CONTENTS

APPENDICES

1. SYSTEM TASKS, EVENTS AND FACILITIES.

2. UNRECOVERABLE ERROR NUMBERS.

3. RRGEL PR' 'TOUT DESCRIBED.

4. PAR AND TAPE LAYOUT ASSIGNMENT TABLES.

5. CURTASK, CURTEX, NUSERTASK COMPARED.

6. MODIFYING THE COMMON AREA.

APPENDICES

1. SYSTEM TASKS, EVENTS AND FACILITIES.

2. UNRECOVERABLE ERROR NUMBERS.

3. RRGEL PRINTOUT DESCRIBED.

4. PAR AND TAPE LAYOUT ASSIGNMENT TABLES.

5. CURTASK, CURTEX, NUSERTASK COMPARED.

6. MODIFYING THE COMMON AREA.

ABBREVIATIONS

SMT - STANDARD MULTITASKING SYSTEM.

DSMT - DEVELOPED SMT.

MMU - MEMORY MANAGEMENT UNIT.

APR - ACTIVE PAGE REGISTER.

PAR - PAGE ADDRESS REGISTER.

PDR - PAGE DESCRIPTOR REGISTER.

PA - PHYSICAL ADDRESS.

VA - VIRTUAL ADDRESS.

O/S - OPERATING SYSTEM.

REFERENCES

1. SMT OPERATING SYSTEM FOR PDP-11

   - RTL/2 REFERENCE : 122 VERSION 2
   - SPL
   - FEBRUARY 1980

2. MODIFIED SMT VERSION 18 SOFTWARE MODULES.

3. DEC Process Handbook.

PART A

A1 General Description

SMT (Standard Multi-Tasking) is a small operating system developed by ICL Limited to run off the DEC PDP-11 range of computers. This report outlines the modifications carried out to SMT to include memory management which enables the operating system to address 128k or 2M words memory. Further features have been added to the o/s, to allow on line loading of new or modified tasks. This manual also outlines some of the concepts used in SMT to achieve task changing as well as how the system is put together.

This report should be read in conjunction with the reference 1 to gain a full understanding of the o/s. This report does not replace Reference 1 but explains the modifications carried out to include Memory Management.

The operating system was renamed Developed SMT (DSMT) to distinguish between the ICL developed o/s and the AECI modified version.

A2 Reason For Modification

The SMT o/s is used extensively within AECI. The o/s has certain limitations which have prevented further features from being added to the computer systems running on SMT. The prime limitations was that the o/s did not utilise the memory management unit (mmu) supplied by DEC, used in conjunction with the PDP-11 processors. This has limited systems to 28k words of memory.

The o/s has now been upgraded to utilize the mmu and to address 128k words using the 18 bit address bus or 2M words using the 22 Bit Bus.

The report outline the procedures to design a system which can cater for tasks up to 16k words long and a common area of 4k words. The method used to change this is descriped in Appendix 6.

PART B

Memory Management Philosophy/Implementation

B1 Description of SMT Prior to Introduction of the Modifications

SMT is a small real time o/s designed to run on the DEC PDP-11 processors. The o/s supports 16 user tasks and provides facilities which control the interactive and communication between the tasks and the outside world.

The user tasks communicate with the o/s via the task and facility control procedures. It is these procedures and the clock task which control the task changing mechanism of SMT. (see Figure B1)

A task change can occur due to 3 possible reasons:

1. A task control proc was called.
2. A facility control proc was called.
3. A clock interrupt occurred which forced the processor to execute thw CLOCK task and then procedes to CHANGE.

(see Figure B2)

After the execution of any one of these procs, the machine procedes to execute the SVC proc called CHANGE which executes as an H task.

The proc CHANGE merely selects which task to run after having firstly checked that the event Q is empty. If the event Q is not empty it calls RETEV which empties the event Q.

If the event Q is empty it checks the STAT word of the highest priority task to see if all the conditions are OK to run it i.e. STAT=0 .... not stopped, or delayed or waiting for an event.

CHANGE continues down the STATAD array (since it was set up in SETFMQ in priority order) until it finds the highest priority task with it's STATUS word OK.

CHANGE then saves the task number in CURTEX and then calls RETFIN which sets up the task environment to run the task.

A task may also be interrupted. When this occurs the task state is saved by a CALL of PIORTL, which uses the hardware stack while executing the servicing routine. After the interrupt servicing routine has been completed, the routine may either terminate by putting an EVENT in the Q which effectively calls CHANGE or a call may be made of RETFIN which continues the execution of the interrupted task. (see Figure B3)

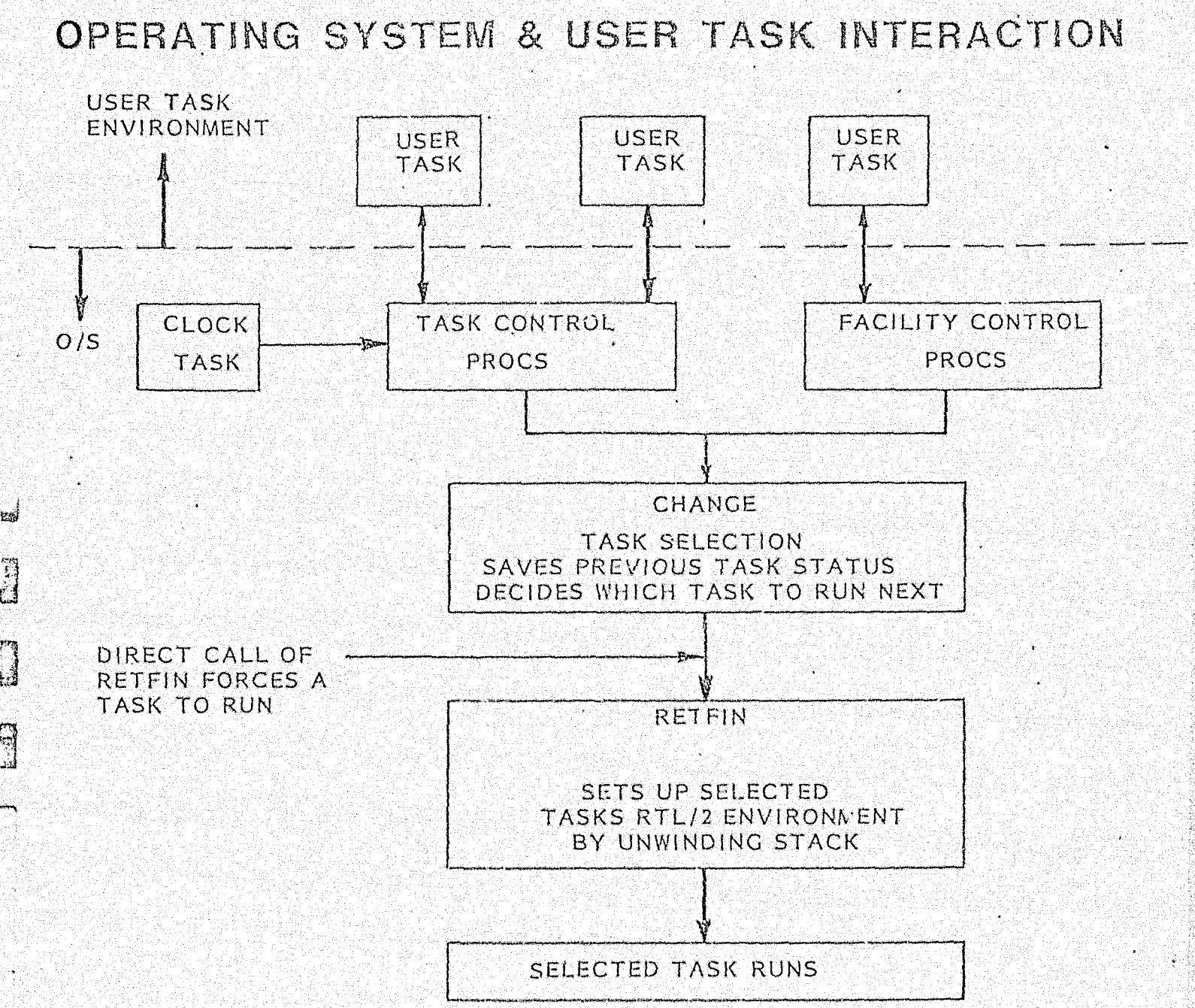
RETFIN takes the task number in CURTEX and replaces the machine registers with the values saved on the stack. Once all the registers have been replaced the task may continue from where it left off.

B2 Hardware Considerations

The mmu has 2 modes of operators. The one being "Kernel" mode and the other "User mode".

To keep SMT as simple as possible it was decided to have all the o/s and user task operates in Kernel mode.

