Since IOG happens at T2 and IAK happens at T6, there is some dead time which can be used. First the logical start page of the MSEG is masked out and saved in X. Then a 20040B is built in S2. This will be used for setting the MEM address register.

22077

Since an interrupt always switches to the system map, MEM must be set back to the user map.

22100-22107

Since this revision of EMAP doesn't use standard MSEGs, the nonstandard MSEG bit must be set. The word is fetched and if that bit is not set, it is set and written back.

22110-22121

Now set the MEAR to map 40B which is the users base page. Get the users physical base page and put it in the log start of MSEG map register. Base page can then be updated by writing into location (log start address MSEG + 1740B + Log start Page MSEG). This was done so that it wasn't necessary to save a different map register and restore it when finished.

22122-22131

Now write into BP the two physical pages which will be mapped in. If the flag is set, the protect bits are set on the second page.

22132-22140

MEAR is set to the MAP at the logical start of MSEG. The physical pages required are in S9 and S10. While loading the map registers, there is some dead time due to the requirement for a READ, WRTE, or RJ30 exactly two micro-instructions before a Q_0, Q_1, or Q_2 micro order. A STC5 will be required, so it is built now.

22141-22147

The address of the return address is in Y. This must be retrieved to do a proper return. The address which must be returned in the B register is the address of the element in the page plus the logical start of the MSEG. The return address is incremented by one because the normal return goes to rtn+1. The JMP IOG with STC5 in the instruction register is required to turn MP back on, since it was turned off by the IOG...IAK combination.

22150-22215

This section does the subscript calculation for non-EMA arrays. There is no magic here.

22216-22245

This is the error return section. The error code is dependent upon which EMA routine was called. However, they all take the same error exit. It is not possible to just check the Counter (which is the low 8 bits of the IR) because it may have been used. Therefore, the opcode must be fetched to determine the proper error return. The error return exits by doing a jump to fetch (location 0) rather than a return because the error routine may have been jumped to by a subroutine (such as EMAS). A RTN would return to the calling micro-program rather than terminating the microprogram.

22246-22260

The buffer length is fetched and saved in S9. The address of the return address is saved in Y. The address of the table is put in X. 1645B (the address of the ID extension address) is built in S3.

22261-22267

The address of the ID ext.. is fetch. If it is zero, then error. Word 1 of the ID ext is retrieved via XLOAD and masked to give the log start address of the MSEG in S.

22270

EMAS will do the required subscript calculations and compute the MSEG required.

22271-22301

The address of the element in the page plus the buffer length gives the number of words which must be mapped. This is converted to pages and saved in S11.

22302-22305

If the number of pages for the buffer plus the offset is bigger than EMA size then error (buffer extends beyond end of EMA).

22306-22322

S1 is the offset in the MSEG to the element. S2 + S9 converted to pages gives the standard MSEG size required to hold the buffer. If this will fit in a standard MSEG, then we can map a standard MSEG as calculated by EMAS.

22323-22331

Must calculate the nonstandard MSEG necessary to hold the buffer. S1 is the logical address of the buffer. S5 is the number of pages needed to hold the buffer. S6 is set to the page displacement into EMA to the page containing the element. The flag is set to tell MMAP that it was called from a micro routine. MMAP02 is an entry point in MMAP. B is set to the address of the Buffer and a normal exit taken.

22332

Come here if the buffer will fit in a standard MSEG. EMAT will map the proper MSEG if required.

22333-22342

This section of code gets the return address and sets P to rtn+1. It also checks to see if a STC5 is in the IR. If not, then a normal return is done. If there is a STC5 in the IR, then mapping occurred and we must return through the base set to turn MP back on.

22343-22350

The flag is cleared so that the mapping routine will not return to a calling routine but will exit. The page displacement is fetched. During the dead time 1645 is built in S3.

22351-22361

Fetch the number of pages and put into S5. Get the ID extension address and put into SP.

22362-22402

The number of pages in a standard MSEG is retrieved from ID.EXT. word zero and saved in P. The logical start of the MSEG is fetched from word 1 of ID.eXT. and put in S9. The physical start of EMA is put in S. The displacement is added to phys. start page to give phys start of MSEG in S7. The displacement is divided by MSEG size to see if a standard MSEG is being mapped (which is true if the remainder is zero) and to get the MSEG number.

22403-22407

The MSEG number (or all 1's if a nonstandard MSEG is being mapped) is put in S4. Subtract the number of pages to be mapped from the MSEG size. If the result is negative then error.

22410-22426

Get the EMA size and put in L. Add the Start page of EMA and save in B. This gives the last physical page which can be mapped. Next, the phys start page of the MSEG is added to the requested number of pages to give the last page requested. If (SPMSEG+REQSIZE)>(SPEMA+EMASZ), the request was for too many pages. If (SPMSEG+STAN.MSEG.SIZE)>(SPEMA+EMASIZE), then map only up to end of EMA, otherwise map a standard MSEG number of pages.

In other words map to the end of EMA or the MSEG, whichever comes first.

22427-22436

The map routines will always map the MSEG size +1 number of pages, if possible. This section checks to see if that last page is with EMA. If it is not then it must be protected. Then a check for a pending halt or interrupt is done.

22437-22442

Memory protect is disabled by forcing and acknowledging a MP interrupt. This is necessary in order to load base page and the DMS registers. The Central Interrupt Register is checked to see if the powerfail fail interrupt received the IAK. This is possible if a powerfail happened after the last interrupt check. Return to the base set if powerfail was pending.

22443

The interrupt forced a switch to the system map. DMS must be set back to the user map.

22444-22456

The constant 20040B will be used to set the Memory Expansion Address Register to the users physical base page. The page number is stored in S6. The content of user map register 1 is saved so that the base page can be accessed through map reg. 1. Map reg. 1 is then set to the users physical base page. Because of the requirement for a READ, RJ30 or WRTE exactly two instructions before some of the DMS instructions, there is some dead time. This is used to build some of the constants which will be required later.

22457-22463

The logical start page of the MSEG is computed to determine the first of the map register to be changed. S3 is loaded with STANSIZE+1 because we always load that many map registers, although some may be protected.

22465-22471

The location 3740B is the start of the user maps in BP (remember the physical base page was loaded into user map reg. 1). The logical start page of MSEG is added to 3740B to give the place to start changing the map registers. This address is saved in S6. The counter is loaded with the number of maps to be done. L is cleared so no read/write protect bits are set.

22472-22503

This heroic loop writes the required map register contents into base page. The counter (previously S5) starts with the number of map register to be loaded bit not read/write protected. The rest of the registers, up to MSEGSIZE+1, are read write protected. If cntr=MSEGSIZE+1 no pages are protected.

22504-22514

This section reads the map reg contents from base page and loads the DMS registers. The counter must be decremented in the loop because ICNT followed immediately by JMP CNDX CNT does not always work. Note that the bottom 8 bits of the address is the negative of the number of times through the loop. In other words, if the counter counted up we stop when it gets to 000 (the last map to be loaded is 1777 in BP). Since the counter must count down, we use the negative of the bottom 8 bits of the address.

22515-22531

Restore user map reg 1. Form ID EXT word 0 (MSEG# etc.) and cross store. In the meantime, build a STC 5 in the IR.

22532

If the flag was set, then we were called by a micro-routine and must return. Otherwise return through IOG to turn on MP.

22540

Resolve computes a 31 bit address from the array subscripts.

22541-22561

The standard MSEG size is fetched and put in S5. The log start of the MSEG is saved in S8. The required page is computed and saved in S2 and A. If the page displacement is greater than EMA size then error.

22562-22575

The offset to the element is divided by MSEG size to give MSEG number. MSEG number is saved in S4. The remainder is subtracted from the page offset to element to give page offset to MSEG. The remainder is converted to words and/ added to the bottom 10 bits of the 31 bit address to give the word offset in the MSEG. B is set to log start MSEG. (Note: B+S1 gives address of element.)

Resolve

22576-22660

This routine converts the array subscripts to a 31 bit address.

EMAT

Check to see if mapping is necessary.

22661-22667

The logical address of the element if formed in S1.
Bit 15 of the ID EXT word zero is checked to see if a nonstandard MSEG is mapped. If bit 15 is set, then must remap.

22670-22674

The MSEG number in ID EXT word 0 is compared to the required MSEG number. If equal, then don't have to remap.

22675-22723

The Phys Start EMA is fetched. Phys start EMA + offset to MSEG gives Start Page MSEG. Next, the EMA size is divided by MSEG to give max MSEG number. If this is the MSEG required we use the remainder as MSEGSIZE, otherwise use the standard MSEGSIZE. An entry into MMAP is used to do the mapping. (Note the flag is set so MMAP returns.)

GETPARM

22724

Start a read in case this address is indirect. Get the results and load M.

22725-22730

If the last address was not indirect, return with a read on the parameter initiated. If it was indirect, fetch its contents and try again. If halt or int. is pending, go service it, unless it is single stepped.

22731-22732

The address of the EMA opcode+1 was saved in Y. P is reset to this value and the interrupt handled. The EMA microcode will be restarted.

22733-22742

This code used when interrupts are checked at various places.

POSDIV

22743-22747

Does a positive divide of A by L. Quotient in A, remainder in B.

GETAD28

22750-22754

Get word 28 of the ID segment. Flows into XLOAD.

XLOAD

22755-22761

Does a cross load. Can't return on the same line as TAB because word Type II may not work right after TAB. Must have SII in S-bus field so we can do conditional jumps baseD ON retrieved value.

22762-22763

If powerfaul occurred before the IAK then come here, reset P go to base set after IAK.

22764-22771

This section of code builds a 102077 in the S register. This can be executed from the front panel to verify proper installation of the ROMs.

TECHNICAL SPECIFICATIONS FOR THE ONLINE GENERATOR, RT4GN

February 4, 1977 - KFH
Updated January 9, 1978 - KFH

TABLE OF CONTENTS

PAGE

1.0 INTRODUCTION 1-1

1.1 General Overview.................................. 1-1
1.2 Operation... 1-1

2.0 GENERATION SEQUENCE................................ 2-1

3.0 FILE INTERFACE..................................... 3-1

3.1 Interface Routines................................ 3-1
3.2 Scratch File...................................... 3-3
3.3 Relocatable Input................................. 3-3
3.4 Answer File....................................... 3-4
3.5 Echo/List Output Tree Struture.................... 3-6
3.6 List File... 3-6
3.7 Echo.. 3-7
3.8 Bootstrap File.................................... 3-7

4.0 ABSOLUTE OUTPUT.................................... 4-1

4.1 Absolute Output File Truncation................... 4-1
4.2 Header Records.................................... 4-2
4.3 Output Routines................................... 4-5

5.0 PROGRAM INPUT PHASE................................ 5-1

5.1 DISPLAY Command Processor......................... 5-1
5.2 REL(OCATE) Command Processor...................... 5-2
5.3 MAP Command Processor............................. 5-2
5.4 LINKS Command Processor........................... 5-3
5.5 /E Command Processor 5-3

6.0 IDENT,LST,FIXUP TABLE STRUCTURES 6-1

6.1 Pointers and Indices.............................. 6-2
6.2 Table Processors.................................. 6-2
6.3 IDENT TAble Format................................ 6-4
6.4 LST Table Format.................................. 6-5
6.5 FIXUP Table Format................................ 6-5
6.6 LST Index For .ZRNT............................... 6-6

PAGE

7.0 PROGRAM TYPES...7-1

7.1 Entry Point Availability Per Program Type 7-1
7.2 Relocation by Types.................................7-4
7.3 Libraries...7-5
7.3.1 Memory Resident Library...........................7-5
7.3.2 Relocatable Disk Resident Library.................7-5
7.3.3 Library Entry Points List.........................7-5
7.4 Undefined Entry Points..............................7-6

8.0 SIZE RESTRICTIONS...................................8-1

8.1 Page Alignments.....................................8-1

9.0 MISC. AREAS...9-1

9.1 Base Page...9-1
9.2 System Communication Area...........................9-3
9.3 Common..9-8
9.4 Configurator..9-10
9.5 Bootstrap and Extension.............................9-10

10.0 TABLE AREAS I AND II...............................10-1

10.1 EQT,DRT,INT..10-2
10.2 Drivers and DVMAP..................................10-3
10.2.1 System Driver Area...............................10-4
10.2.2 Driver Partitions................................10-4
10.3 ID Segments and Extensions.........................10-7

11.0 EXTENDED MEMORY AREA...............................11-1

12.0 PARTITION DEFINITION PHASE.........................12-1

12.1 Program Page Requirements and Sizes................12-1
12.2 System Available Memory............................12-1
12.3 Partition Definition...............................12-3
12.4 Modify Program Page Requirements...................12-8
12.5 Assign Program Partitions..........................12-8
12.6 Memory Protect Fence Table.........................12-9
12.7 Memory Resident Program Map........................12-9
12.8 Setting System Entry Point Values..................12-11

PAGE

13.0 ERROR PROCESSING.................................... 13-1

13.1 Generation Errors........................... 13-1
13.2 File Errors................................. 13-2
13.3 Abortive Termination........................ 13-2
13.3.1 \ABOR.............................. 13-2
13.3.2 \TERM.............................. 13-3
13.4 Misc. Error Processors...................... 13-3
13.5 Error Suspensions........................... 13-4
13.6 Answer File Errors.......................... 13-4
13.7 Driver Partition Overflow................... 13-4
13.8 Error Codes................................. 13-5

APPENDIX A LOGICAL MEMORY LAYOUT........................ A-1

APPENDIX B DISC LAYOUT OF SYSTEM........................ B-1

APPENDIX C LIBRARY LAYOUTS.............................. C-1

APPENDIX D PHYSICAL MEMORY LAYOUT....................... D-1

FIGURES
----

1 ECHO,LIST FILE TREE STRUCTURE....................... 3-6
2 HEADER RECORD #1.................................... 4-2
3 HEADER RECORD #2.................................... 4-3
4 LOGICAL MEMORY MAPS................................. A-1
5 DISC LAYOUT OF RTE-IV............................... B-1
6 DISC RESIDENT LIBRARY............................... C-1
7 PHYSICAL MEMORY LAYOUT.............................. D-1

TABLES
------

1 TABLE ADDRESSING.................................... 6-2
2 IDENT TABLE ENTRY................................... 6-4
3 LST TABLE ENTRY..................................... 6-5
4 FIXUP TABLE ENTRY................................... 6-5
5 PROGRAM TYPES....................................... 7-1
6 PROGRAM TYPE REFERENCES............................. 7-3
7 BASE PAGE FORMATS................................... 9-2
8 ID SEGMENT FORMAT................................... 10-8
9 MEMORY ALLOCATION TABLE............................. 12-4
10 MEMORY PROTECT FENCE TABLE.......................... 12-9
11 MEMORY RESIDENT MAP................................. 12-10
12 ERROR CODES... 13-5

1.0 INTRODUCTION

This section is intended to serve as an aid to the programmer when modifying the online generator program RT4GN. It should be used in conjunction with the generator source listings, as it in no way attempts familiar with RTE-IV and its generation process, and has run the generator. It assumes familiarity with the RTE-IV online generator reference manual outlining the generation process.

