RSX-11M Working Design Document

Document Number: 130-951-009-03

Project Number: P2007580

Date: 17-Jun-74

Authors: M. Pellegrini, D. Cutler

COPYRIGHT 1973, DIGITAL EQUIPMENT CORP., MAYNARD, MASS.

This software is furnished to purchaser under a license for use on a single computer system and can be copied (with inclusion of DEC's copyright notice) only for use in such system, except as may otherwise be provided in writing by DEC.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation.

DEC assumes no responsibility for the use or reliability of its software on equipment which is not supplied by DEC.

1.0 INTRODUCTION AND OVERVIEW

3.0 INTERNAL STRUCTURE, PHILOSOPHY, AND FLOW OF CONTROL.

3.1 Introduction

In this Section we will present the internal operations of the RSX-11M Executive and the routines associated with it (drivers, and common service subroutines). Our exposition will have a tutorial slant. Section 5.0 contains a detailed description of the RSX-11M data structures. In this section we will introduce data structures on an as-needed basis, and discuss the detailed contents of these structures only where needed to render our exposition meaningful.

3.2 Interrupt Mechanisms

RSX-11M is a priority driven, multiprogramming, real-time operating system, and, as with any such system, its principal function is the multiplexing of sharable resources among competing tasks. The multiplexing itself is made possible by the interrupt system of the hardware which causes control to be taken away from user tasks and given to the Executive. It is during this period of interrupt control that the Executive makes its decisions on granting use of shared resources. Understanding the interrupt mechanism is fundamental to understanding the Executive, and once understood, serves as a framework for describing the operation of Executive subsystems (drivers, loader, MCR, etc) and the system as a whole.

3.2.1 Hardware Interrupt Mechanisms - Review & Overview

The PDP-11 family of computers has two classes of interrupts:

1. Processor Traps, and

2. External Interrupts.

Processor traps are not maskable*, that is, when they occur the processor enters the trap sequence of pushing PS and PC onto the current stack, retrieving PS and PC from the proper hardware trap vector, and, if no other interrupts are pending, initiating the processor at the location specified by the trap vector. A table of trap vectors start at location 4 and extend to location 774(8). Processor traps include**:

----

* Maskable means that the condition can be disabled by altering the priority of the processor.

** See the PDP-11 Processor Handbook for a complete list.

BPT Instruction;

EMT Instruction;

IOT Instruction;

TRAP Instruction;

11/40 Floating Point Exception Fault;

Odd address;

Power Fail, and

Illegal Instructions.

External interrupts are hardwired to one of the four bus request levels of the processor. These interrupts are generally associated with I/O devices and are maskable. They can only cause an interrupt when the priority in the Processor Status Word (PS) is less than the priority of the interrupting source. Thus, by setting the processor priority in a trap vector PS word to an appropriate level interrupts equal to, or below that priority are locked out.

Every device (interrupt source) has an associated trap vector in the vector table located in lower memory.

With this sketch of the hardware mechanism, we can examine the interrupt processing in the Executive itself.

3.2.2 Executive Interrupt Processing

Let us assume that all the vectors in the trap vector table have been properly initialized so that when a processor trap or interrupt occurs, an Executive routine will obtain control of the processor*.

On a real memory PDP-11**, only one stack exists, and this stack must be multiplexed to service the user tasks, the Executive, and the Executive Sub-systems.

The RSX-11M System supports n levels of user multi-programming and 250 levels of user priority. The user establishes the priority of his task and the Executive dispatches user tasks based on the highest

----

* Section xxx.xxx describes this initialization.

** See Section x.xxx for the details of mapped memory systems.

priority user task which is ready to run. The System Task Directory (STD), which is composed of one 18-word STD Entry (Task Control Block (TCB)) for each user task in the system, and which is ordered by priority, is scanned to dispatch user tasks.

Having only a single stack also implies a single processor state. The Executive must simulate a two state system. A single word, the stack depth indicator ($STKDP) is used to control this simulation.

When the stack depth indicator is equal to 1 the system is running in the user state; when it's zero or less, it is in the system state. All stack multiplexing is accomplished by testing the contents of this word. In passing, we should note that the priority set in the PS word for user tasks (both privileged and unprivileged) is 0, and for Executive routines, when running interruptable, is either 0, 7 or the level at which the interrupt was taken. It is a design goal to operate the Executive and its associated routines non-interruptable for as short a duration as possible. We currently estimate the system never remains non-interruptable for more than twenty instructions with the typical span of non-interruptable codes being less than ten instructions. Furthermore, the total non-interruptable time will be less than 1/10 of 1% of the total processor time.

External interrupts can occur within the system from either of the simulated states, that is when the stack depth indicator is 1 (user state) or <=0 (system state). Processor traps, which include the EMT instruction used for Executive Directives, only occur in the user state, with the following exceptions:

TRAP Instruction

Powerfail

Describing the RSX-11M interrupt mechanism involves several interrelated routines, and it may be necessary for the reader to make two passes over this section before the process becomes completely clear. To ease your passage, we introduce now a brief description of the basic routines involved.

The RSX-11M interrupt machinery involves the following routines or routine types:

Interrupt processor (both external interrupts and traps);

The Interrupt Save Routine ($INTSV);

The Directive Save Routine ($DIRSV);

The Interrupt Exit Routine ($INTXT);

The Directive Exit Routine ($DIRXT), and

The Fork Processors ($FORK,$FORK0,$FORK1).

Interrupt processors are entered directly and usually call $INTSV or $DIRSV for common save and state switching services; at the completion of these services, the interrupt routines are again given control to complete the interrupt service. The exit routines $INTXT and $DIRXT restore the state prior to switching to the system state, control the un-nesting of interrupts, and make checks on the state of the system (for example, is it necessary to redispatch the processor). The Fork Processors linearize access to shared system data bases. The details of all these routines will emerge in the upcoming narrative.

3.2.3 External Interrupt From The Task State (Stack Depth indicator=1)

The vectors in lower memory contain a PC unique to each interrupting source, and a PS set with a priority of PR7. Hence, when an external interrupt occurs, the hardware pushes the current PS and PC onto the current stack (in this case the users stack) and loads the new PC and PS (set at PR7) from the appropriate interrupt vector. The interrupt routine, then starts executing with interrupts locked out. Interrupt routines may, in fact, be executing in one of three states:

1. At PR7 with interrupts locked out;

2. At the priority of the interrupting source; thus, interrupt levels greater than the source are permitted, or

3. At Fork level which is at PR0.

State 2 is discussed shortly; state 3 will be deferred to Section 3.2.7.2 (Fork Processes); for now, lets look at the PR7 state.

By internal convention, processing in the PR7 state (Interrupt processing state 1) is limited to 100us. If processing can be completed in this time, then the interrupt routine simply RTI's; the interrupt has been processed and dismissed with minimal overhead.

If the interrupt routine requires additional processing time (but does not intend to alter a shared system data base) it calls the routine $INTSV (Interrupt Save). The priority at which the caller is to run immediately follows the call to $INTSV.

Interrupt save uses the specified priority to set up the interrupt routine such that it is interruptable by priorities higher than that of the interrupting source (interrupt processing state 2) and conditionally switches to system state if the processor is not already in system state. The $INTSV algorithm is:

$INTSV: Push R5 and R4 onto the current (user's) stack.

Decrement stack depth indicator

Is the stack depth indicator =0? No; go to 1

Save the current (a task's in this case) stack pointer

Set up the system stack pointer (switch stacks)

1. Load the new processor priority as specified by the caller

Return to caller.

Notes:

The Stack Depth Indicator is zero only when the transition from the user state to the system state has occurred. The user state value of 1 was selected to simplify the decrement, test, and branch which establishes whether a stack switch is necessary.

Pushing of R4 and R5 is done to free these registers for routines processing external interrupts. It is an internal programming convention that assumes these routines will not require more than two registers to accomplish their functions. If they do, they must save and restore any additional registers they use.

Example use of $INTSV*

```
;+
; RK11 DISK CONTROLLER INTERRUPT SERVICE ROUTINE
;
; THIS ROUTINE IS ENTERED VIA THE VECTOR AT LOCATION 220 WHEN AN
; INTERRUPTING CONDITION IS DETECTED IN THE RK11 CONTROLLER. THE
; ROUTINE IS ENTERED AT PRIORITY PR7 WITH ALL INTERRUPTS LOCKED OUT.
;-

$DKINT::MOV     PS,TEMP         ;;;SAVE VECTOR PS WORD
        CALL    $INTSV,PR5      ;;;SWITCH STATES AND PRIORITY TO PR5

;
; CONTROL IS REGAINED AT THIS POINT IN SYSTEM STATE WITH A PRIORITY OF
; PR5. REGISTERS R4 AND R5 HAVE BEEN SAVED AND MAY BE FREELY USED.
;

        MOV     TEMP,R4         ;;;RETRIEVE SAVED PS WORD
        BIC     #177760,R4      ;;;CLEAR ALL BUT CONTROLLER NUMBER
          .
          .
          .
        etc.
```

Implementation note:

The CALL macro in the above example is a special form which is defined in the executive macro file RSXMC.MAC. This file must be concatenated with all assemblies using this form of CALL. The code generated from the macro expansion is:

```
        JSR     R5,$INTSV
        .WORD   ^C<Priority>&PR7
```

3.2.4 External Interrupts From The System State (Stack Depth Indicator <=0)

The code on this interrupt path is identical to that discussed in Section 3.2.3. However, it is not necessary to switch states when $INTSV is called. The current stack is the system stack, and the test on the value of the stack depth indicator will cause the saving of SP

----

* We intend to make extensive use of examples throughout this manual and will repeat coding sequences where necessary to relieve the reader from continually paging to find another related example.

and the switching of stacks to be bypassed. After saving R4 and R5 on the system stack, a return to the interrupt routine is executed.

3.2.5 Processor Traps From The Task State (Stack Depth Indicator=1)

When a processor trap occurs from the task state, the hardware pushes PS, PC, and initiates the routine specified in the associated hardware trap vector. If the trap was an Executive directive (EMT 377), the DPB (Directive Parameter Block) or its address was pushed onto the user's stack prior to the issuance of the EMT. The trap routine, running at PR7 immediately calls the routine $DIRSV (Directive Save) which has the following algorithm:

```
$DIRSV:  Push R5 and R4 onto current (user's) stack

         Decrement Stack depth indicator

         Is the stack depth indicator =0? No, go to 1.

         Save current (user's) stack pointer

         Set up system stack pointer (switch stacks)

     1.  Push R3,R2,R1,R0 onto current (system) stack

         Load new processor priority as specified by the caller

         Return to caller.
```

Notes:

The depth indicator check is made to improve crash analysis; no other decisions are made in $DIRSV since all processor traps, with two exceptions, occur from the task state. The exceptions are handled on exit. All registers are saved; the need for only two registers, R5 and R4 is assumed only for routines processing external interrupts. As with $INTSV the priority at which the caller expects to run immediately follows the call. All processor trap routines, however, run at level 0.

Only one processor trap can be queued for processing in the system at any point in time (ignore, for the moment, the two exceptions we have noted). Since the processor trap occurred in the in task state, entrance to the Executive occurs only when the Executive is idle. While in the System State, only external interrupts can occur. If processor traps occur, then either they are valid exceptions, or the system itself has faulted and will shut down.

Once a valid processor trap is pending, it will be processed to completion before any other system routine is given access to any shared system data base. We will see how this strict sequentiality is effected when we discuss the two exit routines and the fork processors.

