Figure 4.1

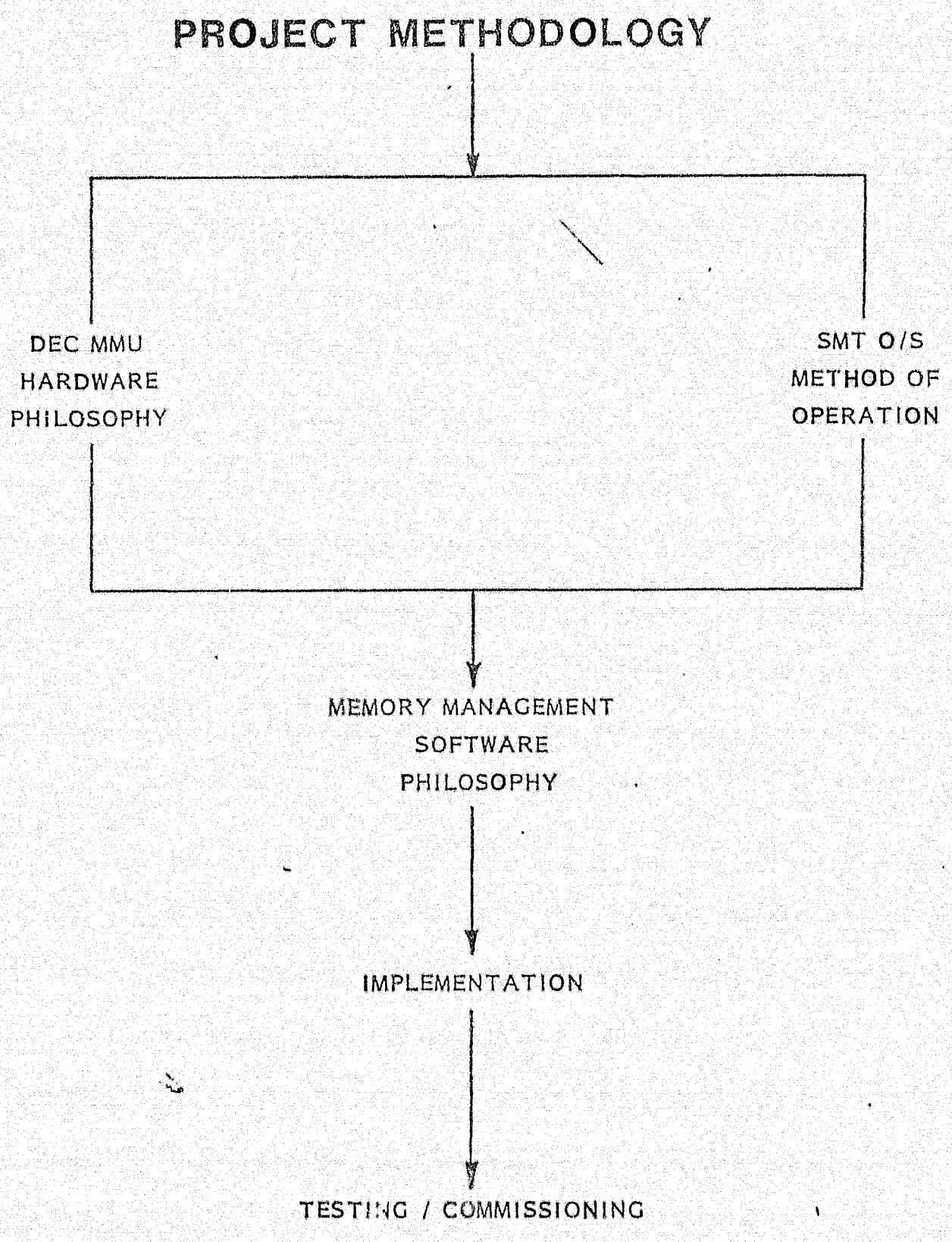
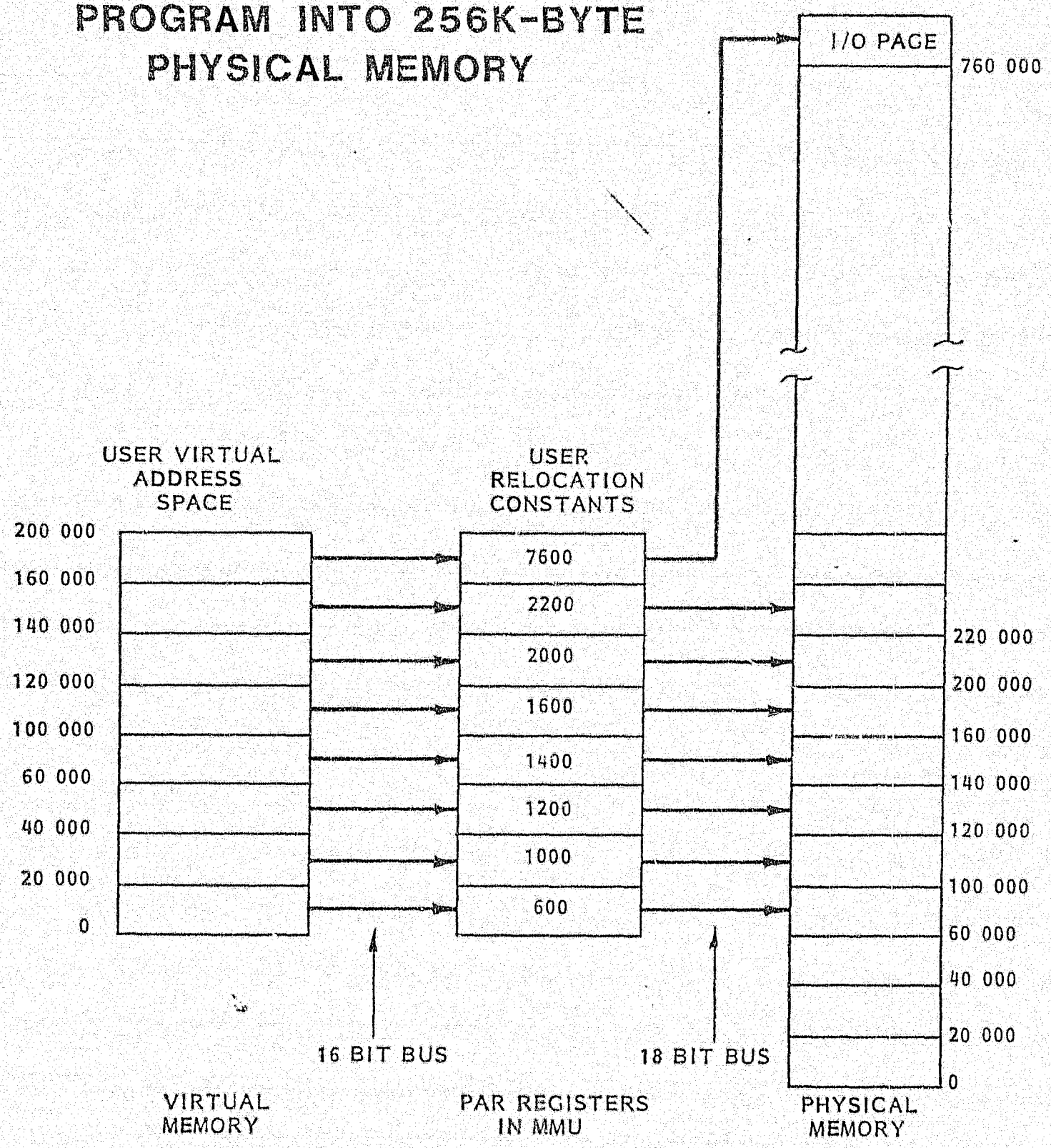

PROJECT METHODOLOGY
DEC MMU HARDWARE PHILOSOPHY
SMT O/S METHOD OF OPERATION
MEMORY MANAGEMENT SOFTWARE PHILOSOPHY
IMPLEMENTATION
TESTING / COMMISSIONING

Figure 4.2


RELOCATION OF 64K-BYTE PROGRAM INTO 256K-BYTE PHYSICAL MEMORY
USER VIRTUAL ADDRESS SPACE
USER RELOCATION CONSTANTS
I/O PAGE
760 000
200 000
160 000
140 000
120 000
100 000
60 000
40 000
20 000
0
7600
2200
2000
1600
1400
1200
1000
600
220 000
200 000
160 000
140 000
120 000
100 000
60 000
40 000
20 000
0
16 BIT BUS
18 BIT BUS
VIRTUAL MEMORY
PAR REGISTERS IN MMU
PHYSICAL MEMORY

in "kernel mode" and thus the tasks operate in the most permissive state at all time. Kernel mode was chosen so that the operation of the memory managed SMT still remained simple. If both "user" and "kernel" modes were used, the efficiency of SMT would have been lost. That is, a great deal of time would have been spent, mapping the kernel into virtual addressing space, every time a kernel primitive is called.