1.1 General Overview

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 online generator a configuration can be achieved under control of the present RTE system, concurrent with other system activities. The online generation process utilizes the file management features of BSM 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.

1.2 Operation

RT4GN is a type 2 or 3 segmented program requiring Table Area II access in RTE-IV. The generator accepts its command input from an "answer" file locatec on disc, a logical unit, 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.

2.0 GENERATION SEQUENCE

LIST FILE NAME?

____

ECHO?

____

EST # OF TRACKS IN OUTPUT FILE?

____

OUTPUT FILE NAME?

____

SYSTEM DISC?

____

for HP 7900/7901 Disc Only:

CONTROLLER SELECT CODE?

____

#TRKS,FIRST TRK ON SUBCHNL?
 0?

________,________

.
.
.

or for HP 7905/7920 Disc Only

CONTROLLER SELECT CODE?

____

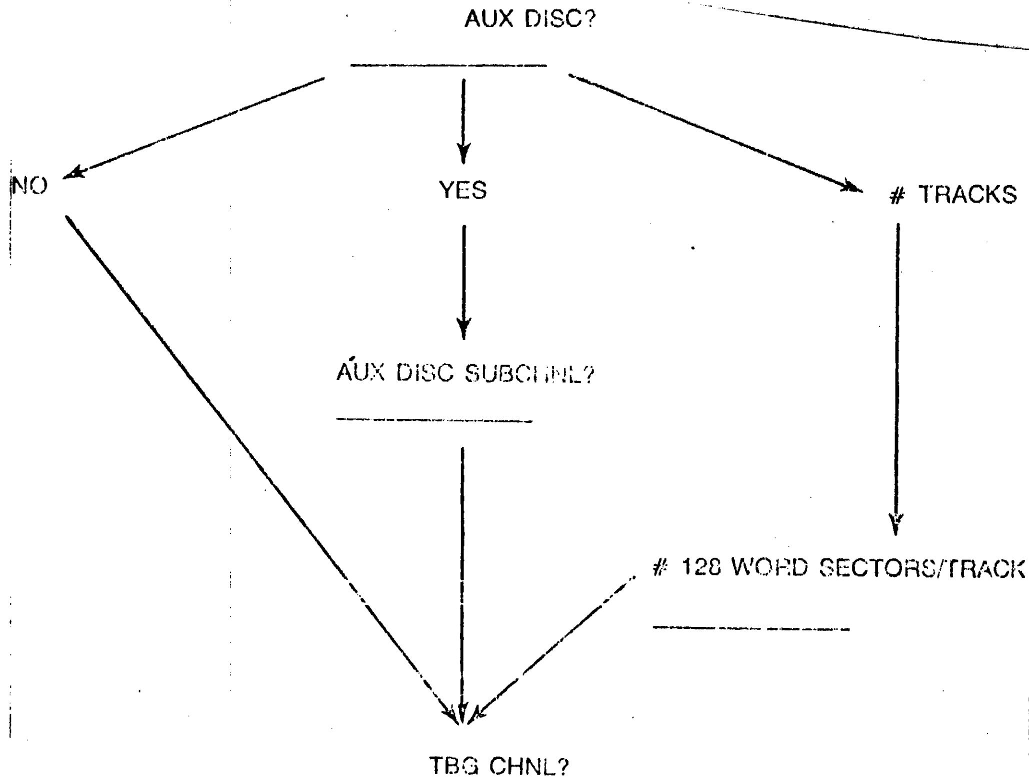
#TRKS,FIRST CYL #,HEAD,# SURFACES,UNIT,# SPARES FOR SUBCHNL:
 00?

________,________,________,________,________,________

.
.
.

SYSTEM SUBCHNL?

____



TBC SELECT CODE?

____

PRIV. INT. SELECT CODE?

____

MEM. RES. ACCESS TABLE AREA II?

____

RT MEMORY LOCK?

____

BG MEMORY LOCK?

____

SWAP DELAY?

____

MEM SIZE?

_____

BOOT FILE NAME?

____

PROG INPUT PHASE:

____
____
____
____
.
.
.
____
____
/E

PARAMETERS

____
____
____
.
.
.
____
____
/E

CHANGE ENTS?

____
____
____
.
.
.
____
____
/E

Table Area I

Equipment Table Entry

EQT 01?

________,________,________,________,________,________,________,________

EQT 02?

________,________,________,________,________,________,________,________

EQT 03?

________,________,________,________,________,________,________,________

EQT 04?

________,________,________,________,________,________,________,________

EQT 05?

________,________,________,________,________,________,________,________

.
.
.

/E

Device Reference Table

1 = EQT #?

________,________

2 = EQT #?

________,________

3 = EQT #?

________,________

4 = EQT #?

________,________

.
.
.

/E

Interrupt Table

```
   4        ENT     $POWR
________,________,________
________,________,________
________,________,________
________,________,________
________,________,________
________,________,________
________,________,________
________,________,________
________,________,________
________,________,________
________,________,________
________,________,________
.
.
.
/E

TABLE AREA I MODULES
  |
  | load map
  v
DRIVR PART 00002
CHANGE DRIVR PART?
--
DP01:
  |
  | load map
  v
SUBSYSTEM GLOBAL AREA
  |
  | 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

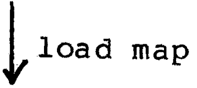


TABLE AREA II

OF I/O CLASSES?

______

OF LU MAPPINGS?

______

OF RESOURCE NUMBERS?

______

BUFFER LIMITS (LOW,HIGH)?

______,______

OF BLANK ID SEGMENTS?

______

OF BLANK SHORT ID SEGMENTS?

______

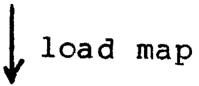
OF BLANK ID EXTENSIONS?

______

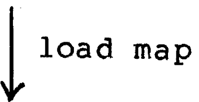
MAXIMUM # OF PARTITIONS?

______

TABLE AREA II MODULES



SYSTEM



PARTITION DRIVERS

DP 02: ↓ load maps

DP 03: ↓

.
.
.

MEMORY RESIDENT LIBRARY

↓ load map

MEMORY RESIDENTS

↓ load map

RT DISK RESIDENTS

↓ load map

BG DISK RESIDENTS

↓ load map

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

1ST PART PG YYYYY
CHANGE 1ST PART PG?

________

SYS AV MEM XXXXX WORDS

PAGES REMAINING:  XXXXX

DEFINE PARTITIONS
PART 01?
.
.
.
SUBPARTITIONS?
.
.
.
/E

MODIFY PROGRAM PAGE REQUIREMENTS?
________
________
.
.
.
/E

ASSIGN PROGRAM PARTITIONS?
________
________
.
.
.
/E

SYSTEM STORED ON DISC
SYS SIZE:  XX TRKS, YYY SECS

RT4GN FINISHED

ZZZZ ERRORS
```

3.0 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 DCB´s 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 TR´s and errors 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 file(s) 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.

3.1 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, and crn is at PARS4+1.

\CRET first calls FOPEN which calls TYP0 - 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 the \CLOS caller the responsibility of 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 TYP0 determines if a lu was specified in the first parameter and TYP0 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.

TYP0 - is passed the DCB address in the A-reg. 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. LU's 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    0     directory entry address
Word 1    0        of file
Word 2    0     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    100001 security codes agree; update open
Word 8    -----
Word 9    ID segment address of generator (from 1717)
Word 10   -----
Word 11   -----
Word 12   -----
Word 13   0 (initial value) in buffer and write flag
Word 14   1 (initial value) record count
```

Special checks are made in determining the EOF control of subfunctions. For driver types >=17 and for DVR05 exclusive of subchannel 0, a 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.

3.2 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 a program's priority or execution interval are changed during the Parameter Phase. If such a modified NAM record does exist for a program, then 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 (eg, A-B) and create a new one. It gets confused if there exist old entries in a file left over from a previous generation.

3.3 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-reg on entry to \RNAM lets us assume 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 merely read the next relocatable record of a file, and optionally to get its position.

3.4 Answer File

Upon start-up, the generator determines thru RMPAR and GETST calls whether an answer file name or lu was specified via the turn-on parameters. If the first parameter was 0, lu 1 will become the default command (answer) lu. If parameter 1 was numeric, that lu will be used for command input (in an MTM environment, this would be the operator console's lu provided no parameters were specified). A dummy DCB will be created in TYP0 for the lu, or the answer file specified in the Namr parameters will be opened via FMP. If an error occurs on the answer file open call, the appropriate error message will be displayed on the console via an EXEC call, and control will be 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, then that lu becomes the error lu provided it represented an interactive device. If it was not interactive, the photoreader for example, then the error lu would default to lu 1.

When an error occurs, the error message (s) is sent to both the list file and the error lu. For many errors, control will be 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 already from an interactive lu then the control is not transferred from it.

All command input is handled by the \PRMT routine, which also issues the prompting message. \PRMT filters the input looking for a !! starting in Column 1 - indicating the operator wishes to abort the generator; or for a , : or TR - indicating 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, Parameter3 the security code, and Parameter4 the CRN:

PARS2,	PARS3,	PARS4	Type:	0=null,	1=numeric,	2=ASCII
PARS2+1	PARS3+1	PARS4+1	0		number	ch 1 & 2
PARS2+2	PARS3+2	PARS4+2	0		0	ch 3 & 4
PARS2+3	PARS3+3	PARS4+3	0		0	ch 5 & 6

Asterisks (*) are not allowed within filenames, security codes or cartridge labels. As soon as 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's the current command input lu (CMDLU) then the response is not sent; otherwise \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, then 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 by the user. (GEN ERR 19 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 will then save 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 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 and handles the recovery. This error won't occur on a POP as the command input source we're

returning to would have already been checked at the original transfer.

3.5

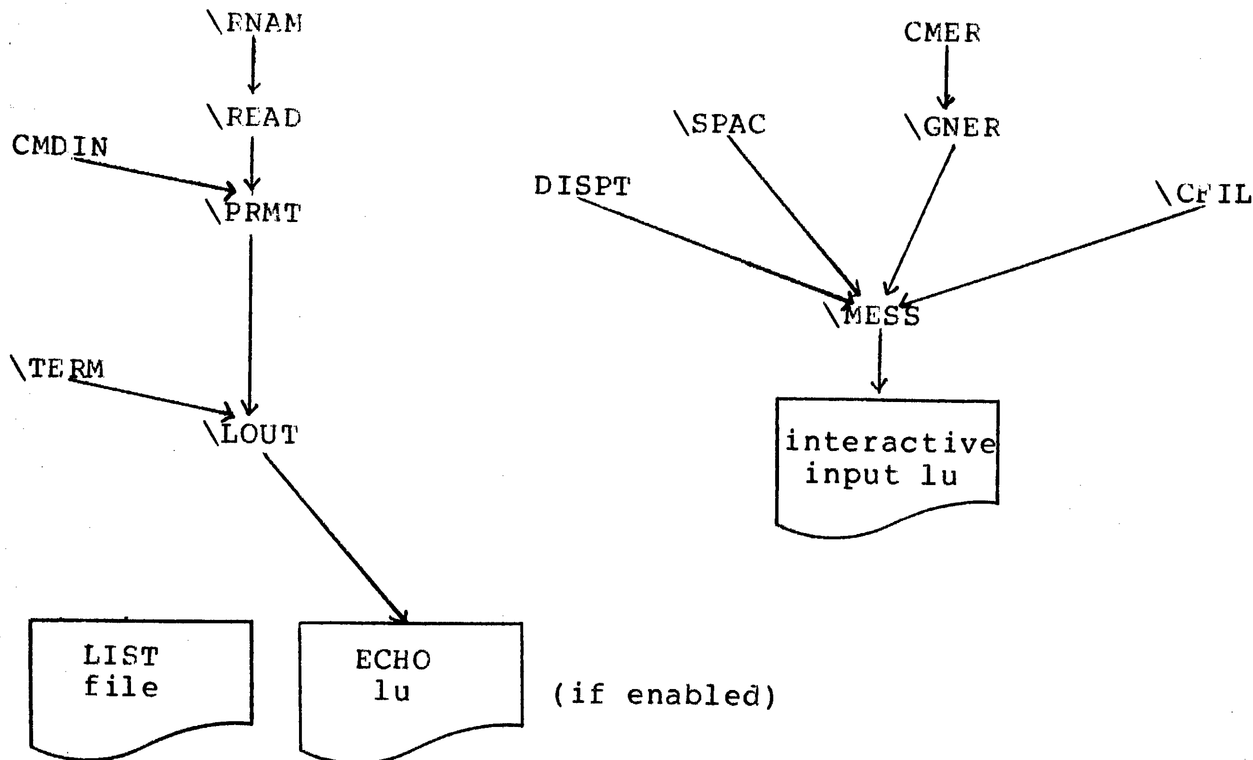


Figure 1 - Tree structure for generator command input and echo/list output routines.

3.6 List File

The operator chooses where they wish to send the generation listed output. If a file name was specified, then a file of size 64 blocks 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 to a non-interactive device, then 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 that time when the resource becomes available.

If the lu was interactive then 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 was not an interactive lu (IALST=0) but the command input/answer file was (IACOM=1), then the line is sent to the operator console (using \IDCB) as well. The status of the command input mode reflected in IACOM changes as TR's are encountered or errors are detected - so it is necessary to perform these checks every time list output is done.

See the Error processing section for the handling of list file errors.

3.7 Echo

The operator must always answer the ECHO? prompt even where not applicable (as when list file is lu of operator console, or generator is to be directed interactively). If the operator requests an echo, then checks are made in \LOUT to see if both IALST and IACOM are equal to 0, meaning neither the listed output or command input are an interactive device. If IALST or IACOM indicate an interactive lu, then 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 done to determine if they reference the same EQT. Note that because echo is dependent on the command input mode it may change as TR's are done to and from the operator console or when error mode is enabled.

3.8 Bootstrap File

At \BOT0,\BOT5 the moving head bootstrap loader may be sent to an FMP file or to an lu: a 0 response by the operator implies no boot is desired. The "0" must be specifically checked for as the

4.0 ABSOLUTE OUTPUT

The absolute output file for the system being generated must be a type 1 file because of its fixed length, quick random access features. Because type 1 files are not extendible, the user must over-estimate the track size of the new system, with the routine \TRUN truncating any excess file space at the end of the generation. The file size specified is checked for a 15 track minimum, as is a size so large that it results in a negative number of blocks (#tracks x 48). When the file created is too small, a GEN ERR 17 will occur at that point where the end-of-file is passed, usually a considerable way into the generation. Because the generator has to abort, the user usually has to learn this lesson only once!

Since RTE requires locations of items on the disc, disc addresses within the generated system are relative to the start of the disc and thus are relative to the start of the absolute output file (exclusive of header records).

4.1 Absolute Output File Truncation

The routine \TRUN is called at the end of the generation during the cleanup processing. It closes the absolute output file, truncating any unused file area. Immediately preceding the call is a forced access to the last record +1 (using \DSKI) to make the truncate work. A special situation results when the area used by the new system is exactly equal to the created size of the file. When the forced access is done, an FMP-12 error occurs because we've overflowed the file. The record was obviously not read in (makes no difference because it was a dummy read anyway), and most important, the track and sector addresses in /ADCB were not set. So /TRUN must check the FMP error code still in FMRR from DISKD's READF call - and if it's negative, no truncation is to be done.

The truncation is done by determining the number of unused blocks in the file, and deleting them in the \CLOS call:

(# blocks in file) - (current block #) = TMP

where the current block # is obtained as follows:

(last file track - first file track)*#sectors per track -
(first file sector + last file sector) *2

4.2 Header Records

The absolute output file contains two, 128-word header records. These are used as a means of passing pertinent information to SWTCH, without SWTCH having to go searching through the absolute output file for it. The track map table (TMT) buffer is used for storage of the header record information as it's being built.

After \STBL send the Track Map Table to the system storage area, it uses the TMT buffer TB30 to send header record #2. In the case of a 7905/7920 this header record will not be identical to $TB32 because it does not contain the number of subchannels, but does contain the spares for each subchannel. See Figures 3a and 3B for the header record #2 formats.

After record #2 is sent, record #1 is built in bits and pieces. The EQT information is obtained while the EQT's are being built. IOADD, the channel, is placed in the high byte, and IOTYP, the EQT type, is placed in the low byte of word pointed to by HEADR (the next header record entry). When the lu of the system console has been assigned, the corresponding channel is retrieved.

Once FSECT has rewritten the track 0 sector 0 boot extension with the correct values, it builds the remainder of record #1 as well. The format of record #1, and where some of its values are obtained, is:

word		Contents		Source
0	*	1 | SYSTEM SUBCHANNEL #		SYSCH
1		SYSTEM DISC EQT #		DRT2
2		# OF EQT'S IN SYSTEM		CEQT
3		PRIVILEGED INTERRUPT CHANNEL		PIOC
4		TBG CHANNEL		TBCHN
5		# OF DISC SUBCHANNELS	SYSTEM CONSOLE CHANNEL	#SUBC/CONTENTS OF TTYCH+3
6		CHANNEL/EQT TYPE OF EQT #1		IOADD/IOTYP
		CHANNEL/EQT TYPE OF EQT #2		.
		.		.
		.		
		.		
		CHANNEL/EQT TYPE OF EQT #n		.

FIGURE 2 - Header Record #1

* bit 15 is set since the absolute output file contains the second header record; this is used so the RTE-IV version of SWTCH can check against RT2GN/RT3GN-produced output files.

HEADER RECORD #2 (128 words)

7900 disc:

```
              -----------------------------
word 0        | FIRST TRACK, subchannel 0 |
     1        | FIRST TRACK, subchannel 1 |
     2        | FIRST TRACK, subchannel 2 |
     3        | FIRST TRACK, subchannel 3 |
     4        | FIRST TRACK, subchannel 4 |
     5        | FIRST TRACK, subchannel 5 |
     6        | FIRST TRACK, subchannel 6 |
     7        | FIRST TRACK, subchannel 7 |
     8        | # of TRACKS, subchannel 0 |
     9        | # of TRACKS, subchannel 1 |
     10       | # of TRACKS, subchannel 2 |
     11       | # of TRACKS, subchannel 3 |
     12       | # of TRACKS, subchannel 4 |
     13       | # of TRACKS, subchannel 5 |
     14       | # of TRACKS, subchannel 6 |
     15       | # of TRACKS, subchannel 7 |
              -----------------------------
```

FIGURE 3A - Header Record #2

or

7905/7906/7920 disc:

```
          ---------------------------------------------
word 0     | STARTING CYLINDER #                       |
     1   * | # SURFACES, STARTING HEAD, UNIT           |  subchannel 0
     2     | # TRACKS                                  |
     3     | # SPARES                                  |
           |-------------------------------------------|
     4     | STARTING CYLINDER #                       |
     5     | # SURFACES, STARTING HEAD, UNIT           |  subchannel 1
     6     | # TRACKS                                  |
     7     | # SPARES                                  |
           |-------------------------------------------|
     8     | STARTING CYLINDER #                       |
     9     | # SURFACES, STARTING HEAD, UNIT           |  subchannel 2
     10    | # TRACKS                                  |
     11    | # SPARES                                  |
           |-------------------------------------------|
     12    |                                           |
     13    |                   .                       |
     14    |                                           |
     14    |                   .                       |
     .     |                                           |
     .     |                   .                       |
     .     |                                           |  subchannel 31
     127   |                                           |
          ---------------------------------------------
```

* format: 15-12: # SURFACES 11-8: STARTING HEAD 3-0: UNIT

FIGURE 3B - Header Record #2

4.3 Output Routines

The following 5 routines control the output of code to the core-image output file.

\ABDO - May be the most useful routine in the generator. It is used to read or write words from the absolute output file, using the memory address of the word in the target system. \ABDO operates using a 3-word map giving the base disc location, the memory address referencing that location, and the highest memory address referenced using this map. \ABDO puts out the current absolute code word (in A-reg) at the memory address (in B-reg) under the current map. Gaps are filled with zero codes if the current word falls beyond the highest previously generated word. For convenience, three "maps" are automatically maintained. These are for the system, a driver partition or a user program, and a user program segment. Subroutines \SYS,\USER, and \SEGS are used to activate these respective maps. Other maps could be constructed and activated by SETDS.

\DSKD - Translates a disc address to a record number in the Type 1 absolute output file, thus satisfying the file's random access nature. FMP READF and WRITE calls are used to access the output file in 128-word (sector-sized) chunks. Note that an FMP-12 error (EOF sensed) is ignored on READF. The reason behind this is that during clean-up, there is a forced access to the last record used, +1, before \TRUN is called. This is to set up the proper track addresses in the DCB \ADCB so \TRUN can truncate the absolute output file to the last referenced record. \DSKI is used to read in that record but it is never rewritten (where the error would be sensed).

In the special case when the header records are written, the memory address in the B-reg is set to negative to indicate record #1. For record #2 the A-reg containing the disc address as well as the B-reg are negative. See Header Record Section for the record formats.

\DSKA - Increments the current disc address to that of the succeeding sector. SDS# is used in determining track crossings and equals the number of 64 word sectors per track.

\DSKI - Controls input from the disc (at specified disc and core addresses) and uses a buffer making the disc appear to have 64 word sectors.

\DSKO - Controls output to the disc as \DSKI controls input.

5.0 PROGRAM INPUT PHASE

During the Program Input Phase, the generator will accept one of many responses to its prompt. Because of this, segment 2 contains its own command scanner and branch mnemonic table. At PRCMD/NXTCM the routine CMDIN is called to retrieve the next command, SCAN is called to determine the correct command processor, and control is sent to that processor (who returns to NXTCM unless a /E was entered). CMDIN issues the prompt and receives the command via \PRMT (which filters and handles all TR's and etc.).

SCAN is used by PRCMD/NXTCM to search for the command keyword, returning a processor index in the A-reg. Scan is also called by some of the individual command processors to search only their portion of the command table such as MODULES, GLOBALS, LINKS, OFF or ALL for the MAP command. PTABL is the branch table for the seven keyword commands (note the existence of entries for both RELOCATE and REL). CMDIN catches blank or comment (*) line, QGETC catches all * following the PIP commands (on the same line), while GETAL catches all those following non-PIP commands - in both cases the remainder of the line is ignored for processing.

The command mnemonic table CTABL contains an entry for each allowable keyword - if an abbreviation is exactly the same as the beginning of a longer keyword (e.g. REL and RELOCATE) the longer must appear first. Each entry in the table contains two pieces of info: bits 15-8 indicate the number of characters in the Ascii keyword, and bits 7-0 contain the offset into table CMTBL of the Ascii value for that keyword. The ordering of entries in both PTABL and CTABL must not be changed, whereas that of CMTBL is of no importance.

When called by the individual processors (of DISPLAY, MAP or LINKS) SCAN operates on their individual keyword tables (LTABS, MTABS and ITAB respectively), again returning the matched keyword index in the A-reg.

5.1 DISPLAY Command Processor

DSPST makes use of the error return (P+1) from SCAN, when neither UNDEFS or TABLE were indicated, and consequently searches the LST for an entry point. If not found, the entry point name followed by an "UNDEFINED" is printed.

LSPST will always send its output to both the operator console and list file. Therefore it is necessary for RT4GN to simulate a TR to the operator console (unless already in the interactive mode), do the display, and then pop that TR from the stack. The operator never realizes that this even happens. If either TABLE or UNDEFS was specified, then the routine EPL is utilized (A-reg on entry = 1 means list LST entry pts, = 0 means list LST undefined's). If a DISPLAY UNDEFS, TR is request, the 'pop' will not be done if there existed any undefs (A-REG is nonzero on return from EPL).

5.2 REL(OCATE) Command Processor

RELST is the most complicated PIP command processor because of the optional module name specification allowed before the file name. Because an lu cannot be specified for relocatable input, special checks are done at CHFNM to insure that one was not entered. \RNAM and \RBIN will catch an invalid file name. Note that because of the special format of the RELOCATE command - no comma until immediately before the filename - \PARS will stick the filename, security code and crn in the usual parse buffer locations (see File Interface). If a module name was specified, RELST stores the name in buffer XNAM; if none specified, XNAM's value is 0. So when a Nam record is read checks are performed at LDRC3 to determine whether or not to load the module. If no module name was specified in XNAM, then the entire file is unconditionally loaded (note that "loaded" here refers to module entries being placed in the LST and IDENT tables - actual relocation is done later). If a module name was specified, then it is determined if there exists a match between the XNAM module name and that of the current Nam. If no match resulted then those relocatable records through the next End record are skipped, otherwise that module only is loaded. Two variables are used to control loading: SERFG = 0 indicated a module is to be loaded, -1 that its to be skipped; NAMR. = 0 when a Nam record is expected (either at beginning of file, after an END record, or in record skipping mode), = 1 means one is not.

5.3 MAP Command Processor

MAPST controls the memory map listing during the relocation phases. The value of MAPMD is stored in bits 3-0 of ID5 of all IDENT entries. MAPMD settings (bit 0 for globals, bit 1 for modules, bit 2 for links) are in effect for all IDENTS entered through subsequent RELOCATE commands. Options can be turned off only by entering a MAP OFF command, and then entering another MAP command listing the desired options (more than one can be specified). A MAP statement is processed left to right - therefore MAP MODULES,

ALL,LINKS,OFF,GLOBALS would result in GLOBALS only being mapped (MAPMD = 000001). The initial (default) value of MAPMD is off.

5.4 LINKS Command Processor

LNKST controls the linkage mode during relocation. As for the MAP command, the option specified is in effect for all IDENTS entered through subsequent RELOCATE commands. The initial (default) value for LNKMD is 0 for base page mode; 1 indicates current page mode.

The value of LNKMD is stored in bit 15 of all IDENT entries. Current page linking is never done on assembled type 3,4 or 5 programs (and their variations).

5.5 /E Command Processor

EOL terminates the Program Input Phase by exiting thru PRCMD´s success return. Before returning however, it calls EPL indicating a listing of the undefined EXT´s (if any) - this listing goes only to the list file (not to operator console, as did the DISPLAY UNDEFS command).

6.0 IDENT, LST, and FIXUP Table Structures

The IDENT table contains an entry for each relocatable module which is specified by RELOCATE commands during the Program Input Phase. There is one entry in the LST for each entry point defined in each relocatable module; entries are also created by the generator (e.g. for $CLAS, $LUSW, $RNTB, $LUAV, $TB31/2) and for those ENT´s suplied by the user during the Parameter Phase. FIXUP entries are used during relocation when an entry point is accessed before it has been defined (no address in the LST).

These three tables are stored in the available memory space starting at the first word following the end of Segment 3, and ending at LWAM. Note that the assumption is made that Segment 3 is the largest segment.

There must exist at least 512 words of undeclared memory in order to insure at least one sector´s worth of words for each table. Initially the space is allocated: 1/4th for the FIXUP table and 3/8th´s each for the LST and IDENT tables. Once a block of space is allocated, it is truncated to a sector-multiple number of words. The block size must also be divisible into the track size (so that when many blocks are swapped out none will cross a track boundary); thus a block may be truncated further by one or more sectors. All truncated words are collected and added to the LST block, as it usually needs the greatest space and is accessed the most. Its block size must still fulfill the above two restrictions, however. The maximum block size is 6144 words (ie, one track´s worth of information).

Six tracks are obtained for the swapping of these tables: 1 for the FIXUP, 2 for the LST, and 3 for the IDENT. If these tracks cannot be obtained, the generator issues the appropriate message and suspends itself (by re-issuing the call without the no-suspend bit) until the tracks become available.

6.3 IDENT TABLE ENTRY FORMAT