Example use of $DIRSV

```
;+
; EMT TRAP PROCESSING ROUTINE
;
; THIS ROUTINE IS ENTERED VIA THE VECTOR AT LOCATION 30 WHEN AN EMT
; INSTRUCTION IS EXECUTED. THE ROUTINE IS ENTERED AT PRIORITY PR7
; WITH ALL INTERRUPTS LOCKED OUT.
;-

$EMTRP::CALL    $DIRSV,PR0          ;;;SWITCH STATES AND PRIORITY TO PR0
        TST     $STKDP              ;EMT EXECUTED FROM SYSTEM STATE?
        BEQ     10$                 ;IF EQ NO
        CRASH                       ;CRASH SYSTEM
10$:    .
        .
        .
        .
        etc.
```

Implementation note:

The CALL macro in the above example is a special form which is defined in the executive macro file RSXMC.MAC. This file must be concatenated with all assemblies using this form of CALL. The code generated from the macro expansion is:

```
        JSR     R5,$DIRSV
        .WORD   ^C<Priority>&PR7
```

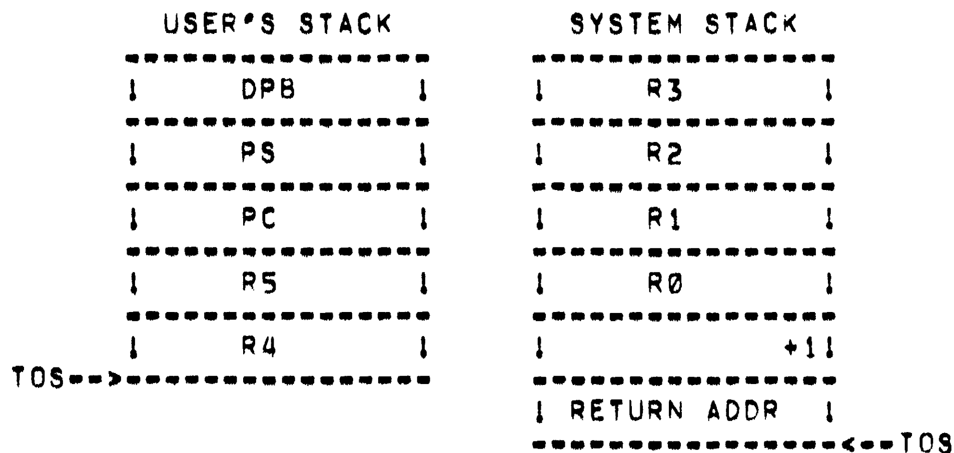
3.2.6 Processor Traps From The System State (Stack Level <=0)

Only two processor traps are valid from the system state: The trap instruction and powerfail. If any other processor trap occurs while in the system state, the system's operation is aborted.

3.2.6.1 Processing For Trap Instructions Which Occur In The System State

The trap instruction is used within the Executive as a core saving technique in returning status following the processing of an Executive Directive. The EMT 377, which is the processor trap used to initiate directives, causes entry into the Directive Dispatcher ($EMTRP) which in turn calls $DIRSV. On return from $DIRSV, but before calling the directive processing routine (and entry to the proper routine is via a CALL), the Directive Dispatcher pushes a value of +1 onto the system stack, and clears the C bit in the PS word stored on the users stack;

then it calls the proper directive processing routine to effect the directive. Figure 3.1 shows the state of the user and system stacks for both the unmapped and mapped systems at the time entry is made to the directive processing routine.



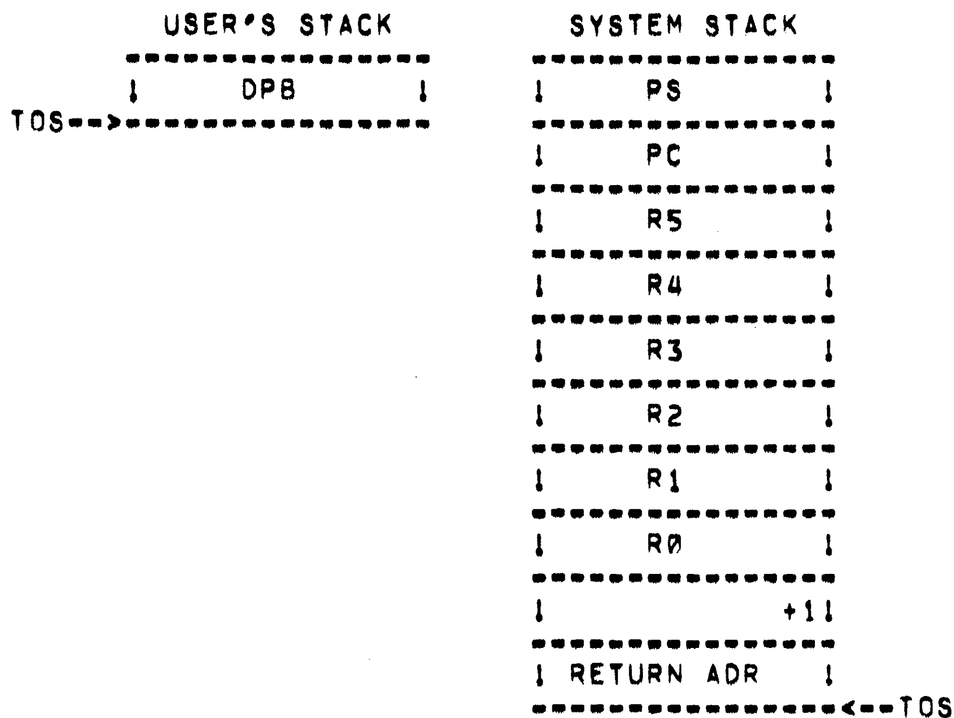


FIGURE 3-1

The Directive processing routine now carries out its function, and in so doing is free to alter any shared system data base, since no other routine will gain access to a shared data base until the directive processing routine is completed. This arrangement of the stack and interface between the Directive Dispatcher and the Directive Processors has two advantages:

1. The normal return for all but a few directives is a +1 status and carry clear. This means the directive routines can return to the dispatcher with an RTS; thus the return path is one word rather than two if a JMP were employed; this scheme probably saves 100 words in the RSX-11M Executive.

2. Internal Executive routines can call the directive processing routines without using an EMT.

If a directive processing routine needs to return a status other than +1, and have carry clear, the routine simply replaces the +1 on the stack with the value it intends to return and then executes an RTS.

Now we come to the use of the trap instruction within the Executive. If a directive processing routine needs to return a status other than +1 and, in addition have carry set, or cleared, based on the status value returned, then it uses the trap instruction with the value of status to be returned in the low order byte of the instruction. When the trap processing routine is entered, it immediately checks for stack depth=0, and if 0, proceeds to reset the stack for correct exiting from a directive processing routine. The low order byte of the trap instruction itself overlays the +1 status currently on the stack; this value is tested and, if minus, carry is set in the user PS word, if plus, carry is left cleared. Then the exiting code of the Directive Dispatcher is entered just as if the directive processing routine had executed an RTS.

If the initial test for a stack depth indicator of 0 fails, then the trap processing routine calls $DIRSV. This call is logically incorrect if the stack depth indicator was less than zero. This programming error is recognized on exit. On return from $DIRSV, the trap processing routine checks the stack depth indicator, and if it is not zero, the system is shut down.

Note that directives are legitimate only from the task state (stack depth indicator=1) so that during directive processing, the stack depth indicator is always 0. Interrupts that occur in system state disappear from the stack before the directive processing sequence resumes following an interrupt. Hence, even though the stack can grow while a directive processor is in control, this growth is transparent to the directive processor. Stating it from a different perspective, interrupts are permitted but the directive processor in control is unaware of them.

Directive processing routines thus have three methods of returning status:

1. For the normal return carry clear and status equal to +1, use an RTS.

2. For carry clear and status other than +1, overlay the +1 status on the stack with the desired status value (status value is at 2(SP)), and RTS.

3. For carry clear or set, and status of one byte, use the trap instruction. This requires more overhead than 1 and 2 above but saves core, and, of course is the required return mechanism if carry is to be set.

Together, these return mechanisms from directive processing routines save between 200 and 300 words in the RSX-11M Executive as compared to returning via jump instructions.

3.2.6.2 Powerfail Processing

When a power failure occurs, the power failure trap processing routine is entered. This routine saves the state of the system, sets up a new power failure trap-vector PC for use when power is restored, then HALTs.

On restoration of power, the state of the system at the time the failure occurred is restored, a flag is set by software indicating that a power failure has occurred, the reschedule pointer* is set to the null task, and the clock is re-enabled. Then, the restoration code issues an RTI, which results in the resumption of the processing that was in progress when the power failure occurred. The specific processing to reflect the occurrence of a power failure will not occur until either Directive Exit is entered or the clock interrupts. In any event, this processing is part of Directive Exit and will be discussed under Directive Exit.

Note that power failure processing is not asynchronous. As much as 1/60 of a second could elapse following restoration before the power failure is acted upon.

The records and logic needed to provide asynchronous processing are simply too large for a system with RSX-11M's core objectives. Our assumption is that asynchronous power-failure processing is not consistent with RSX-11M's objectives or markets.

3.2.6.3 Aborting The System

When a condition is detected that indicates the system should be shut down (crashed in an orderly manner), the detecting routine issues an IOT. The IOT processing routine, as usual, calls Directive Save, and on return checks for a stack depth equal to zero. If equal, normal SST processing is entered. If not equal to zero, the routine $CRASH is entered. Crash displays, on the device whose CSR address is C$$RSH, an appropriate printout which is detailed in Appendix I. After completing the printout, it jumps to a routine called $PANIC, which optionally prints out an edit of selected memory locations. In the minimum system both $CRASH and $PANIC are conditionalized out, and a system crash simply results in a processor halt at location 560(8).

3.2.7 Processing Within Interrupt Routines

In Section 3.2.2 we examined interrupt entry into the RSX-11M Executive. In this section we detail the events which take place following interrupt entry up to the point where the Executive is ready to return control to the task state.

Once the Executive is entered via an interrupt (regardless of the state its in when the interrupt occurs) it will not again return to

----

* The reschedule pointer will be covered when we discuss task switching.

the task state until all system related processing for that interrupt has been completed. From another point of view, the task state is never in control unless the Executive has nothing-to-do (is idle).

While in the task state a single interrupt causes transfer into the system state where the system remains until the interrupt is processed. But while in the system state, repeated interrupts can occur. This implies a fixed interrupt depth of one for the task stack (requiring a task to provide a stack of at least four words in an unmapped system), and implies a variable interrupt depth for the system stack.

If interrupts can occur in the system state, then a mechanism is required to prevent unwanted recursion and improper data base modification. In RSX-11M both of these logical difficulties are resolved by strictly linearizing interrupt processing and access to internal data bases. The mechanisms employed to accomplish this linearization are the system stack, fork processes, and the associated fork list.

3.2.7.1 Queuing Interrupts On The System Stack

In the system state the goal is to operate interruptible as much of the time as possible. Three conditions exist when the system itself runs non-interruptable:

1. The most recent interrupt is being processed at level PR7 and the interrupt routine has not yet returned to an interruptible state.

2. The interrupt routine has dropped from level PR7 to the level at which the interrupt occurred. Priority levels, equal to or less than the priority of the interrupting source are locked out.

3. The system is updating a critical list whose consistency can only be maintained by a non-interruptable instruction sequence. There are two such lists, ahd we will discuss them shortly.

In Sections 3.2.3 and 3.2.4 we examined the code sequence for processing external interrupts and processor traps, as well as the stack additions that occurred during their processing. Interrupt stacking in the system state will occur based principally on hardware interrupt levels. Thus if a level PR4 interrupt is being processed, then a level PR5, PR6, or PR7 interrupt can potentially interrupt this processing and cause context to be stacked and control given to the higher level interrupt routine.

3.2.7.2 Fork Processing And The Fork List

Once an interrupt routine passes from a non-interruptible to an interruptible state via a call to $INTSV, processing is at the same level as the priority of the interrupting source. Along any given interrupt path however, more processing is often required than the goal of minimun non-interuptible code sequences in the Executive permits; along this path the allowable maximum non-interruptible processing time is 500us. Thus, a scheme is required to split interrupt processing routines further, such that part of their execution runs interruptible to any interrupting source. The mechanism for achieving this split is called fork processing. Fork is an internal subroutine with three entry points; fork linearizes accesses to system data bases, thus eliminating unwanted recursion and mulitple-updates of these data bases. An associated list, the Fork List, which is processed FIFO, is used as the linearizing structure.

Interrupt routines are required to adhere to the following internal conventions:

1. Use of any registers except R4 and R5 requires that these registers be saved and restored.

2. Non-interruptible processing must not exceed twenty instructions.

3. All modifications to system data bases must be done via a fork process. The first two requirements are straightforward, so lets turn our discussion to Fork.