The mmu has a status register 0 (SR0) which contains the abort error flag, memory management enable, mode of operating and various other maintenance information. To enable the mmu, bit 0 of SR0 has to contain a zero, otherwise the virtual address coincide with the physical address and the mmu does not map the virtual address to the physical address.

Status register 2 (SR2) contains the last instruction address at the time of a mmu abort. This register can be used to find what caused the mmu error.

If an mmu error occurs the processor traps to location octal 250. A trap handler has been written to save SR0 and SR2 so that when RRGEL, the error reporting procedure, is called they may be printed out. Reference 2.8 should be consulted to obtain details of this.

4.3 MEMORY MANAGEMENT PHILOSOPHY

4.3.1 Introduction

The modifications carried out were minimized so that as few new concepts were introduced to SMT as possible. This section provides the design philosophy used to achieve memory management in SMT.

For the remainder of the thesis, the term "memory management" is construed as implying "memory mapping". As will be seen the final solution did not implement features for the control of memory access which is part and parcel of "memory management", but merely the mapping of the user tasks from physical address space into virtual address space.

4.3.2 User Task Stacks

SMT is a stack-based kernel in which the state of a given task is given by the last link cell on the stack corresponding to that task. Each task has its own stack. With the introduction of memory management the stack could either have been placed in the common area, with the result that a large portion of the common area would have been lost due to the inclusion of all the user task stacks. The second option available, was to have the task stack included in the task image.

The second option was chosen such that each task carries with it in the user task space, (above the common area) its own stack. With the inclusion of the stack in the task image, the start address of the stack has to be passed to the kernel. This is the first piece of information required by the kernel to initialize the task and then to execute it. The kernel links to the task when the system is started up the first time. This is described more fully in section 4.3.6.

4.3.5 The Common Area

The system has been designed with a common area of 12k words (3 pages of 4k words each. see Fig 4.6), i.e. in APR0,1,2 in which we can install the following:

1. SMT kernel (approx. 7k words long).
2. Input/output drivers with buffers defined by the user.
3. The system database.
4. Common data bricks.
5. Common procedure bricks.

Items 3, 4 and 5 are all areas which are used by more than one task. If only one task uses a particular brick it is obvious that the brick should not be in the common area but with the task.

The choice of 12k was made after considering typical databases used on SMT systems at present. The databases looked at, were of the order of magnitude 2.5 - 4 k words.If a database of length 3k words was used, an area of 2k words would be left for common data and procedure bricks, which is adequate. This 12k word common area can be expanded up to 16k words as was done on the Chlorine Plant supervisory system. Full details on how to expand the common area is given in reference 2.8. What has to be borne in mind, is that if a larger common area is used, the task space is reduced from 16k words (4 pages) to 12k words (3 pages), which in any event is still a substantial space for a task.

The abovementioned areas are all linked to form one image which is installed in the bottom 3 PAR pages of virtual address space. This common area remains permanently installed in the bottom 3 pages. Whenever a task change is made, only the task PAR's are swopped and not the common area PAR's. Once having built this common area, the symbols definitions table as the task builder generated by in RSX-11M is used to task build the user tasks.

Note: If a change is made to the common area all the user tasks have to be rebuilt since they all link to the common area.

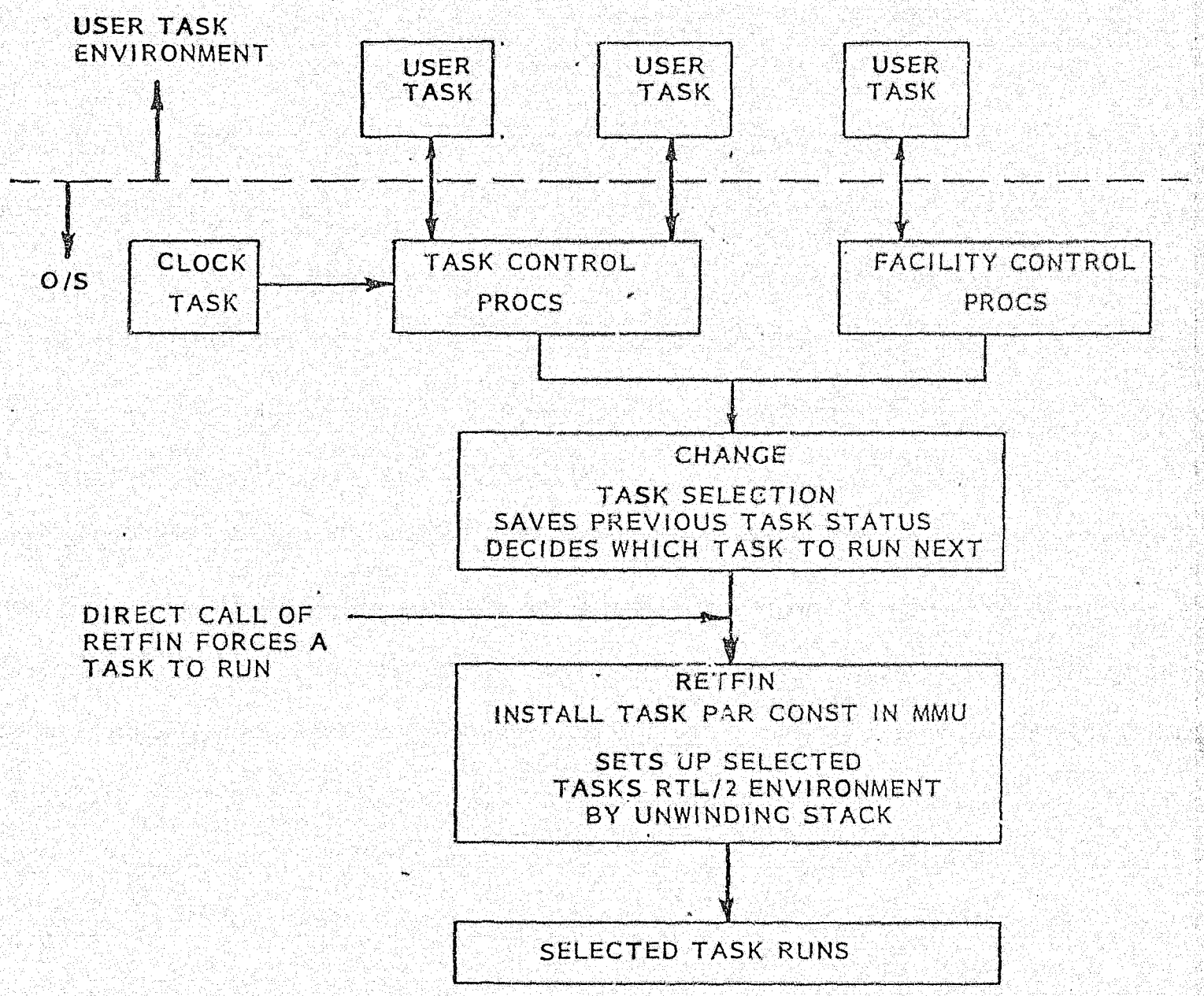
4.3.6 Memory Management Philosphy and PAR Swapping

The basic concepts of memory management lies in mapping. The mmu maps a single virtual space address into a physical space address. The actual location in physical space is given by the constants located in the PAR's.

This idea has been extended to coincide with tasks. The approach used was to assign each user task a set of PAR constants. Every time a task is to run, the kernel installs the user task's relocation constants into the PAR registers. This brings the user task into scope before it procedes to execute the task. This is a simple yet effective method in determining where the task is located in memory.

It seemed logical to put the swopping of the PAR's at the

Figure 4.3



TASK CONTROL PROCS

STOP
START
DELAY
WAIT
WAITFOR
SET
RESET

FACILITY CONTROL PROCS

SECURE
RELEASE

beginning of the procedure RETF1N, i.e. the task PAR constants are installed into the PAR registers bringing the task into scope. Only then is an unwind of the task registers made. (see Figure 4.3)

The installing of the constants at the beginning of RETF1N enables the remainder of RETF1N to remain the same and also SMT is unaffected by the additional instructions. The primitives calling RETF1N remain the same ie it is not necessary to first call a procedure to install the task PAR constants, and then another call of RETF1N to unwind the stack. Another section of the kernel namely the existing interrupt handling mechanism is not affected.

Figure 4.4 illustrates RETF1N prior to the addition of memory management facilities and figure 4.5 that after the facilities have been added.

4.3.5 PAR Assignments

After having decided on the appropriate point for the PAR constants to be swopped, the next question was whicl PAR constants should be swopped. The philisophy used is shown in Figure 4.6 and has previously been discussed in section 4.3.3.

The first three pages APR0, APR1 and APR2 as well as APR7 were made fixed in memory, i.e. they contain the same constants irrespective of which task is running, implying that the kernel will always be in scope.

APR0 and APR1 contain the kernel.

APR2 has been left aside for use as a data.ase/common procedure/common data area.