```
word 1:  ID1 - NAME 1,2
word 2:  ID2 - NAME 3,4
word 3:  ID3 - (15-8) NAME 5
               ( 2-0) USAGE FLAGS:
                      2 this module was loaded as part of a segment
                      1 must load this module (ext defined by it)
                      0 this module was loaded
word 4:  ID4 - (15)   set if main program module
               (14-0) COMMON LENGTH
word 5:  ID5 - (15)   BASE/CURRENT PAGE LINKING FLAG=1/0
               (14)   NEW NAM RECORD FLAG
               (13-4) EMA SIZE
               (3-0)  MAP OPTIONS:
                      2 links
                      1 modules
                      0 globals
word 6:  ID6 - (15)   EMA DECLARED
               (14-10)MSEG SIZE
               (6-0)  TYPE
                      4 SSGA
                      3 REVERSE COMMON
word 7:  ID7 - LOWEST DBL ADDRESS
word 8:  ID8 - DISK LENGTH FOR UTILITY RELOCATABLES
            or MAIN IDENT INDEX FOR SEGMENTS
            or (15-8) PAGE REQUIREMENTS
               (7-0)  KEYWORD INDEX
            or (15)   EQT defined
               (14)   SDA declared
               (13)   SDA own mapping
               (13-0) DRIVER LENGTH
word 9:  ID9  - FILE NAME 1,2
word 10: ID10 - FILE NAME 3,4
word 11: ID11 - FILE NAME 5,6
word 12: ID12 - SECURITY CODE
word 13: ID13 - CARTRIDGE LABEL
word 14: ID14 - RECORD NUMBER
word 15: ID15 - RELATIVE BLOCK
word 16: ID16 - BLOCK OFFSET
```

TABLE 2

6.4 LST ENTRY FORMAT

```
word 1:   LST1 - NAME 1,2
word 2:   LST2 - NAME 3,4
word 3:   LST3 - (15-8) NAME 5
                 (7-0)  ORDINAL
word 4:   LST4 - IDENT INDEX
                 or 2 if COMMON
                 or 3 if ABSOLUTE
                 or 4 if REPLACE
                 or 5 if UNDEFINED
                 or 6 if EMA
word 5:   LST5 - SYMBOL VALUE
                 or IDENT INDEX if EMA
```

TABLE 3

6.5 FIXUP TABLE ENTRY FORMAT

```
word 1:  FIX1 - MEMORY ADDRESS
word 2:  FIX2 - (15-11) instr. code
                (10) byte instr. indicator
                (9) upper BP link indicator
                (2-0) DBL record type
word 3:  FIX3 - OFFSET
word 4:  FIX4 - INDEX OF LST ENTRY REFERENCED
                0 if local symbol
                -1 if .ZRNT
```

TABLE 4

6.6 LST Index for .ZRNT

Since the indices to the LST entries begin with zero (unfortunately) there may be confusion with the value of FIX4,I whose value is either the index of the LST entry referenced, or zero for no reference. It turns out the .ZRNT is always the first entry in the LST because the generator places it there, so it always has the 0 index. Therefore, during the DBL relocation processing in segment 4 (at DBL45 to be exact) when a .ZRNT reference is detected (a special case as it is), then the corresonding FIX4,I entry is set to -1. DFIX will later check this value (where here 0 means to use a zero value), but since .ZRNT is a replace operation (\LST4 = 4) then the bogus -1 value of FIX4,I is never used.

7.0 PROGRAM TYPES

0: SYSTEM
1: MEMORY RESIDENT
2: RT DISK RESIDENT
3: BG DISK RESIDENT W/ TAII ACCESS
4: BG DISK RESIDENT W/O TAII ACCESS
5: SEGMENT
6: LIBRARY/UTILITY
7: UTILITY
8: UTILITY LOAD ONLY TO SATISFY EXTERNAL REFERENCES
9: MEMORY RESIDENT USING BACKGROUND COMMON
10: RT DISK RESIDENT USING BACKGROUND COMMON
11: BG DISK RESIDENT USING REAL TIME COMMON W/ TAII ACCESS
12: BG DISK RESIDENT USING REAL TIME COMMON W/O TAII ACCESS
13: TABLE AREA II
14: TYPE 6 BEING FORCE-LOADED INTO RESIDENT LIBRARY
15: TABLE AREA I
16: SLOW BOOT
17,18,19,20,25,26,27,28: TYPES 1,2,3,4,9,10,11,12 (RESPECT.) with access to SSGA
30: SUBSYSTEM GLOBAL AREA
21-24,29,31-99: UNUSED
(TYPE +80 IS USED TO DESIGNATE AUTO SCHEDULE AT STARTUP, BUT MAY ONLY BE ENTERED IN THE PARAMETER PHASE. +80 IS JUST A FLAG TO THE GENERATOR AND IS NOT STORED IN THE ID-SEGMENT.)

TABLE 5

7.1 Entry Point Availability Per Program Type

Because system entry points are not available to every program type, and entry points in the memory resident library are available to memory resident programs only, certain checks must be made to prevent illegal references. (See Table 6.)

System entry points defined in type 0 and 16 modules can be referenced only by the system and its tables (types 0,13,15 respectively), the slow boot configurator (type 16), by SSGA (type 30) and by type 3 (11,19,27) programs.

Library entry points defined by type 6 and 14 modules can be referenced only by the memory resident library and memory resident programs (types 1,9,17,25). Outside the memory resident area all type 6 and 14 modules are treated as type 7 modules so they will be appended to all programs referencing them.

For all programs, Type 7 modules will be appended to each module referencing them.

System Table Area entry points defined by type 13 and 15 modules can be referenced by anyone. If Table Area II was not included in the user map, cross-map loads must be used to reference it (note that it's write-protected).

SSGA modules (type 30) can be referenced only by modules specifically declaring SSGA access, types 17-20 and 25-28 and another type 30 module.

Checks must be done at load time rather than during relocatable input because of the parameter change capability.

"referencee"

referencer	0 SY	1 HR	2 RTD	3 BGJ	4 BGJ	5 SE	6 LIB	7 UTI	8 UTI	9 MF	10 RTD	11 BGD	12 BGD	13 TAB	14 LIB	15 TA	16 CONF	17 HR	18 RTD	19 BGD	20 BGD	25 HR	26 RTD	27 BGD	28 BGD	30 SSC
0	✓	-	-	-	-	-	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	-	-	-	-	X
1	X	-	-	-	-	-	✓	✓	✓	-	-	-	-	✓	✓	✓	-	-	-	-	-	-	-	-	-	X
2	X	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	X
3	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	-	-	-	-	X
4	X	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	X
5	✓[1]	-	✓	✓	✓	X	✓[2]	✓	✓	-	✓	✓	✓	✓	✓[2]	✓	-	-	✓	✓	✓	-	✓	✓	✓	X
6	X	-	-	-	-	-	✓	X	X	-	-	-	-	✓	✓	✓	-	-	-	-	-	-	-	-	-	X
7	X	-	-	-	-	-	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	X
8	X	-	-	-	-	-	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	X
9	X	-	-	-	-	-	✓	✓	✓	-	-	-	-	✓	✓	✓	-	-	-	-	-	-	-	-	-	X
10	X	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	X
11	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	-	-	-	-	X
12	X	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	X
13	✓	-	-	-	-	-	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	-	-	-	-	X
14	X	-	-	-	-	-	✓	X	X	-	-	-	-	✓	✓	✓	-	-	-	-	-	-	-	-	-	X
15	✓	-	-	-	-	-	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	-	-	-	-	X
16	✓	-	-	-	-	-	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	-	-	-	-	X
17	X	-	-	-	-	-	✓	✓	✓	-	-	-	-	✓	✓	✓	-	-	-	-	-	-	-	-	-	✓
18	X	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	✓
19	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	-	-	-	-	✓
20	X	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	✓
25	X	-	-	-	-	-	✓	✓	✓	-	-	-	-	✓	✓	✓	-	-	-	-	-	-	-	-	-	✓
26	X	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	✓
27	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	-	-	-	-	✓
28	X	-	-	-	-	✓	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	✓
30	✓	-	-	-	-	-	✓[2]	✓	✓	-	-	-	-	✓	✓[2]	✓	-	-	-	-	-	-	-	-	-	✓

✓ allowed references
X disallowed references
\- don't care references

Note 1 - allowed only when loaded with a type 3, 11, 19 or 27 main
Note 2 - all references to type 6 and 14 library routines are treated as type 7 utility references when outside the memory resident area.

Program Type References. TABLE 6

7.2 Relocation by types

type 15's in Table Area I

type 0 drivers in partition #1

type 30's in SSGA

type 0 system drivers

type 13's in Table Area II

type 0 SYSTEM

type 16 configurator

type 0 drivers in partitions #2 onward

type 6 & 14 memory resident library

memory resident's: type 1,(9,17,25)

Real Time DR's: type 2,(10,18,26) plus any type 5 segments

Background DR's: type 3,(11,19,27) plus any type 5 segments

Background DR's: type 4, (12,20,28) plus any type 5 segments

7.3 Libraries

7.3.1 Memory Resident Library (MRL)

The memory resident library contains all type-14 force-loaded modules, and all type 6 modules referenced by type 14 modules or memory resident programs. In order to pick up the type 6 modules, a pseudo-load of all memory resident programs is done.

Because library routines can be referenced only when in the memory resident map, they must be made available to disc resident programs. Therefore, after memory resident loading, all type-6 and 14 modules are demoted by /DEML to type 7 (utility) modules, so they will be appended to all DR programs referencing them. Type 30 SSGA modules are not demoted. If referenced before the MRL is relocated, they are treated as type 7's.

7.3.2 Relocatable Disk Resident Library

The relocatable library contains all type 7 modules. Note that these modules include all demoted type 6 and 14 modules, some of which may have been included in the memory resident library.

7.3.3 Library Entry Points list

The entry points available to the user are sent to the disc in three passes. See Appendix C for the physical disc layout and the entry formats.

PASS 1: all entry points defined by type 0 & 16 (system) modules are sent

PASS 2: all entry points defined by type-30 modules and by type 15 and 13 modules (Table Areas I and II respectively) are sent, in addition to LST types 2 (common), 3 (absolute), 4 (replace).

PASS 3: All entry points defined by type-7 modules (includes all type 6 and 14's).

The output of pass 1 starts on a sector boundary = DSCLB, and contains SYSLN entries. The output of passes 2 and 3 contains DSCLN entries.

7.4 Undefined Entry Points during generation

To recover from an undefined entry point, the user must enter a "DISPLAY UNDEFS,TR" before exiting from the Program Input Phase. These undefined externals will be listed on both the operator console (regardless of echo) and the list file. If undefs existed, a TR will automatically be done to the console for optional recovery.

The LST type for an undefined external will be 5 until that point where it becomes defined. (Note that a CHANGE ENT will do it even after exiting from the PIP). Once exiting from the PIP, it will be treated like a type 4 during program relocation; the value will be zero (a NOP). No error diagnostics will be printed when an unde-

8.0 SIZE RESTRICTIONS

The following limits for an RTE-IV System must be enforced due to the 32K (15 bit) logical address space of 21XX computers, base page ignored. Extended memory areas are not included. P (Area X) is defined as the smallest number of pages which completely contains area X.

SYSTEM:

p (Table Area I) + p (Driver Prtition) + p (Common) +
p (System Driver Area + Table Area II + System + Configurator) $\leq$ 31 pages

MEMORY RESIDENTS:

p (TAI) + p (DP) + p (COM) + p (SDA+ TAII) +
p (Memory Resident Library + Memory Resident Programs) $\leq$ 31 pages

where p (COM) and p (SDA + TAII) are optional

RT AND BG DISK RESIDENTS:

p (TAII) + p (DP) + p (COM) + p (SDA + TAII) +

p (a RT or BG Disk Resident program) $\leq$ 31 pages

LARGE BG DISK RESIDENTS:

p (TAI) + p (DP) + p (COM) + p (a large BG Disk Resident Program) $\leq$ 31 pages

where p (COM) is optional

8.1 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
System Driver Area
Resident Library
Memory Resident Programs (first one only)
Disk Resident programs

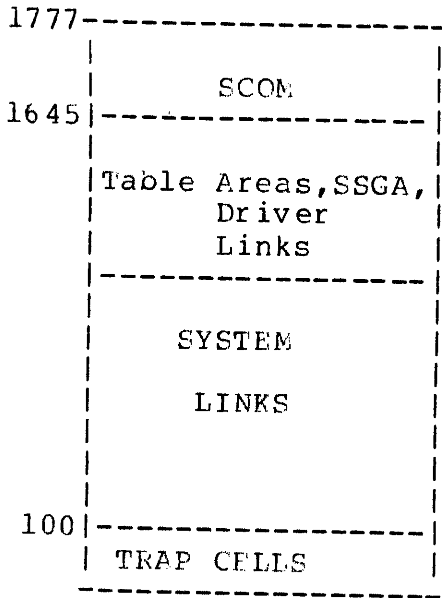
9.0 MISC AREAS

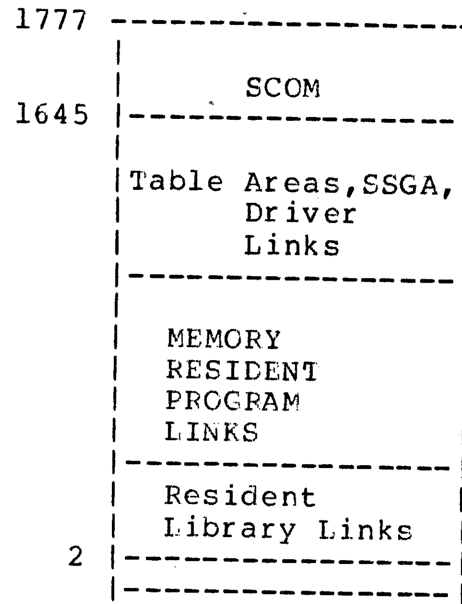
9.1 BASE PAGE

Table 7 describes the various base page formats. Only one (each) system and memory resident base page exists, but each disk resident program has its own copy of base page. The base page links used by a disk resident program are stored in the next disk 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 MRBI 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 disk sector following the last memory resident program's code.

The System Communication Area (SCOM) and all Table Area I, SSGA, Table Area II and drive links are resident in both system and user maps, the SCOM residin in BP locations 1777-1645 and the upper BP links from 1644 downward. After th track 0 sector 0 boot extension has been sent to the disk, 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 disk space. The system links (including configurator) always start at location 100 and grow upward toward the SCOM. The partition drivers links are not allocated until the PRD's have been relocated, so checks are done for overflow of these driver links into the system links. The system base page on disk is updated at the end of the system's relocation for the trap cells and system links. Note that trap cells referencing programs are fixed-up 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 disk 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.





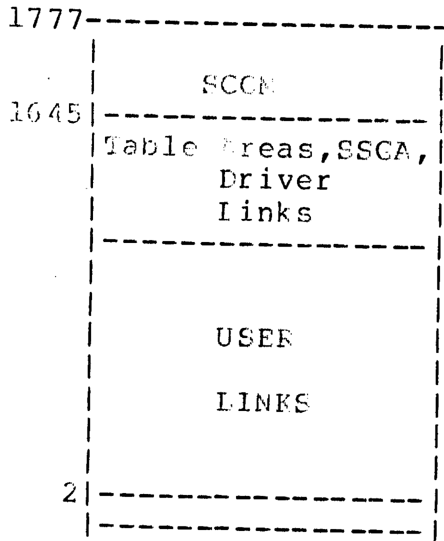


TABLE 7

9.2 SYSTEM COMMUNICATION AREA (SCOM)

The SCOM is built at the end of generation during final clean-up. 133 octal words below the label USRTR are initialized to 0 and overlaid as SCOM is built, transferred to the dummy base page, and then sent to the disk using /ABDO. The base page locations are set by the generator variables as follows:

1645	XIDEX	ID extension address of currently executing program	<-0
1646	XMATA	MAT entry address of currently executing program	<-0
1647	XI	address of index register save area	<-0
1650	EQTA	first word address of equipment table	<-AEQT
1651	EQT#	number of EQT entries	<-CEQT
1652	DRT	first word address of device reference table	<-ASQT
1653	LUMAX	number of logical units in DRT	<-CSQT
1654	INTBA	first word address of interrupt table	<-AINT
1655	INTLG	number of interrupt table entires	<-CINT
1656	TAT	first word address of track allocation table	<-ADICT
1657	KEYWD	first word address of keyword block	<-KEYAD
1660	EQT1	address of SAM#1	<-LWSYS+1
1661	EQT2	# words	<-SAM#1
1662	EQT3	address of SAM#2	<-LWSYS+1+SAM#1
1663	EQT4	# words	<-SAM#2
1664	EQT5	address of SAM#0	<-LWTAI+1
1665	EQT6	# words	<-DPADD-(LWTAI+
1666	EQT7		<-0
1667	EQT8		<-0

1670	EQT9		<-0
1671	EQT10		<-0
1672	EQT11		<-0
1673	CHAN	current DMA channel number	<-0
1674	TBG	I/O address of time base card	<-TBCHN
1675	SYSTY	EQT entry address of system TTY	<-SYSTY
1676	RQCNT	number of request parameters, less 1	<-0
1677	RQRTN	return point address	<-0
1700	RQP1	Addresses	<-0
1701	RQP2	of request	<-0
1702	RQP3	parameters	<-0
1703	RQP4	(set	<-0
1704	RQP5	for a	<-0
1705	RQP6	maximum	<-0
1706	RQP7	of	<-0
1707	RQP8	nine	<-0
1710	RQP9	parameters)	<-0
1711	SKEDD	address of system 'schedule' list	<-SCH4
1712			<-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