Along an interrupt path, control can be taken from a routine only due to a higher priority interrupt pending in the hardware. As discussed previously, these interrupts are kept track of on the system stack. When an interrupt routine needs to transfer from a noninterruptible (including partial non-interruptibility) to an interruptible state, or modify a system data base, it must call Fork. Fork, however, cannot be called directly from an interrupt routine; it must first switch to system state by calling Interrupt Save and then call fork.

Fork has three entry points depending on the size of the fork block being provided by the caller for context storage.

$FORK requires a 4-word fork block which contains:

A forward link in the fork list

PC of the caller

Saved R4

Saved R5

$FORK is for I/O drivers and the fork processor assumes R5 is loaded with the unit control block (UCB) address.

$FORK1 requires a 3-word fork block containing:

A forward link in the fork list

PC of the caller

Saved R5

$FORK0 requires a 2-word fork block containing:

A forward link in the fork list

PC of the caller

$FORK when called:

Stores the return PC, R5 and R4* into the fork block. Appends the supplied fork block to the fork list, and jumps to Interrupt Exit.

By virture of calling fork, the routine is now interruptible and its access to system data bases is strictly linear. The interrupt processing routine is now in state 3. (Refer to Section 3.2.3 for a discussion of states 1 and 2). The Fork List is a list of system routines waiting to complete their processing, in particular, waiting to access a shared system data base.

When the Fork routine, after placing the fork block in the fork list, jumps to $INTXT the stack items for the interrupt routine are removed from the stack. In effect the fork list is a secondary interrupt queue (stack) whose members are processed FIFO, and obtain processing time only if the system stack is empty.

Note that the context saved for a caller of $FORK depends on which entry point is called, and the context saved is all that is needed to restart routines on the fork list.

----

* R5 and R4 will or will not be stored depending on which variant of Fork is called.

Example call to $FORK

```
;+
; RK11 DISK CONTROLLER INTERUPT SERVICE ROUTINE
;
; THIS ROUTINE IS ENTERED VIA THE VECTOR AT LOCATION 220 WHEN AN
; INTERRUPTING CONDITION IS DETECTED IN THE RK11 CONTROLLER. THE
; ROUTINE IS ENTERED AT PRIORITY PR7 WITH ALL INTERRUPTS LOCKED OUT.
;-

$DKINT::MOV     PS,TEMP             ;;;SAVE VECTOR PS WORD
        CALL    $INTSV,PR5          ;;;SWITCH STATES AND PRIORITY TO PR5
        MOV     TEMP,R4             ;;;RETRIEVE SAVED PS WORD
        BIC     #177760,R4          ;;;CLEAR ALL BUT CONTROLLER NUMBER
        ASL     R4                  ;;;CONVERT TO WORD INDEX
        MOV     CNTBL(R4),R5        ;;;RETRIEVE ADDRESS OF UCB
        TSTB    RTTBL+1(R4)         ;;;DRIVE RESET IN PROGRESS?
        BEQ     50$                 ;;;IF EQ NO
        .
        drive reset processing code
        .
        .
50$:    CALL    $FORK               ;;;CREATE A FORK PROCESS

;
; CONTROL IS REGAINED AT THIS POINT WITH ALL INTERRUPTS ALLOWED.
;
```

3.2.8 Exiting The System State

In Section 3.2.2 - 3.2.7.2 we covered the Executive's processing of interrupts, and the methods employed to linearize their processing so as to minimize the non-interruptible time spent in the system state. In several places we referenced two routines $INTXT (Interrupt Exit) and $DIRXT (Directive Exit). These routines result in the sequential removal of all items on the system stack, then all items on the fork list. It is these two routines to which we now turn.

The Executive's strategic objective is to return to equilibrium (the idle state) as fast and as efficiently as possible; its tactics to achieve equilibrium is to is to service all routines on the system stack first. These routines are usually running at some level of non-interruptibility. when the system stack is cleared of pending requests, the Executive then services the pending requests on the fork list. When both the fork list and system stack are empty, the Executive will either return to the task state or if no task is active, drop into the Null Task.

$INTXT is transferred to by external interrupt processing routines that are running on the system stack at the priority of the interrupting source (state 2 for those interrupt processing routines).

$DIRXT has the task of servicing the fork list and, when the Executive has no more work to do, restoring the task state. $DIRXT is entered by trap routines, fork routines, and by $INTXT.

3.2.8.1 Interrupt Exit ($INTXT)

The $INTXT algorithm is as follows:

$INTXT: Lock out interrupts

Is stack depth indicator=0? No, go to 1.

Is fork list empty. Yes, go to 1.

Allow interrupts.

Store R3,R2,R1,R0 on the current (system in this case) stack

Jump to $DIRXT (Directive Exit).

1. Increment stack depth indicator.

Restore R4 and R5 from current stack and RTI.

Notes:

Interrupts must be locked out to insure a consistent check of the stack depth indicator and the contents of the fork list. The same type of lockout occurs in Directive Exit. There are two non-interruptible code spans used to check and update the fork list mentioned in Section 3.2.7.1., one in $FORK, and one in $DIRXT. The saving of R3 thru R0 is prepatory to the jump to $DIRXT which expects these registers on the stack. Note that the path through the Executive which find both the fork list empty and the stack depth indicator equal to 0 is fairly common. As can be seen, this is the minimum overhead path.

3.2.8.2 Directive Exit

The $DIRXT algorithm is as follows:

$DIRXT: Lock out interrupts.

Is the fork list empty? Yes, go to 1.

Update Fork list pointers.

Allow interrupts.

Restore Fork context.

Call routine whose fork context was restored.

Go to $DIRXT.

1. Is rescheduling required ($RQSCH not=0)? No, go to 2.

Allow interrupts.

Clear $RQSCH.

Save context of current task.

Locate a ready-to-run task.

Restore context of new task.

Go to $DIRXT.

2. Is the power failure flag set? No, go to 3.

Allow interrupts

Call power failure processing.

Go to $DIRXT

3. Restore task stack pointer.

Increment stack depth indicator.

Restore R4 and R5 from user stack and RTI.

Notes:

$DIRXT calls both waiting fork processes and the powerfail routine. These routines terminate via a RTS instruction. On return $DIRXT again cycles looking for work.

The task reschedule pointer $RQSCH controls the redispatching of the processor. It points to the location in the STD list where $DIRXT should begin its scan for a task ready to use the processor.

$RQSCH is set when a change of state has occured in the system that might cause a task other than the one currently in control to obtain processor time. Examples are I/O done, clock queue runout, or a task doing an EXIT. The word is reset by $DIRXT just prior to its dispatching a new task.

3.2.9 Interrupt Processing Summary

Seven routines or groups of routines not only comprise the interrupt system but can be said to practically comprise the entire Executive itself.

External Interrupt Routines

Trap Routines

Interrupt Save

Directive Save

Fork and Fork Processes

Interrupt Exit

Directive Exit

External interrupts cause traps to external interrupt processing routines which run in one of three states:

1. Non-interruptiple at PR7.
   They run here when initially entered.

2. Interruptible by priorities higher than the interrupting source.

   Both states 1 and 2 are linearized being queued and dequeued from the system stack.

3. Fully interruptible as fork processes.

Trap routines ,of which only one may occupy the system stack during any given passage through the Executive, operate at priority level zero, need never call Fork, and operate entirely from the system stack.

Interrupt save is called by external interrupt routines when they make a transition from state 1 to state 2.

Directive save is called by trap routines.

Fork creates a fork process for external interrupt routines.

Interrupt Exit unstacks waiting routines executing from the system stack, and when the system stack is empty drops into Directive Exit.

Directive Exit has the job of giving control to waiting fork processes, processing power failure, and redispatching the processor.

The Executive structure has an implied sequentiality which obviates the need for any explicit synchronizing mechanisms. System routines which follow the internal conventions of the Executive never need concern themselves with multiple-update of shared system data bases. In tending toward the idle state the Executive gives priority to routines on the system stack, then to fork processes.

3.3 I/O Processing

3.3.1 Goals

The I/O interface is 100% compatible with RSX-11D. Within the requirement of compatibility, three goals guided the design:

1. The total memory for I/O processing (data structures plus drivers) must be reduced by 50% vs RSX-11D.

2. The I/O data structures should have substantial flexibility for adding future devices, or for altering the service discipline of existing devices.

3. Throughput should equal or exceed RSX-11D's.

3.3.2 I/O Philosophy and Functional Overview Of Its Implementation

To meet its stated goals, the RSX-11M I/O system attempts to centralize common functions, thus eliminating the repetitive appearance of code, which is almost identical in form and function, from appearing in each and every driver in the system. To achieve this, a substantial effort has been expended in the design of RSX-11M's I/O data structures which are used to drive the centralized routines. The effect is to substantially reduce the size of individual I/O drivers; an RSX-11M driver is typically one-fourth as large as its RSX-11D counterpart. Of course, the centralized code, and an increase in the size of RSX-11M's data structures compared to RSX-11D's reduces our effective size reduction. But, on the balance, we have substantially reduced the size of our total I/O processing core requirements while at the same time produced a more understandable, maintainable, and enhanceable I/O system (obviously, our subjective judgement).

The user interface to the RSX-11M I/O system consists of logical unit numbers (LUNs) and a single active I/O directive QUEUE I/O. (The directives ASSIGN LUN, GET LUN INFO, etc. do not initiate I/O transfers).

In RSX-11M all the preliminary processing antecedent to actually queuing an I/O request is performed by the QIO directive processing code which is driven from the I/O data structures; this code calls ancillary routines for centralized services. When a driver finally receives an I/O order, it has very little to do other than set up the status registers and issue the order.

Termination processing is equally modular and centralized. The driver is entered, performs some cleanup operations, and calls centralized routines for obtaining pending I/O orders, performing AST processing,

etc. The driver is only concerned with the most intimate and specific details of the actual hardware interface in respect to the execution and completion of I/O transfers. Using this centralization philosophy, RSX-11M keeps both driver size and non-interruptible processing time small.

3.3.3 RSX-11M I/O Data Structures

The static* I/O data structures consists of three distinct entities:

1. A Device Control Block (DCB);
2. A Unit Control Block (UCB), and
3. A Status Control Block (SCB).

Although each serves a specific function, and the components of each, in general, reflect these functions, the coherence achieved by a strict set of rules for determining into which data structure a specific unit of information would be placed was ultimately sacrificed to core savings and code efficiency. The exceptions, however, are few, and the functional purpose of each data structure is reflected by the units of information which compose them.

3.3.3.1 The Device Control Block (DCB)

One device control block exists for each device type attached to the system. Its function is to describe the static characteristics of both the controller and the units attached to the controller. All the DCB's in the system are singly linked. The DCB contains such information as:

The device mnemonic (Two ASCII characters)

The lowest and highest Unit Numbers on the respective Controller

The address of the first UCB

The Length of Each UCB

The next DCB Pointer

The Legal Function Mask

----

* Static in the sense that they are established at SYSGEN and only another SYSGEN can expand or contract the number of I/O units served by the structures.

The Control Function Mask

The No-Op'd Function Mask

The File Function Mask

The Pointer to the Driver Dispatch Table

All these information units are static and are used principally by the QUEUE I/O directive processing code to prepare a Queue I/O request for a device driver. The details of QUEUE I/O Processing are in Section 3.3.5.

3.3.3.2 The Unit Control Block (UCB).

One unit control block exists for each physical device unit attached to the system. Many of its information units are static, though it does contain a few dynamic parameters. For example the redirect pointer which reflects the result of a Redirect MCR command.

The UCB contains device unit specific data, such as unit status, physical unit number, and unit characteristics.

3.3.3.3 The Status Control Block (SCB).

One status control block exists for each device controller in the system. This is true even if the controller handles more than one device unit (like the RK Controller). Line multiplexers such as the DH11 and DJ11 are considered to have a controller per line since all lines may transfer in parallel.

Most information in the SCB is dynamic. It contains the following information about the currently active unit:

The Interrupt Vector Address

The Controller Bus Request Priority

Timeout Counts (Initial and Current)

The Address of the Control Status Register

The Address of the Current I/O Packet

Storage For A Fork Block

The I/O Queue Listhead

The Controller Status (Busy/Idle)

The Controller Index

As can be seen, the SCB is quite dynamic, making it possible to maintain control of the current I/O in progress on the controller. The presence of the fork block storage in the SCB implies I/O linearity for the processing at fork level on a given controller. We have here a specific example of how multiple updates and recursion are controlled. The driver for a specific device type never concerns itself with unwanted recursion or multiple updates. Once a driver is in a fork level process, further I/O processing, which may involve updating a shared data base, is automatically locked out by the structure of the system itself.

