Memory Management Extensions to the SRI Micro Operating System for PDP-11/23/34/35/40

IEN 136

1st MAY 1980

S.R. Wiseman
B.H. Davies

ROYAL SIGNALS & RADAR ESTABLISHMENT
North Site, Leigh Sinton Rd., Malvern, Worcs., UK.

CONTENTS

PART 3 - HOST MACHINE ASPECTS

0. Introduction

This document describes the requirement, design and implementation of memory management extensions to the Stanford Research Insitute's Micro Operating System (MOS) for PDP-11 minicomputers. The document is divided into three parts. Part 1 covers the rationale behind the way in which the memory management facility of the PDP-11 is used. The results of the work fall naturally into Parts 2 and 3. Part 2 describes the modifications to MOS itself and the debugger which are independent of the operating system of the host machine on which the target system is generated. Part 3 describes the Linking and Loading modules which are dependent on the operating system of the host machine. The host machines used at RSRE are PDP -11/34 and 40s running RSX-11M v3.2. Also we have provision in our standard MOS for writing processes in a high level language, CORAL 66, which requires the use of an auxiliary stack pointed to by R0. This facility may be excluded from the extended MOS by the use of a configuration switch.

PART ONE

SYSTEM DESIGN

1.1 Requirement

For a number of planned real-time software projects the 28k words of code and data space accessible without memory management in the PDP-11 is inadequate. Therefore the question arises is it possible to use the memory management registers to provide more code and/or data space up to 124k words, with minimal overheads in the form of context switching and with no modifications to presently written processes?

1.2 Overview of Solution

The reason for the provision of a two state memory management system in the PDP-11s is not only to allow access to memory above 28k words but also, by the use of kernel and user modes, to provide protection between processes and the core resident part of the operating system. This usually means that the I/O page is non-resident when a user process is running and that I/O is handled in kernel mode. Thus in a real-time system there may be unacceptable overheads due to the considerable amount of context switching involved in this protection mechanism. However if one is willing to dispense with interprocess and operating system protection a simple and elegant solution which produces no detectable overheads is possible.

The basic concept of the solution is that while a process is running there is no need to have in scope any other process so that a snapshot of the 32k virtual memory at any one time will look like an ordinary MOS system configured for one process. The only extra executable code involved is in the scheduler for paging out the process that has just finished running and paging in the process that is about to be run. The I/O page, operating system, handlers and common buffer area are always resident otherwise access to these entities would have to be mediated by the operating system so that they could be paged in with consequent increase in overheads. A typical MOS process runs for something like 1 millisecond and the additional paging overhead is 4 MOV instructions taking about 20 micro seconds, a 2% overhead. A major change in the MOS layout occurs in the positioning of the stacks. Because we want the permanently resident part of MOS to be as small as possible and never greater than 8k words the stacks have been removed from the Process Control Tables and are paged in and out with their processes.

1.3 Use of the Memory Mangement Facility

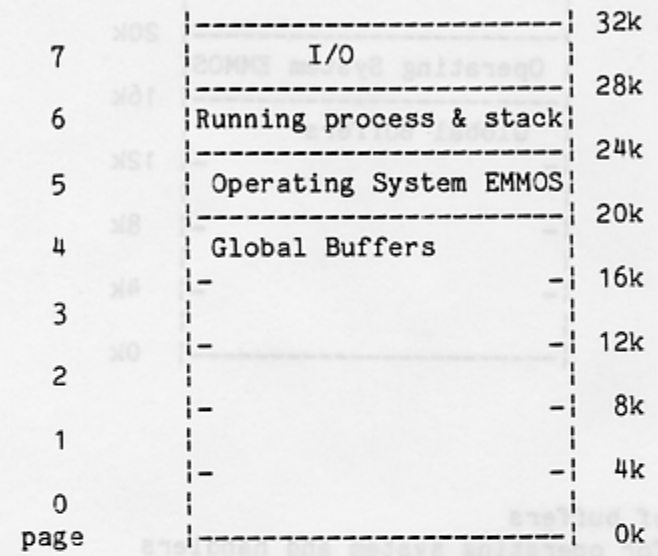
At any one time only the running process will reside in virtual memory, the others will be suspended and so have no need to be accessible. Permanently resident in virtual memory will be the vector area, global buffer area, operating system including all handlers, and the I/O page. Because the maximum size of a page is 4K words and it's position is fixed in virtual memory, the allocation of these pages is somewhat restricted. However a trade-off between global buffer space,

operating system size and process space is possible to some extent, and the following configurable options are catered for:-

Options where code and stack space of any process does not exceed 4k words:-

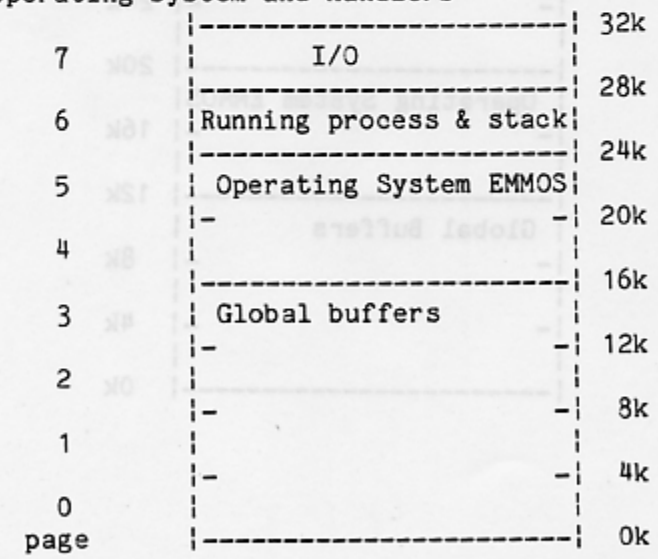
Option 1

20k words of buffer space
<4k of operating system and handlers



Option 2

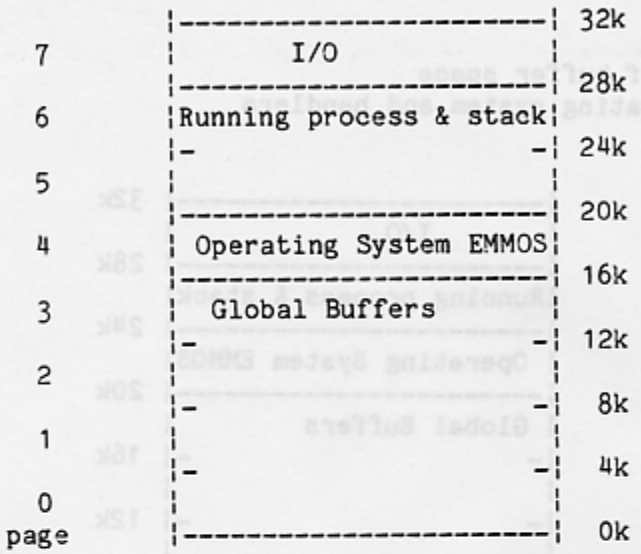
16k words of buffers
<8k operating system and handlers