```
1717 XEQT   ID segment addr. of current program                 <-0
1720 XLINK  ID segment linkage                                  <-0
1721 XTEMP  ID segment temporary                                <-0
1722 XTEMP  ID segment temporary                                <-0
1723 XTEMP  ID segment temporary                                <-0
1724 XTEMP  ID segment temporary                                <-0
1725 XTEMP  ID segment temporary                                <-0
1726 XPRIO  ID segment priority word                            <-0
1727 XPENT  ID segment primary entry point                      <-0
1730 XSUSP  ID segment point of suspension                      <-0
1731 XA     ID segment A-Register at suspension                 <-0
1732 XB     ID segment B-Register at suspension                 <-0
1733 XEO    ID segment E and overflow reg. at suspension        <-0
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 interface card (PI)            <-PIOC
1740 IDSDA  disc address of first ID segment                    <-DSKSY
1741 IDSDP  position within sector of first ID segment          <-IDSP
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 FTORG 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

1755 TATLG negative length of track assignment table <- -(DSIZE+DAUXN)

1756 TATSD number of system disc tracks <-DSIZE

1757 SECT2 number of sectors per track on lu 2 (system) <-SDS#

1760 SECT3 number of sectors per track on lu 3 (aux.) <-ADS#

1761 DSCLB disc address of entry point library <-DSKLB

1762 DSCLN # of user available entry points in library <-LBCNT

1763 DSCUT disc address of reloc. disk resident library <-DSKUT

1764 SYSLN # of system entry points in library <-SYCNT

1765 LGOTK load and go: lu, starting track, # of tracks <-0

1766 LGOC current load and go track/sector address <-0

1767 SFCUN log source: lu, disc address <-0

1770 MPTFL memory protect on/off flag (0/1) <-0

1771 EQT12 <-0

1772 EQT13 <-0

1773 EQT14 <-0

1774 EQT15 <-0

1775 FENCE memory potect fence address <-0

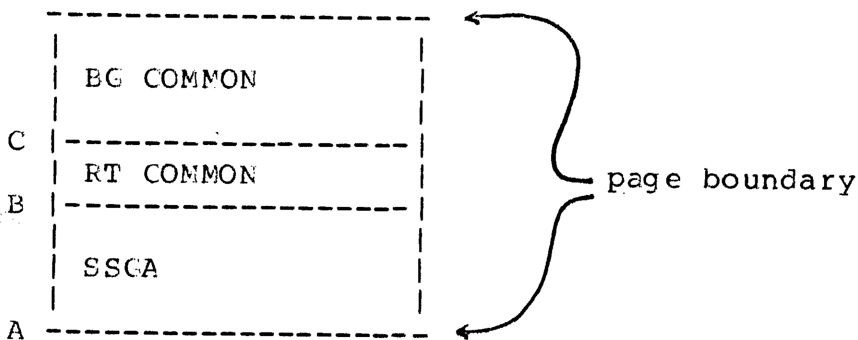
1776 <-0

1777 BGLWA last word memory address of BG partition <-LWASM

9.3 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 can "see" (has map entries for) the others, only the memory protect fence table can provide any protection (see its format under MPFT writeup).

The order of the three areas was chosen such that a hierarchical protection is preserved:



The memory protect fence will be placed at A, B, or C if a program is using COMMON.

When the IDENT's are scanned for ID segment allocation at the end of the PIP, the common sizes of each program stored in \ID4 bits (14-0) is used to set the maximum RT and BG common sizes, COMRT and COMBG respectively.

Starting 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 is displayed, the user is given the option of increasing it, and the starting address is displayed:

```
RT COMMON   XXXXX <------- decimal words
CHANGE RT COMMON?
NNNNN       0 means no change; GEN ERR 14 issued on an invalid response
RT COM ADD XXXXX <------- octal address
```

RTCAD is set to the RT common starting address from PPREL, and COMPT, the number of words, maybe updated; BGBND is set to the starting address of BG Common, PPREL+COMPT. 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<--------decimal words (new size)
CHANGE BG COMMON?
NN         0 means no change; GEN ERR 14 issued on an invalid response
BG COM ADD  XXXXX
BC COMMON XXXXX<----decimal words
```

BG Common size is increased in page multiples so COMBG has NN*1024 added to it, and is redisplayed.

9.4 CONFIGURATOR PROGRAM

The configurator program is a special system module of type 16. It has access to all the system entry points and is loaded immediately after the system in what will later be the beginning of 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 or a GEN ERR 18 will result and the generation will be aborted. The memory above 77577 must be reserved for the disc ROM loader. The last word of both the system code and configurator code must be saved (in LWSYS and LWSLB respectively) in order 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 the Table Area II entry point $SBTB which is the first word of the following 6 word table:

```
$SBTB  disc address of driver partitions
       # of pages for all driver partitions
       disc address of memory resident base page
       # of pages for memory resident base page
       disc address of memory resident library and programs
       # of pages for all memory resident library and programs
```

The values are stuffed into $SBTB when the generator sets the values of all the Table Area II entry points specified in $$TB2 at the end of the partition definition phase (see Section 12.8).

9.5 BOOTSTRAP AND EXTENSION

The generator builds both the track 0 sector 0 boot extension and moving head bootstrap loaders for either the 7900 or 7905/7920 discs. Generator segment 1 builds the 7900 bootstrap loader whereas segment 7 builds it for the 7905 etc. The generator stores the system subchannel disk specifications in the bootstrap loader (i.e., first track, # of tracks, starting head, # surfaces, etc.), and for the moving head bootstrap loader, configures the disk 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. The generator also sets the following values in the boot extension:

Values	
TBASE U#NIT B#MSK SKCMD R#CMD	7900 only
BHD# #HDS WAK SKCMD AD#RC R#CMD S#TAC	7905/20 only

10.0 TABLE AREAS I AND II

The generator-built track map table ($TB31 or $TB32), EQT's and extensions, DVMAP table, DRT, INT and all type 15 modules will be loaded and/or stored in Table Area I (in that order). Table Area I exists in all maps. The user-available entry points to system code will be loaded into Table Area I from the type 15 module $$TB1. Note that all user-defined track map tables will have to 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 map
$MPFT table
Track Allocation Table
$$TB2 entry points
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. Since Table Area II is included only in the system, memory resident (optional), and type 2 RT and type 3 BG program address spaces, those type 4 BG programs wanting to access any Table Area II entry points must do so via cross-map loads.

All external references from the Table Areas are resolved thru 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.

10.1 EQUIPMENT TABLE (EQT), DEVICE REFERENCE TABLE (DRT), & INTERRUPT TABLE (INT) SIZES

The maximum number of EQT and DRT entries is 63 and 255, respectively. Since both the EQT and DRT entries are sequentially prompted, the generator will issue a GEN ERR 35 for all entries past the 63rd or 255th until a /E is encountered. The size of the DRT is always twice the number of LU's 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
--------------------------------------
| Subchannel |          | EQT number  |
--------------------------------------
  of device               of device
```

The INT contains entries for each channel from 6B thru 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 thru 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 2's 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 types 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).

10.2 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's, 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's 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 to be stored
word 2: instr. 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 so specified by an S or M in their EQT definitions as an SDA type. Those drivers 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 in addition to or in place of the S may be specified.

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 the driver's IDENT word 8, bit 15 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, then the new type must match that of the old. i.e., bits 14 and 13 of the current entry must match the values to be set by the new entry, otherwise a GEN ERR 23 would be issued and the EQT would have to be redefined.

The system disc driver cannot reside in SDA. When an EQT's select code matches the "CONTROLLER CHNL?" response, the system disc EQT is assumed and a check is made to make sure that SDA was not declared for this driver - or a GEN ERR 23 occurs.

The first half of the driver map table DVMAP is dynamically built in a buffer as the EQT's are defined. DVMAP consists of two consecutive chunks of size CEQT (the number of EQT's). After all the EQT's 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 user by RTIOC. A 64-word buffer (maximum # of EQT's) 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  14               8   7                    0
-----------------------------------------------
| 1 | IDENT index of driver                    |   for a partition resident
-----------------------------------------------    driver (PRD)
```

or:

```
 15                                   0
--------------------------------------
| 1 |                           | 1 |   for a System Driver Area driver
--------------------------------------
  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:

```
            9                         0
--------------------------------------
|         | physical starting page    |
--------------------------------------
             of driver partition
```

When a PRD is relocated into a partition, all the 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.

10.2.1 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.