3.3.4 Some Sample I/O Structures

Figure 3-2 shows the data structure that would result for three terminals each interfaced via a DL11 asynchronous line interface. The structure requires three UCBs and three SCBs since the activity on all three units can proceed in parallel. In Figure 3-3 the internal data structure for an RK disk controller with three units is depicted; note that only one SCB exists because only one of the three units may be active at any time.

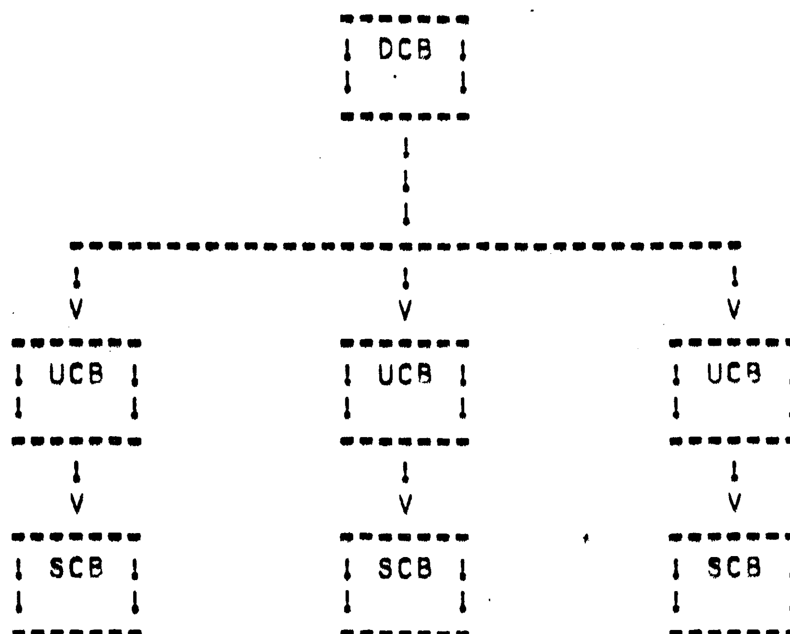


FIGURE 3-2

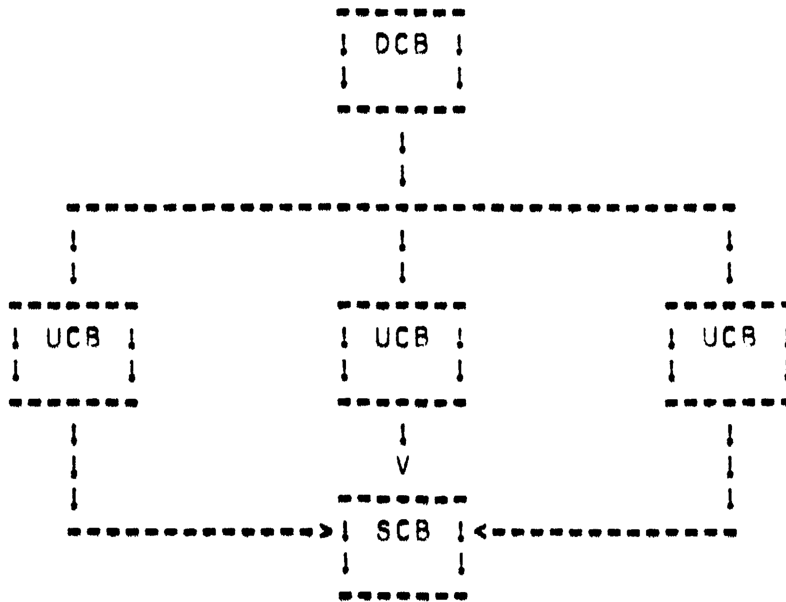


Figure 3-3

3.3.5 Queue I/O Directive Flow

The Queue I/O directive requires the issuer to construct a twelve word directive parameter block as shown in Figure 3-4.

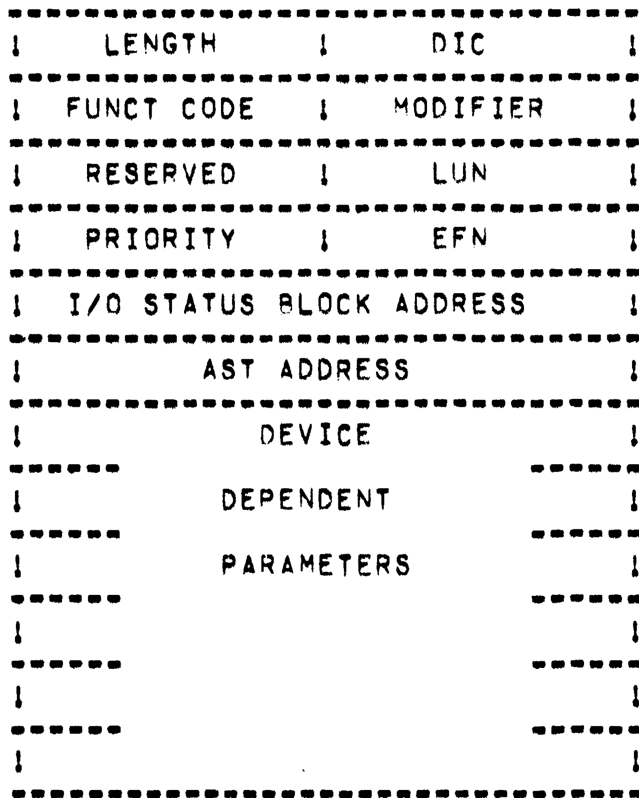


Figure 3-4

The parameters have the following interpretation.

Length (required):

The length of DPB. For QIO always equal to twelve words.

DIC (required):

Directive Identification Code. For QIO, the value is a 1.

Function Code (required):

The code of the requested I/O function (0 thru 31.).

Modifier:

Device dependent modifier bits.

Reserved:

Reserved byte and must not be used.

LUN (required):

Logical Unit Number.

Priority:

Request priority. Ignored by RSX-11M, but space must be allocated for RSX-11D compatibility.

EFN (optional):

Event flag number.

I/O Status Block Address (optional):

This word contains a pointer to the I/O status block which is a 4-byte device dependent I/O completion data packet formatted as:

Byte 0

I/O status byte

Byte 1

Augmented data supplied by the driver

Bytes 2 and 3

The contents of these bytes depend on the value of byte 0. If byte 0=1 then these bytes contain the processed byte count. If not=1, then the contents are device dependent.

AST Address (optional):

Address of an AST Service Routine.

Device Dependent Parameters:

Up to six parameters specific to the device. Typically these are:

Buffer address

Byte count

Carriage control type

Logical block number

Any optional parameters that are not specified must be filled with zeros.

When the QIO directive is issued, QIO directive processing code receives control and processes the request as follows:

1. [Perform first level validity checks]

QIO examines the LUT (Logical Unit Table) in the task header for a LUN match.

It checks if the LUN is legal for the requesting task. If the LUN is greater than the available LUT slots established for the task the directive is rejected. Given a valid LUN, a check is made to determine if a valid UCB pointer exists in the LUT for the specified LUN. If none exists the directive is rejected. If the LUN and UCB pointer are valid, the redirect algorithm is entered.

1a. [Redirect algorithm]

The UCB is located and the re-direct pointer checked. If the re-direct pointer points to the UCB which contains it, then the final link in the redirect chain has been found. Else the redirect UCB address is obtained and the test is repeated. This search continues until the last re-directed UCB is found. Then, the backpointer to the DCB is used to locate the DCB and a check on the device name is made. If the name is TI, then the address of the UCB which is to be used to control I/O transfers is in the TCB of the requesting task. This pointer was implanted by MCR when it requested the task and the chaining process described links the requesting task to the terminal which requested it. If the name is not TI, the final UCB in the redirect chain is used for the I/O Transfer. The EFN and I/O status block (IOSB) address are validated and the priority is ignored. If any of these checks fail, the directive is rejected. If the checks pass, the directive status is set to +1 and the IOSB, if specified, is cleared.

2. [Obtain storage for, and create an I/O driver queue entry]

After the first level checks prove positive, QIO obtains an 18-word storage block from dynamic storage. Into this block, which we will refer to as an I/O packet, QIO inserts the following (the source of the data is specified with each data item)

Function Code
  Copied from the DPB

Contents of the first LUN word
  Established by the redirect algorithm

Address of the second LUN word
  Calculated from the requesting task's header address

EFN
  Copied from the DPB

Priority
Copied from TCB of the requesting task

Address of the Task Control Block
The TCB address of the requesting task

Virtual address of the I/O status block
Copied from the DPB

Relocation Bias of the I/O status block*
Calculated

Address of the I/O status block*
Calculated

AST address
Copied from the DPB

Device Dependent Parameters
Copied from the DPB

3. [Validate the function request]

Using the legal-function mask in the DCB, QIO determines if the requested function is legal. If it is not legal, go to 9.

4. [Check for no-op'ed function]

Using the no-op'ed function mask in the DCB, check if the requested function is to be no-op'ed. If yes, go to 9.

5. [Check for a Control Function]

Using the control function mask in the DCB, determine if the requested function is a control function. If yes, go to 8.

6. [Check for a file system function]

The next tier of function checks determine if the function is a file system function. If it is, a check is made to see whether the device to which the request is directed is file structured. If it is we go to 8. If the device is not file structured, then the function requested must be either a read virtual or write virtual. The request is mapped to its logical counterpart (read or write logical) and processing proceeds at step 7.

----

* These exist to satisfy requirements of the mapped system. A separate section will detail the differences between a mapped and unmapped system.

7. [Transfer function processing]

If the function is legal, but not a no-op or control function, then it's a transfer function, and address checks are made on the buffer, count, and modulus requirements*. If any of these fail, go to 9.

8. [Packet Queuing]

QIO checks the control bits in the UCB which determines if it will queue the packet and then call the driver, or call the driver and let the driver queue the packet. The call to the driver is via the pointer in the DCB to the driver dispatch table entry addresses, namely:

INITIATOR

CANCEL I/O

POWER FAILURE

DEVICE TIMEOUT.

In this case the initiator entry point is called, and normal path QIO directive processing is complete.

9. [Function is illegal, no-op'ed, or invalid parameters]

If any of these cases pertain, QIO calls I/O Done and passes a status code. A special entry point is used which causes I/O Done to bypass clearing of the unit and controller busy flags, clearing of which occurs along I/O Done's normal path.

Notes:

By reviewing the algorithm for the QIO directive processing, the reader should note the care taken to relieve the driver of validation processing. It's through the centralization of validation processing that driver code is substantially shortened and structurally simplified. A beneficial fallout of this strategy is that drivers are not called with requests that are going to either fail on a pre-issue validation check, or not result in the issuance of an actual I/O (like no-op and control functions). This substantially reduces internal overhead.

3.3.6 Issuing I/O

----

* Modulus checks exist for devices which have boundary alignment requirements.

QIO calls the driver at the initiation entry point. We will avoid the subtlety of when packet queuing does not occur and assume it's queued. The driver algorithm is as follows:

1. [Get an I/O request]

   When the driver is called, it immediately calls the internal routine Get Packet ($GTPKT). Get Packet is discussed in Section 3.3.6.1. $GTPKT either delivers a packet, or returns busy. If busy, go to 3.

2. [I/O Issue]

   The driver builds the actual I/O order and issues it. It then returns, in this case to QIO. QIO returns the directive status to the user issuing the original QIO directive and clears the directive from the stack by returning to the directive dispatcher.

3. [GTPKT returns busy]

   If $GTPKT finds the controller busy, thus making it unable to return an I/O packet to the driver, it simply returns a busy indication. The busy indication simply informs the driver that it cannot at this time issue an I/O order. The driver returns to QIO, which returns the directive status to the user issuing the original QIO directive and clears the directive from the stack by returning to the directive dispatcher. The original I/O is in the driver queue, and the issuer can take appropriate action (Waitfor or continue).