Options where code and stack space of any process does not exceed 8K words:-

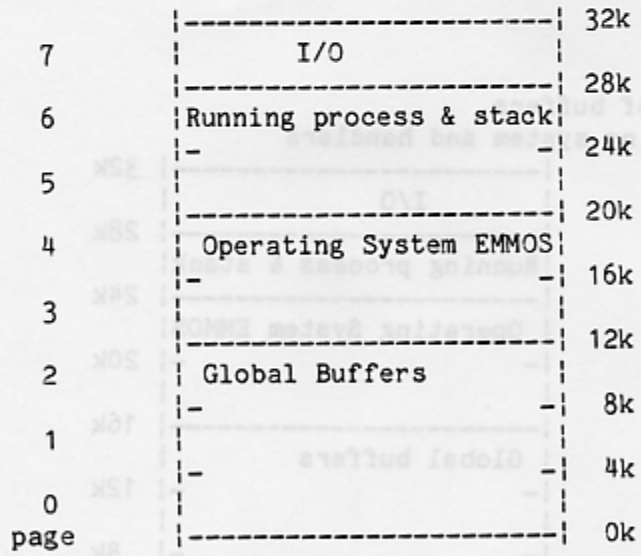
Option 3

16k words of buffer space
< 4k words of operating system and handlers



Option 4

12k words of buffers
<8k words for operating system and handlers



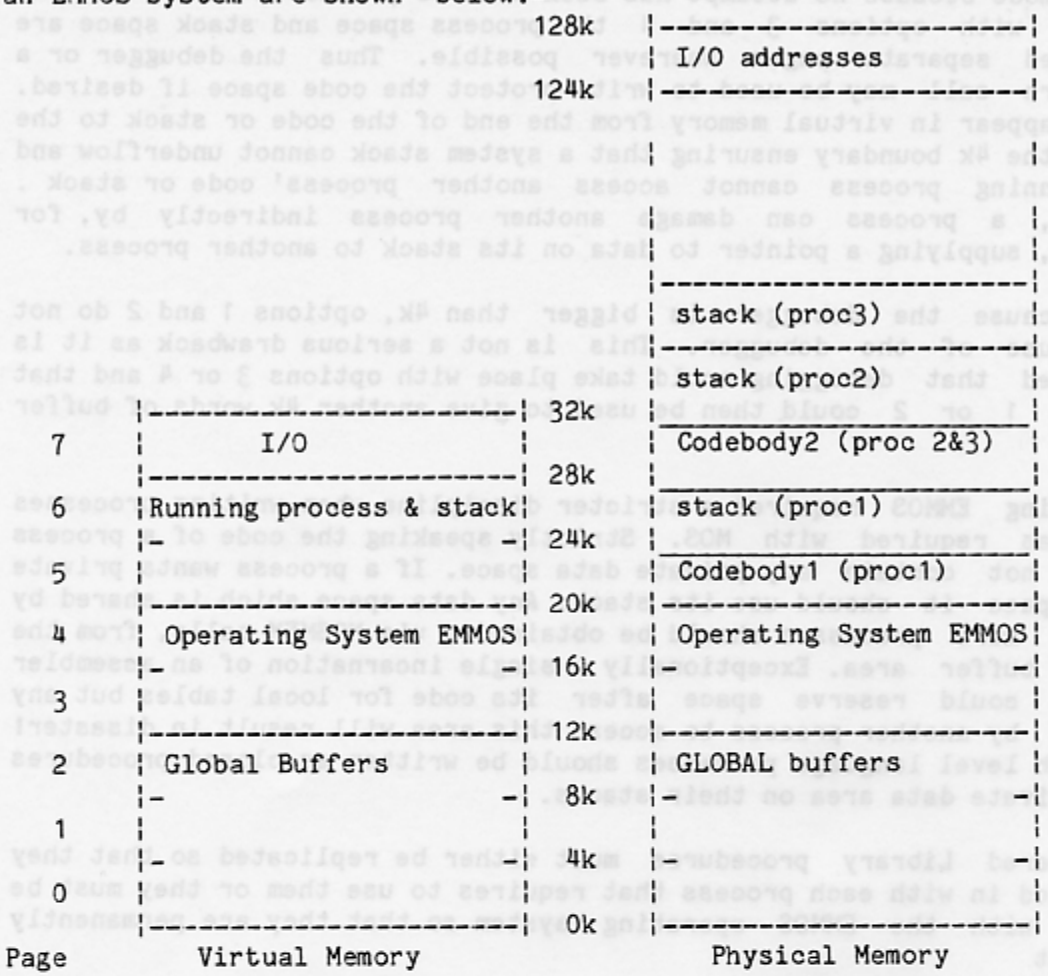
Only kernel mode is used by EMMOS, even user processes run in kernel mode because no attempt has been made to produce a secure system. However with options 3 and 4 the process space and stack space are allocated separate pages wherever possible. Thus the debugger or a procedure call may be used to write-protect the code space if desired. Holes appear in virtual memory from the end of the code or stack to the end of the 4k boundary ensuring that a system stack cannot underflow and the running process cannot access another process' code or stack . However, a process can damage another process indirectly by, for example, supplying a pointer to data on its stack to another process.

Because the debugger is bigger than 4k, options 1 and 2 do not allow use of the debugger. This is not a serious drawback as it is envisaged that debugging could take place with options 3 or 4 and that options 1 or 2 could then be used to give another 4k words of buffer space.

Using EMMOS requires a stricter discipline when writing processes than was required with MOS. Strictly speaking the code of a process should not contain any private data space. If a process wants private data space it should use its stack. Any data space shich is shared by two or more processes should be obtained , via MOSMEM calls, from the global buffer area. Exceptionally a single incarnation of an assembler process could reserve space after its code for local tables but any attempt by another process to access this area will result in disaster! All high level language processes should be written as closed procedures with private data area on their stacks.

Shared Library procedures must either be replicated so that they are paged in with each process that requires to use them or they must be linked with the EMMOS operating system so that they are permanently resident

A typical example of real and virtual memory allocations for an EMMOS system are shown below:-



The linker/loader system used with EMMOS should be reasonably intelligent! It is it's job to reserve physical memory in accordance with the stack size requested for each process. In particular it must evaluate the total memory requirements of a process that has more than one incarnation and determine whether or not some or all of the code has to be replicated.This may arise when the stacks of all the incarnations of a process plus the code will not fit into the paging window. The linker/loader is also responsible for outputting error messages when the process and stack requirements cannot be satisfied by that particular configuration of EMMOS.

PART TWO

TARGET MACHINE ASPECTS

2.1 Alterations to MOS

Perhaps the most vital alteration is that made to the scheduler. A context swap now involves saving the suspended process' window and restoring the new process' window. This however, only involves about four move instructions extra, for a typical configuration (8K processes).

The major alteration is, however, to the Process Control Tables (PCTs) and consequently to the PCT initializer. In particular the stacks for a process no longer reside here, hence a process' PCT is relatively small. Extra fields in the PCTs give the memory management window necessary to 'page-in' the process and the ends of the system and CORAL stacks. The PCT initializer now has to cope with setting up these fields, which it does by consulting the load map, supplied by the linker/loader.

Additions have also been made to the $CREAP macro. The sizes of the system and CORAL stacks are given to the $CREAP which in turn passes the information to the linker/loader (depending on the method used).

It has also been necessary to offer modified synchronous I/O routines 'SOUT' and 'SIN'. The original versions can be used if the I/O is performed on buffers from the permanently resident buffer pool, however they will not work if the I/O buffer is taken from the local code or data space. This is because the local area may be paged-out when the interrupt routine tries to get the next character, causing it to pick up rubbish or trap out. The new versions will copy the local I/O buffer into a global buffer before initiating the I/O, thus ensuring that it is always paged-in. This is likely to be the main overhead of the system.

Finally, the memory sizing performed in MOSMEM is no longer required. This is because the position of the global buffer pool is fixed by the page allocation scheme.

2.1.1 The Process Control Table

The following fields have been added to the PCT:

```
pct_mmr : ARRAY [ first_page..last_page ] OF window_type;
pct_r0e, pct_r6e : virtual_byte_addresses;
```

where

```
TYPE window_type = RECORD par, pdr : INTEGER  END;
TYPE virtual_byte_addresses = 0..#177777;
```

The following field has been removed from the PCT:

```
pct_stk : ARRAY [ 1..stk_len ] OF BYTES
```

pct_mmr is an array containing the window required to 'page-in' the process. The bounds are not 0..7 because it is known that most pages remain constant, only pages first_page to last_page are changed since this is the amount of virtual memory allocated to a process. Each element of the array is four bytes long and contains the page address register (par) and page descriptor register (pdr) values for that page.

pct_r0e and pct_r6e contain the virtual byte addresses of the ends (smallest numbered address) of the R0 (CORAL) and R6 (system) stacks. These values are used to detect stack overflow.

Since the stacks no longer reside in the PCT the field pct_stk has been removed.

The new fields in the PCT are set up directly from the Load Map produced by the linker/loader.

2.1.2 The Scheduler

A context swap now involves saving and restoring the memory mapping, as well as the general register values. We can actually avoid saving the mapping when a process is suspended because normally it remains constant. The protection on a page is the only thing likely to change and if this is done using the supplied routines then we can still avoid doing the saving.

When the process is made runnable it is first 'paged-in' by restoring it's memory mapping. To ensure that the stack pointer and it's stack are restored together, interrupts are disabled earlier than in the old version. If an interrupt is allowed to occur between 'paging-in' the stack and restoring the stack pointer then disaster will obviously follow.

Stack overflow, on both the R0 (CORAL) and R6 (system) stacks, is checked for when the process is suspended. The ends of the stacks are marked with overflow detect words, the address of which is held in the PCT. It is a simple matter to check if these words are still intact. If the system has no CORAL processes or library routines then the checks made on the R0 stack can be omitted by setting a switch in the EMMOS configuration file.