APR7 has also been made fixed in memory and is the system I/O page.

This implies that individual user tasks may have a maximum length of 16k words each, occupying APR's 3,4,5 and 6. With modifications the common area may be expanded to 16k words length and individual tasks of maximum length 12k words (See Ref 2.8).

Figure 4.7 and 4.8 summarise the above mechanism, illustrating two tasks residing in virtual space and how they map to physical addresses.

4.3.6 Linking of User tasks to the Kernel

The original SMT together with all the user tasks were built as one image. The stack and task entry points of each task were passed to the kernel at task build time, via a pointer called TKDFN.

The new kernel has the facility to reload tasks at run time. This necessitated the concept of linking the kernel to the tasks to be changed.

Figure 4.4

RETFIN PRIOR TO ADDITION OF MEMORY MANAGEMENT FACILITIES

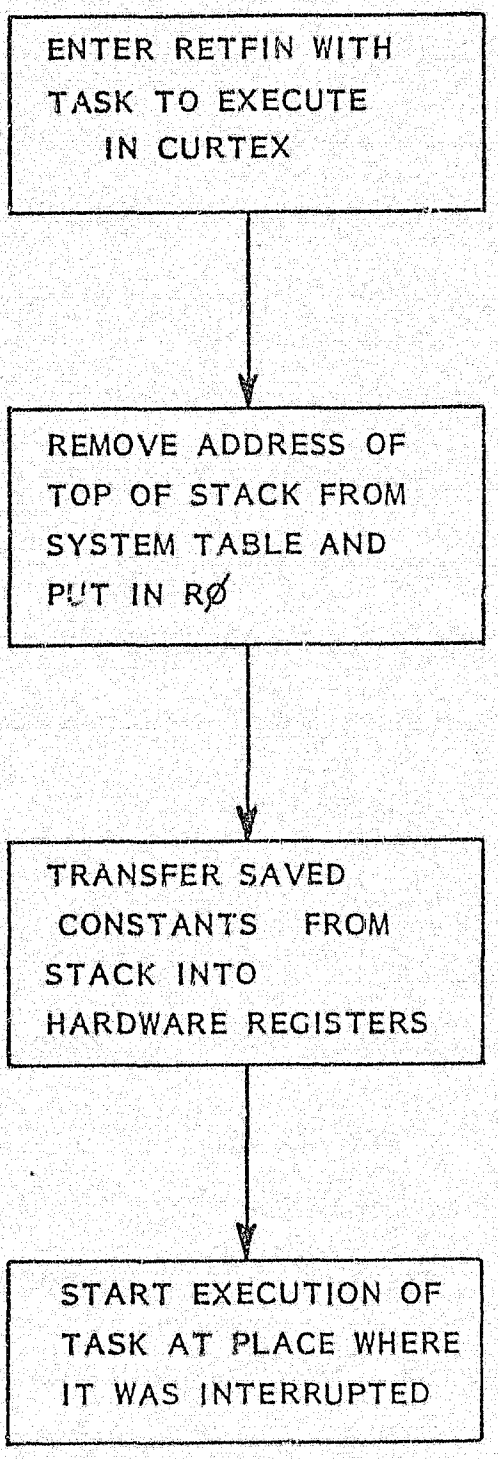


Figure 4.5

RETFIN AFTER THE ADDITION OF MEMORY MANAGEMENT FACILITIES

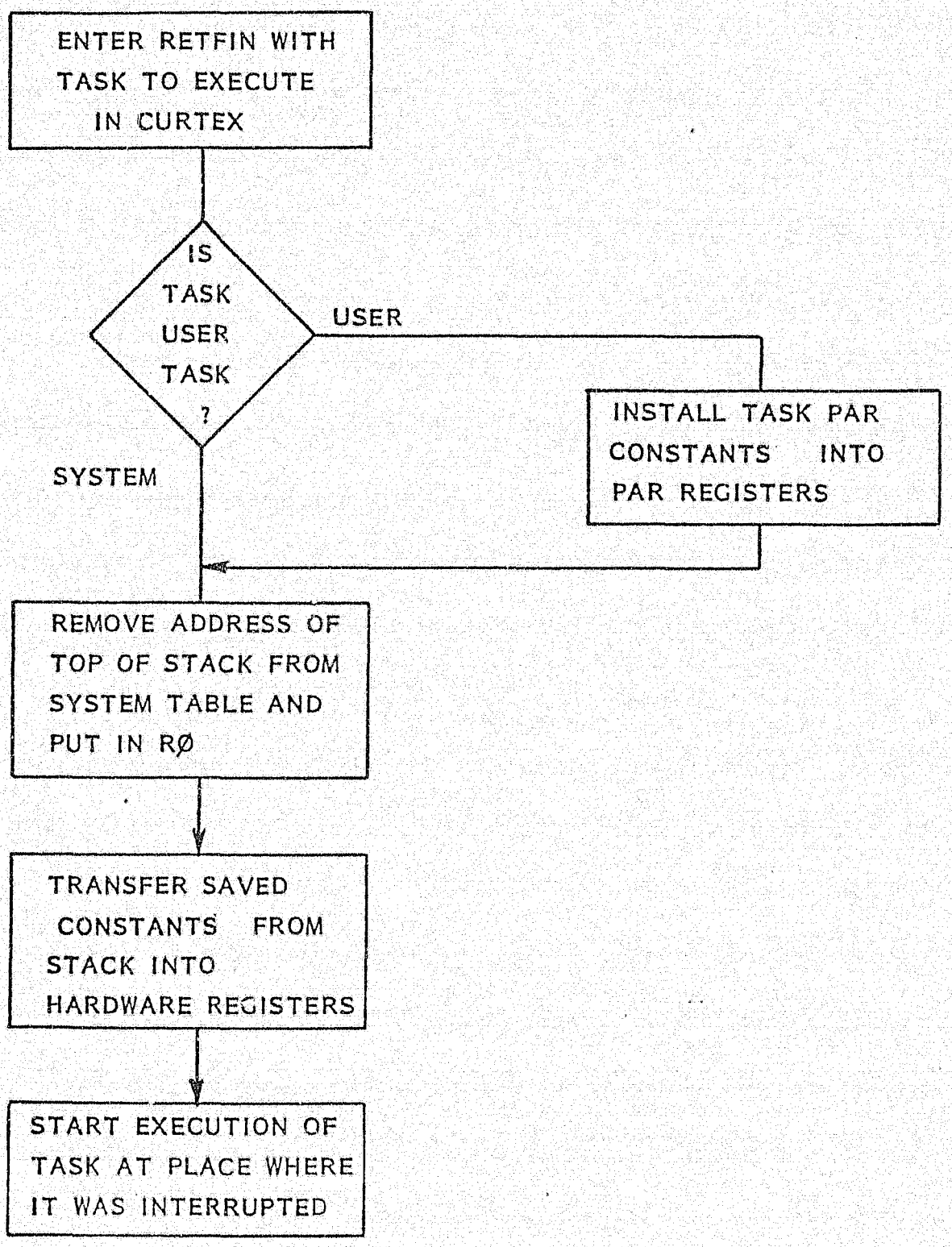
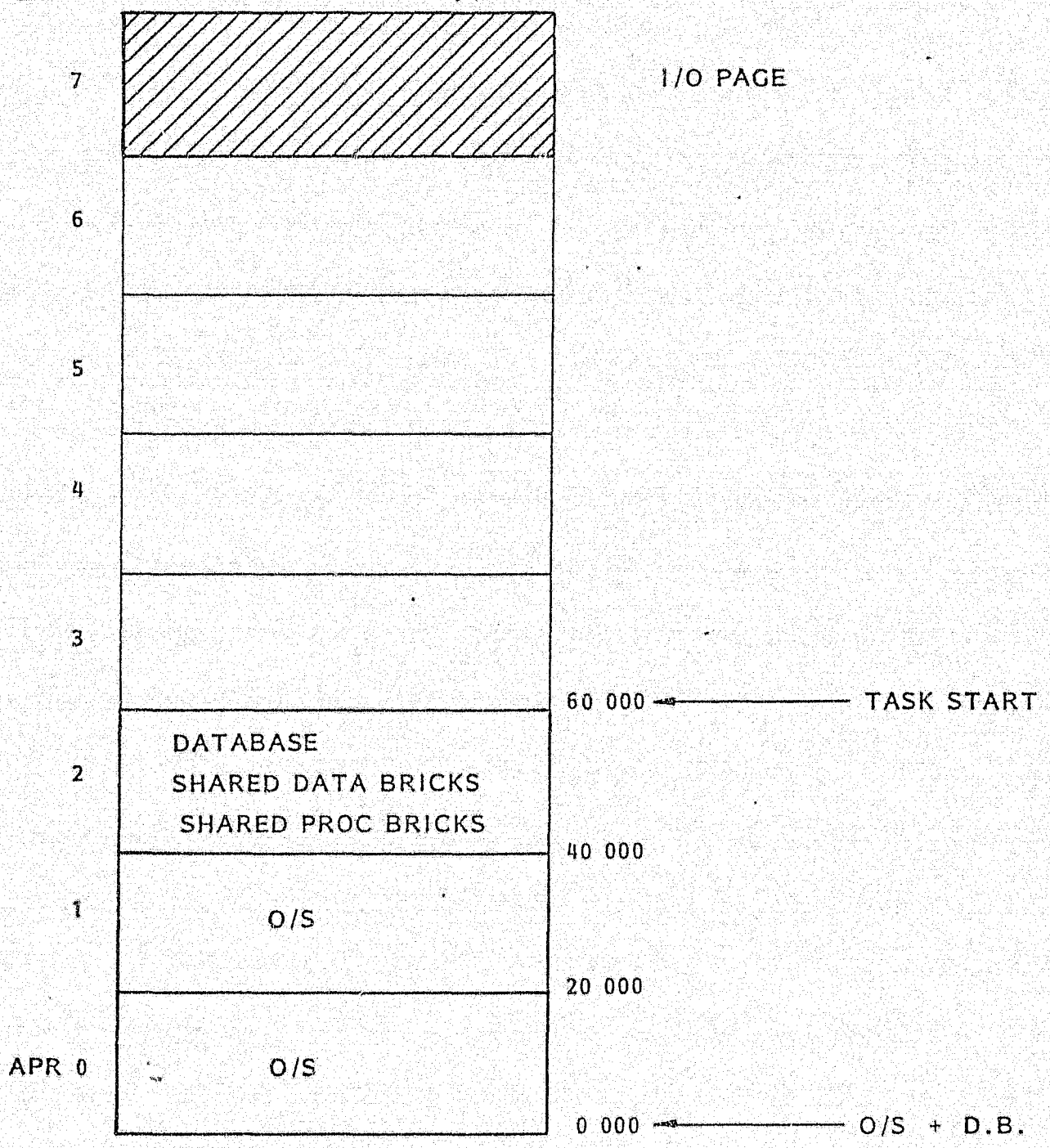