3.3.6.1 Get Packet Routine

Get Packet ($GTPKT) is called when an I/O driver needs work. This occurs following a QIO call on the driver, and after a driver has processed an I/O termination. $GTPKT does not know about the initiation cause of a call upon it, it simply attempts to find work for the driver and proceeds as follows:

1. [Scan the driver Q]

   $GTPKT is passed a UCB address, which it uses to locate the SCB and the I/O queue. If the controller is busy, $GTPKT returns busy to the caller. If the controller is free, a queue scan is begun for the highest priority request that the driver can initiate. Note that the queue is already in priority order and the question to be resolved is whether an entry represents work the driver can do. Also note, that if a controller is free, all units on the controller are free. Each queue entry must be checked to determine if the unit to which it is directed is attached. If it is attached, $GTPKT

must check if the attached task is the same as requesting task. If it is, it returns the packet to the driver. If it is not, it continues the queue scan. It will either find work, or return busy.

3.3.7 Termination Processing

On an I/O interrupt (specifically a termination in our present discussion) the driver is entered directly. The driver first calls $INTSV and then $FORK. On return from $FORK access to shared data bases has been linearized and the driver may finish processing of the I/O request. The routine that performs this processing is called I/O done.

I/O Done proceeds as follows:

The controller and unit are unbusied. (Both the controller and unit require busy indicators to enable the Executive to identify the controller-unit busy relationship if a power failure occurs).

The relocation bias and IOSB address in the I/O packet are used to locate the IOSB. If an IOSB was specified then the final status is stored. I/O Done then decrements the outstanding I/O count (the I/O count is used to prevent task checkpointing if outstanding I/O's are pending for the task). If the count goes to zero, and the task was blocked for I/O rundown, the task is unblocked.

If a checkpoint request was pending for the task, an internal routine is called that will initiate the checkpointing process.

A check is then made to determine if an AST address was specified, if not, the I/O packet is released to dynamic storage, a significant event is declared, and I/O Done returns to the driver.

If an AST address was specified, the I/O packet is used for the required AST dynamic storage and it is linked into the AST listhead for the task. A significant event is declared and I/O Done then returns to the driver. On return, the driver will jump to its initiator entry point with the address of the just unbusied device UCB in R5. The initiator calls $GTPKT and the process of looking for work begins again. It should be noted that once underway, either by a call from QIO, or entry from an interrupt termination, a driver propagates its own execution by cycling back to its initiator entry point looking for more work.

3.3.8 I/O Processing Summary

RSX-11M I/O drives itself from four data structures:

The Device Control Block;

The Unit Control Block;

The Status Control Block, and

The 18-word driver queue I/O packet.

Centralized routines facilitate both initiation and termination processing. And, finally, the fork structures used by the drivers along paths requiring more than 500us of processing both linearize access to the I/O data bases, and decrease the non-interruptible time within the system itself.

3.4 MCR - Monitor Console Routine

MCR is the collection of functions that make it possible to operate and control the RSX-11M system from a terminal device. As the link between the collection of services provided by RSX-11M and users who want to make use of these services, MCR provides a number of services, specifically:

Services 1-16 run as MCR overlays.

1. ABOrt

   The task name submitted with the command will be aborted.

2. ALTer

   The priority of the named task is altered.

3. CANcel

   Periodic rescheduling is terminated for task name submitted. The task itself remains in the STD, and may be active or inactive.

4. DEVices

   Symbolic names of all devices known to the system are displayed on the requesting terminal. The display includes any device redirections.

5. FIX

   The named task is fixed in memory. This task cannot be checkpointed, and will remain in memory at task exit.

6. LUN

   A list of Logical Unit Numbers and their associated device symbolics is displayed for the task name submitted with the command.

7. OPen

   Open is used for examination and/or modification of main memory.

8. PARtitions

   This command outputs a description of each partition and sub-partition in the system. The list also specifies if the partition is a task or common partition.

9. REAsign

Reassign de-assigns a Logical Unit Number from one physical device and assigning it to another for the named task.

10. REDirect

Redirect makes posible the redirecting of all I/O requests from one physical device to another having compatible characteristics.

11. REMove

The task named is deleted from the STD. The task so removed is unknown to the Executive and exists only as a disk image.

12. RUN

This command enables a task to be scheduled in terms of:

a. A delta time from now,or

b. A delta time from clock unit synchronization, or

c. Absolute time of day, or

d. Immediate execution.

With options a, b, and c periodic rescheduling is provided.

13. SAVe

This command preserves an image of the RSX-11M Executive on disk such that a hardware bootstrap or the BOOt MCR function can reload and start up the system.

14. SET

Set terminal and device parameters.

15. TASks

This command outputs a description of every task which exists in the STD.

16. UNFix

Unfix reverses the effect of FIX.

Services 17-22 run as tasks.

17. BOOt

The Boot Command will bootstrap an RSX-11M system from a file that was either created by the SYSGEN process or the SAV MCR function.

18. DMOunt

    Declares a volume logically off-line.

19. INItialize

    Creates an RSX-11M structured volume.

20. INStall

    The task contained in the file specified in the command is entered into the STD, and the task header is initialized.

21. MOUnt

    Declares a volume logically on-line.

22. UFD

    This command creates a User File Directory (UFD) and enters its name into the Master File Directory (MFD).

3.4.1 Structure And Operational Environment Of MCR

MCR is an RSX-11M task which operates out of a subpartition which is part of a main partition occupied by the File System*. TKTN, the task termination notification task, also operates out of a subpartition of the File System partition. The partition is set up so that the file system is checkpointable and either TKTN or MCR can checkpoint the File System. Thus, the file system will be swapped out and MCR swapped in when an operator request occurs.

MCR is a tree structured task, and its structure is depicted schematically in Figure 3-5.

----

* We mean that part of the file system referred to as File Control Primitives.

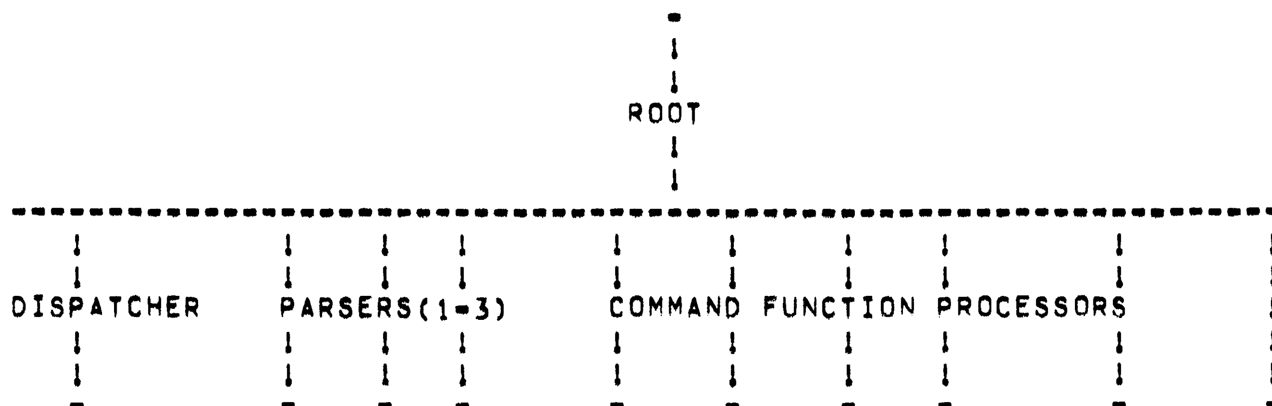


Figure 3-5 MCR Tree Structure

The command function processors are those that process the first 16 console services listed in Section 3.4. The remaining console services run as tasks and not as integral parts of MCR. MCR, in fact, does not distinguish between these task functions, and tasks that it initiates as a result of recognizing an MCR request for functions 17-22 listed in Section 3.4. The console language syntax is defined such that if the first three characters of an input line are not part of the defined command language, then MCR will attempt to initiate the task named

...XXX

Thus, the task named ...JIM can be initiated by entering

JIM

to MCR, or by entering

RUN ...JIM

to MCR.

3.4.2 The Terminal Driver and MCR Initiation

The terminal driver is intimately integrated into the operation of MCR. Since RSX-11M accepts and acts upon unsolicited input from any operator terminal, it is the function of the terminal driver to know when it is receiving input destined for MCR.

When a character on an operator terminal is struck, the resulting interrupt initiates the terminal driver. (Remember the device is full duplex and the keyboard cannot be locked to prevent input when the

device is, in fact, involved in an I/O operation). The driver then acts on the input as follows.

1. [Check the device state]

   Is the device busy. No, go to 3.

2. [The device is busy]

   If the driver was sending output (in an output state) when the character was entered, an input request flag is set in the appropriate UCB and the driver continues sending the output stream. When the output request is finished, processing continues at 5.

   If the terminal was in an input state the character is accepted. Go to 6.

3. [Device is not busy]

   Note, if the device was not busy, the incoming character is the first character of an input line.

   Was the input character a Control-C? (Control-C is an explicit request to communicate with MCR). If the character was a Control-C, the terminal driver executes a fork and execution continues at 4.

   If the first character is not a Control-C, then a check is made to see if the device is attached. If yes, then ignore the character (unsolicited input to MCR on an attached device is not permitted).

   If the device is unattached then it will be considered the beginning of unsolicited input to MCR. Go to 4.

4. [Fork level processing]

   The driver has transferred to fork level because it needs a buffer, and it can only get a buffer at fork level (shared system tables must be altered to obtain a buffer). In addition to getting a buffer, the fork level terminal processing code must check for a rare race condition.

   After the arrival of the Control-C (or a non Control-C character if the terminal is not attached) and between the time the fork is executed and control is regained in the driver, it is possible that the device may have returned to the busy state. This is because we may have just unbusied the device for a previous request when the input interrupt occurred. The interrupt code finds the device free and executes a fork. But before control is regained at fork level, execution is continued in the driver for the previous

request. The driver jumps to the initiator entry to propagate its execution and thus may find another waiting I/O request which it will begin processing since the device is free. Thus the fork routine must recheck the state of the device. If it is busy the input is ignored and the driver returns (exits) from fork level. Else an attempt is made to obtain a buffer for the unsolicited input.

5. [Buffer Acquisition]

If the buffer acquisition attempt is unsuccessful, the driver ignores the input and exits.

If a buffer is obtained, the driver sets up to start an unsolicited input request by initializing various pointers and setting the status of the controller and unit to busy.

If the initial input character was Control-C, then

MCR>

is echoed to signify an explicit request to input to MCR.

Else the input character is stored in the buffer and echoed on the initiating terminal.

The driver returns (exits) from fork level.

6. [Character processing]

Once the terminal driver has determined that input coming from an operator terminal is destined for MCR, it will transfer subsequent characters into the buffer acquired in Step 5. It also echoes the incoming characters. The acceptance of input will cease if:

a. The buffer is filled (the buffer has room for 80 characters) but the maximum accepted depends on the device:

72 for KSR

72 FOR VT05B

80 for LA30S

80 for LA30S

b. An end of line character is entered. The valid end of line characters are:

Control-Z

Carriage Return

ALT-Mode (codes 33, 175, and 176)

7. [Interrupt from a character echo]

Is the device in input mode? If no go get another character from the user output buffer and echo it. If the device is in input mode, is end-of-line set? If no re-enable the keyboard interrupt and exit from the interrupt. If end-of-line is detected, then fork.

8. [End-of-line processing - fork level]

Was the input solicited or unsolicited? For unsolicited input, deposit the UCB address and the terminating character into the input buffer and link the buffer into MCR's input queue, then request MCR to run. The driver itself clears control and unit busy and returns to its initiator entry point.

For solicited input I/O Done will be called. First, the number of characters entered is determined and the buffered input is moved to the soliciting task's input buffer. The driver input buffer is released and I/O Done is called with the second I/O status word equal to the number of bytes entered. The left byte of the first I/O status word is set equal to the terminating character and the right byte to +1. The driver then jumps to the initiator entry point to propagate its execution.