10.2.2 DRIVER PARTITIONS

The defaulted driver partition size is two pages - large enough to hold any HP partition-resident driver. As many drivers are relocated into 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 a lot of leftover page space can be used. For partition-resident drivers greater than 2 pages, the DP size must be overridden in order to accomodate it. Otherwise, if the driver overflows a DP at relocation time the generation will be aborted 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?
```

A 0 response implies no change, otherwise the new size must be >= DPLN and less than 17 (a blue-sky estimate). If an invalid response is entered then a GEN ERR 01 is issued and the query re-prompted. The new value of DPLN is used to set the Table Area II entry point $DLTH. The last word occupied by Table Area I is founded 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 "sent" to the disc by updating the relocation addres PPREL. When \ABDO is called the next time, 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 #) 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 begins at the beginning of the IDENT table stopping when a driver's size specified in \ID8 of its entry indicate 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, otherwise the scan through the IDENT table is continued.

Note, that however, that a driver may still overflow a partition. This may happen when referenced subroutines are appended to the driver during relocation. Upon overflow, the violating driver is 'backed-up' over an error (actually a warning) is issued, and the IDENT table scan is continued, searching for another driver that will fit. 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 up 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 #1 zero-fill is done to the last word of a DP, but for the other DP's zero-fill is done only to the last word of the last page used. With this feature pages may be saved where one or more complete pages of a DP are unused.

For each new DP, the scan is then begun at the beginning of the IDENT table for the next unrelocated partition-resident driver. If none exist, the driver partitions are done. The fixup table is cleared before the memory resident library is built.

For driver partitions #2 onward the \ABDO specificatin 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 and physically in pages above the system area. From then on, when each new DP is started, the DP map's 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 it's starting physical page is set: PAGE#<---PAGE# + number of pages required by previous driver partition (<DPLN). PAGE# is initially set, for driver partition #2, to $ENDS which is the physical page immediately following SAM#1. (See the physical memory map in Appendix D.) 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 made during PRD relocation for overflow of links into those occupied by the system -A GEN ERR16 results and the generation is aborted.

Note, a user-entered disc Track Map Table (e.g. $TB31 or $TB32) must be defined as a type 15 module so it will be placed in Table Area I for access by drivers.

10.3 ID SEGMENTS AND EXTENSIONS

During the construction of Table Area II, space is reserved for long ID segments (33 words long), mem. res. ID segments (29), short ID segments (9), and ID extensions (3). Long ID segments are allocated to real-time and background disk resident programs; mem. res. 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 thru the queires:

# of BLANK ID SEGMENTS?
# of BLANK SHORT ID SEGMENTS?
# of BLANK ID EXTENSIONS?

A GEN ERR 60 is issued if the total number of long and mem. res. ID segments is >254. If more than 254 long ID segmented are required before any blanks are requested, then the generator aborts after giving the GEN ERR 60. A GEN ERR 01 is given if the number of ID EXTENSIONS exceeds the number of long ID segments.

The keyword table and ID extension table ($IDEX) have one word allocated for each ID segment and ID extension, respectively - plus one "stop" word equal to zero. The keyword table entries are set to the ID segment addresses as the ID segments are being built, or during final cleanup for the blank ID segments generated. The ID extension table and the ID extensions preceed the keyword table and ID segments. When built, the ID extension table ($IDEX) is initialized to the addresses of the ID extension entries (3 words each). See the EMA section for the description of the values stored in the ID extension entry.

ID segment entries are built as follows:

ID SEGMENT WORDS SET DURING GENERATION

Word	Contents
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 gives NAME1 & 2
13*	ID2 gives NAME3 & 4
14*	ID3 (15-8) gives NAME5; ID6 (2-0) gives TYPE
15	optionally set bit 0 if the scheduled program
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	** (see below)
22*	low main address from PPREL
23*	high main address from TPREL
24*	low BP address from PBREL
25*	high BP address from TBREL
26	main disk address from DSKMN
27*	0
28	ID EXT# & EMA SIZE - see EMA section
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)

Words 0–25 (bracketed): MEMORY RESIDENT ID SEGMENT plus words 30, 31, 32

* short ID segments

** RP bit 15 may be set during the Partition Definition Phase.
 # pages required (14-10) set at end of main program load by IDFIX; for EMA programs includes MSEG size; may be changed for non-EMA programs during the Partition Definition Phase.
 MPFI (9-7) set at end of main program load by IDFIX.
 Partition # (5-0) may be set if assigned during PDP.

TABLE 8

12.3 PARTITION 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 (DPAPF, 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 thru its absolute memory address, updating the code on the disk via \ABDC. See Table 9 for the format of a MAT entry.

The user is prompted for the definition of each partition starting with "PART 01?", and stopping when a "/E" is entered by the user. The physical pages will be sequentially allocated to the MAT entries and the first -1 link will thus indicate the end of the defined partitions. The user enters the number of pages, partition type (either RT, BG 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 a RT partition, bit 15 is set in MAT word 5 (cleared for BG partitions). If a reserved partition bit 15 is set in MAT word 4. If the partition size is greater than the maximum addressable size MAXPG, then the user is asked if they want to define subpartitions. A NO response simple results in a large unchained partition being defined. If the user responds YES then that partition becomes a mother partition with bit 15 set in word 3 of its MAT entry and the MAT subpartition link word (6) of the mother partition is initialized to point to itself. At this time only can subpartitions for a mother partition 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 then 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 will be issued and the partition must be redefined, it can however be larger 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 - a GEN ERR 46 results on that subpartition definition causing the overflow, and that partition must be redefined. If the type is RT or BG, then we've left the subpartition mode and are proceeding as normal.

MAT ENTRY FORMAT

```
          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 |                 |  NUMBER PAGES              |
         --------------------------------------------------------
     5   |RT |                                       | Rd Comp|
         --------------------------------------------------------
     6   |             Subpartition Link Word                   |
         --------------------------------------------------------
```

M = Mother partition bit
D = Dormant bit
R = Reserved bit
C = Chain in effect bit
RT= Real-time partition bit
Rd= Read completion

TABLE 9

12.3.1 The following sequence occurs for a partition definition:

A. If NEXTP=MAXPT then set the XX in the prompt to blanks.

B. Send the prompt "PART XX?" for the definition of partition XX and get the response. If "/E" is entered, the generator proceeds to the partition cleanup at Q. If NEXTP > $MNP then no more MAT entries can be entered and the generator issues a GEN ERR 49 and goes back to B.

C. Retrieve the partition size, subtract 1 for the base page, and store in DPSIZ. DPSIZ must be between 1 and 1024 pages, else issue GEN ERR 45 and go to A.

D. Retrieve the partition type (RT,BG or S); if neither RT,BG or S then issue a GEN ERR 46 and go to A (after clearing SUBS?).

E. If S then SUBMD must = 1 indicating subpartitioning enabled, else issue a GEN ERR 46 and go to A.

F. If DPSIZ+1> MOMSZ (size of mother partition) then issue a GEN ERR 56 and go to A.

G. If DPSIZ + SUBT+1 (current subpartition page total) > MOMSZ then issue GEN ERR 56 and go to A.

H. Increment # of pages covered by subpartitions, SUBT<--SUBT+DPSIZ+1; set type of subpartition to that of mother, DPTY<--MOMTY. Go to J.

I. If RT or BG then its a regular partition (i.e., non-subpartition) and SUBMD is set to 0 (may have already been in regular mode). Set DPTY to 1 for RT so bit 15 of word 5 can be set.

J. If DPSIZ > MAXPG (largest logical partition size) then this partition may be split into subpartitions. Subpartitions themselves may be >MAXPG, but they cannot be further subdivided. Therefore the subpartition mode flag SUBMD is checked; if = 0 then the generator is in regular mode and this partition may be subpartitioned - so SUBS? is set to 1 indicating that the user is to be asked if they want to define subpartitions at L.

K. Retrieve reserved flag. If one entered, set bit 15 for word 4 (DPRSV<--0, -1 otherwise).

L. If SUBS? = 0 then goto N, else prompt the user "SUBPARTITIONS?".

M. If NO then go to N. If YES: enable subpartition mode SUBMD<--1; store address of current (mother) MAT address in MOMAD; save mother partition size for subpartition checking, MOMSZ<--DPSIZ+1; clear subpartition total, SUBT<--0; 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).

N. Build the new MAT entry (words 0 & 3 are completed during partition cleanup).

word 0: set to 0 to indicate a defined entry.
word 3: DPMOM is used to (optionally) set bit 15 if a mother partition
word 4: DPRSV is used to (optionally) set bit 15 if a reserved partition, and DPSIZ is stored in bits 9-0.
word 5: DPTY is used to (optionally) set bit 15 if a RT partition
word 6: if SUBMD=1 then set to MOMAD, else 0. This will set the SLW (Subpartition Link Word) to point to itself if the mother partition, or to the mother MAT entry if a new subpartition at the end of the chain.

O. If SUBMD=1 and SUBS?=0 then at least one subpartition has been defined. The current subpartition must then be linked to the previous MAT entry (which is either the previous subpartition in the chain or the mother partition). Since CURMAT is the memory address of the current MAT entry, then $(CURMAT-1)<--CURMAT.

P. SUBS?=0, DPMOM<--0. Bump NEXTP. Go to A.

Q. Partition Definition Cleanup. The MAT is scanned, summing up the individual partition sizes, until the first undefined entry is found (link word 0 = -1) or the end of the table is reached. Only the regular and mother partition sizes are included in the total, and 1 is added to each of these sizes because the base page was not included in the size stored in word 4. Subpartition sizes are not included, their pages having already been included in the mother partition; a subpartition MAT entry is detected by word 6 (SLW) being nonzero and the mother bit (15) not being set in word 3. If the total number of pages occupied by the defined partitions (DPTCT) does not equal the number available (DPARE), then a GEN ERR 53 is issued and all the partitions must be redefined.

12.3.2 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 thru 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., respectivel 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 then it's the $RTFR list, and the remaining go into the $BGFR list. The type of list being threaded is irrelivant 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 sub-partition entires 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.

12.4 MODIFY PROGRAM PAGE REQUREMENTS

The IDENT entry for the named program is retrieved; a GEN ERR 48 is issued if the program name can't 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's ID segment is built, the keyword offset is stored in the program's IDENT entry word 8 bits (7-0). The routine IDFND retrieves the program's ID segment address by going thru \ID8 and the keyword's value stored on disc. Before the program's ID segment word 21 can be updated, the new page size must be verified. The program's 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) then a GEN ERR 51 results. A program's page requirements, stored in its 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 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.

12.5 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's 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 (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.

12.6 MEMORY PROTECT FENCE TABLE

The 6 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 10 - MPFT

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 BGEND (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.

12.7 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 having one word per physical register. The map is built as follows:

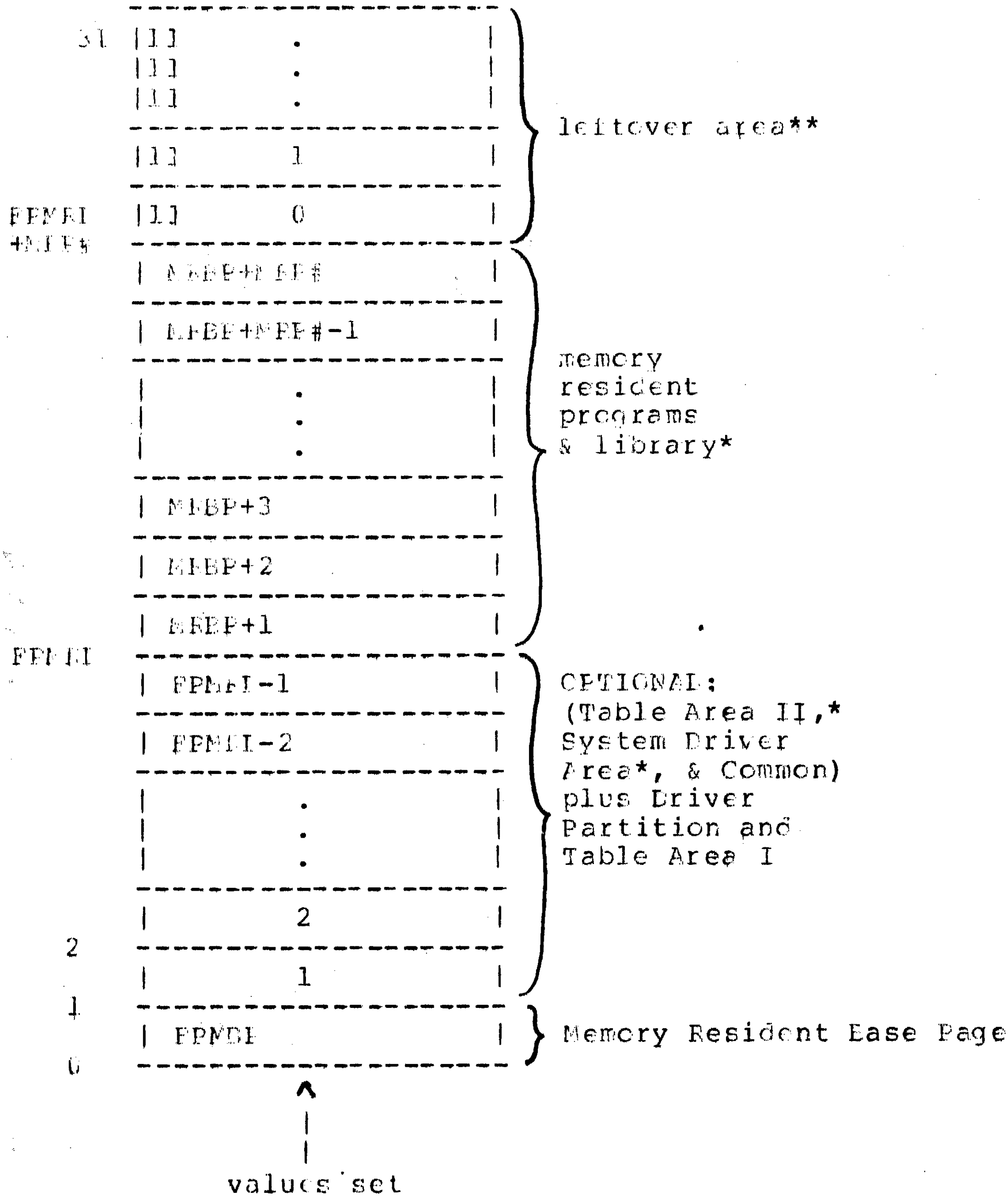


TABLE 11 - Memory Resident Map

*System Driver Area, Table Area II, and the 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 FPMBL. Words 1 thru FPMBL-1 are thus set to their logical and physical page addresses, 1 thru FPMBL-1. If the System Driver Area and Table Area II are to be included in the map (MRTA2=1) then their pages are write-protected. MRP# contains the number of pages occupied by the memory resident library and programs. The map words FPMBL thru (FPMBL + MRPGS-1) are thus set to their corresponding physical pages, MRBP+1 thru (MRBP+MRPGS). The library pages (FPMBP+1 to FPMBP-1) are always write-protected. The remaining map words (FPMBL+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 MFMP entry).

12.8 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 5 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 DP programs without common

$PLP, load point address for privileged DP programs

$LEND, last word +1 address of memory resident library

$BLLO, negative lower buffer limit

$BLHI, negative upper buffer limit

when done setting the above values 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
```

13.0 ERROR PROCESSING

There are two classes of errors that occur during generation: FMP ERR's resulting from files being accessed through FMP calls, and GEN ERR's 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 the lu 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 them 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 the 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).