Figure B1



TASK CONTROL PROCS

STOP
START
DELAY
WAIT
WAITFOR
SET
RESET

FACILITY CONTROL PROCS

SECURE
RELEASE

OPERATING SYSTEM & USER TASK INTERACTION

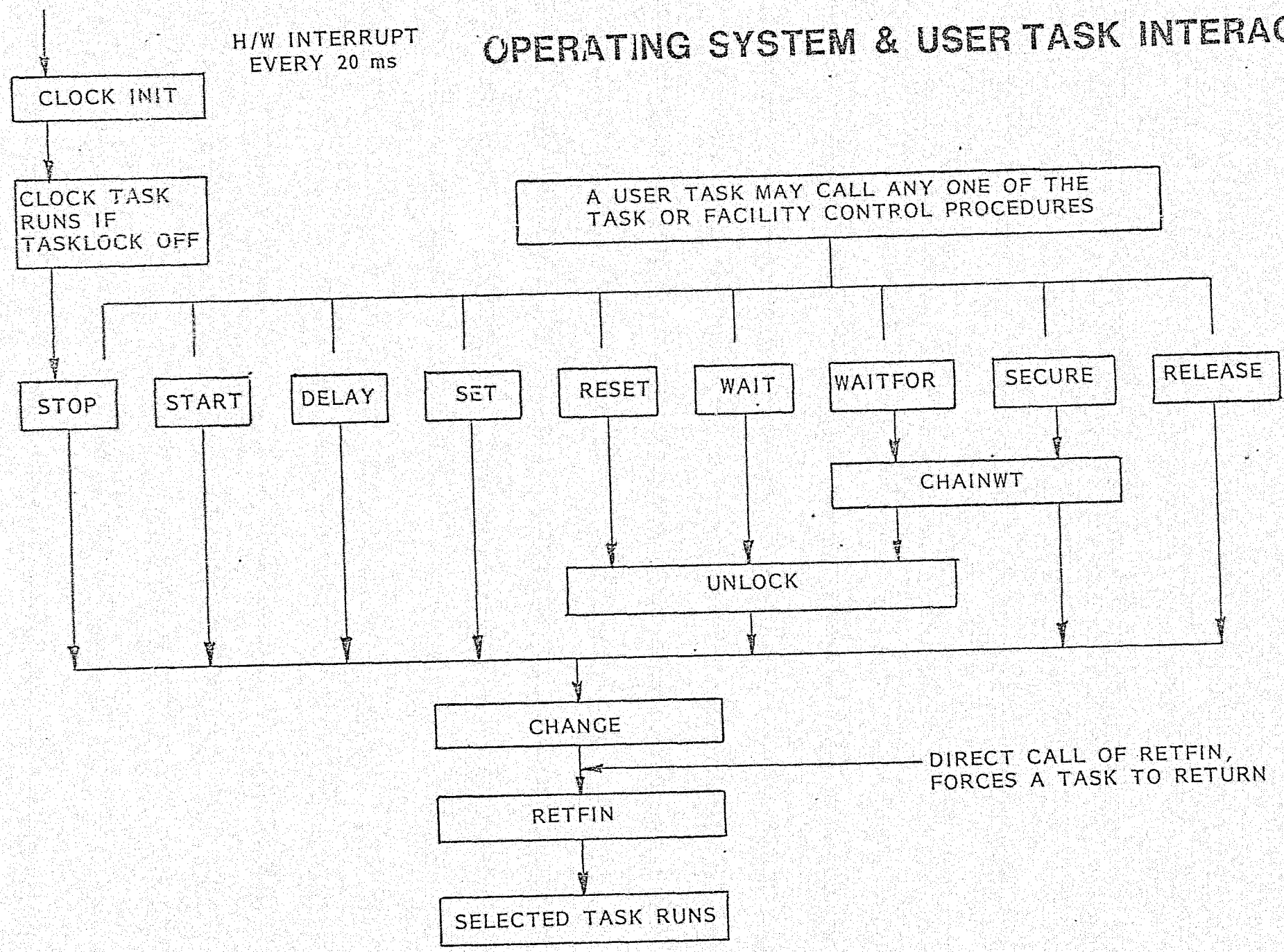
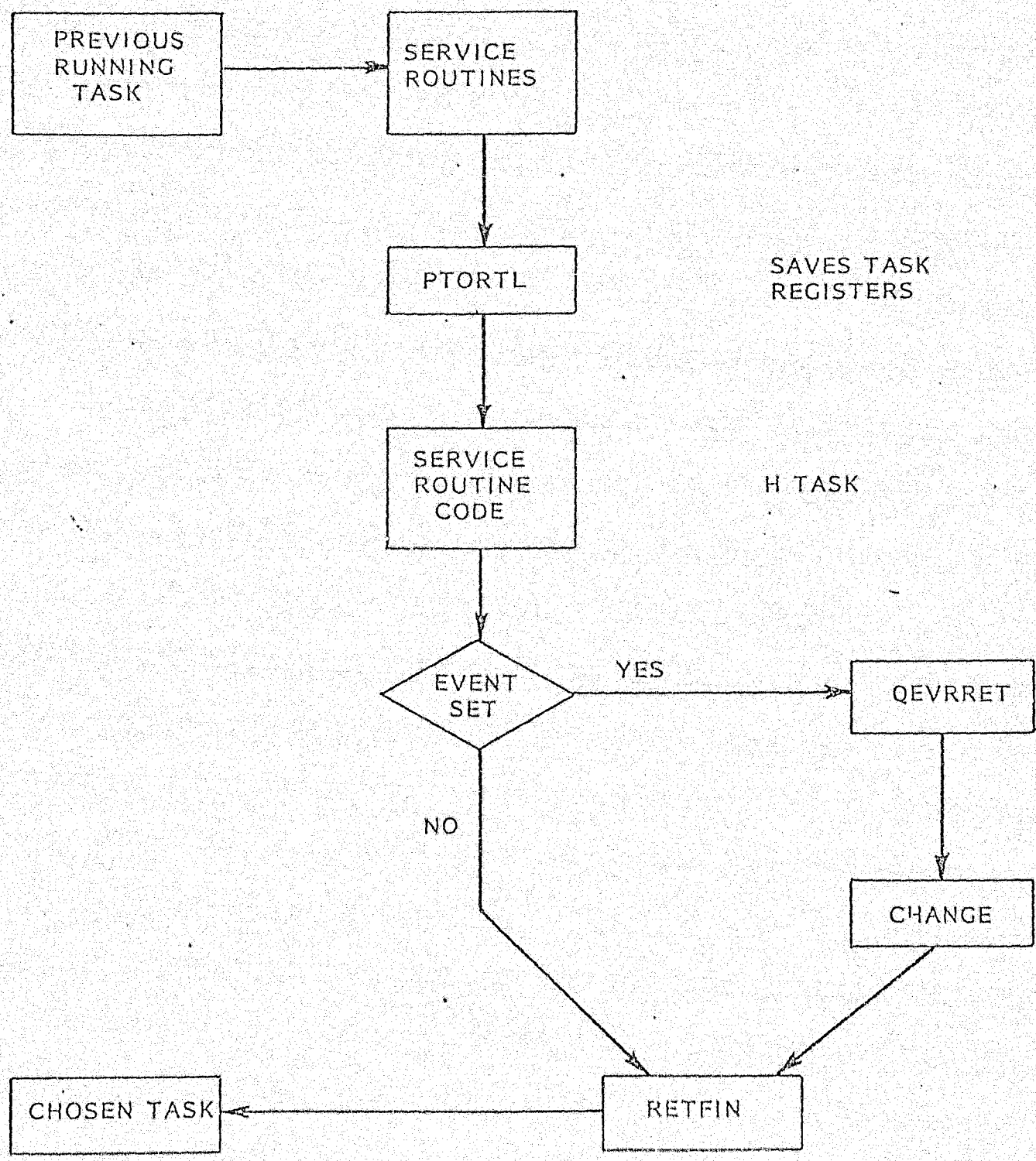


Figure B2

Figure B3

S-TASK CHANGE DUE TO AN INTERRUPT



This implies that bits 12, 13, 14, 15 of the PSW has to be 0 to operate in the Kernel mode.

The virtual addressing space of the mmu has a fixed length of 32k words. This length is divided into 8 sections, each of 4k words length. Each section has 2 registers associated with it namely the PAR (Page address register) and the PDR (Page description register).

The PAR contains a mapping constant to which the virtual address is added. This new address is the new address in physical memory.

The PDR provides information to the processor on the sections accessibility (read/write, write only etc) length etc.

For simplicity sake the PDR have been made fixed so that for all the tasks the same constants apply i.e. all APR section are 4k long, read/write accessible and upward expanding.

The mmu has a status register 0 (SRO) which contains the abort error flag, memory management enable mode of operating, various other maintenance information.

To enable the mmu bit 0 of SRO has to contain a zero, otherwise the virtual address coincide with the physical address u mmu does not map the VA to the PA.

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 SRO and SR2 so that when RRGEL is called they may be printed out.

Reference 3 should be consulted to obtain details of the above.