3.4.3 MCR Operation

After the request of MCR by the driver, the file system is swapped out and MCR is swapped in. Control is passed to the MCR root segment which calls the Dispatcher (DSPTCH) overlay. DSPTCH, via a privileged subroutine ($SWSTK), switches to system state. The call to this routine includes a parameter which is the address where the caller wants to return when it switches back to task state. The state switching routine performs the switch and resumes processing in the caller immediately following the call. When $SWSTK is called, it sets up an interrupt entry to the system. Interrupts are locked out while it pushes the passed return address and the PS on the stack. $SWSTK then calls interrupt save ($INTSV). On return from interrupt save R3, R2, R1, R0 are pushed onto the stack and now the stack state simulates that of an EMT. $SWSTK now calls the caller who resumes execution one instruction past the call. When the calling routine finishes, it returns, which takes it back to $SWSTK. $SWSTK jumps to Directive Exit which redispatches the processor. The effect of this is to resume the caller in task state at the passed return address.

MCR now proceeds as follows:

1. [Request an unsolicited input queue entry]

   The Dispatcher calls the Queue Removal routine ($QRMVF). $QRMVF will attempt to remove a buffer and deliver it to the Dispatcher. If no buffer is available (carry set return from $QRMVF) the Dispatcher exits. The buffer is formatted as shown in Figure 3-6.

```
<----WORD---><----WORD---><---UP TO 80 BYTES--->
------------------------------------------------
! LINK TO    ! UCB OF      ! COMMAND INPUT      !
! NEXT BUFFER! INPUT DEV   !                    !
------------------------------------------------
```

   Figure 3-6 Input Buffer

   The queue empty condition will never occur on an initial call to MCR, since MCR is not requested unless something is in the queue. MCR will remain resident until it has processed all the entries in the unsolicited input queue.

   Note that the Dispatcher, during buffer requisition, is operating at system level, and all queue entries are done at fork level. Thus the buffer removal process is linearized with buffer item entry.

   If DSPTCH gets a buffer, it saves the buffer address in a memory location and does a return. This return takes DSPTCH back to task state where the processing of the buffer begins.

2. [Process a [Buffer]

   On return to task state, the dispatcher scans through the buffer and

   compresses out redundant spaces and/or tabs
   converts an Escape character to a Carriage return
   truncates trailing spaces and/or tabs

   If no line terminator is found in the buffer, a CR is inserted as the 80th character. Finally the actual line terminator (either CR or ESC) is saved so it can be restored in the message if the message must be routed to a task other than MCR itself.

   The dispatcher then converts the first three characters to RAD50 and begins to search two internal tables for an MCR function with a matching name.

3. [Searching the function tables - Table descriptions]

MCR has two function tables; one for privileged commands, and one for non-privileged commands.

Privileged commands are those whose unrestricted use could cause privacy violation or system failures and they can only be executed from a privileged terminal. Privileged terminals are identified by a bit in the UCB. These terminals are established at SYSGEN or by the SET command.

Both tables contain a 5-word packet for each command in the class (privileged or non-privileged). The packet appears in Figure 3-7.

```
-----------------------------------
!    RAD50 CMD NAME (3 CHARS)     !
-----------------------------------
!  INDEX INTO COMMAND OVERLAY     !
-----------------------------------
!     ADDRESS OF PARSER TABLE     !
-----------------------------------
!   RAD50 COMMON OVERLAY NAME     !
-----------------------------------
!   INDEX INTO COMMON OVERLAY     !
-----------------------------------
```

Figure 3-7 - Function Table Entry

The table is designed with the assumption that a given MCR function would not require more than three overlays to carry out its intent. Thus, the table entries correspond to three overlay types:

Command overlay,

Parser overlay, and

Common overlay.

The use of these overlay types is in general observed, but exceptions occur and they will be noted.

Typically, any command that can be processed in a single overlay, and whose size is such that it requires all or nearly all of the max overlay size (600 wds) will be classed as a command overlay.

Parsers for the commands are distinct entities and are grouped into overlays. Generally, a given parser services more than one command but three parsers currently service all

the commands. The parser entry is a pointer to a parser table entry shown in Figure 3-8.

```
---------------------------------
!  RAD50 PARSER NAME (3-CHARS)  !
---------------------------------
!   INDEX INTO PARSER OVERLAY   !
---------------------------------
```

Figure 3-8 - Parser Table Entry

Since three parsers service all the commands it is more economical in storage space to point to the parser table rather than include the name and the index in the main function table.

The index is used as the entry point into the parser where the parsing for a given command begins. This is required since a parser can, and generally does, contain parsers for more than one command.

The common overlay is used when the processing for a command is small enough to make it practical to group more than one command into a single overlay. This grouping saves space since ten words are required by the Overlay Runtime System for each overlay in a tree structure. The Index serves the same purpose as the index in a parser overlay.

Note that a command overlay also contains an index. The value of the command overlay index is generally zero. But to maintain the coherence of the table processing commonality, and to allow for flexibility, the index is included.

3a. [Look up and start a function other than an MCR internal function]

The dispatcher then looks in the privileged command table for a name which matches the first three characters in the input buffer. This table contains all the privileged MCR commands. These are noted in the commands listed in Section 3.4.1.

Internally, privileged terminals are identified by a bit in the UCB. The bit is set at SYSGEN or from a privileged terminal using the SET MCR command.

If the command is not found in the privileged command table, the non-privileged command table is searched.

If the name is not in either table, then the dispatcher will prefix three periods to the three buffer characters and using these six characters will search the STD looking for a match

on the name. If it does not find the name it will display an error message on the initiating terminal. If it finds the name, it will request the function to run, supplying as an argument to the requested task the UCB address that was in the input buffer. The UCB address is inserted into the TCB of the requested task as its TI (terminal input) pseudo device. If the attempt to request the task fails an error message is displayed, the buffer is released and MCR exits. Having discovered a non-internal MCR function, MCR must prepare to pass the buffer, since the initiated task is going to issue a GET MCR COMMAND LINE directive. To pass the buffer MCR uses three words in System Common. These words are:

1. The TCB address of the requested task;

2. The address of the command buffer, and

3. The byte count of the number of input characters in the buffer.

MCR fills these words, making synchronizing checks that they are free, since only one triplet exists for passing buffers to a requested task. Thus, until the buffer is emptied, other completed buffers in the queue will be waiting.

Eventually (and this should be soon) the requested task will start running, and issue a GET MCR COMMAND LINE directive. The directive processing then tests for a match on the TCB address in SYSCM and the TCB address of the requesting task. If they match the buffer is passed to the task by copying it into the DPB of the directive. The directive status is set to the byte count, the buffer is released and the TCB address in the SYSCM triplet is cleared. The TCB address being zero is an indication to MCR that the triplet is free.

3b. [Start an internal MCR function]

Once a name match has been found in the command table, the Dispatcher copies words 1, 2, 4 and 5 of the function table entry and both words of the parser table entry for this command into the MCR root segment. Now the Dispatcher scans the function table entry as follows:

3c. IF

A parser exists

THEN

Go to 4

ELSE

IF

a command overlay exists

THEN

Go to 3d

ELSE

IF

a common overlay exists

THEN

Go to 5

ELSE

Abort

3d. Form overlay name, construct required overlay information packet, and enter the root at the point where overlay loading is performed.

4. [Parser functions]

The selected parser will parse the buffer and, if the parse is successful, it will jump back to the root to load the desired function. If the parse fails, the parser deposits an error number in the root and jumps to the entry $ERLD in the root which loads the error overlay.

Ultimately the root will initiate another routine, either the error routine or the requested function.

5. [Function routines]

These routines may further check the input and find errors. If errors are found, the function sets up the error routine and jumps to the root to load an error overlay. If it succeeds, the function will release the buffer and enter the root at the point where the root will reload the dispatcher.

6. [Error Overlay]

The error overlay contains all error messages and the code needed to format the error message from the error number deposited in the root by the MCR component discovering the error.

7. [Final Exit]

The dispatcher calls the queue routine to obtain another buffer, if one is found the cycle of name table scanning resumes (starting at step 2). If no buffers are waiting, MCR exits.

3.5 Partitions

The user area of RSX-11M is divided into partitions. In unmapped systems tasks are bound to a specific partition and must execute from that partition. In mapped systems tasks may be installed and subsequently executed in any partition provided the partition is large enough and sufficient checkpoint space is available in the task image.

A partition always consists of at least a main partition; the main partition can be subdivided into as many as seven subpartitions. The execution of tasks within the main partition and its subpartitions is mutually exclusive. This means that if a task is executing in the main partition no tasks may be executing in any subpartition of the main partition. Contrarywise, if a task is executing in a subpartition no task requiring the main partition may be executing. The subpartitions, however, can all execute in parallel.

Subpartitions exist so as to make it possible to reclaim and subdivide large partitions that service nonrealtime tasks like language translators.

If a main partition task is fixed in memory then no other main partition or subpartition task may execute until the task is unfixed. A fixed subpartition task will exclude the execution of a main partition task and other tasks wanting to execute from the subject subpartition. Other subpartitions operate independently.

An identical set of conditions apply to tasks that are not checkpointable.

The manipulation of a partition and its subpartitions becomes more intricate when the tasks occupying them are not fixed in memory and are checkpointable. Any number of tasks can be waiting for use of the partition or subpartition. Given that no tasks running out of a partition are fixed in memory, and that all are also checkpointable, the sole determiner of which queued tasks waiting for the partition will run is priority.

When a task is requested it is entered into the appropriate partition wait queue and a sequence is initiated to determine if the existence of this task in the partition wait queue will require checkpointing of the task currently occupying the partition. If the task entering the wait queue is of higher priority then that of the current task, checkpointing will proceed.

When a task that is not fixed in memory exits, the partition wait queue for the partition being freed is examined for the highest priority task waiting for the partition, and when located this task is loaded and put into active competition for processor resources.

Before proceeding to the detailed algorithms used to service a partition the key concepts are listed below:

* A partition consists of a main partition and up to seven subpartitions.

* Execution between a main partition and its subpartitions is mutually exclusive.

* Execution among subpartitions may proceed in parallel.

* Tasks fixed in memory or not checkpointable lock up the partition or subpartition from which they are executing.

* Any number of tasks may be waiting for a partition.

* Task access to a partition is based on priority. Tasks not fixed in memory or not checkpointable will be rolled out to make room for higher priority tasks waiting to occupy the partition.

3.5.1 Partition Control Data Structures

The data structures which service the partition concepts outlined above are shown in Figure 3-9.

```
        -------
SPARHD !       !---
        -------   !
                  !  ------    -----    -----
                  -->!PCB !--->!TCB!--->!TCB!
                     !MAIN!    !   !    !   !
                  ---!    !    -----    -----
                  !  ------
                  !
                  !  ------
                  -->!PCB !
                     !SUB !
                  ---!    !
                  !  ------
                  !
                  !  ------
                  -->!PCB !
                     !SUB !
                  ---!    !
                  !  ------
                  !
                  !  ------    -----    -----
                  -->!PCB !--->!TCB!--->!TCB!
                     !MAIN!    !   !    !   !
                  ---!    !    -----    -----
                  !  ------
                  !
                  !  ------
                  -->!PCB !
                     !SUB !
                  ---!    !
                  !  ------
                  !
                  !  ------
                  -->!PCB !
                     !SUB !
                     !    !
                     ------
```

Figure 3-9 - Partition Data Structure

The very composition of the structure provides insight into both the extent of the service and how internally the service is propagated to the user level.

A word is maintained in system common ($PARHD) that points to the first main Partition Control Block (PCB) in the system. From this PCB are linked the PCB's for any subpartitions defined for this main partition. All PCB's defining other partitions and subpartitions are similarly linked. The last PCB is a subpartition PCB except in the case where there are no subpartitions of the subject main partition.

In this case the main partition PCB links to the next main partition PCB. The data structure is then repeated until we run out of PCB's to link together. Notice that the TCB's of tasks waiting to occupy either the main partition or a subpartition are always linked from the main partition PCB. The TCB's are ordered by priority and each contain a pointer to the PCB of the partition for which they are waiting.

The scheduling of a partition is done by the next task ($NXTSK) routine. It is the function of $NXTSK to select the next task in the list of TCB's linked from the main partition wait queue that is to occupy the main partition or a subpartition of the main partition. Note that this process is independent of which task in the system will gain control of the CPU next.