2.1.3 The $CREAP Macro

This macro initializes parts of the PCT, the rest is done at run time. Two extra parameters are now required, which specify the size, in words, of the R0 (CORAL) and R6 (system) stacks. To avoid confusion the parameters should be called by name, the default size is zero words:

```
eg:     $CREAP  G802,<UX25  >,,DV.IMP+0,DV.IMP+1,R0SIZE=200,R6SIZE=20
        $CREAP  G802,<UX25  >,,DV.IMP+2,DV.IMP+3,R6SIZE=30,R0SIZE=350
```

The macro converts the size into memory blocks (40 octal words) and places the information, along with the process id and name, in the .PSECT LDRCON. This information is used by the linker/loader in a way that depends on the methods being used. Note that different incarnations of the same process can have different sizes of stacks. In the above example (all numbers are octal) the first process would have

```
        ceiling( 200/40 ) =  4 memory blocks = 200 words for R0
        ceiling(  20/40 ) =  1 memory block  =  40 words for R6
```

and the second process would have

```
        ceiling( 350/40 ) = 10 memory blocks = 400 words for R0
        ceiling(  30/40 ) =  1 memory block  =  40 words for R6
```

Note: "ceiling(r)" is a function that rounds up real numbers to the next largest integer.

2.1.4 Input and Output Routines

The synchronous I/O routines, SIN and SOUT, have to be modified for use with EMMOS. The user interface is unchanged, but the method of operation is slightly different. SOUT now grabs a global buffer, copies the string into it from the user's buffer and then performs the SIO on the global buffer, to output the characters. SIN grabs a global buffer, calls SIO to fill it and then copies the string into the user's buffer.

This is necessary because if the string to be output is in the process' local area, it will be 'paged-out' with the process. SIO uses a pointer to the string (a virtual address) to get the characters, so if it does this when the initiating process is 'paged-out', it will either pick up rubbish or cause a memory management error. Similarly for SIN.

The copying would not be required for a string already in a global buffer, because the buffer will always be 'paged-in', regardless of which process is running. In this case the MOS versions could be renamed and used to avoid a loss of efficiency.

The same problems apply to the asynchronous I/O routine, SIO. However, any copying from local data space to a global buffer that is needed, is left up to the user. Checks have been included in the code that ensure both the IORB and the data area are in the global buffer pool or the resident EMMOS code. These checks can be omitted for a program known to work by setting a switch in the EMMOS configuration file.

2.1.5 Memory Sizing

This is now not carried out. The limits of the global buffer area are assumed to be the ends of the pages allocated to buffers (less space for the vectors if this includes page 0). It is assumed that the hardware has been configured in a sensible manner.

2.1.6 Protection Routines

Two routines are supplied, one to inspect the access control field (ACF) of a page, the other to change the ACF. Under the memory management system each page can be either

1.	non-resident	-all attempts at access will cause a trap
2.	read-only	-attempts to write into a location in the page will cause a trap
3.	(unused)	-same as non-resident
4.	read/write	-neither reads nor writes are trapped

The two routines will only work for one of the process pages. They both return an integer result. This equals 'true' if the page specified is a process page, else it equals 'false' and the ACF is not changed or inspected. The pre-defined constants, 'non resident', 'read only', 'un used' and 'read write', should be used to specify and compare the ACFs.

```
INTEGER PROCEDURE change page protection ( VALUE INTEGER page, access );
```

If the page specified is a process page then the ACF of that page is changed to the given access. The copy of the memory management registers held in the PCT table for this process, is also updated.

```
INTEGER PROCEDURE current protection ( VALUE INTEGER page;
                                       LOCATION INTEGER pct, pdr );
```

If the page specified is a process page then pct and pdr are set to the values of the ACF field of the memory management registers for that page, taken from the PCT window area and active page registers respectively. The two different versions are supplied in case one of them has been corrupted. Both should always be the same.

CORAL Constant Values

true	1
false	0
non resident	0
read only	2
un used	4
read write	6

cont.

The assembler versions of these routines are completly re-entrant and run to completion, so it is possible to have one copy of the code in the resident part of the system which is shared between all processes. The routine to change the page protection is called $CHAPP and the routine to inspect the current protection is called $CURPP, they have the same effect as the CORAL versions but are called as follows:

```
Change Page Protection :
        called by JSR   PC,$CHAPP
        with    R2 = number of the page to be changed
                R3 = the new access of that page
        returns with
                R1 = TRUE or FALSE
                R0 and R2...R6 unchanged

Inspect Current Page Protection :
        called by JSR   PC,$CURPP
        with    R2 = number of page to be examined
        returns with
                R1 = TRUE or FALSE
                R3 = value from PCT
                R4 = value from active PDR
                R0, R5 and R6 unchanged


Assembler Constant Values

        TRUE    1
        FALSE   0
        NONRES  0
        READON  2
        UNUSED  4
        RWRITE  6
```

2.1.7 System Configuration

The page allocation scheme, operating system options and various buffer addresses, are specified in the configuration file, EMMSYS.SML .

2.1.7i The Page Allocation Scheme

The virtual memory is divided up into 8 pages of 4K words, numbered 0..7. Page 7 is always allocated for the I/O page, the remaining seven are split between the global buffers, the operating system and the running process (including it's stacks). The order is critical, buffers should be at the low numbered end, then the system, then the process and finally the I/O page. The exact split is given by specifying the first and last page used by each part.

eg	buffers	0 - 3	emmos	4 - 5	process	6 - 6
or	buffers 0 - 2		emmos	3 - 4	process 5 - 6	

When decreasing the space allocated to the global buffers, be sure that the workspace buffers (see 2.1.7iii) are still located before EMMOS, otherwise EMMOS may be loaded on top of one of them before it's finished with. Note: different linking/loading schemes may lend themselves to a different ordering of buffers, system and process, in which case minor alterations to the PCT initializer may be necessary. The I/O page must always be page 7.

2.1.7ii Operating System Options

EMMOS: EMMOS/MOS selection switch
= 0 for EMMOS
= 1 for MOS (SAVMAP and SET250 have no meaning)

SAVMAP: save mapping when process suspended switch
= 0 when a process is suspended the window is saved as part of the process' volatile environment. This allows a process to change the window dynamically (eg the protection). Note, only that part of the window for the pages allocated to processes is saved and restored, so changing other pages will affect all processes.
= 1 the window is not saved each time the process is suspended. The original value is reloaded when the process is run again, erasing any changes made.

It is recommended that SAVMAP = 1 and the supplied protection routines are used to change the page allocation, with the other fields left well alone.

SET250: memory management error trap initialization on/off
= 0 when the system is initialized the trap is set to a local handler.
= 1 the trap is left untouched, this option should only be used when the debugger is used (MUD will set up it's own handler).

COPYIO: sin/sout version selection switch
= 0 (OR EMMOS = 0)
use the versions of sin and sout that copy to/from a global buffer before initiating the I/O.
= 1 (AND EMMOS = 1)
the MOS versions will be used

NOCORL: CORAL process indicator
= 0 no processes are written in CORAL, or use CORAL libraries, so the R0 stack is not checked for overflow each time a process is suspended. The stack is still set up as usual. The R6 stack is still checked.
= 1 There is a process using CORAL, so the R0 stack will be used. Both stacks are checked for overflow each time the process is suspended.

CHKSIO: SIO buffer address checking on/off
=0 Each time SIO is called the IORB and the data addresses are checked to ensure that they are in the global buffer pool or the resident EMMOS code. An attempt to use a local iorb or data area will cause a BUGHLT.
=1 No checks are performed. The use of a local IORB or data area is likely to cause the system to crash.

2.1.7iii Workspace Buffer Addresses

The linker/loader and system initializer must agree on the postion of the load map. It's address is given by MAPADD in the system configuration file. The linker/loader is given this information in some implementation dependant way.

Note that in the RSX linker/loader system, the area specified by MAPADD is also used by the loader for other workspace buffers during load time, because of this MAPADD must specify an area of store, lying between the end of the loader and the beginning of where EMMOS will be loaded, of about

25 * max_number_of_processes (decimal) words long

The loader ends around 032000, without removing redundant libraries, so even for a max number of 30 processes (25 * 30 decimal is about 1400 octal), giving MAPADD = 32000 would mean the tables would end way below 040000. This allows us to load a system with only 8K of buffers.

2.2 The EMMOS Debuggger - MUD

MUD (Marvellous Universal Debugger) is split into two parts, RESMUD and MUD proper. RESMUD is a small program written in MACRO-11 that is permanently resident in virtual memory. It acts as the interface between the operating system and the debugger, MUD, which is only brought into virtual memory when it is needed.

MUD is loaded by the loader in the same way as any other overlay. Normally , however, overlays that are not $CREAPed do not have entries in the load map, but the loader creates a special entry for the debugger. RESMUD uses this entry to find out where MUD has been loaded, and transfers control to it. For this reason RESMUD must be the first module loaded, even before MOSPM, if the debugger is to be used.

If RESMUD is not included in the system, then MUD will be ignored if it is itself included, and the debugger cannot be used. EMMOS will ignore MUD, unless a process is foolishly creaped with the name MUDSYS!