Figure 4.6

VIRTUAL ADDRESSING SPACE ALLOCATION & PHILOSOPHY



START ADDRESS OF ALL TASKS THE SAME ie $60\ 000_8$

ACTION OF MMU ON VIRTUAL ADDRESS TO PHYSICAL ADDRESS MAPPING

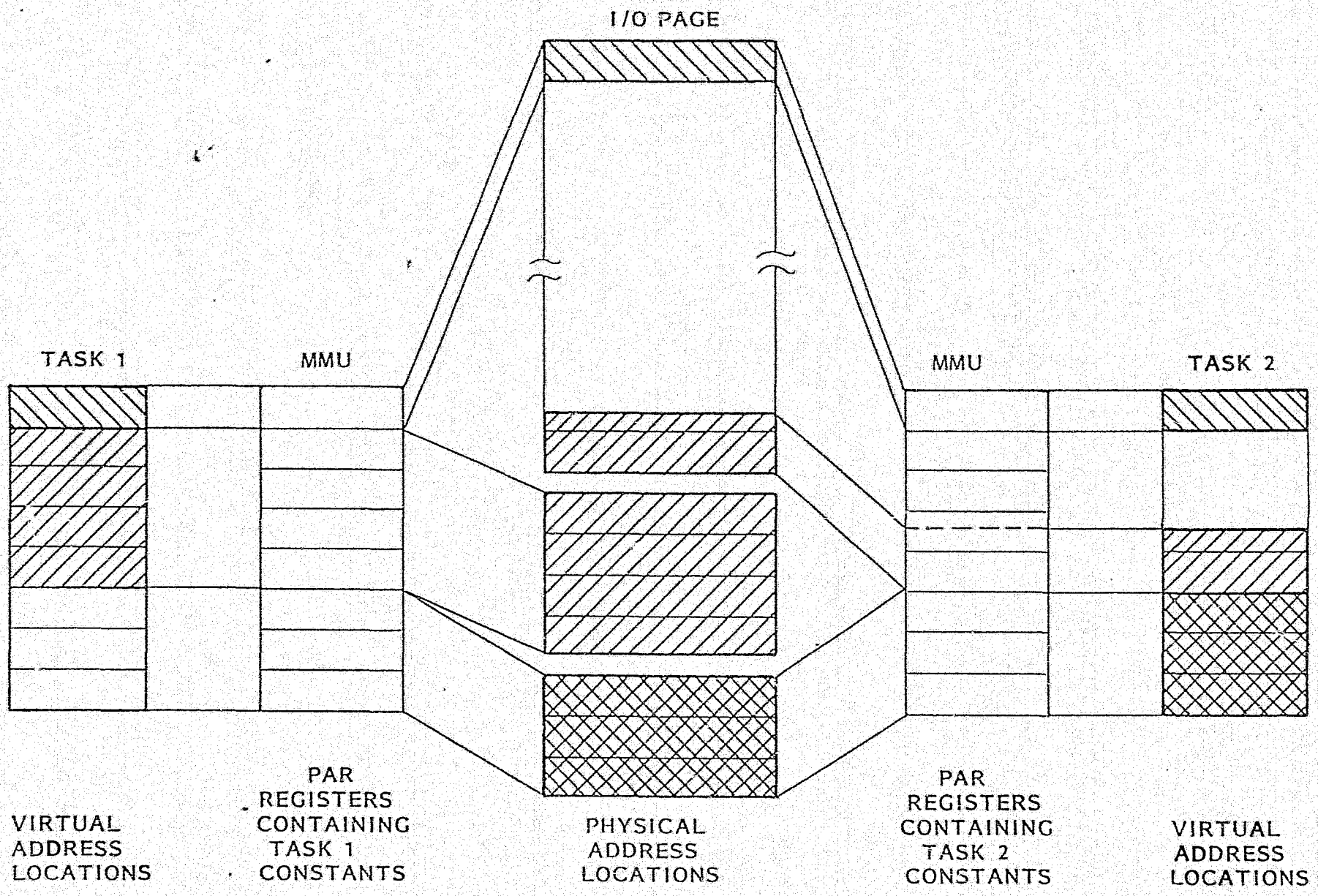


Figure 4.7

ACTION OF MMU ON VIRTUAL ADDRESS TO PHYSICAL ADDRESS MAPPING

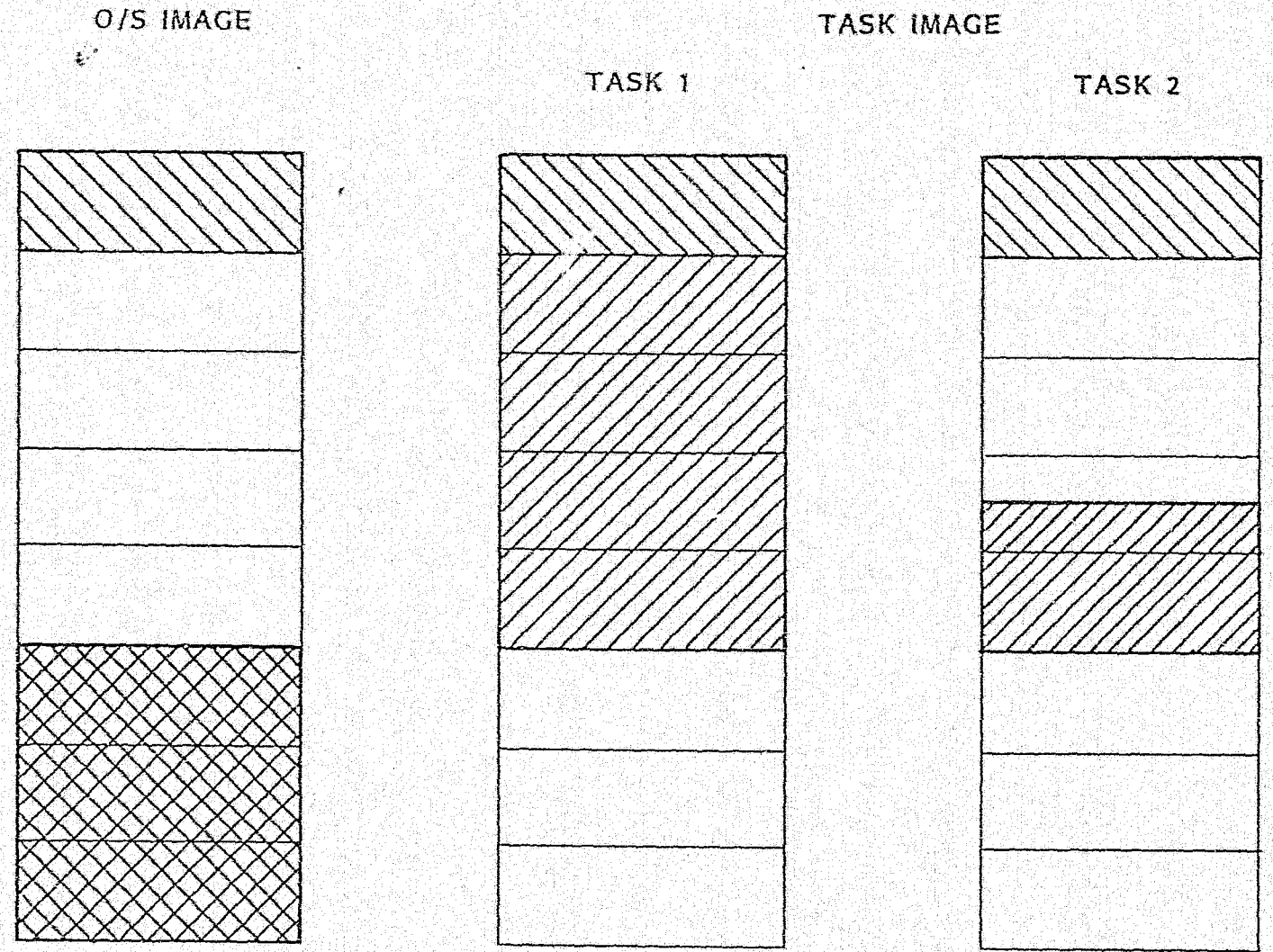


Figure 4.8

ACTION OF MMU ON VIRTUAL ADDRESS TO PHYSICAL ADDRESS MAPPING

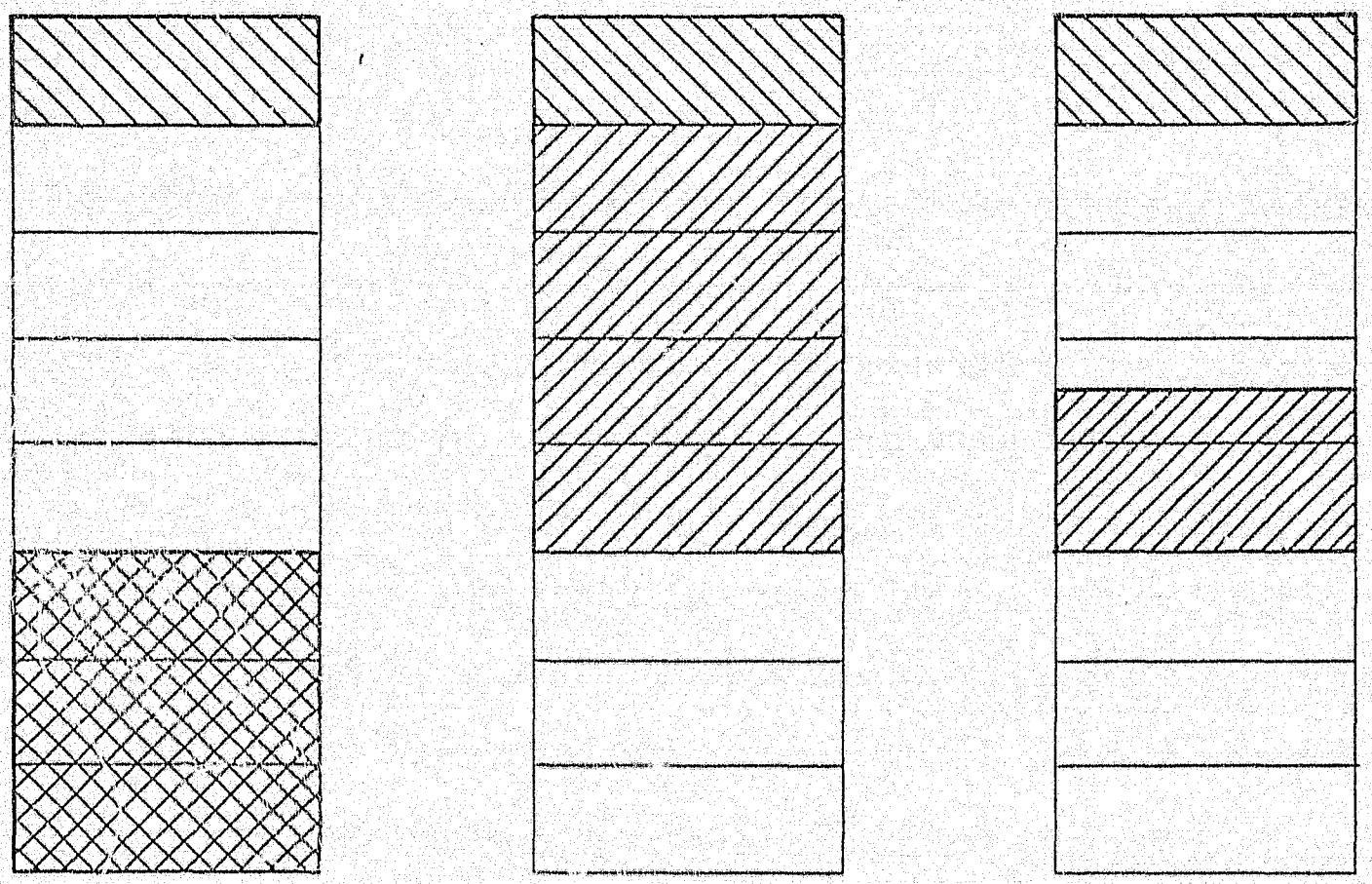


Figure 4.8

If the common area and all the user tasks are created as seperate operations, a technique had to be devised for making the user tasks known to the kernel. For a task to be known to the kernel, two entry points have to be known, namely the top of the stack and also the starting point of the task.

Therefore these two parameters have to be passed from the user task space to the kernel. The kernel has to know where to find these two points and the method used was to build each task with a dummy module (SMTTE) at the base of each task. This module contains 2 integers, these being pointers to the stack and task entry point (See Figure 4.9). These 2 variables are put in the databrick (in SMTTE) at task build time by RSX11M - the supporting development environement. Each task is task built individually with the symbol definition table produced by RSX-11M of the common area, so as to satisfy all the cross-refencing from the user task to the procedures in the kernel.