There are four specific events which can result in a change of control in a partition.

1. Task exit of a nonfixed task.

   $NXTSK must now look for another task waiting for the partition.

2. The loader has completed bringing a task into memory.

   A new task is now ready to compete for the processor.

3. An initiation type request occurs.

   This can be:

   a. A RUN command from a terminal .

   b. A RUN or REQUEST Executive directive.

   In this case several checks must be made by $NXTSK. (we will discuss these checks shortly)

4. The outstanding I/O count for a checkpointable task waiting for swap out has gone to zero.

   The loader must checkpoint the task and then call $NXTSK to select the next task to occupy the partition.

Given these preliminaries we can now examine the algorithms used to service partitions.

3.5.2 Partition Algorithms

1. [servicing initiation requests]

Is the requested task active or currently being fixed in memory? If yes, return carry set. (The request is redundant since the task is either running or in process or being set-up to run).

If the task is in memory, it is fixed in memory and a partition allocation pass is not required. The following task set up is performed.

> An initial stack is constructed that will cause the task to start executing at its transfer address.
>
> The task status word is set to active.
>
> A conditional schedule request is set that will force redispatching of the processor if the task just made action is of higher priority than the currently running task.

If the task is not in memory, then it is entered by priority in the Partition Wait Queue and $NXTSK is executed.

It should be noted that a task fixed-in-memory remains physically in control of memory even if the task exits. A task that is not fixed-in-memory is removed from memory at task exit thus freeing the memory in which it resides.

2. [$NXTSK - Select the next task to run in a partition]

$NXTSK begins a scan of the main partition wait queue seeking to determine if the partition, for which a task is waiting, is free. Note that a given task is not necessarily the one most recently inserted into the partition wait queue. Examination of the partition data structure will reveal that all TCB's, both for main and subpartitions, are linked from the main partition wait queue.

$NSTSK checks if the partition is free. No, resume scan. Yes, insert the TCB address into the PCB declaring ownership of the PCB. Set the busy flag in both the main and subpartition. Remove the TCB from the partition wait queue, insert it into the loader queue.

A TCB is always linked into the STD. A given TCB may also be linked into either the loader queue or the partition wait queue.

The loader is then requested and $NXTSK tries to allocate another task within the partition. The allocation process continues until either all waiting tasks have been assigned to partitions or an allocation failure occurs.

3. [$NXTSK Checkpointing - Main partition requests]

Whenever $NXTSK finds a task waiting for a main partition that is busy it must make a checkpoint decision. For a main partition request, it proceeds as follows:

3a. Is the task waiting for the main partition of sufficient priority to pre-empt the task currently occupying the main partition or a subpartition; no exit, the partition cannot be allocated to the waiting task.

3b. Is the task occupying the partition in any of the following states?

Not checkpointable

Fixed in Memory

Being Fixed in Memory

If yes, exit the partition cannot be allocated to the waiting task. No, go to 3a and check the next subpartition. Eventually, either every subpartition is found to be checkpointable, or the main partition cannot be freed and $NXTSK exits.

3c. [Checkpointing subpartitions]

Once $NXTSK determines the main partition can be made available by checkpointing the tasks in the subpartitions it starts the checkpointing process as follows:

Is the I/O count for the task occupying subpartition =0. No, go to 3d. Yes, execute the checkpoint initiation routine.

Move the TCB occupying this partition into the loader queue, set a bit establishing the task is checkpointed, and request the loader. No further action on this subpartition is taken until the loader recalls $NXTSK after the checkpointing process is completed.

3d. Set a bit which indicates the task is to be checkpointed when its I/O count reaches 0. (This bit is tested by I/O Done, and if set, I/O Done calls the checkpoint initiation routine in 3c above).

4. [$NXTSK checkpointing - Subpartition requests]

If a subpartition is requested the essentials of the algorithm in 3 above is followed, but only the requested subpartition need be checked.

Steps 1-4 complete the description of partition management. But since the loader is intimately involved with the algorithms, it will be described in this section.

3.5.3 The Loader

The loader, which is a resident RSX-11M system task, has three functions:

1. Loading new tasks (initial reading of disk image);

2. Checkpointing tasks (writing task checkpoint image to disk), and

3. Resuming checkpointed task (reading task checkpoint image back from disk).

The loader has a single objective; to empty the queue of tasks waiting for its attention. The queue is serviced FIFO and 2 bits in the task status word, the checkpoint bit and the out of memory bit, determine the loader action on an entry.

CKPT on
OUT OF MEM on
The task is read back into memory from its checkpoint area.

CKPT on
OUT OF MEM off
The task is written from memory into its checkpoint area.

CKPT off
OUT OF MEM on
The task is read into memory from its load image.

CKPT off
OUT OF MEM off
This combination is illegal.

When the loader removes the next entry from its queue, it assumes memory is available if the task is to be read; after the loader writes a task into its checkpoint area Release Partition is called which in turn calls, $NXTSK to select the next task that will occupy the partition.

On reading a task into memory, the loader:

1. Marks it in memory (i.e. clears OUT OF MEM), and

2. Builds a initial stack (on initial reads only)

The marking of the task in memory has the effect of unblocking the the task so it can compete for system resources. As soon as a task is requested it is considered initiated. It will not compete for system resources however, until the loader marks it in memory. After completing the service for a queue entry the loader looks for more work, and when the queue is empty, it exits.

Though the loader and $NXTSK call each other they are almost totally ignorant of each others function. The call from the loader to $NXTSK is such that $NXTSK can't distinguish it from other events that trigger $NXTSK. And, the loader makes very few decisions as evidensed by the fact that the description of its algorithm is best presented in narrative form.

4.0 FAULT ISOLATION - SOME GENERAL HINTS

4.1 Introduction

Though RSX-11M is a real time, multiprogramming system, it has, in the real memory version, no "brickwall" protection. The basic machine configuration which RSX-11M supports, has only a single program state, and no memory protection. The lack of these features simply mean that the user state tasks can fault such that the system itself faults. In these circumstances, it becomes extremely important to develop the skills and discipline needed to rapidly isolate the source of a system failure. This section is a first attempt at recording the experience gained thus far in isolating software faults that occur in RSX-11M. Ultimately this section should evolve into a handbook for field software specialists. To reach this much needed state, it will require inputs and suggestion from all members, of the RSX-11M team.

4.2 Fault Classifications

Three culprits can be identified when the system faults:

1. A user state task has faulted such that it causes the system to fault;

2. The RSX-11M system software itself has faulted, or

3. The host hardware has faulted.

The immediate action on the part of the programmer subject to one of these errors is to determine which of these three cases is the source of the fault. Our prime concern will be the procedures which may help the programmer uncover the fault source. The repair of the fault itself is assumed to be the programmers responsibility.

Faults manifest themselves in roughly three ways (and they are listed here in order of increasing difficulty of isolation)

1. The system displays the CRASH printout and halts. The CRASH printout is discussed in Section x.x.

2. The system halts but displays nothing.

3. The system is in an unintended loop.

4.3 Immediate Servicing

RSX-11M can be built to contain resident crash reporting and panic dump routines, and our comments assume such a system. (It should be noted that the minimal system will not have space for these routines.)

The immediate aim, regardless of the fault manifestation, is to initiate the crash reporting and panic dump routines.

4.3.1 Case 1 - The System Has Displayed the Crash Printout

In this case, all the basic information describing the state of the system has been displayed. We will pick up the actual Crash printout after we have described how to invoke Panic Dump for cases 2 and 3.

4.3.2 Case 2 - The System Has Halted - No Information Displayed

Before taking any action preserve the current PS and PC (i.e. examine and record). The procedure depends on the particular PDP-11 processor. For all processors, the PC is displayed in the address register. The PC can also be obtained as follows:

For an 11/45

1. Enter a 7 in the console Switch Register
2. Depress Load Address
3. Depress Register Examine
4. The PC is displayed in the data lights. Record the PC.

For an 11/40 or 11/10

1. Enter 777707 in the console switch register
2. Depress Load Address
3. Depress Examine
4. The PC is displayed in the data lights. Record the PC.

Now obtain the PS, the procedure for which, is identical in all systems.

1. Enter 777776 in the console switch register
2. Depress Load Address
3. Depress Examine

4. The PS is displayed in the data lights. Record the PS.

Next invoke the Panic Dump routine by entering 40(8) in the switch register, Depressing Load Address, and then Start.

40(8) is the address of a JMP to the Panic Dump Routine in any RSX-11M system.

The Panic Dump saves R0-R6 and then halts awaiting dump limits to be entered via the console switch register. The PS is cleared when START is depressed, and the original PC is destroyed. Thus the importance of recording these vital pieces of debugging information before initiating the Panic Dump.

Dumps of selected blocks of memory may be obtained using the following procedure:

1 - Enter the low dump limit in the console switch register and depress continue. The processor will immediately halt again.

2 - Enter the high dump limit in the console switch register and depress continue. The dump will begin on the device whose CSR address is D$$BUG (usually 177514 which is the line printer). At the end of the dump the processor will again halt awaiting the input of another set of dump limits.

   To reach the same status arrived at with crash reporting in Case 1 above, enter the dump limits 0-520(8) when the panic dump first halts. This will dump the system stack and the registers.

4.3.3 Case 3 - System Is In An Unintended Loop

Proceed as follows:

Halt the processor

Record PC, and PS as in 4.3.2 above.

After recording the PS and PC, the programmer may want to step through a number of instructions in an attempt to locate the loop.

After the attempt to locate the loop transfer to the panic dump routine as in Case 2.

This brings us to an equivalent status for the three fault situations.

4.4 Other Pertinent Fault Isolation Data

Before proceeding with the task of locating the fault, the programmer is strongly advised to dump system common (SYSCM). He can accomplish this by looking for the file SYSCM in the Executive load map listing and entering the appropriate limits to the Panic Dump Routine. SYSCM contains a number of critical pointers and listheads.

In addition the programmer should dump the dynamic storage pool and the device tables. The dynamic storage region is the module INITL and the device tables are in SYSTB.

The programmer now has:

PS

PC

The Stack

R0-R6

The Dynamic Storage Region

The Device Tables, and

System Common

This data is a minimal requirement for effective fault isolation. If an individual programmer plans to consult with other group members on the source of a fault, he should do so only after he has collected this data. Eventually, we will require that all SPR's supply this information.

4.5 Fault Tracing

Three pointers in SYSCM are critical in fault tracing.

$STKDP - Stack Depth Indicator

This data item will indicate which stack was being used at the time of the crash. As will be seen, this plays an important role in determining the origin of a fault. The following values apply.

+1 - User (task state) stack

0 or less - System stack

STKTCB - Pointer To the current Task TCB

This is the TCB of the user level task in control of the CPU.

$HEADR - Pointer To The Current Task Header.

Locating the header provides some other useful data. The first word in the header is the users stack pointer the last time it was saved. If the Stack Depth is +1 then the user has managed to crash the system. In a system with brickwall protection (for example, the mapped RSX-11M system), the user should not be able to crash the system. Even if the user stores in the Executive, the crash will not occur until the state is switched and then the system will crash. Such a fault may prove difficult to locate.

If the user branches wildly into the Exec it will terminate the user task, but the system will continue to function (possibly erroneously). Knowing the users stack pointer provides one more link in the chain which may lead to the resolution of the fault.

4.5.1 Tracking Faults Following An Automatic Display Of The System State (Case 1)

First examine the system stack pointer. Usually an Executive failure is the result of an SST type trap within the Executive (other than the specialized use of the trap instruction).

If an SST does occur within the Executive, then the origin to the call on the crash reporting routine will be in the SST service module. (The crash call is initiated by issuing an IOT at a stack depth of zero or less.)