The debugger is able to reference the whole of the 18 bit physical store without the user knowingly changing the memory management registers.

2.2.1 A Summary of MUD Commands

The name of a command is one or two letters long, in upper or lower case, as follows:

Print Commands:

pb	print breakpoint locations (see breakpoint commands)
pr	print the value of a relocation register (see relocation commands)

Spy Commands:

s	spy on store
sr	spy on the registers
sw	spy on the memory management window

Breakpoint Commands:

b	set a breakpoint
k	kill a breakpoint
pb	print breakpoint locations (see print commands)

Enter User Program Commands:

g	go to a specific address
gs	go to a specific address single stepping
c	continue from a breakpoint
cs	continue from a breakpoint single stepping

Relocation Commands:

r	set a relocation register
pr	print the value of a relocation register (see print commands)

Exit Command

x	leaves MUD and halts

Special Commands

i	change I/O channel
e	evaluate an arithmetic expression
f	find a memory location containing a given value

2.2.2 MUD Command Syntax

The syntax is fairly simple and is quite free with spaces. A command consists of the command name, one or two letters(upper or lower case), optionally followed by one or two operands.

```
<name>
<name> <op1>
<name> <op1> <op2>
```

The two operands and the name may be seperated by commas or spaces. The operands specify 18 bit addresses or numbers in octal. Their syntax is:

```
<operand> ::= <repeat symbol>
          or= <number> <relocation> <mode>

<repeat symbol> ::=  *

<number>  ::= <18 bit octal number>
          or= null

<relocation> ::=  .  <relocation register>
             or= null

<relocation register> ::= <octal digit>
                      or= null

<mode> ::= V
       or= P
       or= null
```

If the octal digit is ommitted, it is assumed to be zero. If the relocation dot is not present, then the operand is assumed to be an absolute physical address. If the octal digit is missed out, then relocation register one is assumed. Hence the following examples all mean the same:

```
s100.,150.
s100.1  150.
s 100.   ,   150.1
s 100.1   150.1
```

2.2.3 The Relocation Mechanism

Tnere are seven relocation registers supplied (r1..7) which can be used to modify an address in a command. Values are put into these registers by the 'r' command (qv). To specify that an address is to be relocated it is followed by a dot and perhaps a digit. The digit is the name of the relocation register to be used and is one by default. When relocation is specified, the contents of the relocation register are added to the address giving the actual address used. For example:

```
r2,100  - sets relocation register 2 to 100
s20.2   - spys on location 20+100=120
r350    - sets relocation register 1 to 350
s30.    - spys on location 30+350=400
```

Many commands allow an asterisk to be used as an operand, this usually means 'all' in some way, for example:

```
k100    - kill the breakpoint at location 100
k *     - kill all breakpoints
```

2.2.4 Addressing Modes

An operand can refer to a virtual location by following the relocation indicator by a 'V' (the default mode is 'P', so normally addresses are physical). The corresponding physcial location will be computed using the user's window. Page lengths are ignored.

For example, if the base address of page 5 (virtual addresses 120000-137777) is 120100 and relocation register 2 contains 120000, then the following are equivalent:

```
s120110   - physical address by default
s120110p  - physical address specified
s120010v  - pa = 010 + base(5) = 120110
s110.2    - pa = 110 + 120000  = 120110
s110.2p   -   . . . . physical specified
s10.2v    - va = 10 + 120000 = 120010
          - pa = 010 + base(5) = 120110
```

2.2.5 Command Specification

pb print breakpoint locations

```
pb      prints out a list of all the breakpoints
        that are set, showing their location as an
        absolute address and as an offset +
        relocation.
```

pr print relocation registers

```
pr      print out the value of relocation register 1
pr 'd'  print out the value of relocation register 'd'
pr *    print out the value of all the relocation
        registers.
```

r set a relocation register

```
r 'n'   set relocation register 1 to 'n'
r 'd' 'n'       set relocation register 'd' to 'n'
r * 'n'         set all relocation registers to 'n'
r *     note that 'n' is zero by default so that
        r* will set all relocation registers to
        zero. Initially all are zero anyway.
```

x exit

```
x       leaves MUD and halts. This instruction would only
        be used when running under a monitor operating system.
```

cont.

s spy on store

s 'l' spy on the store starting at location 'l'. The address is displayed followed by the contents. The contents can be changed by typing a number followed by <return>, in which case the new value will be displayed. To advance to the next location just type space, to retreat to the previous location type tab. To finish the spy type <return>. Note that typing <return> with no number does not change the value in store.

```
eg
mud> r100
mud> s20.
000120> 123456  23
000120> 000023   000122> 077177
mud>
```

this leaves location 122 unchanged, but puts 23 into location 120.

s 'l1' 'l2' spys on locations 'l1' to 'l2', but does not allow you to change any values. Effectively this is just a display or dump command. 'l1' should be less than or equal to 'l2' or nothing will happen. To stop the printing hit <escape>. This is handy if you specify an enormous amount of printing by mistake.

sr spy on registers

sr spys starting at register 0
sr 'd' spys starting at register 'd'
This is similar to spy on store, but you can only progress forward one register at a time, by typing <space>. After reaching register 7, the program counter, you get to look at the psw. The only way to spy on the psw is to first spy on register 7. If you space forward from the psw you get back to register 0, to get out of the spy type <return>.

cont.

sw spy on the window

sw spy on the memory management registers. At present you are restricted to looking at the registers for the pages allocated to processes, which under EMMOS should be the only ones to change.
The registers are automatically split into their composite fields for displaying
and changing. To go forward to the next field, type <space>, to go to the next page, type <tab> and to exit type <return>. Numeric fields are changed by typing a number followed by <return>, other fields are changed by typing one character.
the expansion direction, ed> is either up=u or down=d
the access control is either non resident = n
read only = r
or read/write = w
an access control of 'unused' indicated an error in the memory management registers. The written into bit cannot be examined or changed.

b set a breakpoint

b 'l' puts a breakpoint at location 'l'. Note that you are not allowed to place a breakpoint on top of another breakpoint. It is possible to set breakpoints on top of emt instructions, but, once they have been reached, it is not possible to continue or single step from them.

k kill a breakpoint

k 'l' kills the breakpoint at location 'l'. If the program is halted at the breakpoint, it is not possible to continue from there, use 'go' instead.

k * kills all breakpoints. If the program had halted at a breakpoint it will not be possible to continue, use 'go' instead.

cont.

g go to command

g 'l' jumps to location 'l' and starts executing. 'l' must be a virtual address, mapped according to the window. The registers are loaded as set.

gs 'l' as above, but single stepping. Only one instruction is executed before control is returned to MUD. When single stepping all breakpoints are temporarily removed, so no breakpoints as such will be hit.

c continue command

c continue from the breakpoint last reached or at the next instruction if previously single stepping.

cs as for continue but single stepping (see 'gs' above)

Note that it is not possible to continue from a breakpoint that has just been killed, but you can use 'g' instead with 'l' as the contents of the program counter, R7.

i change I/O command

This command has a special syntax, because it is used to redirect the input and output of MUD from one terminal to another.

i change I/O to the console

iv 'n' change I/O to vdu 'n'

is sink the output. All output will be thrown away, including echoing. Input remains on the same channel.

f find a value

f 'v' searches the physical memory, stating at location zero, for a word containing the value 'v'. Every so often a message will say "still looking" if one has'nt been found yet. To stop the search early hit <escape> to break in.

f 'v' 'l' As above but the search starts at 'l' instead of zero.

cont.

e evaluate an expression

This command evaluates an 18 bit expession, using octal or decimal numbers or the contents of a relocation register. There is one accumulator, acc, which holds an 18 bit number. The operations that can be performed on it are:

^ val	Load the acc with 'val'
+ val	Add 'val' into the acc
- val	Subtract 'val' from the acc
P	Convert the least significant 16 bits of acc into an 18 bit virtual address
V	Treat the value in acc as an 18 bit physical address and convert it into a 16 bit virtual address.

The values are given by:

# octal number	- an 18 bit octal number, the # can be omitted
$ decimal number	- a 15 bit decimal number, ie positive only
R num	- the 18 bit value of relocation register 'num', num is 1 by default

Several commands can appear on one line, seperated by commas or spaces if it is necessary to avoid ambiguity. The value in acc is displayed when the end of the line is reached.

The value in acc is zero the first time the 'e' command is used. Subsequently the value is that left in it after the last use of 'e'.

2.2.6 Using MUD

If RESMUD is linked in before EMMPM and MUD is present as an overlay, then the debugger will be entered once the system has been loaded. If not the debugger cannot be used.