B3 Memory Management

B3.1 Introduction

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

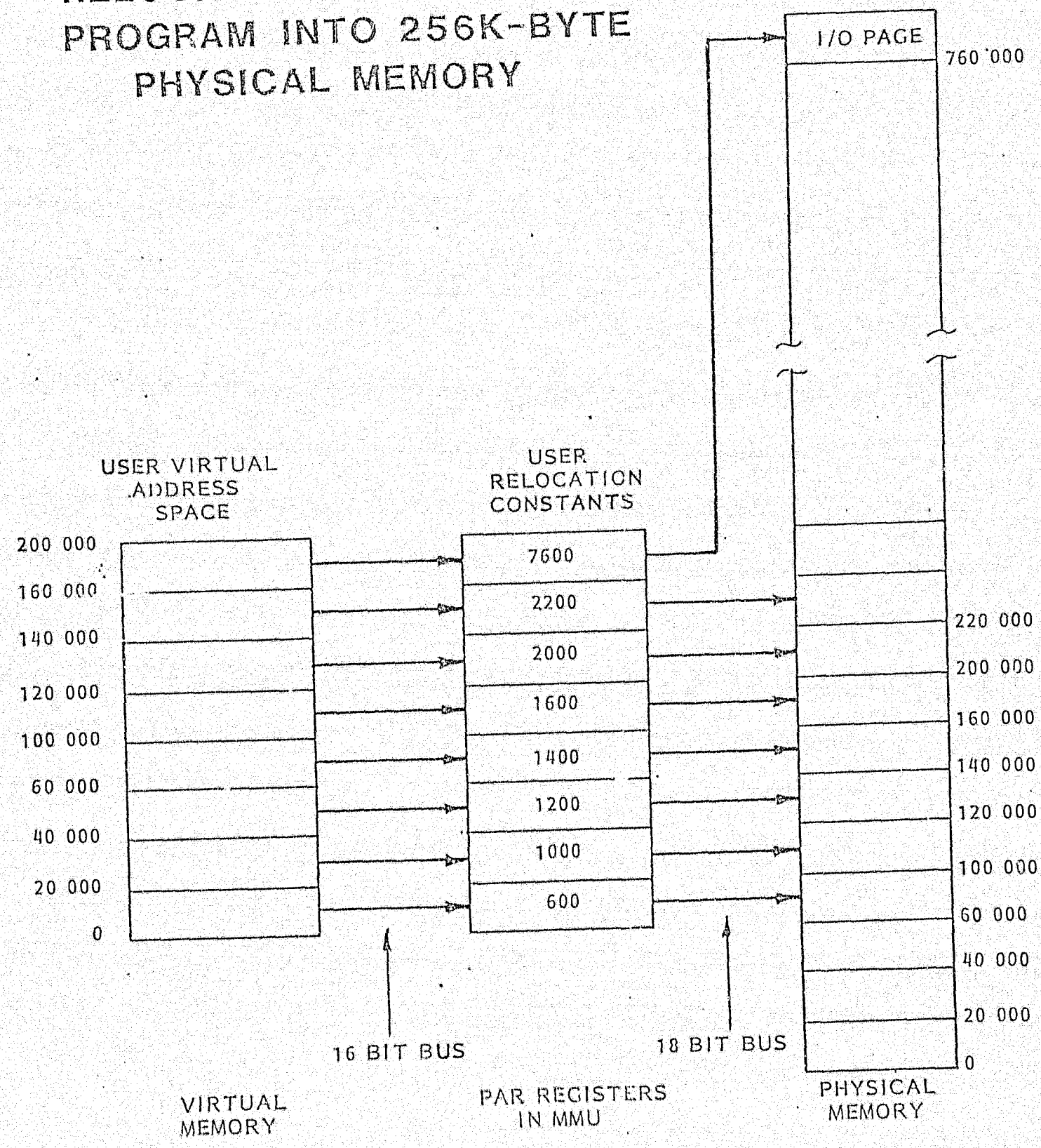
B3.2 Memory Management Philosphy and PAR Swopping

The basic concepts of memory management lies in mapping. What the mmu does is take a single virtual space address and maps it 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 operating system 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 sitting in memory. (see Figure B4)

Figure B4

RELOCATION OF 64K-BYTE PROGRAM INTO 256K-BYTE PHYSICAL MEMORY



It seemed logical to put the swopping of the PAR's at the beginning of RETFIN, 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.

By doing it in the manner described above, the whole interrupt handling mechanism is not affected.

B3.3 PAR Assignments

After having decided where the PAR constants are to be swopped the next question was which PAR constants should be swopped.

The philosophy used is shown in Figure B5.

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

APR0 and APR1 contain the operating system.

APR2 has been left aside for use as a database/common proc/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.

Figure B6 and B7 illustrate two tasks residing in virtual space and how they map to physical addresses.

B4 User Task Stacks

SMT supports RTL/2 is a stack based operating system in which the state of a given task may be given by the last link cell on the stack.

Each task has to have 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.

Thus each task carries with it in the user task space, (above the common area) its own stack. The operating system link to the stack at initialization time. (See Section B6 on run time linking).

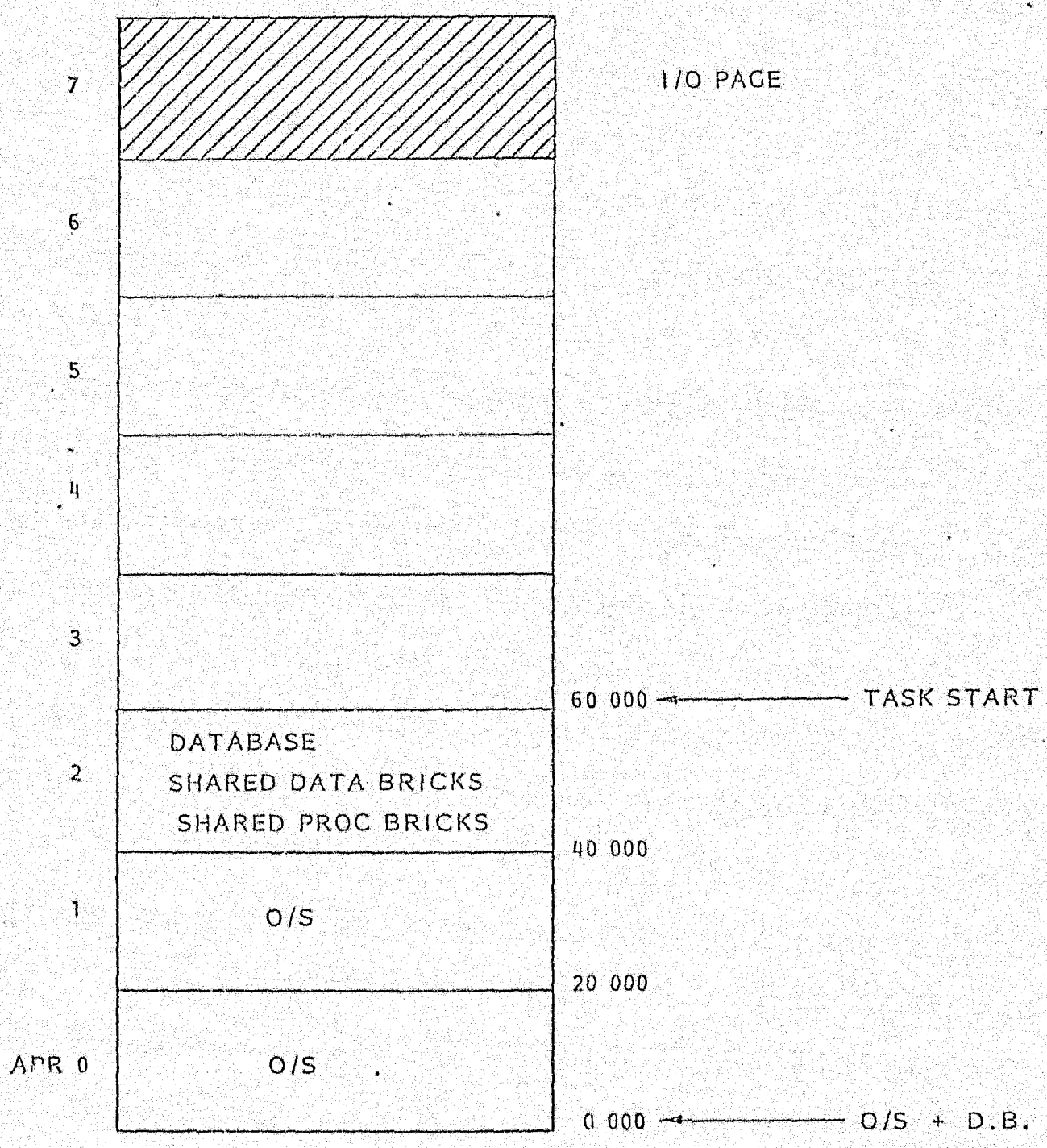
B5 The Common Area

The system has been designed with a common area of 12k words i.e. in APR0,1,2 we can install the following:

1. SMT operating system (approx. 7k words long).
2. I/O drivers with buffers defined by the user.
3. Database.
4. Common data bricks.
5. Common proc bricks.