At initialization time when the kernel runs for the first time, the kernel takes the stack address (which is also the pointer to the SVC DATA area) of each task and puts it in an internal table called CELL in a data brick called TASKDATA. This can only be done dynamically and below a description is given of the method used to implement this dynamic task initialization.

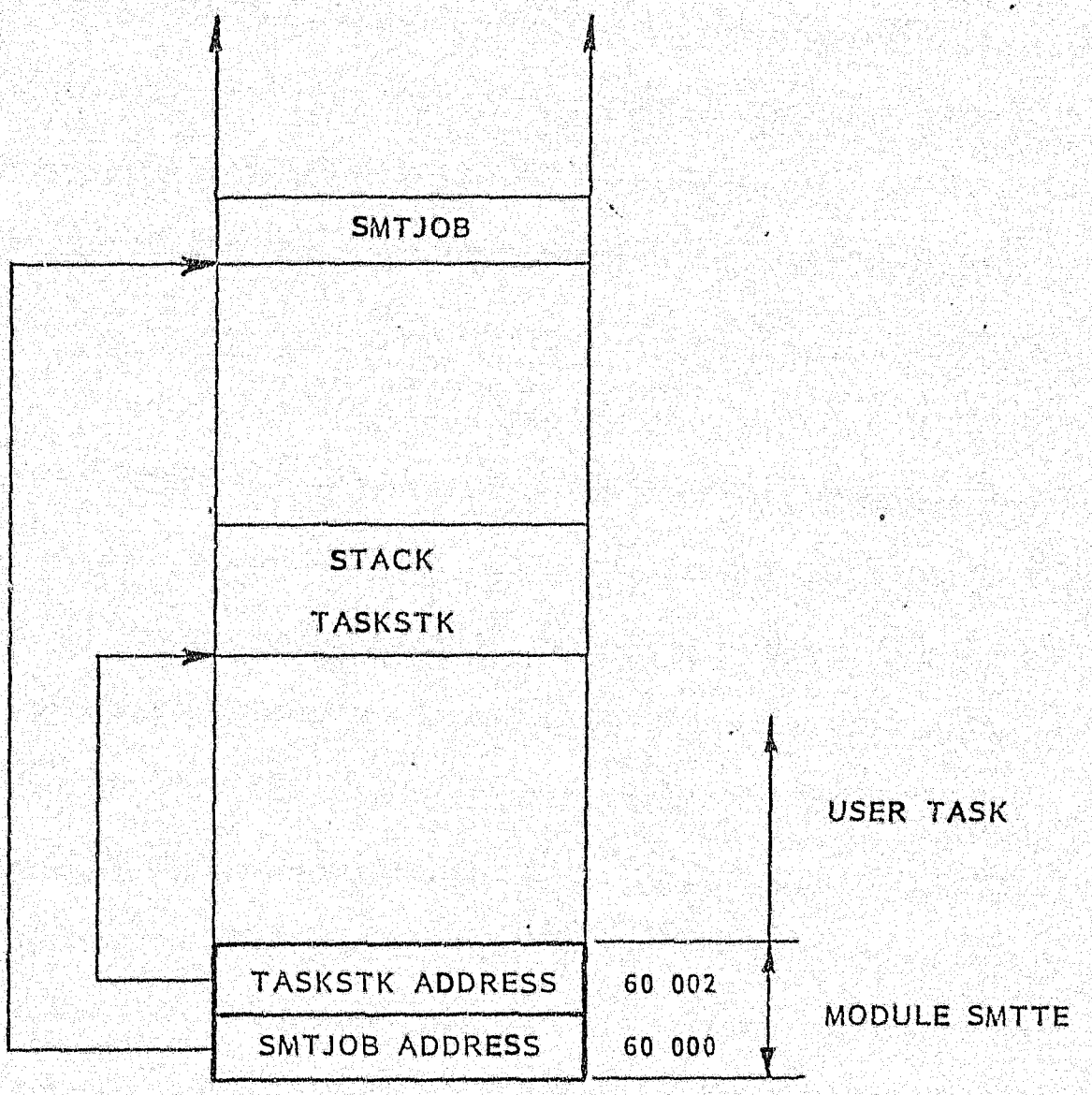
The kernel has to know that the task exists and also where it resides in memory. The technique used was to put a databrick containing the PAR constants at a place known to the loader task at system start up. The procedure followed is to first load the databrick first containing the PAR constants and then to use the information in this databrick to load the kernel and then to load the user tasks one at a time. As each task is loaded, the task PAR constants are installed into the mmu so as to bring the task into scope and then the two entry points are removed from SMTTE. Each task is initialized by a stack initialization procedure called USERTASKINIT. This procedure initializes the table CELL in TASKDATA and also puts the task entry point onto the task stack. See Figure 4.10 for the various steps in initializing a task. After all the user tasks have been loaded, the system scheduler table is set up.