When control is passed from MUD to the user program, by using the go or continue commands, the system error traps are set so that any error will be handled by MUD. This remains so until the user program changes them. Note that while MUD is running the traps are set to different handlers which detect whether the debugger itself is in error. If so check that RESMUD has been assembled with the current page configuration.

We can return to the debugger at any time via the console emulator. Find the address of the global symbol $MUDIN, it is part of RESMUD and so is permanently resident, it's virtual and physical address is therefore the same. Use the console emulator to jump to this location and MUD will restart.

2.3 Linker and Loader Functions

In general a system's linker and loader perform two logically distinct functions, though in practice the border between these functions may be less clear. In a virtual memory system it is usually the operating system that allocates physical memory, but in a monitor system this may be done by the linker. There are a number of functions that the linker and loader are required to perform on an EMMOS system. Some of these functions require a degree of intelligence that is not usually found in either linkers or loaders. These intelligent functions may be incorporated in either the linker or the loader, or distributed between them. When presenting these functions we will describe them all as being part of the linker, for ease of understanding. The loader then, just copies an output image file from the linker, word for word into the 128K physical memory. However, linkers are complex pieces of system software, that are difficult to write or modify. Therefore, in practice, one usually takes the best of the functions that the linker has to offer and writes a loader program which incorporates those remaining.The functions that the linker/loader must perform are listed below:

1. Resolve all global references and relocate the object code.
2. Allocate physical memory for the system, buffers, process code and stacks. This takes into account the page configuration scheme.
3. Construct a Load Map which specifies where processes are in physical memory and where the ends of their stacks are in virtual memory. This is passed on to the loaded system.

2.3.1 The Linker

We can consider the input to the linker to be a set of Relocatable Binary (RLB) modules, with some system of reference resolving that need not concern us. A process' body is made up of one or more RLB modules, with sharing allowed. EMMOS is similarly composed. To specify how the system and process bodies are constructed we supply the linker with a Link Control File, which specifies which modules make up the system and process bodies, and the order in which they are to be linked to form the body. We also need a way of uniquely identifying a process body, so that when we create a process we can specify which code body to use.

Here we give an example of what this file could look like. The line headed 'SYSTEM' would define the construction of the resident operating system part of EMMOS and lines headed 'BODY' would each describe a process body, it's name given in brackets:

```
SYSTEM:          EMMPM-EMMTTY-.........-EMMMEM;
BODY(X25):       X25CODE;
BODY(PRINTR):    IOCODE-IOLIB;
END:
```

cont.

The EMMOS linker must also allocate physical memory for the process code bodies and stacks. In order to do this it needs to know which code bodies will be incarnated as which processes and how much stack to give those processes.T This information could go in the Link Control File. As an example of how it could be represented, we consider the example body description given above, and create some processes with those bodies:

```
PROC(X25-A):    "X25"    ,SYSTEM=200
PROC(X25-B):    "X25"    ,SYSTEM=200
PROC(PRINTR):   "PRINTR",SYSTEM=100,CORAL=400
END:
```

Here we describe three processes, called X25-A, X25-B and PRINTR. The first two are both incarnations of the body "X25" and both have 200 words for their system stack and no CORAL stack. It's not necessary for them to have the same size stacks, though some implementations could impose this as a restriction with no great hardship. The third process, "PRINTR", is the one and only incarnation of the body "PRINTR". It has 100 words for it's system stack and 400 words for it's CORAL stack.

The linker still requires information on the page configuration before it can allocate physical memory. This information is not strictly necessary. We could allocate the code and stacks for each process as one lump of physical store and if it is too large for the configuration, then the system will fail at run time. By knowing the page allocation scheme however, the linker can be quite subtle and avoid unnecessary duplication of code and could even arrange for the code and stacks to be on separate pages if the sizes are right. (For more details see 3.2.7/8). To complete our example we shall give an idea of how the configuration could be specified in the Link Control File:

```
PROCS 5..6,     BUFFERS 0..2,   EMMOS 3..4;
```

Note that page 7 must always be allocated for the I/O page and that the section using page 0 will lose the first 1000 octal bytes for the vector area.

In our example, if the body "X25" was less than 4K words then it would fit on one memory page. Since the stacks for "X25-A" and "X25-B" are quite small, they too fit on one page. The linker should then only put one copy of the body into physical memory and arrange the mapping of both the processes such that page 5 maps onto it. Page 6 would map onto the processes stack areas, which would of course be different physical memory locations.

Once the linker has finished allocating physical memory it needs to be able to tell the EMMOS initializer what the memory management window for each process is and where it's stacks are. It does this by including a table with this information in, in the Load Image File. This table must be loaded at a place in memory where the initializer can find it, say somewhere in the global buffer pool.

2.3.2 The Loader

The Loader takes the Load Image file as input. This file is produced by the linker and contains absolute binary data with an indication of where in physical memory to load it. The loader need not know anything about the processes or the system it is loading. However there needs to be some way of telling where the system's entry point is. Note that since the entry point will be a virtual address it is necessary to include an initial memory mapping as part of it's definition.

PART THREE

HOST MACHINE ASPECTS

3.1 Linking Using the RSX Task Builder

The RSX task builder(TKB) is used as the linker. It is because not all the functions we require are supplied by this linker, that our loader is more complicated then usual. The Task Builder is used to perform the relocating and global reference resolving parts of the required actions. The physical memory allocation, process incarnation and the production of the Load Map are all actions that are performed by our loader. This does mean that the loader is complicated, but less overall effort is required to write it than to completely re-write the linker.

3.1.1 Use of the Task Builder

The task builder is able to handle overlays in several ways. It can produce disk resident overlays, in which "paged-out" segments reside on disk, or memory resident overlays, in which all segments reside in physical memory but not all are "paged-in" into virtual memory. TKB will also allow your task to be some combination of the two. Overlays are paged in and out by special calls to RSX which TKB automatically inserts into the object code. It is possible to instruct TKB to leave out this code, allowing the user to handle his own paging. Overlays are characterized by being relocated at the same place in memory. it is this feature that EMMOS exploits to get all the process' bodies relocated at the same base address. We do however require that the linker does not put any auto loading code in for the overlays.

Amongst the various tables hidden in the task image, is the window description table, this does not appear for disk resident overlays, so we chose to use memory resident overlays throughout. Also, when using memory resident overlays, each layer of the overlay tree is relocated on a page boundary, which is another requirement for EMMOS. There is an option, when running a task under RSX, of allowing it to be swapped in and out of physical memory (checkpointed). A checkpointable task has a large space in it's task image file, so as EMMOS has no need of it, the task must be built not checkpointable (/-CP switch). We will assume then that the task is not checkpointable, with manually loaded (though we shall not include the calls to do it) memory resident overlays.

The first snag is met when we try to get information about the size of the overlays and their position within the task image. The tables that contain this information follow on directly after the code of the root segment, but there is no indication as to how long the root is, except in these tables. Normally their position is known because the overlay loading mechanism references them, but we've done away with this. Ideally the root segment would be the EMMOS system, but there is no clean way of finding out how long it is, so we chose to not allocate page 0 to the system. Instead the root segment does not contain any code, only some tables of known length used by the loader. Unfortunately, because TKB will relocate the next overlay layer on page 1, we have used up 4K of virtual space just for the interrupt vectors etc. To avoid this we allocate the global buffer pool space to pages 0,1.... because no code is loaded for it. The problem remains to persuade TKB to relocate EMMOS at the end of the buffers. We could do this by extending the root segment to occupy all these pages and then just ignore it all when loading, but this would mean the task image file would be enormous. Instead we use dummy overlay layers. Although each layer contains only one word, it is sufficient to persuade TKB to relocate the next layer at the next page boundary.

It is because of such devious means of getting the overlay relocated to the desired base, that all the calculations performed by TKB to produce physical memory load addresses are useless. Instead of wasting large areas of physical memory the loader does the calculations again. This also means that we can handle the process' stacks nicely.

Finally there is the problem of uniquely identifying each overlay, so a process can specify which is to be it's code body. TKB gives each overlay a name, which is the first six characters of the first object file's name in the overlay, however this may not be unique. We have then two options; use the name given by TKB or use a position numbering scheme. Despite it's drawbacks of non-uniqueness, the former was chosen. Names are easier to understand then numbers, and the possible inclusion of the debugger may well upset the numbering, though it is a special case. The use of six character names for the overlays led us to adopt the <name> field of the $CREAP macro to specify which code body to use. Other schemes may require another field in the macro call.

3.1.2 The Format of the Task Image File

The information required to complete the Overlay Description Table is scattered across the first few blocks of the task image file. The format is dependant on various switches and the form of the ODL file. There follows a brief description of the parts relevant to the loader, assuming it was built in the required manner.

BLOCK 0:

Contains no useful information, except perhaps the creation date of the task image.