Figure B5

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

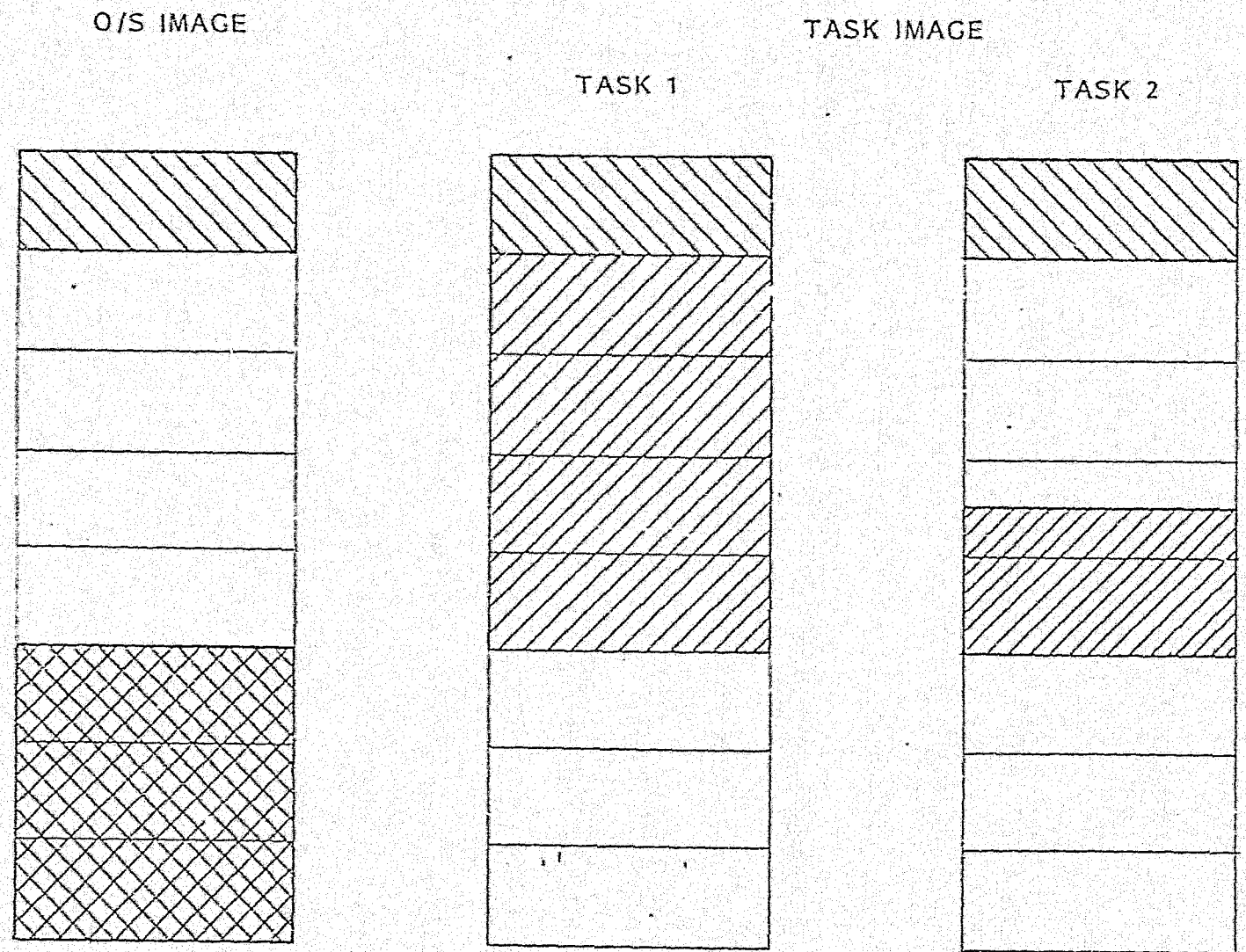


Figure B6

ACTION OF MMU ON VIRTUAL ADDRESS TO PHYSICAL ADDRESS MAPPING

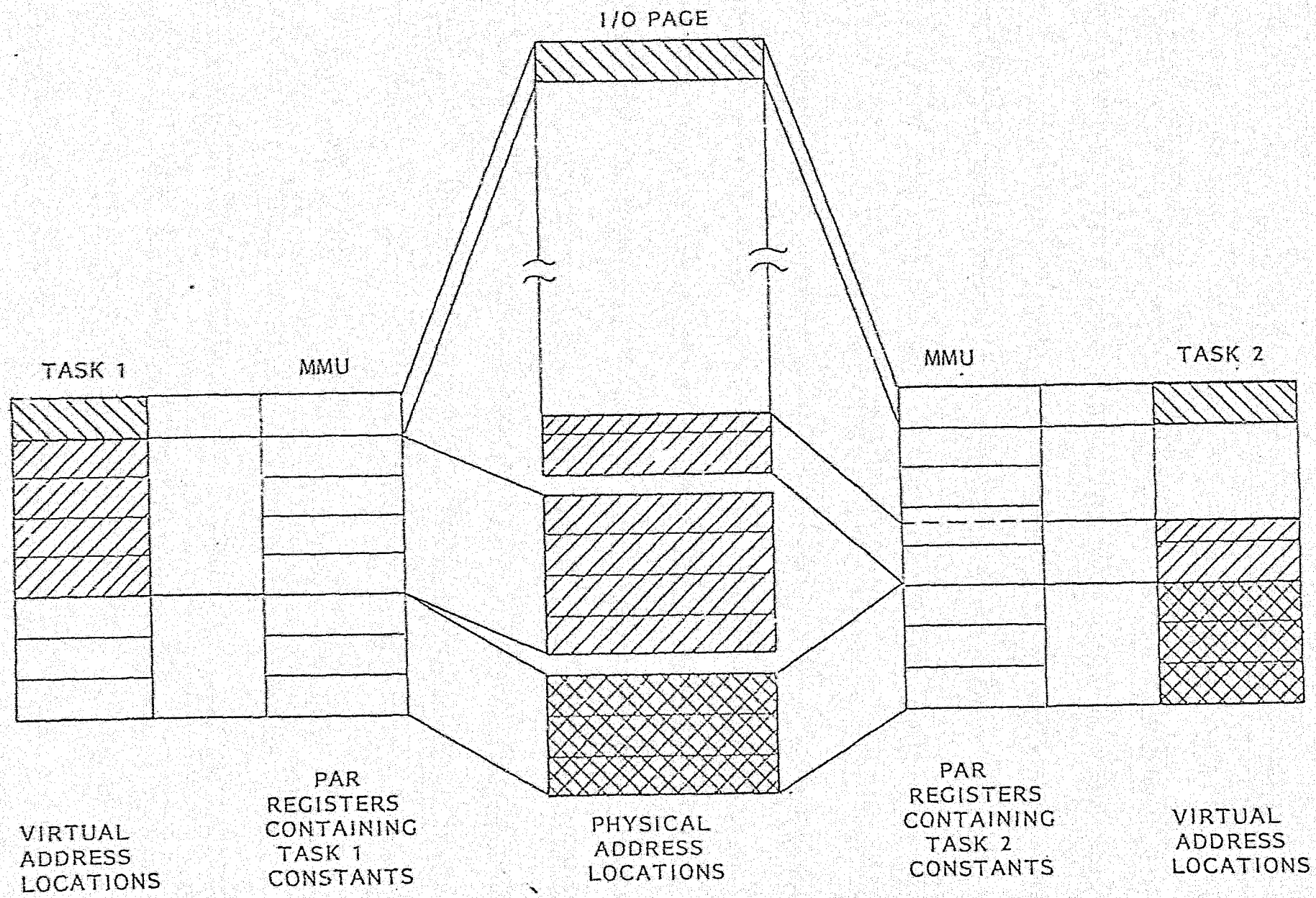
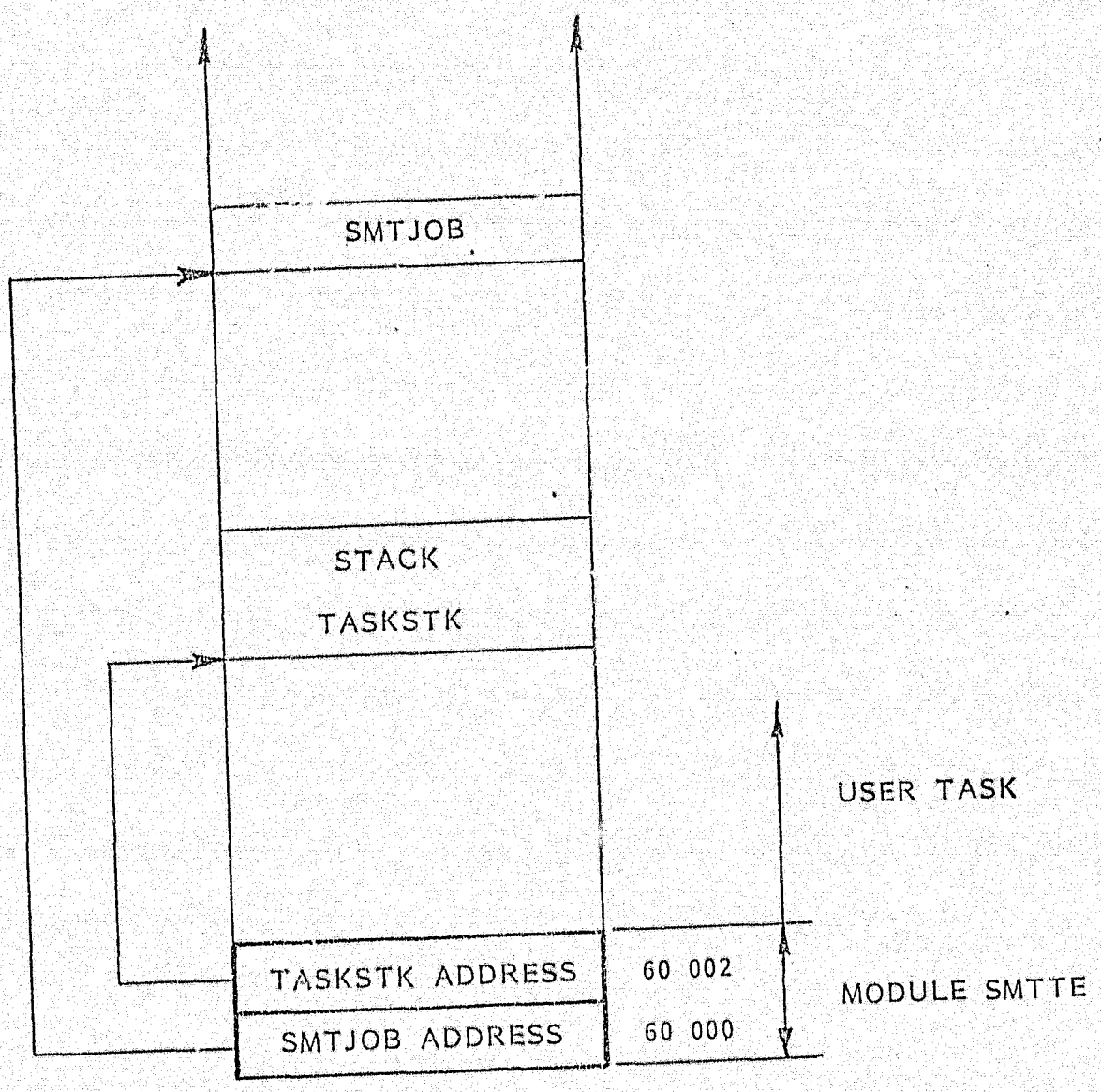