13.1 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, then it implies that we have 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.

13.2 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 \FMER. 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-Reg. From the DCB words 0 and 1, the file directory entry address or the lu is retrieved. 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 then 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-Reg DCB address is zero indicating a check following an OPEN or CREAT call, then 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 who 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 occured.

13.3 ABORTIVE TERMINATION

13.3.1 \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 down the problem. After outputting the message, \TERM is called for clean-up before termination. The abort may result if there exists a problem 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 imcomplete disc swap of a table block - this could be an actual generator problem or it could be a hardware problem. Past experience recommends one to check the hardware first on all GEN ERR 00's. Obviously this error should never occur - so only strange conditions will cause it.

13.3.2 \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, \FDCB, \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.

13.4 MISC. ERROR PROCESSORS

\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 where the transfer of control is done to the console. The caller of \INER then reissues the questions for the corrected response from the operator.

\IRER calls \GNER for the irrecoverable errors 07, 12 and 21, followed by a call to \TERM to perform clean-up and abortion.

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 was entered, or when an FMP ERR-XX FNAME occurred on a file referenced in a RELOCATE command. NXTCM then prompts (-) for the next command.

13.5 ERROR SUSPENSIONS

The generator detects two error conditions which 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 6 scratch tracks and they are not available, then it issues the message "GENERATOR WAITING FOR TRACKS", and reissues the EXEC 4 call with the wait bit set. The same sequence of operation occurs 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.

13.6 ANSWER FILE ERRORS

When doing transfers within TRCHK, special processing must be done for PUSH/POP errors. At POPER, 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" to be done on return in some cases. When an invalid lu was specified on a PUSH, at TR3 RECOV is again called before issuing the GEN ERR 20; the same holds true at TR4 when an error occurred on the new input file, only here we have to save the error code while RECOV is being called.

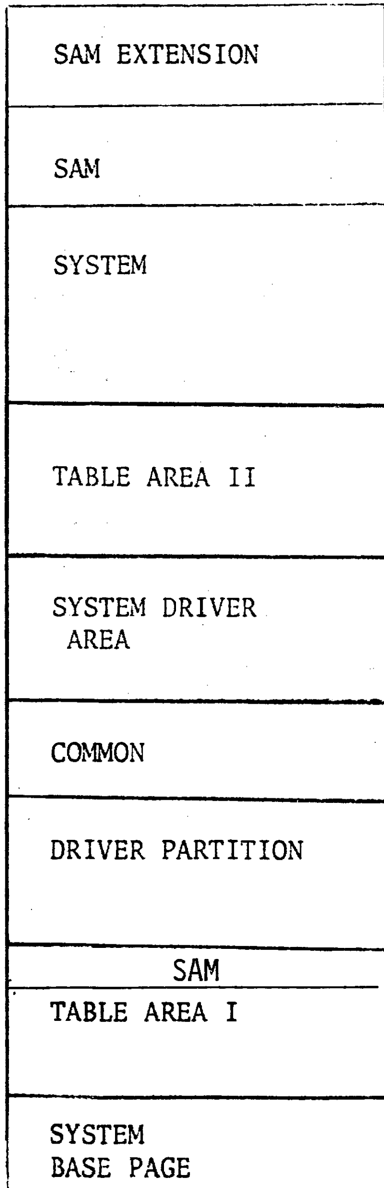
When an invalid lu or file was specified in the turn-on parameters, STRT2 issues its own errors rather than call \GNER or \CFIL before the answer file and IACOM have been established. Once the lu of the operator console has been determined (default is 1) the Ascii of that lu is stored in the "TR,XX" message to be used later with all "TR,ERRLU" calls to TRCHK.

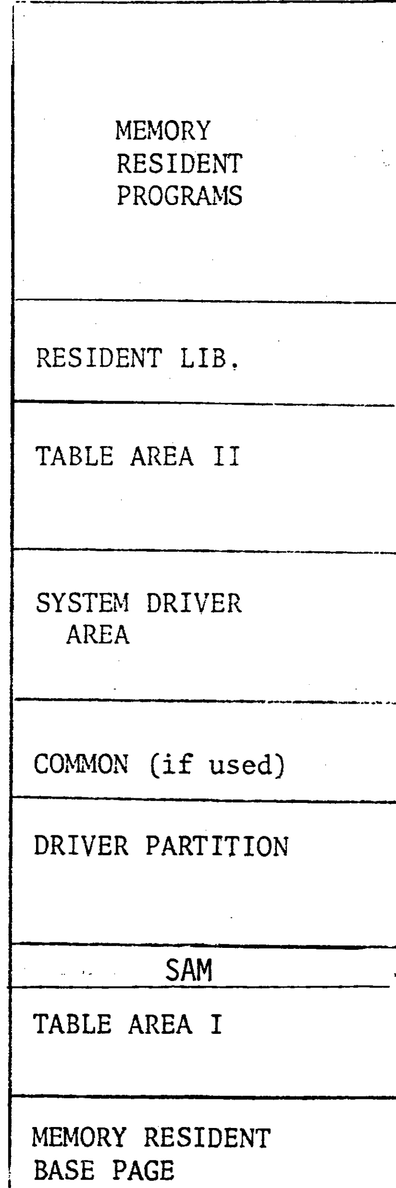
13.7 DRIVER PARTITION OVERFLOW

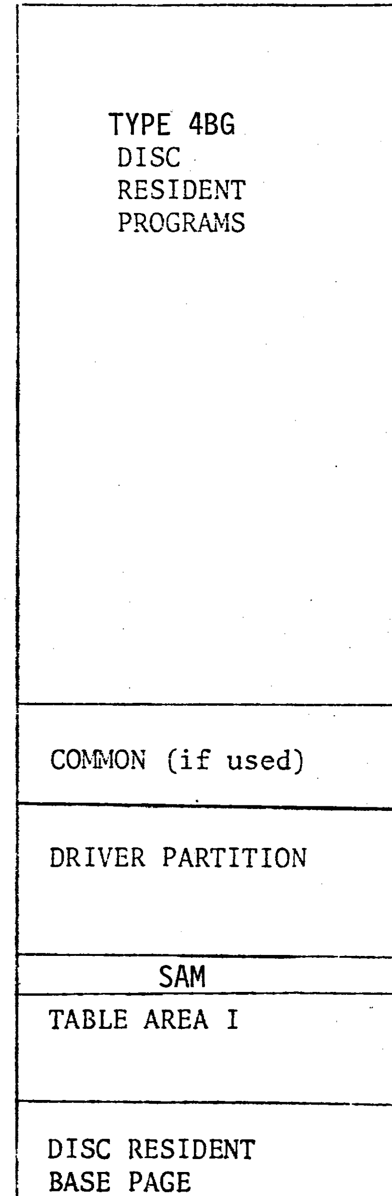
When multiple drivers are being relocated into a driver partition and a driver overflows the logical memory space reserved for the DP, a warning message of the form:

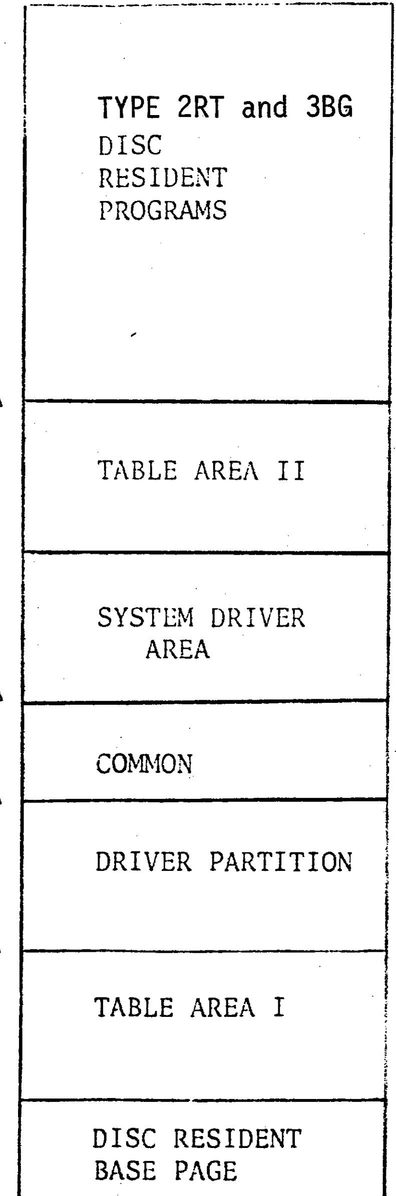
´DRIVER PARTITION OVERFLOW´

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.





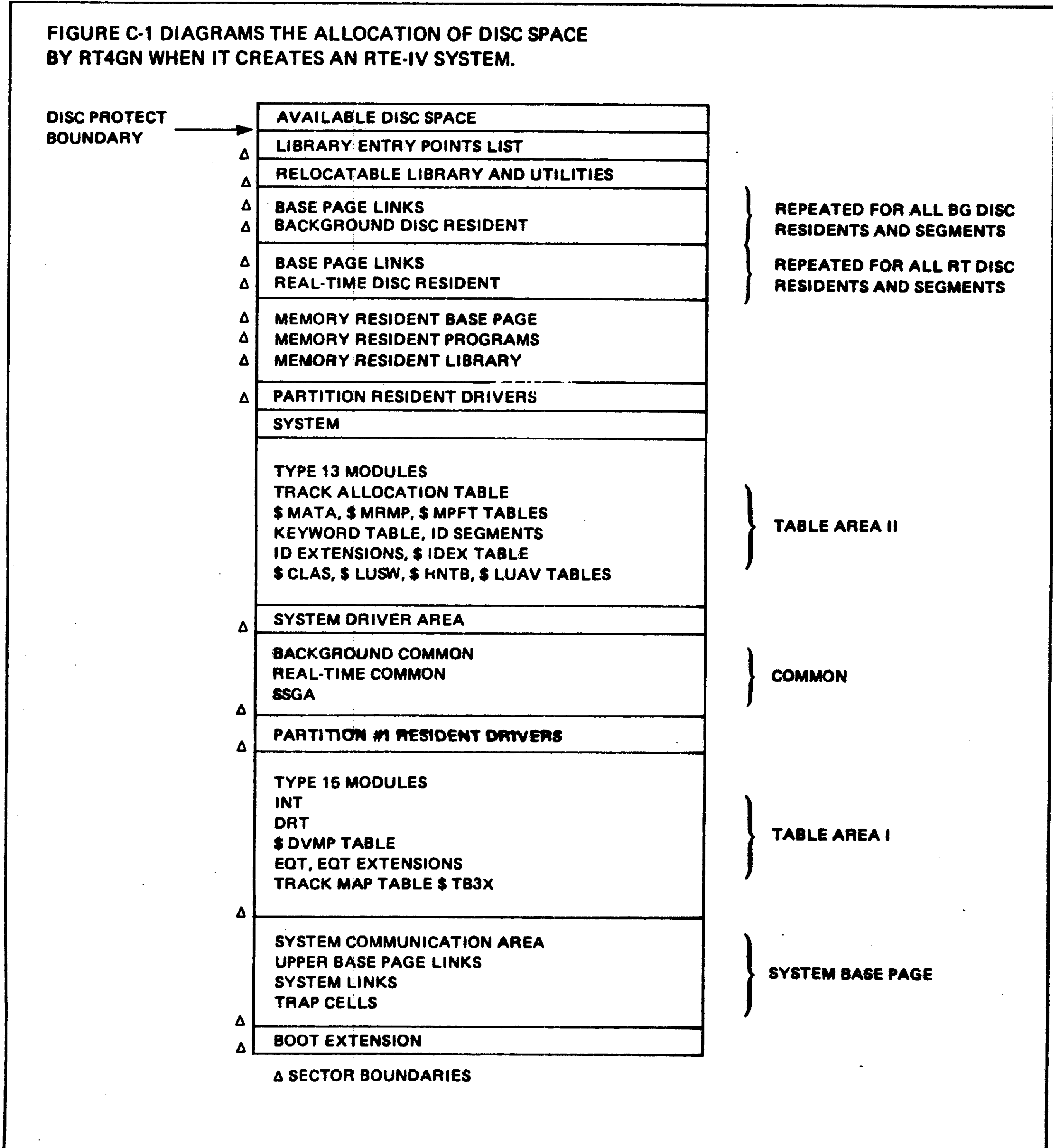




Δpage boundaries

APPENDIX A

DISC LAYOUT OF AN RTE-IV SYSTEM

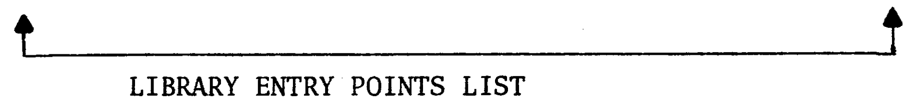


DSCUT DSCLB SYSLN DSCLN end of system

Disc Resident Relocatable Library	System Only Entry Points	User-Available System Entry Points	User Entry Points	Free Track Area
all type 6,7,& 14 programs	type Ø,16 modules entry points	type 13,15 & 30 modules entry points; all ABS,RP, COMMON, entry points	all type 6,7, & 14 modules entry points	