BLOCK 1:

No useful information.

BLOCK 2:

This is the start of the task header. The word at byte offset 72 octal is the "header guard word pointer". This contains the byte offset, within the next block, of the "guard word", this immediatly preceeds the code of the root segment and is how the loader finds the root.

BLOCK 3:

This contains the start of the root segment. The first word of the root follows the "guard word". The root segment is used to transmit configuration details and the process description table to the loader. Immediatly following these tables is 29 words of rubbish (code used to load the overlays under RSX), the segment descriptor table, one word of rubbish and then the window descriptor table. These two tables are used by RSX to define the positions of the memory resident overlays in physical memory, EMMOS uses them to construct the Overlay Description Table, as follows:

segment descriptor table -
each entry is 9 words long, there is one entry for each overlay and the root

word offset 1 the virtual base address
word offset 6&7 the segment's name in radix 50 format.

window descriptor table -
each entry is 10 words long, there is one entry for each overlay

word offset 2 overlay's size in memory blocks (64bytes)

NEXT BLOCK BOUNDARY:

The code of the first overlay begins at the first word of the next disk block. Subsequent overlays start on block boundaries following on in order.

3.1.3 The Overlay Description Language

The Task Builder takes a series of object files and links them together to produce a Task Image. This is the exact image of what is to be loaded at run time (under RSX). EMMOS exploits the memory resident overlay capabilities of the Task Builder in order to relocate the system kernel and processes' bodies in the desired place. The arrangement of the overlays is described in the Overlay Description Language (ODL) file, the structure of which is as follows:

```
module1 - module2 - ..... - moduleN
```

specifies that the N modules are to be concatenated in the given order

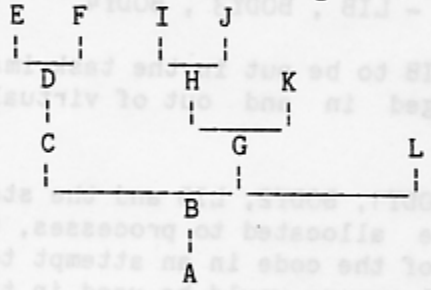
```
( module1, module2, ..... , moduleN )
```

specifies that the N modules will overlay each other in virtual memory. The modules not paged-in will be resident on disk or, if they are memory resident overlays (indicated by a ! before the open bracket), in physical memory.

These concatenation and overlay operations are combined to describe the overlay tree structure, eg:

```
A - B - !( C - D - !( E , F ) , G - !( H - !( I , J ) , K ) , L )
```

this specifies the following tree



At any one time during runtime, the memory will contain those modules lying on one of the paths from the root to a leaf.

For ease of writing, it is possible to give parts of the overlay tree a name, using the factor command (.FCTR). so a complete ODL file for the above example could be:

```
.ROOT           A - B - !( OV1 , OV2 , L )
OV1:    .FCTR   C - D - OV11
OV2:    .FCTR   G - !( OV22 , K )
OV11:   .FCTR   !( E , F )
OV22:   .FCTR   H - !( I , J )
.END
```

3.1.4 The EMMOS ODL File

```
.ROOT       ROOT - !( OV1 - !( OV2 - !( EMMOS - !( PROCS ) ) ) )
EMMOS: .FCTR  EMMPM - FIRST - ......................
PROCS: .FCTR  BODY1 , BODY2 , ................, BODYn
```

The factor EMMOS describes all the system modules, which are all concatenated. The factor PROCS describes each of the different code bodies for the processes. Normally only one code body is required at a time so the modules overlay each other, but it is possible to concatenate processes which are to access common routines.

For example, suppose we have four code bodies BODY1, ... , BODY4 which are each $CREAPed only once to give processes PROC1, ... , PROC4. If these processes are completely independant then we write:

```
PROCS: .FCTR  BODY1 , BODY2 , BODY3 , BODY4
```

Suppose we have a library module, LIB, that is shared by PROC1 and PROC2, then we could write:

```
PROCS: .FCTR  BODY1 - LIB , BODY2 - LIB , BODY3 , BODY4
```

This would cause two copies of LIB to be in physical memory. We could however write:

```
PROCS: .FCTR  BODY1 - BODY2 - LIB , BODY3 , BODY4
```

which would cause one copy of LIB to be put in the task image, but PROC1 and PROC2 would now be paged in and out of virtual memory together, which may not be desirable.

Note that if the total size of BODY1, BODY2, LIB and the stacks for PROC1 and PROC2 exceeds the space allocated to processes, then the loader will duplicate some or all of the code in an attempt to fit it in. It may be then, that more physical memory would be used in trying to share LIB than if we specified duplication in the first place.

cont.

The modules ROOT, OV1, OV2, ... OVn are required by the system to ensure that the operating system and the process bodies are relocated in the correct place. ROOT also transmits information from the $CREAP macro calls to the loader. The Task Builder relocates the first "layer" of the overlay tree (the root) at location zero. Successsive layers are relocated at the start of the next page boundary (a page is 4K words long). So, because each of these modules is quite small, ROOT is relocated at 0, OV1 at 20000, OV2 at 40000 etc. If we require the operating system to be relocated at 60000, we put it in the next layer (as in the above example). Suppose we want EMMOS to start on page 4 (location 100000), and it is over one page long so the processes will be on page 6 (location 140000), we would need to pack out pages 0..3 thus

```
.ROOT   ROOT - !( OV1 - !( OV2 - !( OV3 - !( EMMOS - !( PROCS ) ) ) ) )
```

For technical reasons it was not possible to have the operating system at the low numbered end of memory, so that is where the buffers have been put. Dummy overlays are used to pad out the buffer area because they cause the task image to be much smaller than if we had a .BLKW . The latter would cause it to contain about 12K of zeroes!

3.1.5 Installation of the Debugger

The EMMOS debugger, MUD, is in two parts, a resident part RESMUD and an overlay part MUD. To be able to use the debugger both parts must be included in the ODL file. RESMUD must appear before EMMOS but MUD can be put anywhere amongst the PROCS, eg:

```
.ROOT   ROOT - !( OV1 - !( OV2 - !( RESMUD - EMMOS - !( PROCS , MUD ) ) ) )
```

3.1.6 Creating the Task Image File

Once the form of the ODL file has been worked out the task builder is run under RSX-11 by:

```
TKB     EMMOS , EMMOS /NOSP = EMMOS /MP
```

The first output file is for the Task Image (.TSK) file, the second is the task map (.MAP) file, which is spooled by default. The MP switch specifies that the input file is an .ODL file.

The .MAP shows the structure of the task image and gives the virtual addresses of the modules, global symbols and .PSECTs. This link map should be used in conjunction with the loader's load map, which specifies physical addresses of code and stacks, when debugging. The .TSK file now needs to be copied to the RT-11 format disk which contains the loader using FLX. The system is now ready for loading and running.

3.2 The EMMOS RSX Task Image Loader

This section describes the operation of the loader for the Extended Memory MOS system. The loader itself is loaded from a floppy disk in RT-11 format, using a fairly standard bootstrap loader. The EMMOS loader then loads the system from an RSX-11 Task Builder Image file (stored in RT-11 format) on the same floppy disk. Control is then passed to the operating system or to the EMMOS debugger, MUD, if it is loaded.

3.2.1 Inputs and Outputs

The input to the loader is an RSX-11 Task Builder Image file stored on a floppy disk in RT-11 format. This file contains:

1. The Configuration Table (see 3.2.3)
2. The Overlay Description Table (see 3.2.4)
3. The Process Description Table (see 3.2.5)
4. Absolute binary code for the operating system and processes.

The loader produces a table, the load map, which it places in memory and displays on the console, that describes the positioning of all the processes in physical memory.

The address of the entry point to the loaded system must be stored in the first word of the system. This is set by MOSPM or, if the debugger is included, RESMUD.

3.2.2 The Console Load Map

This is a human readable form of the LOAD MAP that is supplied to the loaded system. There is one line for each process describing the position of it's code and stacks.

The first column gives the process id (pid), the next two the limits of the code and the fourth the limits of the two stacks combined.

If none of the pages containing code, contain any stack then the third column will read "unshared", that is, there is no page that contains both stack and code. However, if there is only one page for processes then this must contain all the code and all the stack, in this case the third column will read "page is shared".

If the code is longer than one page, and it is necessary to have a page containing both stack and code, then the third column will be the limits of the piece of code that is sharing with the stacks. This may be a copied piece of code.

The limits are given in terms of physical memory blocks, so a limit of

001200 - 001250

for the code would mean that the code starts at physical memory location 120000 and ends somewhere between 125000 and 125077.