Figure B7

Figure B8


ACTION OF SMTTE
SMTJOB
STACK
TASKSTK
USER TASK
TASKSTK ADDRESS
60 002
SMTJOB ADDRESS
60 000
MODULE SMTTE

Item 3,4 and 5 are all areas which more than one task uses. 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 above mentioned areas are all task built to form one image which is installed in the bottom 3 PAR pages.

Once having built this common area, the symbols definitions table 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.

B6 Linking of User tasks to o/s

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

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

What was decided is to build each task with a dummy module (SMTTE) at the base of each task. This module containing 2 INTS, these being the stack and task entry point. These 2 variables are put in the data brick at task build time. Each task is task built individually with the symbol definition table produced by RSX-11M (See Figure B8) of the common area.

At initialization time when the o/s runs for the first time, the o/s takes the stack address (which is also the pointer to the SVC DATA area) of each task and puts it in the table called CELL in TASKDATA.

Each task is initialized by the stack initialization proc USERTASKINIT. (see Section C2.3). This proc initializes the table CELL in TASKDATA and also puts the task entry point onto the task stack.

PART C

NEW OR MODIFIED PROCEDURES BRICKS, DATA BRICKS AND MODULES

	NAME	N/M*	MODULE	REFERENCE
New SMT Modules	– DSMTU2			C1.1
	– DSMTB4			C1.2
	– TUDRV			C1.3
	– SMTCTLA			C1.4
	– SMTTE			C1.5
New or Modified SMT procs	– SETFMQ	M	DSMTB1	C2.1
	– USERTASKINIT	N	DSMTB1	C2.2
	– SETPAR	N	DSMTB1	C2.3
	– STPAR	N	DSMTB1	C2.4
	– GETSTEP	N	DSMTB1	C2.5
	– SET STATADS	N	DSMTB1	C2.6
	– LOADER	N	DSMTB4	C2.7
	– RETFIN	M	DSMTB1	C2.8
	– LOADER CONTROL	N	DSMTB4	C2.9
	– TU58	N	DSMTB4	C2.10
	– WORKTU	N	TUDRV	C2.11
	– REGIN	M	DSMTB1	C2.12
	– INTPTS	M	DSMTB1	C2.13
	– ERPRIN	M	DSMTB2	C2.14
	– START	M	DSMTB2	C2.15
	– INSTRUCT	M	SMTCTLA	C2.16
New/modified SMT data	– LDRDATA	N	DSMTU2	C3.1
	– SMTDATA	N	DSMTU2	C3.2
	– STEP	N	SMTTE	C3.3
	– TEP	N	DSMTB1	C3.4
	– TASKDATA	M	DSMTB3	C3.5
	– TU58LDR	N	DSMTB3	C3.6
	– TUDRIVER2	N	DSMTB2	C3.7

* NEW or MODIFIED

C1 New SMT Modules

C1.1 DSMTU2

This module contains two data bricks namely SMTDATA and LDRDATA. These bricks contain all the data required to bootstrap the system as well as all the use task data.

This module will always be located between octal 400 to octal 1400.

This module is loaded at bootstrap time as well as everytime, a task is loaded using the LOADER TASK.

For a complete description of the data bricks see Section C3.1 and C3.2.

C1.2 DSMTB4

This module contains the LOADER task and all the procs required to control the incoming data from the TU58.

C1.3 TUDRV

This module contains a TU58 loader program. See WORKTU in Section C3.

C1.4 SMTCTLA

The control A task has been removed as a system task from SMTB2 and now appears as a user task.

There are limitations to doing this, that is, the user can now look at data locations in the common area, i.e. o/s, database, common data bricks and common procs. Data bricks in the user task area is not accessible, with this version of memory managed SMT.

C1.5 SMTTE

This module is used when a user task is built into a task image. This module contains a data brick STEP which installs the address of the current task stack and task entry point at the bottom 2 locations of the task image. See Section 3.3.

C1 New SMT Modules

C1.1 DSMTU2

This module contains two data bricks namely SMTDATA and LDRDATA. These bricks contain all the data required to bootstrap the system as well as all the use task data.

This module will always be located between octal 400 to octal 1400.

This module is loaded at bootstrap time as well as everytime, a task is loaded using the LOADER TASK.

For a complete description of the data bricks see Section C3.1 and C3.2.

C1.2 DSMTB4

This module contains the LOADER task and all the procs required to control the incoming data from the TU58.

C1.3 TUDRV

This module contains a TU58 loader program. See WORKTU in Section C3.

C1.4 SMTCTLA

The control A task has been removed as a system task from SMTB2 and now appears as a user task.

There are limitations to doing this, that is, the user can now look at data locations in the common area, i.e. o/s, database, common data bricks and common procs. Data bricks in the user task area is not accessible, with this version of memory managed SMT.

C1.5 SMTTE

This module is used when a user task is built into a task image. This module contains a data brick STEP which installs the address of the current task stack and task entry point at the bottom 2 locations of the task image. See Section 3.3.

C2 NEW OR MODIFIED SMT PROCEDURE BRICKS

C2.1 SETFMQ MODIFIED DSMTB1

SETFMQ being the STARTUP task is the first task to run when the system is started or restarted. The function of SETFMQ has not been altered, but the proc has been greatly modified to facilitate the initialization of the user tasks.

SETFMQ has been divided into 3 sections namely:

1. the static initialization of the arrays, queue etc.
2. the dynamic initialization of the system tasks.
3. the dynamic initialization of the user task.

(See Drawing C1)

Once all the system tasks have been initialized i.e. (I=NSTSK) the STATADS array is set up and contains only the system tasks in it. The reason for doing this is that once the loader starts loading the user tasks, the manner in which it operates, requires that other tasks are running, since it is an interrupt driven loader.

While the LOADER is loading the user tasks the system tasks are running, especially the fall back task.

For this same reason the STARTUP task now has a priority higher than the FALLBACK task. If this was not the case, the processor would never return to SETMQ since the FALLBACK task would always run.

When SETFMQ is completed the task is STOPPED and will never be started again. This implies that the STAT word of SETFMQ will never be 0 and then it will never run. i.e. the FALLBACK task will run in the normal manner.

The only time SETFMQ will run after startup is when it becomes the SET task. There is no problem here in that when a call of RETEV is made, RETEV calls the SET task via RETFIN and not via CHANGE. This implies that RETEV forces the processor to run the SET task and does not use the CHANGE option to check the tasks STAT.