4.4 CONCLUSION

The method used to implement memory management is described in detail in this chapter. The design philosophy used and the reasons for using the selected techniques are described. Discussion is also made on how the modifications affect the run-time aspects of the kernel. A brief look at the system startup or initialization was made in. For a full description of this aspect of system generation refer to reference 2.8.

Figure 4.9

ACTION OF SMTTE



THE INITIALIZATION PROCESS OF A USER TASK

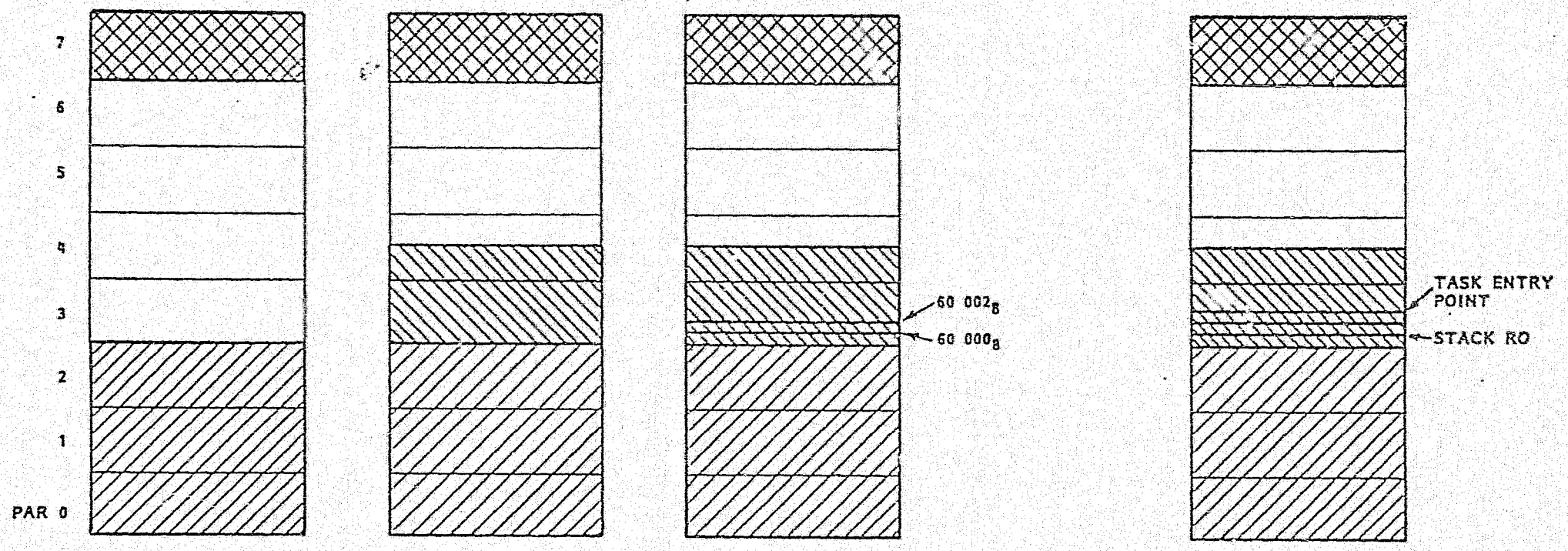
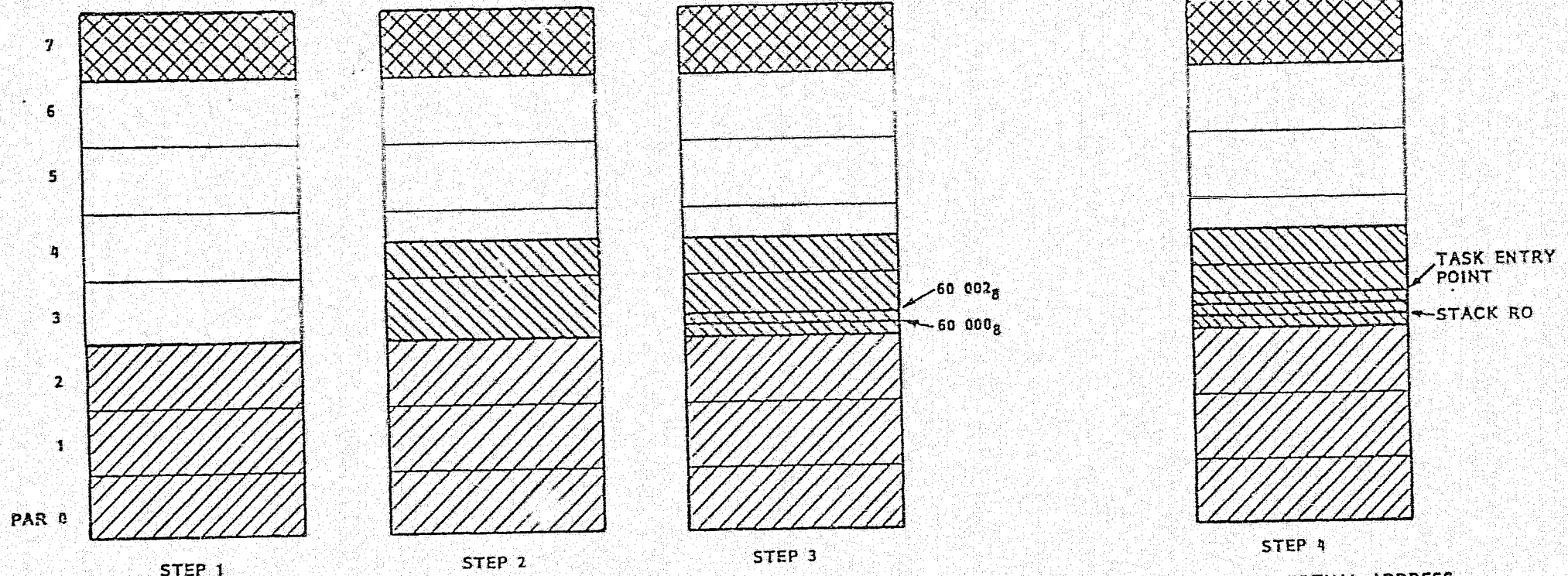


Figure 4.10

STEP 1	STEP 2	STEP 3	STEP 4
COMMON AREA & I/O PAGE PERMANENTLY INSTALLED PAR 3-6 NOT SET UP	INSTALL TASK & PAR 3-6 CONSTANTS IN MMU PUTS TASK IN SCOPE	LOCATION 60 000 CONTAINS PROC ENTRY ADDR = SMTJOB 60 002 CONTAINS STACK ENTRY ADDR = RO THESE TWO ADDRESSES PUT INTO DATABRICK TEP FOR MANIPULATION	-STACK VIRTUAL ADDRESS SAVED IN CELL (TASKDATA) -STACK INITIALIZED -PROC ENTRY ADDRESS PUT ON STACK