track boundary

NOTE: SYSLN & DSCLN contain the number of 4-word entries



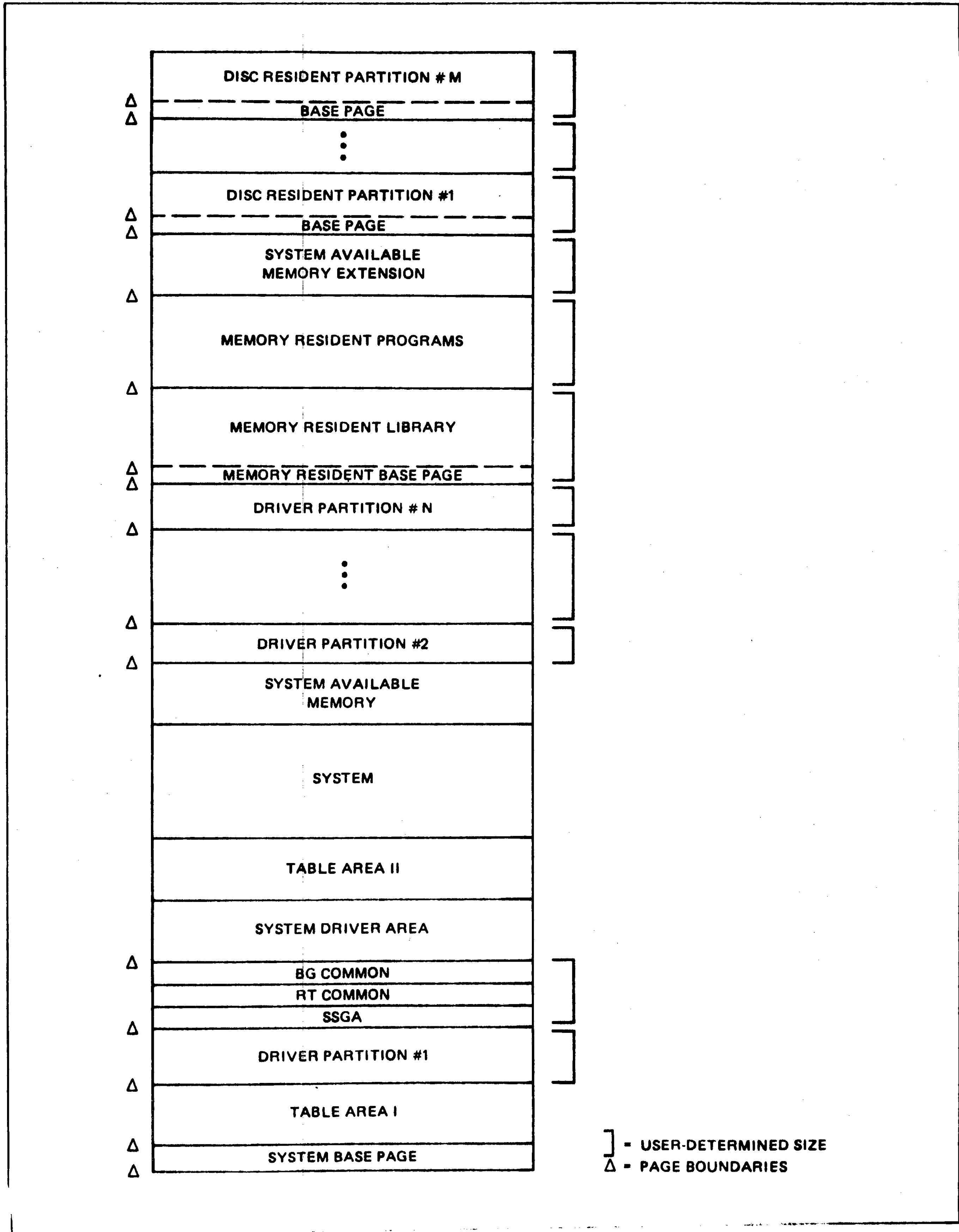
LIBRARY ENTRY POINTS LIST

FORMAT:
- word 1 name 1,2
- word 2 name 3,4
- word 3 name 5, flag bits
- word 4 value

flag bits:

000	memory resident
001	disc resident
010	common
011	absolute
100	replace

▲ sector boundaries



APPENDIX D. **RTE-IV PHYSICAL MEMORY ALLOCATION**

MTM TECHNICAL SPECS

MN
1/18/78

A number of changes have been made to the modules that interface to MTM and several new system library routines have been created. The new system library routines were sufficiently discussed in the MTM ERS. This section discusses the changes to the scheduler, MESSS, and the flow of control between PRMPT and RPN.

SCHEDULER

Two changes have been made in the scheduler. The first is in the $TYPE routine, the second is the programmatic schedule request (EXEC 9, 10, 23 and 24) at $MPTS.

$TYPE accepts operator commands and is the system console's interface to $MESS which processes these commands. $TYPE now screens the operator input commands for 'ON' or 'RU'. If either of these commands were entered and were successful, then session word 3 (ID segment word 32) will have a -1 placed in it.

Session word 3 will be used two ways. If the sign bit is clear the program is in the session mode. If the sign bit is set, the the word has the negative LU number of the console at which the program was scheduled. Thus $TYPE places a -1 in the word meaning that the program is not in session and was scheduled at LU 1.

Another change in the scheduler was made at $MPT5 which handles EXEC schedule requests. Processing was added to propagate session word 3 from father to son. The father was scheduled from and thus which LU to issue messages to.

MESSS

MESSS is a system library routine that allows users to interface to $MESS in the scheduler. MESSS is used by many programs to schedule programs when the 'ON' or 'RU' command is issued at an MTM terminal. MESSS now performs the same operation as $TYPE, that is, if the 'ON' or 'RU' was entered and was successful (no returned message from $MESS) then the program to be scheduled will have its session word 3 set to the negative MTM LU number.

PRMPT - RPN

The guts of MTM is really two programs, PRMPT and RPN. PRMPT is a program that is scheduled by interrupt. This means that PRMPT must be relocated at generation time and then one entry XX,PRG,PRMPT be made in the Interrupt Table phase of generation for every select code address, XX, where the user wishes MTM terminal handling.

PRMPT is the program that issues the XX> prompt and issues a class I/O read on the terminal. RPN does the class get and handles any input typed into the terminal. The one exception to this is for MTM scheduling a copy of FMGR. This is PRMPT's responsibility.

The flow of control in MTM actually starts before the interrupt. As mentioned PRMPT must be set up at generation time. The terminal must also be enabled before MTM will do any processing. Typically a terminal is enabled with the :CN, LU#, 20B command. It may be disabled with the :CN, 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's in 2'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. MTM 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.

PRMPT

PRMPT takes the address of word four of the EQT and calls TRMLU. TRMLU is a new system library routine. Its responsibility is to match up the EQT address to an interactive LU number. This is done by comparing the passed EQT address to the contents of the device reference table (DRT). Recall that the lower six bits of the DRT has the ordinal number of the EQT associated with that LU. TRMLU also insures that the LU number returned is an interactive LU. This is done by checking for driver type. If the driver is DVR00, DVR05 and subchannel zero, or DVR07 then the device is interactive.

When TRMLU returns PRMPT has both the LU number and the EQT. Checks are made to insure that the EQT and LU are up. The availability bits are checked in the EQT and the sign bit is checked in part two of the DRT to insure that the LU is up. A check is also made to see if the CRT LU is loccked. If the LU is locked bits 10-5 of the DRT will have the resource number (RN#) of the lock. If that field is non-zero, then the LU 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.

Next a check is made to see if a FMGXX exists, if so the prompt XX>FMGXX is sent to the terminal and FMGXX is scheduled with the list device set to XX. The schedule request uses the string pass feature to send FMGXX to the HI file before control is transferred to the terminal. Lastly PRMPT reenables the terminal and terminates saving resources.

If FMGXX does not exist or is busy, then the prompt XX> is issued, a class read is performed on terminal LU XX, RPN is scheduled passing the class number. In the case of DVR00 and DVR05, the terminal is disabled to avoid mutiple prompts from being written. For DVR07, an edit mode control request is made.

PRMPT does perform one other task. After the very first successful class read, which also requests the class number, PRMPT saves the returned class number in $MTM in Table Area l. This insures that aborts of PRMPT or RPN do not also lose the class number.

RPN

RPN is the MTM module that does class gets with wait. That is, it receives all input to the terminal, screens it, and passes it on to the operating system via direct call to the operating system routine $MESS.

When RPN is first scheduled, it picks up the class number and tracks down the ID address of FMGR, SMP, and D.RTR for later use.

RPN then performs a class get to get the input data. This get is performed over and over again so that RPN is always get suspended.

The system reschedules RPN whenever there are any input operator commands to process. RPN, on reschedule performs the same EQT, LU, and LU lock checks as does PRMPT. If I/O to the CRT is possible, execution continues. If the EQT or LU is down, the request is ignored and RPN goes back to the class get to suspend itself.

If I/O is possible then three commands are screened. BR, AB, and FL commands are handled locally in RPN. However, if no FMGXX exists only the FL command is honored. In this case, BR and AB are just passed on to the system.

If the input command was not AB,BR, or FL, then the command is sent to the operating system module $MESS. When $MESS returns any messages returned from the system are set to terminal XX with a class write call. This insures buffered writes to unbuffered devices. Lastly, the terminal is reenabled if the driver was DVR00 or DVR05 and a class get is performed so that RPN may suspend itself.

NOTE that RPN does class gets to retrieve its own class write information and PRMPT's class read information.

ASSEMBLER

2. Introduction

There are two major changes made in the RTE-Assembler. A pseudo opcode EMA was added to the assmebler. This opcode allows the user to declare an external memory array. Next, I/O from and to LS/LG areas is replaced by I/O from and to file manager files using the compiler library. List output can also be directed to a file manager file. The cross-reference generator XREF is also changed to do file I/O.

3a. EMA Pseudo Opcode

The EMA instruction is defined as:

Label EMA EMA size,MSEG Size.

EMA is assigned 5 as an opcode identifier. In pass 1 of the assembler, the processing for the EMA instruction is done in the EMP processor. Here EMCNT flag is checked to determine if it is 0. If this flag is non-zero, then another EMA instruction was encountered previously. Since only one EMA instruction per program is allowed, an 'IL', illegal instruction error message is printed for the second 'EMA' instruction encountered. The CHOP routine is called next to evaluate the two operand values. EMP checks to make sure the two values returned by CHOP are absolute, a 'UN' - undefined symbol error is printed otherwise. The EMA size must be positive and less than 1024. The MSEG size must be positive, less than EMA size (unless EMA size is 0) and less than 32. If any of the above conditions are not met, 'M'- illegal operand error is printed. An EMA instruction must have a label. If a label is not present an 'LB', label not present error message is printed. The symbol type assigned to the EMA label is 4 - the same as that for an external symbol. The undefined bit, bit 15 of word 1 of the symbol table entry is set to distinguish between an external symbol and an EMA label whenever necessary. The label for an EQU to EMA label is given the symbol type 5 with the undefined bit set. The starting address of the external memory array is the beginning of the first page in free available memory and can be defined only at load time.

The assembler creates a special 7-word EMA binary record with the record identification number as 6. Refer to the RTE manual for a description of the EMA binary record. This binary record is set up and output at the beginning of pass 2, just after all the EXT binary records are output. The relocation indicators in the DBL records for instructions using EMA label are: 4 for instructions using EMA label and 5 for EMA label with offset. The assembler does not make a distinction between EXT and EMA symbols while processing the memory reference instructions.

APPENDIX:

OPCODE Table Format:

Each entry has the following format:

<table>
<tr><td>1st CHAR</td><td>2nd CHAR</td></tr>
<tr><td>3rd CHAR</td><td>CODE</td></tr>
<tr><td colspan="2">INSTRUCTION FORMAT</td></tr>
</table>

Code is one of 64 (8-bit) identifiers used to process the OPCODE during assembly. Instruction Format is the instruction format or the location of the processor for the opcode.

SYMBOL TABLE

Contains:

a. Labels
b. External symbols
c. COMMON names

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>U</td><td colspan="3">WORDS</td><td>E</td><td colspan="3">TYPE</td><td colspan="8">1st CHAR</td></tr>
<tr><td colspan="8">2nd CHAR</td><td colspan="8">3rd CHAR</td></tr>
<tr><td colspan="8">4th CHAR</td><td colspan="8">5th CHAR</td></tr>
<tr><td colspan="16">VALUE</td></tr>
</table>

U = undefined bit: 0 = Symbol defined
1 = Symbol undefined or EMA (symbol type 4 or 5)

WORDS = Number of words in entry (2-4)

E = Entry point bit: 1 = Symbol has been declared an entry point
0 = Not an entry point

TYPE = Symbol Type:
0 = Absolute
1 = Relocatable
2 = Base Page Relocatable
3 = COMMON name
4 = External or EMA
5 = Label equated to External symbol or EMA label
6 = Code replacement (ENT)
7 = Literal

VALUE = Symbol value:

a. Absolute value (type 0)
Value relative to relocation base for types 1, 2, 3
External or EMA symbol ordinal (types 4, 5)

b. The value will contain the location of the literal relative to the end of the main program at object time.