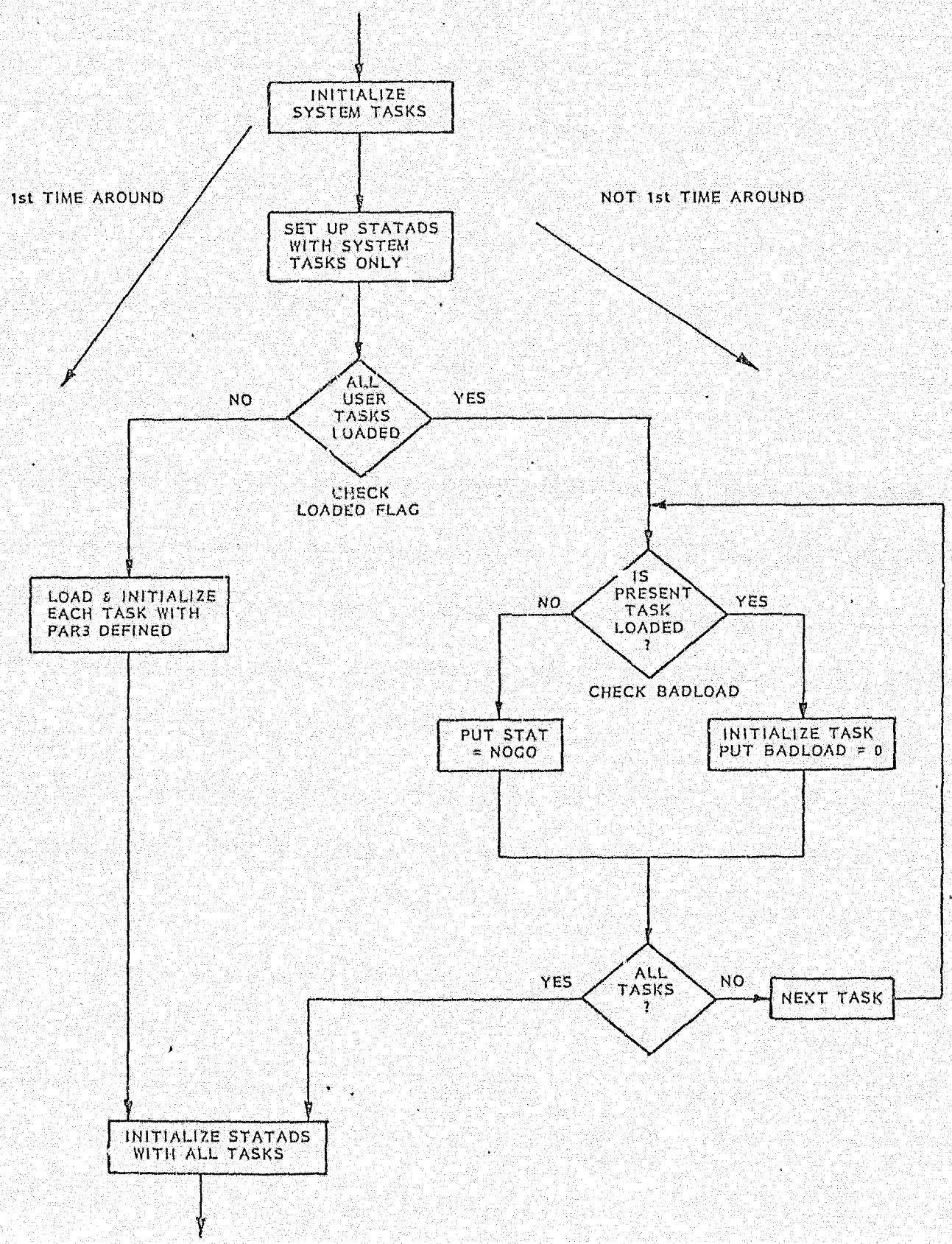
The processor first checks whether all the tasks have been loaded previously by checking the flag LOADEDFLAG. The flag is statically loaded = 0 and is made = 1 once all the tasks have been loaded the first time.

If LOADEDFLAG is = 0 it then checks the tasks first PAR value (PAR3) whether the task exists. If the task exists PAR3 is non zero. If it does not exist it is = 0. If the PAR3 value does exist a call of LOADER is made which loads and initializes the task. In this manner all the tasks are loaded one at a time. When all the tasks are loaded LOADEDFLAG is made = 1.

If LOADEDFLAG = 1 it implies task has been loaded. It then does a further check to see whether the task was loaded sucessfully the first time. Each task has a byte called BADLOAD. This is non zero if the task was not loaded successfully. Therefore if a task was not loaded sucessfully the stack is not initialized and the STAT word is made non zero (i.e. STOPPED).

Figure C1

MODIFICATIONS TO SETFMQ TO LOAD USER TASKS



The other possiblity is if the task had already been loaded sucessfully. If this is the case, the task stack is initialized by a call of USERTASKINIT. (see section C2.2).

Once all the user tasks have been initialized and loaded (if run for the first time) the array STATAD is set up again by a call of SETSTATAD proc (see section C2.6). But now the array is set up with both the user and system tasks.

The STARTUP task ends off with a call of STOP which STOPS itself and this part of the proc will not run again.

C2.2 USERTASKINIT

The procedure used to initialize a stack for a user task has been removed, from SETFMQ and put into another proc brick. The reason for seperating the procedure, is because a user task can now be loaded on line. Once loaded the task stack has to be reinitialized. This is done by a call of USERTASKINIT from the loader proc. This would not be possible if the stack initialization procedure was left within SETMQ.

USERTASKINIT controls the 3 steps necessary to initialize a user task stack as outlined in Section B6 and Figure C2.

The proc firstly installs the task PAR constants in the mmu which puts the task in scope. This is done by a call of SETPAR.

It then extracts the 2 entry points required to initialize the stack from the bottom of the task image by a call of proc GETSTEP.

With the entry points in data Brick TEP a call of the original STKINIT initializes the user task stack.

New SMT Procs

C2.3 SETPAR NEW DSMTB1

When the system is being initialized for the first time, the processor executes the proc SETFMQ which is the STARTUP task.

SETFMQ initializes all the stacks i.e. sets up the SVC data area pointer, R0 as well as the virtual start address of the base proc.

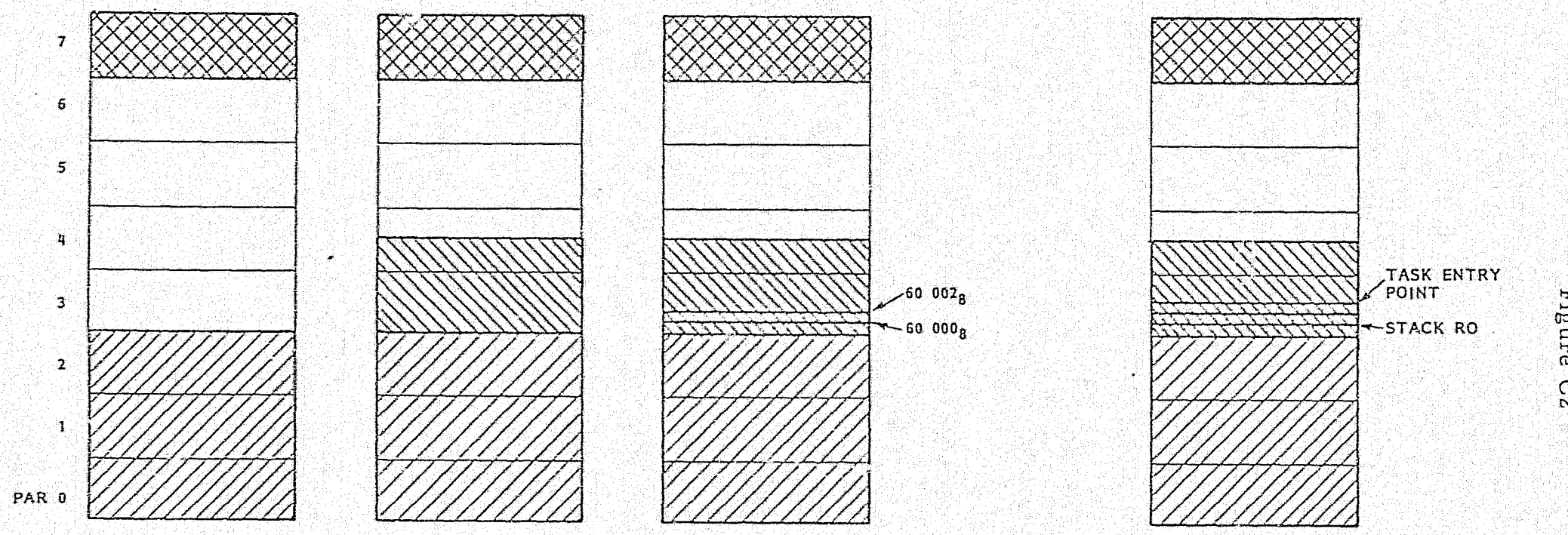
To initialize the user task, the USERTASKINIT proc first calls SETPAR to install the task PAR constants in the mmu. This is done by jumping to the label STPAR (see section C2.4). On returning from STPAR the user task stack and procs will be in scope and the stack can be set up.

The parameter passed to the proc is the USERTASKNO.

C2.4 STPAR

This procedure is used to set up the PAR's from the data contained in LDRDATA for a given task number. This proc is used at initialization and task changing time. Only user task PAR's are changed. This task was not written in RTL/2 since and RTL/2 stack environment does not exist in RETFIN to support stack manipulations.

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 C2

The proc users 3 words of stack for the calculations of the record length. The stack used is the stack of the precious running task. The locations used on the stack are above the saved register and so the security of the task is not endangered.

This PAL proc is called either from SETPAR during start up time or from RETFIN at task change time.

C2.5 GETSTEP

At the STARTUP phase when all the tasks are being initialized GETSTEP is used to get the stack and task entry point from the first two locations of the task image. This proc moves the entry points from virtual locations 60000 and 60002 to the data brick TEP, the data being manipulated by STKINIT().

C2.6 SETSTATADS

SETSTATADS is the procedure which sets up the STATADS array in priority order.

The STATADS array is an array used by CHANGE to determine which task to run.

The array is an array of pointers which point to the task STAT word. The array is arranged in task priority order with the highest priority at the top and going down with decreasing task priority. This array is set up by the STARTUP task (SETFMQ).