BEGIN SETFMQ	SETPAR	GETSTEP	STKINIT
	USERTASKINIT		

THE INITIALIZATION PROCESS OF A USER TASK



STEP 1

COMMON AREA & I/O PAGE PERMANENTLY INSTALLED
PAR 3-6 NOT SET UP

STEP 2

INSTALL TASK & PAR 3-6 CONSTANTS IN MMU
PUTS TASK IN SCOPE

STEP 3

LOCATION
60 000 CONTAINS PROC ENTRY ADDR * SMTJOB
60 002 CONTAINS STACK ENTRY ADDR = RO

THESE TWO ADDRESSES PUT INTO DATABRICK TEP FOR MANIPULATION

STEP 4

-STACK VIRTUAL ADDRESS SAVED IN CELL (TASKDATA)

-STACK INITIALIZED

-PROC ENTRY ADDRESS PUT ON STACK

BEGIN SETFMQ	SETPAR	GETSTEP	STKINIT
	USERTASKINIT		

Figure 4.10

CHAPTER 5

SUMMARY AND DISCUSSION

5.1 USER EXPERIENCE IN INSTALLATION

The modified kernel was tested on a process control application in AEGI Ltd. It was necessary to carry out the following modifications to the existing software in order to make is suitable for operation with the new kernel:

1. All the common data areas were grouped into one module.

2. All the common procedures were grouped into one module.

3. All the device drivers were grouped into a module.

4. All the user tasks were modified to be compatible with the new kernel's method of generating user tasks.

The conversion took approximately 1 week to complete. The software was loaded onto an off-line computer and it was checked and verified that operation was identical to the original system. The software was then loaded into the plant (i.e. on line) computer and has run without fault for the previous nine months prior to the submission of this dissertation.

The increase in overhead was measured and found to be minimal (of the order of 1%). This corresponds to the increase calculated in Appendix A.

5.2 SUMMARY OF KERNEL ENHANCEMENTS

The kernel was additionally enhanced to include an interrupt driven loader program and driver to interface to a DEC TU58 system. This driver is used to load the system at bootstrap time and it may also be used to load individual tasks. The driver may also be used by application tasks to transfer data to tape.

The old SMT was previously built as one image and then loaded into the machine. In the new memory managed version of SMT each task is built seperately and at startup, tasks are loaded one at a time under the control of the SMT startup program. This feature has made it possible to do modifications to a task on a host machine and to load the affected task into the target machine on-line. This feature is useful in that a single task can be loaded into the machine rather than loading the complete system.

5.3 SUGGESTIONS FOR FUTURE WORK

The project has been extremely successful in implementing the run-time aspects of the memory-managed SMT.

There are however areas which could be studied further, the most important being the programming support environment for the development of systems operating under SMT.

The original SMT kernel has an S-task used primarily for the operator to

communicate with the kernel. This task also has the ability to look at the contents of memory locations. This task in the new kernel has been configured as a user task and thus can only look at memory locations in the common area. It could be modified to look at any locations in physical memory thus enhancing its usefullness.

5.4 CONCLUSION

In conclusion, the addition of memory management facilities to SMT has been successful, with the new kernel operational in one on-line computer system of AECL Ltd. Although the memory managed SMT has a slightly increased overhead, the implementation is extremely efficient and the basic design concepts of the SMT kernel have been retained.

The enhancements and new features described in this report have made the SMT kernel more powerful and flexible and have enabled the full capability of the DEC hardware to be utilized. Further features have been added to the kernel to allow on-line loading of new or modified tasks.

APPENDIX A

INCREASED OVERHEAD TIME CALCULATIONS

This appendix indicates the method used to calculate the increase in overhead due to the introduction of memory management activities into SMT.

The scheduling or task changing mechanism in SMT is split into 2 procedures, namely the procedure CHANGE and the procedure RETFIN.

Figure A1 indicates how these two are related and where the statements increasing the overhead are situated.

The procedure CHANGE is entered after a call of a primitive (eg DELAY, WAIT etc) as outlined in chapter 3. The next task to run is chosen and a call of RETFIN is made.

RETFIN then proceeds to set up the RTL/2 environement for the chosen task. Two sections of logic have been added to RETFIN and it is this logic that has increased the overhead, the remainder of the procedure has remainded the same. The first stage checks to see if the task selected, is a user task or a system task. If it is a system task, the task is in scope and thus PAR constants need not be loaded into the machine registers. While if it is a user task the second stage is entered into.

In the second stage, the memory management registers are loaded , bringing the task into scope. The remainder of the procedure stays the same.

The calculation assumes that the module is compiled using the extended instruction set option (E1).

No of new instructions in the first stage

If a system task	- instructions	: 1
	- execution time	: 14.5 micro Seconds
user task	- instructions	: 3
	- execution time	: 20.5 micro Seconds

No of new instructions in the second stage

If a user task	- instructions	: 10
	- execution time	: 90 micro Seconds

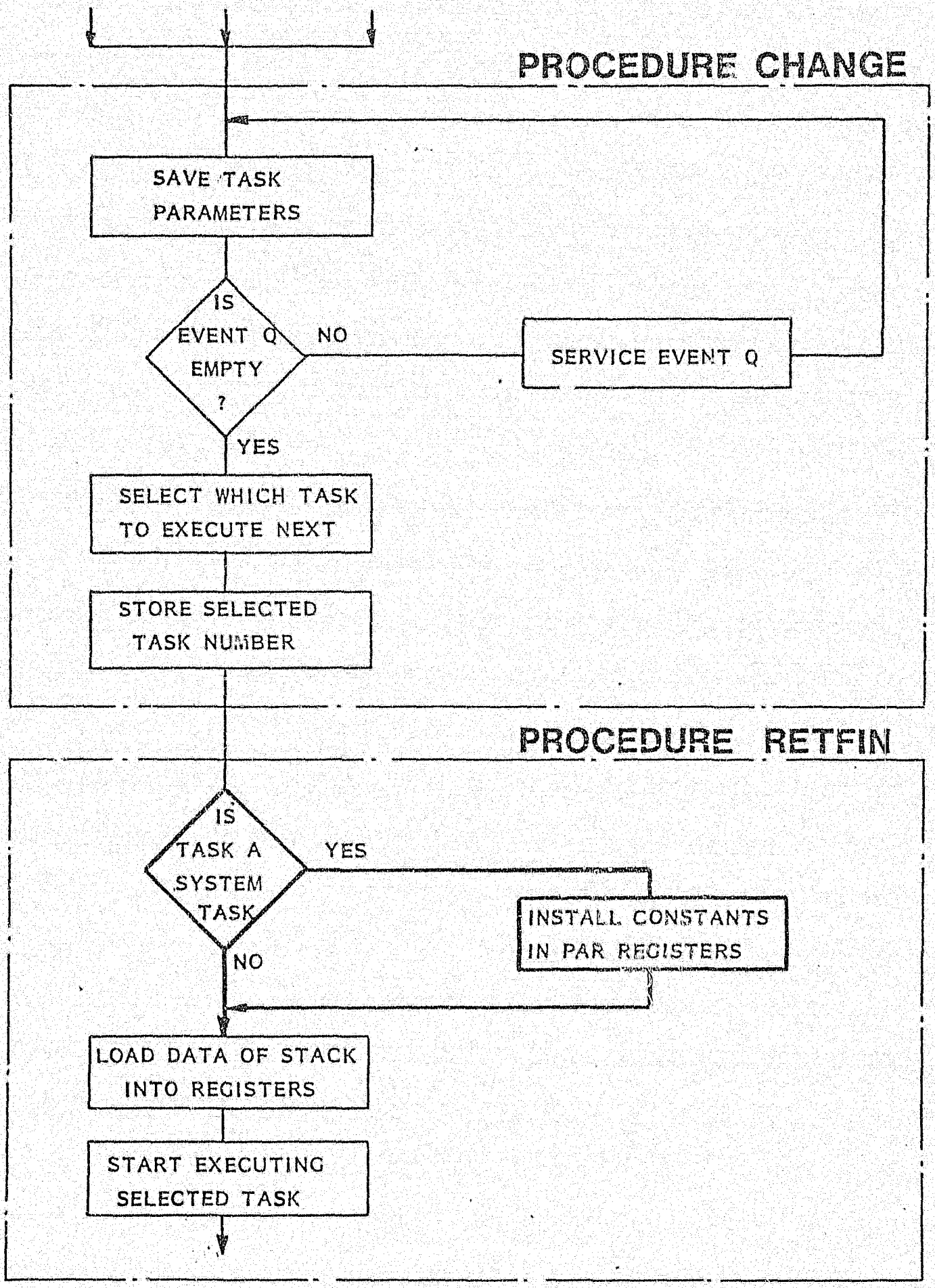
Increase in overhead time for a task change to:

A user task	- 4.5	microSeconds/task change
A system task	- 110.5	microSeconds/task change

An estimate of the number of times this will occur in one second, gives an indication of the overhead increase.