If the task image does not contain an overlay for a process' body, then the load map entry for that process will be of the form:

pid - ABCDEF has'nt got a body

If one or more processes lack bodies then the load will be abandoned, once the load map has been printed, with the message:

"process requires body" (see ERRORS)

Note: the printing of the Load Map can be abandoned by hitting <escape>.

3.2.3 The Configuration Table

This table resides in the root segment of the task image, immediately before the Process Description Table. It is used to tell the loader the page allocation scheme and the maximum number of processes. The information is passed to it at run time so that the loader need not be re-compiled for every change in the configuration.

The format is:

```
VAR config_table :RECORD
                    max_number_of_processes,
                    load_map_address,
                    first_mos_page, last_mos_page,
                    first_process_page, last_process_page
                                  : INTEGER
                END;
```

3.2.4 The Overlay Description Table

This table is constructed by the loader from the task image's window and segment descriptor tables, that follow the root segment. Each overlay named by the task image (except the dummy ones and MOS) has an entry of the form:

```
TYPE overlay_description = RECORD
                               name : ascii;
                               size : mem_blocks;
                               disk_address : INTEGER
                           END;

TYPE ascii = ARRAY[ 1:6 ] OF CHAR;

TYPE mem_blocks = 0..number_of_process_pages * #200 -1;
```

The table is given by

```
odt : ARRAY[ 1..number_of_overlays ] OF overlay_description;
```

odt[ov].size contains the size of the overlay segment (that is of the code) in 32 word memory blocks.
odt[ov].name is the overlay's name, taken from the overlay description language file at build time.
odt[ov].disk_address is the number of the disk block where the overlay starts.

Note that the disk address field is not used in the current version of the loader. It remains as a left-over from a version that would not support down line loading.

3.2.5 The Process Description Table

This table resides in the root segment of the task image, following the Configuration Table. It is produced by the $CREAPs used to create the processes. It is used to tell the loader which processes are required, which overlay will supply the code and the size of their stacks. There is one entry in the table per process, each of the form:

```
TYPE process_description = RECORD
                               pid : INTEGER;
                               name : ascii;
                               r0_size, r6_size : mem_blocks
                           END;

TYPE ascii = ARRAY [ 1..6 ] OF CHAR;

TYPE mem_blocks = 0..number_of_process_pages * #200 - 1;
```

The table is given by

```
VAR pdt : ARRAY [ 1..number_of_processes ] OF process_description;
```

pdt[i] is the entry for the ith process,
so pdt[i].pid = i for all i IN 1..number_of_processes.
The name field contains the six character name of the overlay which is the code for the process. This is taken from the second parameter of the $CREAP macro call.
pdt[i].r0_size and pdt[i].r6_size contain the required sizes, in 32 word memory blocks, of the stacks.
Note that the macro converts the sizes from words to memory blocks before placing the values in the table, using

```
size_in_blocks := ceiling( size_in_words / 32 ).
```

3.2.6 The Load Map

The load map describes the loaded system to the initializer. It is placed in memory by the loader at an agreed address before it gives control to the system. There is one entry in the table for each process, and a special one for the debugger if it is included in the task image. The form of a table entry is:

```
TYPE load_description = RECORD
                            pid,
                            r0_low, r0_hi,
                            r6_low, r6_hi : INTEGER;
                            window : ARRAY
                                     [ first_page..last_page ]
                                     OF window_type
                        END;
TYPE window_type = RECORD par, pdr : INTEGER END;
```

The map is given by:

```
map : ARRAY [ 1..number_of_processes + 1 ] OF load_description
```

map[i] is the entry for the ith process, so map[i].pid = i, for all i IN 1..number_of_processes. map[i].pid = -1 for the debugger entry. Any unused entries have pid = 0.

r0_low and r0_hi contain the virtual byte addresses of the lowest and highest numbered words in the R0 (CORAL) stack for the process. Similarly for r6_low, r6_hi.

window describes the mapping from virtual memory onto this process. There are two words for each page allocated to a process, the page address register and the page descriptor register.

3.2.7 The Physical Memory Allocation Algorithm

The loader should attempt to arrange the loading of the Load Image into physical memory in a way that will reduce the amount of physical memory required. The total amount of virtual memory allocated to a process for it's code body and stacks will be fixed (= num_pp memory pages), but for each process the sizes of it's code and the total size of it's two stacks will vary.

If at all possible the loader will arrange for the stacks and the code to be on different pages in virtual memory, allowing the process to put read-only protection on it's code. This is only possible if:

pages(code_size) + pages(stack_size) <= num_pp

where pages(i) = the size i rounded up into a whole number of memory pages. Of course if pages(code_size + stack_size) > num_pp then not enough virtual memory is availiable for this process. However if we have:

pages(code_size + stack_size) = num_pp

then some of the code and some of the stack space must share one of the pages. The shared page cannot then have read-only protection placed on it. If a process has already been created with this code body then physical memory following the code body will have been allocated as a stack. We cannot then have the stack for the next process following directly on from the code, so it may be necessary to copy part of the code body and arrange that the memory mapping maps the two parts of the code onto consecutive virtual memory locations.

E.G.

Suppose we have two pages for a process, num_pp = 2, and we have a code body which is $CREAPed twice, as PROC1 with stacks S1 and as PROC2 with stacks S2. If the code body is larger than one page then the second page must be shared between the stacks and some of the code.

First the loader copies the code body into physical memory, then it allocates the stacks for PROC1, the window for PROC1 will be (a..b,b..d).

```
--a--------------b----c------d----f--------------
  :    code size       :  S1  :        ....physical memory
-------------------------------------------------
  |<- 4K words -->|<-- 4K words -->|
```

Now for PROC2. If pages(code size + S1 + S2) <= num_pp, then the loader can allocate the stack following on directly from S1, thus the window for PROC2 will be (a..b,b..e).

```
---a-------------b----c------d--e-f--------------
   :   code size      :  S1  :S2:
-------------------------------------------------
```

Note that PROC2's window allows it to (illegally) access the stack of S1. If however pages(code size + S1 + S2) > num_pp then the loader must take a copy of the piece of code that overlaps the second page, (copy from b..c to d..g), thus the window for PROC2 will be (a..b,d..h), and the copy will be mapped into the same virtual locations as the original.

```
---a-------------b----c------d--e-fg--h------------
   :   code size      :  S1  :copy :S2:
-------------------------------------------------
```

Now PROC1's stack is not in the window of PROC2 so neither process can use the other's stack.

The loader will try and fit as many stacks as possible into the page without copying a piece of code, but it wont try to optimize the ordering. If the code size is smaller than a page and the stacks are bigger then the whole of the code will be copied.

3.2.8 The Logical Operation of the Loader

```
get process description table from task image;
construct overlay description table;
load EMMOS;
FOR each overlay
DO
    load the code;
    FOR each process
    DO
        IF name of process = name of overlay
        THEN
            IF size of code in pages + size of stacks in pages
                <= number of pages for processes
            THEN
                code and stacks are on completely different pages;
                set up the window for the process;
                leave room for the stacks in physical memory

            ELSF (size of code + stacks) in pages
                = number of pages for processes
            THEN
                one page contains stack and code;
                IF stack will fit on the end of the last
                   incarnation's window
                THEN
                    leave room for the stacks after last stack;
                    set up the process' window
                ELSE
                    make a copy of the piece of code that must
                    share a page with some stack;
                    leave room for the stacks after the copied code;
                    set up the window for the process
                FI
            ELSE
                process is too large
            FI
        FI
    OD
OD
```

3.2.9 Error Messages

i. System Traps

These indicate something has gone drastically wrong with the loader, hopefully they will only occur after modifications have been made and not thoroughly tested. They may also occur if the loaded system goes wrong before it has set the trap locations to its own trap handlers.

```
cpu error: pc = nnnnnn   ps = mmmmmm
reserved instruction: pc = nnnnnn   ps = mmmmmm
```

These messages occur when a trap to #004 and #010 occur. The contents of the program counter and processor staus word, as placed on the stack by the trap, are displayed.

```
memory management trap 250
nnnnnn at mmmmmm
```

This message occurs when something goes wrong with the memory management. The contents of the two status registers are displayed. nnnnnn = SR0 which is further expanded in full following the error message , and mmmmmm = SR2, the virtual address associated with the error. Following the written expansion of SR0, the contents of the kernel mode memory management registers are displayed.

```
power failed, start again
```

the power to the processor failed, re-boot the system. NB this has not been tested!!!!

ii. Loader Errors and Messages

test: eof

The end of the task builder image file has been reached prematurely.

test: get has failed

An error occurred in the transfer of data from the floppy disk. Try again.

test: rewind

The loader requires the input task image file to be rewound. This is only an error if the file is not on a rewindable device, eg down line loading. It is caused by an error in the loader.

cant find 'task file name'

The file is not on the floppy disk, check the file name specified in the loader is the same as you copied to the disk, and the disk is in the correct drive (drive 0). Remember that the name specified in the loader must include any blanks necessary to pack the name to 10 chars, in the form ABCDEF.EXT

too short!