CHANGE starts at the first address in STATADS and checks if the STAT word is = 0 i.e. ready to run. It continues down the table until it finds the highest priority task with its status OK.

The STATADS array is setup twice once after the system tasks have been initialized and once after the user tasks have been loaded. See SETFMQ for the reason why this is done.

The parameter passed to SETSTATADS is the number of tasks to be installed in the STATADS array.

C2.7 LOADER
(see Figure C3 for philosophy)

This proc is used to load tasks into memory either at startup or while the system is on line. The proc can either be called from the STARTUP task (SETFMQ), or the LOADER task TU58. Two parameters are passed to LOADER, one being the task number to load and the second, a flag which indicates who called LOADER. The flag is set to:

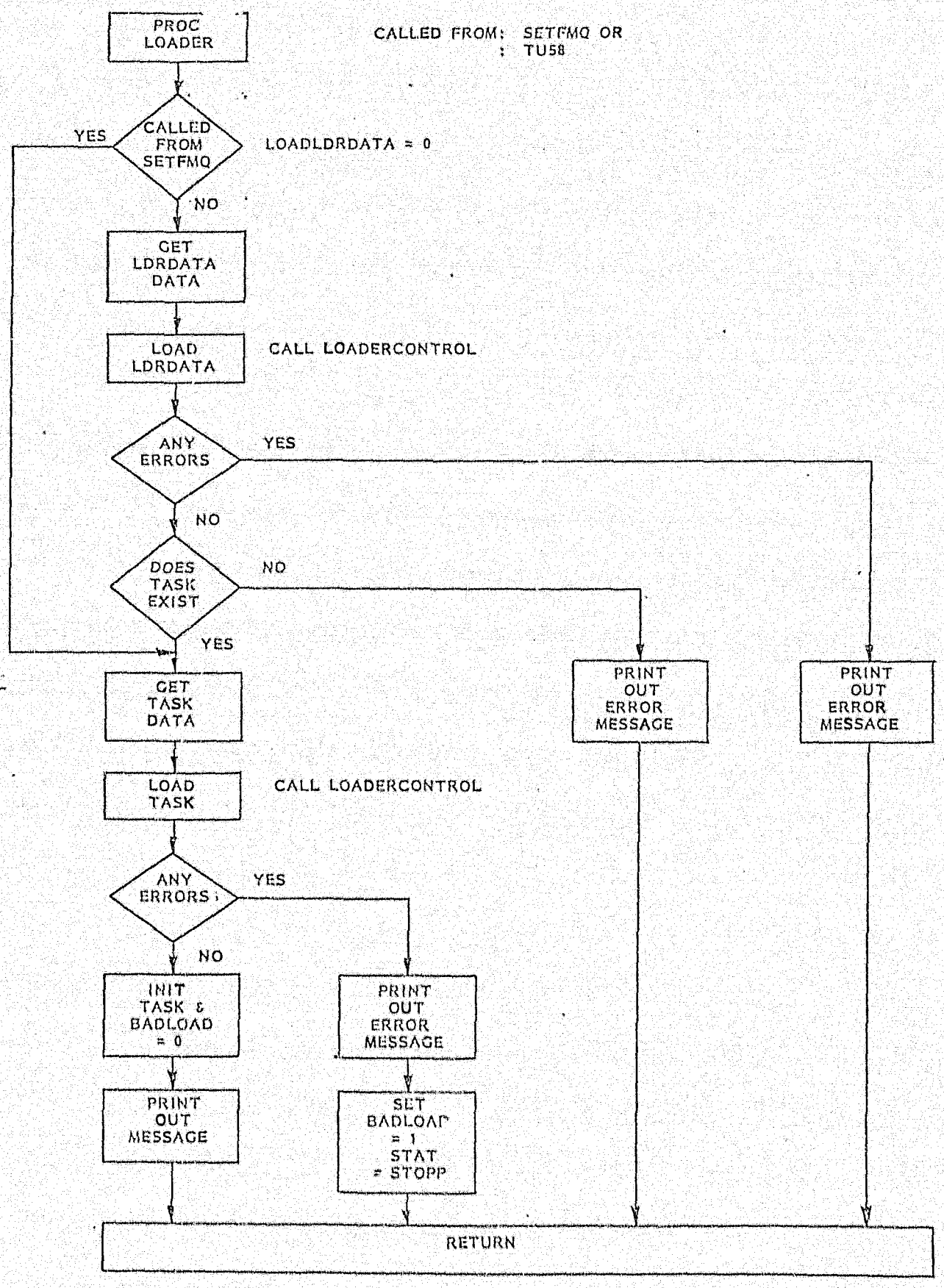
1 if called from TU58
0 if called from SETFMQ

The flag is used for the following purpose: If LOADER is called from SETFMQ the data brick LDRDATA is not loaded from block 1 of the tape and only the task is loaded. Thus when the system is loaded for the first time all the user tasks are loaded one at a time.

If LOADER is called from TU58 it means that an on line load has been requested. In this case the data brick LDRDATA is loaded first and

Figure C3

LOADER PROC PHILOSOPHY



the data contained in it is used to load the requested task. This enables the task STAT word and the task length to be changed in LDRDATA off line and this new value to be used by the on line loader program.

The data required to load the task or LDRDATA is set up in this proc and then a call of the LOADERCONTROL proc is made which actually control the transfering of the task from the tape to the memory locations.

This program also communicates the success or failure of the load to the operator.

If a task was not loaded sucessfully the BADLOAD byte associated with the task is made = 1 and the STAT word of the task is made = STOPPED.

This implies that the task will not be able to run and cannot be STARTED (see mod to START section C2.15).

If a task was successfully loaded the task stack is initialized by a call of USERTASKINIT and BADLOAD (Task Number) is made = 0 and so the task can be STOPPED and STARTED as at will.

C2.8 RETFIN

RETFIN is the proc which sets up the RTL/2 environment to run a selected task. RETFIN restores the machine registers to the state they were in, prior to an interrupt or a task change.

The proc does not select which task to run. The proc is entered with the task number to run in CURTEX. This proc may either be called from CHANGE or any other task. What the proc does is the following:

It takes CURTEX and decides firstly whether it is a system or a user task. If it is a user task, the proc makes a jump to STPAR and loads the PAR values into the respective registers. The task will now be in scope as well as its own stack.

The following will now be carried out whether it is a user or a system task:

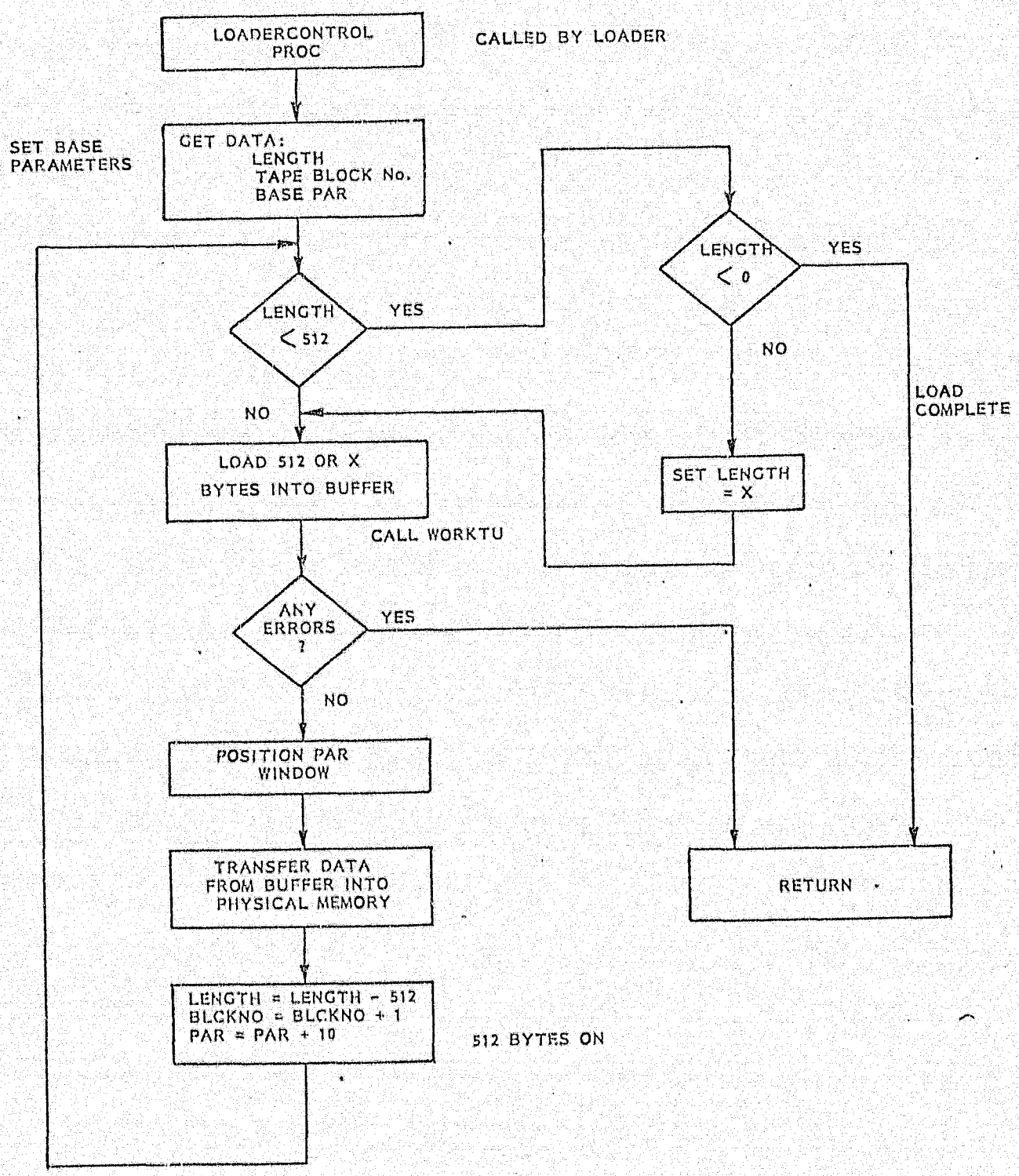
The base address of the task stack is then removed from CELL and put into R0. The stack pointer is then removed from SX6 and put into R6 and the remaining stored task registers are loaded into the machine registers. The task starts off with an RTI.