The clock task will execute every 20 mS , ie 50 times per second. If the user tasks are entered after each clock task execution, this implies that the user tasks may be entered into 50 times a second. This is true, if the user tasks never have time to complete, ie the processor is running at 100 % load. If the system load is 50% and the remainder of the time(50%), the fall back task executes, this implies that the user tasks will execute 25 times a second and

Figure A1


PROCEDURE CHANGE
SAVE TASK PARAMETERS
IS EVENT Q EMPTY ?
NO
SERVICE EVENT Q
YES
SELECT WHICH TASK TO EXECUTE NEXT
STORE SELECTED TASK NUMBER
PROCEDURE RETFIN
IS TASK A SYSTEM TASK
YES
INSTALL CONSTANTS IN PAR REGISTERS
NO
LOAD DATA OF STACK INTO REGISTERS
START EXECUTING SELECTED TASK

the system task will execute 25 times a second.

Thus

CLOCK TASK EXECUTIONS	50 x 4.5 =	225.0
FALL BACK TASK EXECUTION	25 x 4.5 =	112.5
USER TASK EXECUTION	25 x 110.5 =	2762.5
TOTAL IS		3100.0 micro seconds

as a percentage per second:

$$(3100.0) \times 10^{-6} \times 100$$

$$= 0.3\ \%$$

REFERENCES

1. Real-time Computing.
2. Real-time Operating System.
3. RTL/2.
4. Task Scheduling.
5. Memory Management.
6. Real-time Languages.
7. Special Issue of Journals.
8. Books.

1. Real-Time Programming

1.1 K J Kylstra "Real-Time Programming"
5th IFAC/IFIP International Conference on Digital Computer Application to Process Control.
June 1977.

1.2 J G P Barnes "An Introduction to Real-Time Programming"
SPL International.

1.3 I MacLeod "A case study in Real-Time Programming using RTL/2"
Real-time programming tutorial, Computer Society of South Africa, August 1980.

2. Real-Time Operating Systems

2.1 V Haase "Facilities Required in Operating Systems for real-time Language Programming, Computing with Real-Time Systems Volume 1 681-31.022 (41) London.

2.2 D A Anderson "Operating Systems", Tutorial Series No 9, COMPUTER, June 1981.

2.3 F van der Linden and I Wilson "Real-time executives for microprocessors" Microprocessors and Microsystems, Vol 4 No 6, July/August 1980.

2.4 P Elzer and R Roessler "Real-time languages and operating SYstems", Digital Computer Applications to process Control, VAn Nauta Lemke edition, North-Holland Publishing Company, 1977.

2.5 JP Collins & MM Lewitt , "An object Oriented Operating System for Micro computers" Computer Design, June 1982.

2.6 INTEL "iAPX86/30 Operating System Processor", INTEL Corporation 1981.

2.7 RC Varney "Towards the understandability of an Operating System", The Computer Journal, Volume 19 No3.

2.8 C Charalambous "The Memory Managed SMT - DSMT - User Manual", AECI Engineering Department, September, 1983.

2. Real-Time Operating Systems

2.1 V Haase "Facilities Required in Operating Systems for real-time Language Programming, Computing with Real-Time Systems Volume 1 681-31.022 (41) London.

2.2 D A Anderson "Operating Systems", Tutorial Series No 9, COMPUTER, June 1981.

2.3 F van der Linden and I Wilson "Real-time executives for microprocessors" Microprocessors and Microsystems, Vol 4 No 6, July/August 1980.

2.4 P Elzer and R Roessler "Real-time languages and operating SYstems", Digital Computer Applications to process Control, VAn Nauta Lemke edition, North-Holland Publishing Company, 1977.

2.5 JP Collins & MM Lewitt , "An object Oriented Operating System for Micro computers" Computer Design, June 1982.

2.6 INTEL "iAPX86/30 Operating System Processor", INTEL Corporation 1981.

2.7 RC Varney "Towards the understandability of an Operating System", The Computer Journal, Volume 19 No3.

2.8 C Charalambous "The Memory Managed SMT - DSMT - User Manual", AEGI Engineering Department, September, 1983.

3. RTL/2

3.1 J G P Barnes. "The Use of RTL/2 for Multitasking".
Minicomputer Forum, London, February 1975.

3.2 I Gray. "Summary of Aims and Progress of an RTL/2 Project".
Computing with Real-Time Systems. Volume 1 681.32.022(41) London.

3.3 J G P Barnes. "The Development of RTL/2".
The Future of Real-Time Languages in Process Control. The British Computer Society, September 1979.

3.4 J G P Barnes. "The Standardizat' of RTL/2".
Software Practice and Experience.

4. Task Scheduling

4.1 E A Parish Jnr. and V K L Huang. "A Schedule for Real-Time Task Control in Microcomputers".
IEEE Trans. on Industrial Electronics and Control Instrumentation Volume IEC1-25, No.1, February 1978.

5. Memory Management

5.1 D .I Tanner "Clearing Up the Confusion : Virtual vs Mapped Memory". Computer Design October 1976.

6. Real-Time Language

6.1 "A Comparison of Two Real-Time Programming Languages" J R Taylor UK Atomic Energy Authority, Papers on Real-Time Programming.

6.2 J G P Barnes "Real-Time Language for Process Control". The Computer Journal Volume 15 No.1.

6.3 T J Williams "Needs and Desires for a Standardized Real-Time Programming Language". IFAC/IFIP First Software Symposium for Computer Control, May 1976.

6.4 J G P Barnes "Real-Time Primitives and the US HOL Proposals" SACAC Symposium on Real-Time Software for Industrial Applications Pretoria S A, November 1978.

6.5 N Wirth "Towards a Discipline of Real-Time Programming". Communications of the ACM August 1977 Volume 20 No.8.

7. Special Issues of Journals

7.1	Computer	May 1977	Stack Machines
7.2	Computer	June 1974	Operating Systems
7.3	Computer	October 1976	Operating Systems

8. Books

8.1 S T Allworth "Introduction to Real-Time Software Design".

8.2 P B Hansen "Operating System Principles".
Prentice Hall Inc, New Jersey 1973.

8.3 SPL REVIEW " RTL/2 SPECIAL" SPL ref RTL/2600/1ed/175

8.4 G. Goos & J. Hartmanis "Microcomputer System Design"
Springer-Verlag, New York, 1981.

8.5 J G P Barnes "RTL/2 Design and Philosophy", London, Heyden, 1976.

8.6 RTL/2 TRAINING MANUAL, ICI Ltd , Mond Division, Runcorn,1974.

8.7 RTL/2 STANDARD STREAM I/O, ICI Ltd, Mod Division, Runcorn, 1974.

THE FOLLOWING DOCUMENT ATTACHED IS REFERENCE 2.8 MENTIONED IN THE THESIS ENTITLED "THE ADDITION OF MEMORY MANAGEMENT FACILITIES TO A REAL-TIME OPERATING SYSTEM".

SMT OPERATING SYSTEM FOR PDP-11

UTILIZING MEMORY MANAGEMENT

DATE : 29th SEPTEMBER 1983

PURPOSE : THIS MANUAL DESCRIBES THE MODIFICATIONS TO THE OPERATING SYSTEM CALLED SMT, TO INCLUDE MEMORY MANAGEMENT. SMT IS WRITTEN MOSTLY IN RTL/2, TO RUN ON DEC PDP-11's. THE DEVELOPMENTS CARRIED OUT NOW ENABLE SMT TO ADDRESS 128K WORDS (18 BITS) OR 2M WORDS (22 BITS).

AUTHOR : C CHARALAMBOUS

REF : SMT/DSMT/CC

AECI ENGINEERING
P O BOX 796
GERMISTON
1400
TRANSVAAL
SOUTH AFRICA

CONTENTS

PART A - Introduction to the Developed SMT - DSMT

A1 General Description.
A2 Reason for the Modifications.

PART B - Memory Management Philosophy/Implementation

B1 Description of SMT prior to the Modifications.
B2 Hardware Considerations.
B3 Memory Management.

B3.1 Introduction.
B3.2 Memory management and PAR Swopping.
B3.3 P/R Assignments.

B4 User Task Stack.
B5 The Common Area.
B6 Linking of User Tasks to Operating System.

PART C - Modification to SMT/SMT Maintenance Information

C1 SMT Modules.
C2 SMT Procedures.
C3 SMT Data Bricks.
C4 SMT Notation.

PART D - General Topics

D1 Keyboard Commands.

PART E - Building SMT Systems

E1 Defining the Common Area.

E1.1 Introduction.
E1.2 Editing.
E1.3 Set up DSMTU2.
E1.4 Compiling Common Area Modules.
E1.5 Modifying the Common Area.

E2 Defining the User Tasks.

E2.1 Introduction.
E2.2 Editing.
E2.3 Compiling.

E3 Task Building.
E4 Bootstrap Procedure and User Task Data Brick.
E5 Loading Images onto Tape.
E6 Loading Tasks on Line.
E7 Initial Loading and Bootstrapping of DSMT.