The file is less than 4 blocks long, so cannot possibly be the correct task image.

first dummy overlay not found

the dummy overlay that is used to pack out the buffer area is not present in the task image. Check the file is a task image and that the ODL file correctly specifies the dummy overlays.

mos is larger than nn pages long

the resident part of EMMOS is too large, more than the nn pages specified in the configuration file. Make sure that the processes have been put in as overlays and not in the resident part.

window and segment descriptors of unequal size

The loader has lost it's place in the task image, or the ODL file does not specify the overlays in the correct format.

ABCDEF is too large

The named overlay is too large for the configuration, even without stacks. Check that the overlay names are seperated by commas in the ODL file.

process nn too large

The process with pid = nn has stack and code too large for the configuration. Check that stack sizes have been given in words in the $CREAP macro call.

ABCDEF has'nt been made a process

The named overlay does'nt appear in any of the $CREAPs. The overlay is loaded but otherwise ignored, this is not a fatal error.

process nn: code duplicated

The loader was forced to make a copy of some of the process' code because the code had already been $CREAPed and the stack and code needs to share a page. This is not an error.

debugger has been loaded nnnnnn - mmmmmm

The overlay part of the debugger has been loaded, starting at memory block nnnnnn and finishing in block mmmmmm. This is not an error.

ABCDEF has incorrect base address = nnnnnn

The named overlay has been relocated at nnnnnn by the Task Builder. This does not agree with the virtual page allocation scheme given. Check that the ODL file is correct, that the right number of dummy packing overlays are present and that all overlays are not too large for the configuration.

debugger is too large

The debugger takes up more space than is allowed for processes. it is still loaded into physical memory, but only those pages normally allowed for processes will be paged into virtual memory, this will almost certainly cause MUD to crash. Use the console emulator to enter the system instead of MUD. Alternatively, remove the debugger or allow more pages for processes. This is not a fatal error.

the debugger cannot be used

Either the resident part or the overlay part of the debugger are missing and hence the debugger cannot be used. If the debugger is needed then include the missing name in the ODL file. This is not a fatal error.

non-unique overlay ABCDEF

The ODL file describes a configuration such that two overlays have the same name. The loader requires all overlays to have unique names so it can unambiguously select the code body for a process. Try re-ordering the overlay description in the ODL file. If the effending overlay is a common library, put it's name after the code body, otherwise the file may need to be copied to one of a different name.

process requires body

at least one process has specified a body name that does not appear in the Task Image as an overlay. An indication of which process is causing the problem can be found from the Console Load Map (qv). Make sure the name specified in the $CREAP macro call is that of one of the object files, check in the task map for the names actually used. If the debugger is included, make sure it, and all other overlays, are separated by commas in the ODL file.

3.2.10 Tracing

The loader and it's libraries are liberally scattered with code to produce vast amounts of output, that describes in detail what is being processed. These pieces of code are normally excluded by surrounding them in comment brackets. The start of a piece of trace code is indicated by {trace and the end is followed by ecart}. To include the trace statements change {trace to {trace} and ecart} to {ecart}. This will make it easy to restore them later.

The output produced should be self explanatory, except for PROCEDURE waiting; which displays the message 'waiting' and waits until any character is typed in at the keyboard. All this does is slow down the output, since <cntrl-s> and<cntrl-q> are not implemented.

3.2.11 Changing The System

The loader need not be re-compiled every time the system configuration is changed. This is because the configuration information is placed in the root segment of the task image file at link time.

3.3 Process Creation

Processes are created at compile time by calling the $CREAP macro in the system configuration module. The $CREAP macro defines the attributes of a process; it's entry point, the name of the overlay that contains it's code body, the devices associated with it and the size of the CORAL and System stacks.

Note, the name given is the name of the overlay in which the code (and therefore the entry point) is found. This name is that of the first module given in the overlay subtree. E.g. If we have a library module LIB and a process body module BODY1, then in the ODL file we could write:

```
        PROCS:  .FCTR   BODY1 - LIB , .........
```

in which case the $CREAP call would be

```
        $CREAP  $ENTRY , <BODY1> , ............
```

or alternatively

```
        PROCS   .FCTR   LIB - BODY1 , .........
```

in which case the $CREAP call would need to be

```
        $CREAP  $ENTRY , <LIB   > , ............
```

For this reason all the overlay names must be unique, othewise the loader would not know which code body to give a process. Now suppose we have another process body module, BODY2, which also uses LIB. We could not write:

```
        PROCS:  .FCTR   LIB - BODY1 , LIB - BODY2 , .........
```

since both overlays would be called LIB. Instead write either

```
        PROCS:  .FCTR   BODY1 - LIB , BODY2 - LIB , .........
```

with

```
        $CREAP  $ENT1 , <BODY1 > , ......
        $CREAP  $ENT2 , <BODY2 > , ......
```

or

```
        PROCS:  .FCTR   LIB - BODY1 - BODY2 , ......
```

with

```
        $CREAP  $ENT1 , <LIB   > , ......
        $CREAP  $ENT2 , <LIB   > , ......
```

or some other combination.

Remember that the name must be of exactly six characters. Pad out with blanks if necessary.

The stack sizes are specified using named parameters, these are the only extra parameters of $CREAP due to the EMMOS extensions. The sizes are given in words, but are automatically rounded up into memory blocks (32 words). A zero length R0 stack is acceptable for a non-CORAL process, but the R6 stack should always have some space. If the size is not specified it is set to 32 words by default. Note the stack sizes are given seperately for each process, not for each code body, so two incarnations of the same body could have different stack sizes, though it would be a bit odd.

APPENDICES

APPENDIX A Pitfalls.

A.1 Summary

Processes must ensure that all communication is performed via the global buffer pool.

Shared overlaid library procedures must not use the CORAL global vector for their entry.

CORAL code bodies incarnated more than once should not preset variables.

All local variables must be taken from the stack, in code bodies incarnated more than once.

All IORBs and their data buffers must be located in permanently resident memory (ref 2.1.4).

A.2.1 Use of local space for inter-process communication

Obviously processes with different code bodies cannot use their local static space (ie not the stack) for communication, but at first sight it would appear that different incarnations of the same body could communicate using variables in the body. Although sometimes possible, this is extremely configuration (and even loader) dependant. The problem arises when there is not enough room to fit all the code body (which includes local static data) onto process pages of it's own. The loader may need to take a copy of some of the code, which may include local data. In this case there would be more than one copy of the local data, so if two processes use it for communicating, they would actually be using two different locations.

Therefore, processes must ensure that all communication is performed via the global buffer pool.

A.2.2 Use of shared, overlaid CORAL libraries.

All CORAL procedures, common variables etc., must use unique indices to the global vector, to define themselves (unless some trick is being performed). This remains true even if the procedure is defined in an overlay, because there is only one global vector. If a library module containing some CORAL procedures only appears in one overlay, then the global vector will contain the virtual addresses of the entry points. However, the procedures will only be in virtual memory when a process which is an incarnation of that overlay is running, so only these processes can use the procedures.

This is ok, the problem arises when two or more overlays use the library module. If the overlays are such that when relocated, the entry points of the procedures are at different virtual locations, the global vector mechanism wont work. The entry point recorded will probably be that for the last overlay processes by the linker, so the first overlay will crash when it tries to call the procedure.

To prevent this all copies of the library module must be relocated at the same virtual locations, either by placing it on permanently resident EMMOS pages or putting it first in the overlay, though this may cause problems with the overlay naming convention. A more satisfactory solution in many ways, would be to not use the global vector for the procedure entry. The offending procedures would use genuine names for their entry points, not vector indices. This is achieved in CORAL by using a large index, outside the bounds of the global vector.

A.2.3 Use of static variables

Any data space allocated in a code body, which is incarnated more than once, will be used by more than one incarnation, unless the loader causes a copy to be made. The processes cannot then use this space for private local variables, because another may use the same physical memory for it's private local variables. Neither can it be used for shared variables, because the loader may have copied them (see 'use of local space for inter process communication'). Constants are ok since all incarnations will require them to remain constant. Beware of string constants though, although they really should remain constant, some unhealthy procedures actually change some parts of them. Such practices should be banned.

In particular, note that the local variables can be created in devious ways, in CORAL a preset variable becomes an OWN variable by being allocated not on the stack, but in the local static space. Presetting and variables in the outermost block must be avoided for multi-incarnate processes.

Always take local variables from the stack.

Don't Panic !