C2.9 LOADERCONTROL
(see Figure C4 for philosophy)

This proc does the control of the absolute loader in WORKTU after having been called by LOADER. This proc loads the specified task in 512 byte lengths i.e. 1 tape block at a time. The block is loaded into a buffer under interrupt control in proc WORKTU. After having been loaded, this proc then sets up the PAR window - PAR6 choosen to point to the address where the program is to be relocated to. Once the relocation is completed, the pointers are set up to point to the start of the buffer again and the bottom of PAR6. The relocation constant is then incremented by 10 that is the bottom of PAR6 now

Figure C4

LOADERCONTROL PROC PHILOSOPHY



points 256 words up in memory.

The load is complete after no more 512 byte chunks exist.

The LOADERCONTROL proc assumes that the PAR areas, i.e. pages, are 4k words long and are consecutive in memory.

C2.10 TU58
(see flowchart i.e. Figure C5)

This proc is now a system task. The objective of having this task is that individual tasks can be loaded on li e.

If from the CTLA task the operator enters an LT the TU58 task is started. The TU58 task then prompts the operator for a task number. The task then STOPS the task required to be loaded so that the task will not run while it is being replaced and also makes the tasks BADLOAD byte non zero so the task cannot be STARTED.

A call is then made of the proc LOADER which loads the required task. A request to load a task from TU58 also loads the data brick LDRDATA.

Finally the TU58 task is put back to sleep.

C2.11 WORKTU

This proc is the absolute loads for the TU58 used to load the user tasks at startup or on line.

The proc initializes the TU58, receives data from the TU58 and controls the protocol required by the TU58 tape drive.

WORKTU after having been called by LOADERCONTROL initializes the tape driver and receives all the data from the tape driver under interrupt control. This implies that an RTL/2 environment has to exist to operate this driver and also that the system has to be running.

WORKTU expects that the mode IOPKT contains all the details of the task to be loaded.

C2.12 BEGIN

The proc BEGIN is used to initialize the machine before the STARTUP task run has been modified so that the mmu is also initialized and enabled. The PAR constants of the common area's are put into the mmu PAR registers and the PDR registers are all initialized to be R/W upwards expanding.

This has to be done since the common area PAR's are not swopped at task changing time and remain constant, unless the machine is reconfigured.

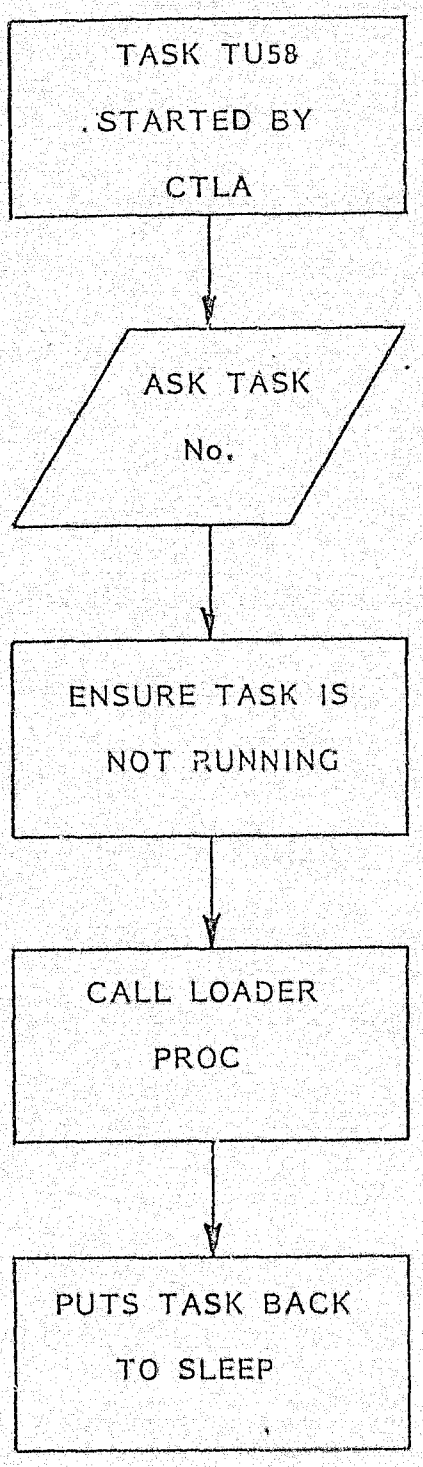
C2.13 INTPTS

INTPTS is a dummy proc which contains all the interrupt servicing routines.

The mmu has a trap associated with it, in the event of a mmu fault

Figure C5

LOADING FROM CTLA TASK-USER OPTION



CHECKS WHETHER
IN BOUNDS

STOPS TASK
PUTS BADLOAD = 1
ie.CANNOT BE STARTED

LOADS TASK
(SEE LOADER)

or violation e.g. a write was attempted into a read only section etc.

This trap is in location octal 250.

The servicing routine written for this trap is the following: The error number allocated to a 250 trap is 36, i.e. when a call of RRGEL is made the error number will be 36.

The routine also saves the two mmu status registers SR0 and SR2 which will give an indication of where the fault occureed since SR2 freezes the contents of itself at the occurence of a mmu fault.

The mmu is reenabled after the recording of the data and a jump to the error handling mechanism is made.

C2.14 ERPRIN

The error printing task has been modified to print out the mmu statu registers SR0 and SR2 when ever the error printing task is started by RRGEL.

C2.15 START

The START proc has been modified so that when a request is made to start a user task the proc first checks whether the BADLOAD byte of the task is zero. A zero means that it was loaded sucessfully. This check has been put in so that a task which has been loaded with an error cannot be started from another task if the LOADER proc STOP's the task.

If an attempt is made to start a user task with a BADLOAD byte not = 0 an error 37 is made by RRGEL.

C2.16 INSTRUCT

The operator can communicate directly with the system and perform certain functions. These functions include setting up of the date and time, the starting and stopping of tasking and also the looking at of memory locations.

The modifications carried out to this system task is that it now appears as a user task and not a system task i.e. it does not reside in the common area.

The modification to the GTLA task is that it now starts the loader task up by entering LT after the normal GTLA prompt.

One major limitation is putting the GTLA as a user task is that the user can only look at memory locations located in the common area i.e. o/s, database, common procs and I/O channels.

C3 NEW OR MODIFIED SMT DATA BRICKS

C3.1 DATA LDRDATA NEW DSMTU2

This data brick contains all the data required to fully specify a user task, for loading into the machine as well as all the user task data. This data previously appeared in SMTU1.

The items are:

ARRAY (NUSERTASKS) INT BLOCKNO	Each element defines the location on the TU58 tape of the corresponding task.
ARRAY (NUSERTASKS) INT TAPE LENGTH	Each element defines the length of the corresponding task in bytes on the TU58 tape.
ARRAY (NUSERTASKS) BYTE USERPRIO	Each element defines its priority of the corresponding task at initialization time.
ARRAY (NUSERTASKS) BYTE USERSTAT	Each element defines the status of the corresponding task at initialization time.
ARRAY (NUSERTASKS) APSET CURAPSET	This is an array of records defining the PAR constants for the corresponding tasks. The mode description of APSET is the following:

MODE APSET (INT PAR3,PAR4,PAR5,PAR6)

Each element of this mode contains the constants that have to be placed in the mmu to relocate the tasks in physical memory.

The first 2 arrays are used to load the task off the TU58 tape and the remaining arrays are used for task initialization and task changing.

C3.2 DATA SMTDATA NEW DSMTU2

This data brick is used to define the length of the common area in bytes and also where the common area is located on tape (i.e. block number).

These 2 pieces of information are used by the bootstrap program. The reason for putting this data here is that if the common area is changed, SMTDATA can be changed and not the bootstrap program.

The items are:

INT SMTPLOCKNO	Where common area is on tape.
SMTLENGTH	How many bytes on tape.

C3.3 DATA STEP NEW SMTTE

SMTTE is used to determine the location of the task entry point SMTJOB and the stack entry point TASKSTK. When built with the task and the .STB file of the common area the address of the two mentioned entry points is put into the two locations at the bottom of the respective task. The operating system looks at these two locations at start-up to determine what the virtual address of the