A call on crash also occurs in the Directive Dispatcher when an EMT was issued at a stack depth of zero or less, or a trap instruction was executed at a depth of less than zero. The stack structure in the case of an internal SST fault is as follows:

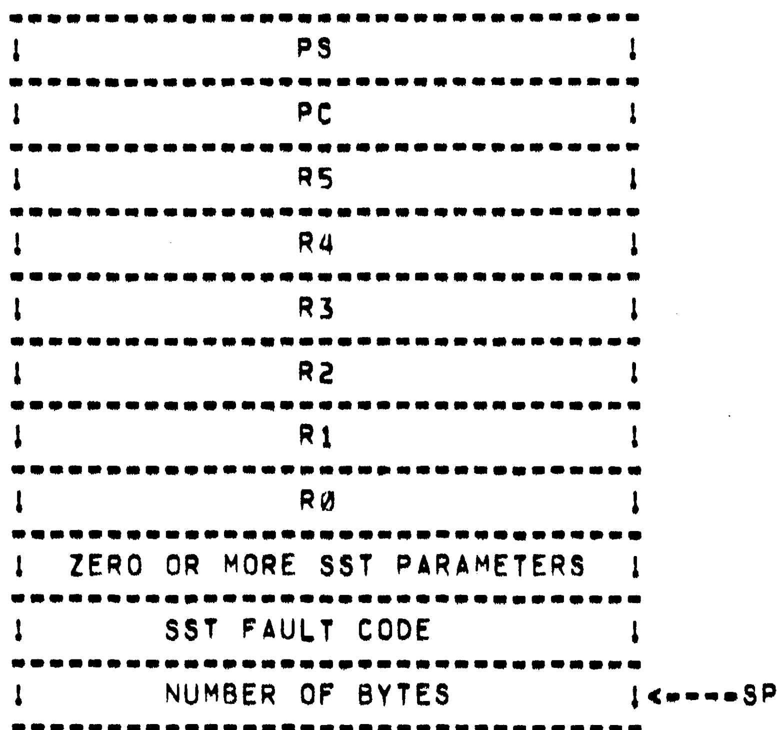


Figure 4-1

The PC points to the instruction following the one which caused the SST failure. The number of bytes is the number of bytes that are normally transfered to the user stack when the particular type of SST occurs. If the number is 4, then just the PS and PC are transfered and there are no SST parameters. The definition of each fault code can be found in the file ABODF.

If the failure is detected in $DRDSP the stack is the same as Figure 4-1 except the number of bytes, SST fault code, and SST parameters are not present. The crash report message, however, will indicate the failure occurred in $DRDSP.

There is one SST-type failure that will not have the stack structure of Figure 4-1 and that is stack underflow. To distinguish the two cases, determine where the crash actually occurred. If it occurred in $DRDSP, or was a normal SST crash, then Figure 4-1 is the stack structure preserved. If it was a non-normal SST, then Figure 4-2 is the preserved structure.

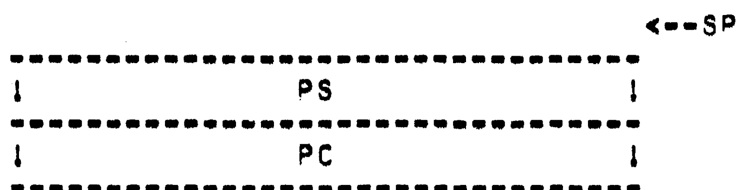


Figure 4-2

Non-normal SST failures occur when it is not possible to push information on the stack without forcing another SST fault. When this occurs, a direct jump to the crash reporting routine is made rather than an IOT crash. The PS and PC on the stack are those of the actual crash, and the address printed out by the crash reporting routine is the address of the fault rather than the address of the IOT that crashes the system. Note that the crash reporting routine removes the PC and PS of the IOT instruction from the stack which in this case is incorrect. Thus the stack pointer will appear to be 4 greater than it really is (i.e. as in figure 4-2).

The programmer now has all the information he needs to isolate the cause of the failure. From this point on he must rely on his own experience and knowledge of the interaction between his program and the services provided by the Executive.

4.5.2 Tracking Faults When The Processor Halts Without Providing A Fault Display

Tracking starts with an examination of $STKDP, $TKTCB, and $HEADR. The difficulty in tracking failures in this case is that the system stack is not directly associated with the cause of a failure.

By examining $STKDP, one can determine the system state at the time of failure. If it was in user state, the next step is to examine the users stack. The examination process focuses on scanning the stack for addresses which may turn out to be subroutine links which will ultimately lead to a thread of events isolating the fault. This is essentially the same aim in looking at the system stack if $STKDP is zero or less.

Frequently a fault will occur such that the SP points to Top of Stack (TOS)+4. This results from issuing an RTI when the top two items on the stack are data; this will result in a wild branch, then, most probably, a halt. Figure 4-3 shows a case, where two data items are on the stack when the programmer executes an RTI.

TOS points to a word containing 40100. Suppose that location 40102 contains a halt. This indicates that the original SP was four bytes below the final SP and fault tracing should begin from the previous SP.

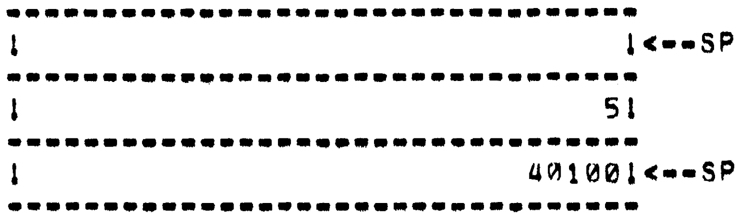


Figure 4-3

4.5.3 Tracking Faults When An Un-intended Loop Has Occurred.

After halting the processor, we are roughly in the same state as the preceeding section. Some specific suggestions are to check for a stack overflow loop. Patterns of data successively duplicated on the stack indicated a stack looping failure.

The console lights may also indicate the origin of a problem. A tight, glimmering pattern may indicate hardware, while a blinking, more elongated, yet repetitive pattern may indicate software.

5.0 DATA STRUCTURES

5.1 Partition Control Block (PCB)

```
         ---------------------------------
P.LNK   !        PCB  LINK WORD          ! 0
         ---------------------------------
P.NAM   !      PARTITION NAME            ! 2
         -------               ---------
        !          (RADIX 50)            ! 4
         ---------------------------------
P.SUB   ! PNTR TO NXT SUB-PARTITION PCB  ! 6
         ---------------------------------
P.MAIN  ! POINTER TO MAIN PARTITION PCB  ! 10
         ---------------------------------
P.REL   !        PHYSICAL ADDRESS        ! 12
         ---------------------------------
P.SIZE  !  SIZE OF PARTITION IN BYTES    ! 14
         ---------------------------------
P.WAIT  !     PARTITION WAIT QUEUE       ! 16
         ----------          ----------
        !            LISTHEAD            ! 20
         ---------------------------------
P.BUSY  !     PARTITION BUSY FLAGS       ! 22
         ---------------------------------
P.TCB   !     TCB ADDRESS OF OWNER       ! 24
         ---------------------------------
P.STAT  !  PAR STATUS   !# APR'S TO LOAD! 26
         ---------------------------------     --- MAPPED
P.PDR   !    CONTENTS OF LAST PDR        ! 30    ! SYSTEM
         ---------------------------------       !  ONLY
P.HDR   !  ADDR OF HDR IN MAPPED SYSTEM  ! 32    !
         ---------------------------------     ---
```

5.1.1 Partition Control Block Details

P.LNK

Description:

Pointer to next partition. The PCB's are linked in physical address order, highest to lowest. If a main partition has subpartitions these are linked in the PCB chain off the main partition in highest to lowest address order. The last subpartition of a main partition will either end the PCB chain or link to the next main partition. A main partition with no subpartitions either links to the next main partition or ends the chain.

Initialization/Access:

P.NAM

Description:

Radix 50 partition name.

Initialization/Access:

P.SUB

Description:

Points the next subportion. Structured and used analgously to P.LNK when manipulating a chain of subpartition PCB's.

Initialization/Access:

P.MAIN

Description:

Backpointer from a subpartition to its parent main partition.

Initialization/Access:

P.REL

Description:

Partition base relocation bias. In a mapped system P.REL is the bias; in an unmapped system, P.REL is the actual partition address.

Initialization/Access:

P.HDR (unmapped system only)

Description:

Task Header pointer.

P.SIZE

Description:

Size of partition in bytes.

Initialization/Access:

P.WAIT

Description:

A pointer to a list of tasks awaiting the use of the partition. The list is ordered by priority and is searched to determine which task should be in control of the partition.

Initialization/Access:

P.BUSY

Description:

A two byte field. The first byte, the busy status, is the inclusive or of the state for the main partition and all its subpartitions. The second byte, the busy mask, contains a busy(1) not busy(0) setting for the main partition and its seven subpartitions.

Initialization/Access:

Under complete control of the set command processor.

P.TCB

Description:

TCB address of partition's owner

Initialization/Access:

P.STAT

Description:

The status bits have the following meaning:

Bit	Symbolic	Meaning
0-2	PS.APR	Starting APR number for non-PIC libraries
3	Reserved	
4	Reserved	
5	Reserved	
6	PS.PIC	Library is Position Independent 1=Yes
7	PS.COM	Partition is a COMMON/LIBRARY partition

Initialization/Access:

P.NAPR

Description:

Number of APR's to load.

Initialization/Access:

P.PDR (mapped systems only)

Description:

Contents of the last PDR.

Initialization/Access:

P.HDR (mapped systems only)

Description:

Address of the Task Header.

5.2 Task Control Block

```
         --------------------------------
T.LNK    !      UTILITY LINK WORD       ! 0
T.PRI    --------------------------------
T.IOC    !  I/O CNT    !  PRIORITY      ! 2
         --------------------------------
T.TCB    !     ADDRESS OF THIS TCB      ! 4
         --------------------------------
T.NAM    !          TASK NAME           ! 6
         ----------     IN     ----------
         !           RADIX50            ! 10
         --------------------------------
T.RCVL   !                              ! 12
         ------- RECEIVE LISTHEAD -------
         !                              ! 14
         --------------------------------
T.ASTL   !                              ! 16
         ---------AST LISTHEAD-----------
         !                              ! 20
         --------------------------------
T.EFLG   !          TASK LOCAL          ! 22
         ------- EVENT FLAGS 1-32--------
         !                              ! 24
         --------------------------------
T.UCB    ! UCB ADDR OR REQUEST TERMINAL ! 26
         --------------------------------
T.TCBL   !     TASK LIST THREAD WORD    ! 30
         --------------------------------
T.STAT   !       TASK STATUS WORD       ! 32
         --------------------------------
T.LBN    ! LOGICAL BLOCK !TSK STATUS EXT ! 34
         ----    -------------------------
         !  NUMBER OF TASK LOAD IMAGE   ! 36
         --------------------------------
T.LDV    !  UCB ADDRESS OF LOAD DEVICE  ! 40
         --------------------------------
T.PCB    ! PCB ADDRESS OF LOAD PARTITION ! 42
         --------------------------------
```

TASK STATUS WORD BIT DEFINITIONS

BIT	SYMBOLIC		
0	TS.CKD	Checkpoint Disable	1=yes
1	TS.CKR	Checkpoint Request	1=yes
2	TS.CKP	Checkpointed	1=yes
3	TS.OUT	Out of core	1=yes
4	TS.FXD	Fixed in memory	1=yes
5	TS.BFX	Task Being Fixed	1=yes
6	TS.CHK	Task checkpointable	0=yes
7	TS.AST	AST in progress	1=yes
8	TS.ACP	Ancillary control processor	1=yes
9		Reserved	
10		Reserved	
11	TS.REM	Remove task at exit	1=yes
12	TS.MSG	Abort Message being outout	1=yes
13	TS.DST	AST recognition disabled	1=yes
14	TS.RDN	I/O rundown in progress	1=yes
15	TS.EXE	Task in execution	0=yes

TASK STATUS EXTENSION BYTE BIT DEFINITIONS

BIT	SYMBOLIC		
0	TS.WFR	Task in WAITFOR	1=yes
1	TS.SPN	Task suspended	1=yes
2	none	Saved TS.WFR on AST	
3	none	Saved TS.SPN on AST	
4	TS.MCR	Task requested as MCR function	1=yes
5	TS.ABO	Task marked for abort	1=yes
6	TS.PRV	Task is privileged	1=yes
7	TS.HLT	Task being halted	1=yes

5.3 Task Header

Label	Contents	Offset
H.CSP	CURRENT SP	0
H.HDLN	HEADER LENGTH IN BYTES	2
H.PCBT	TASK PCB ADDRESS	4
	LOW VIRTUAL ADDRESS	6
	HIGH VIRTUAL ADDRESS	10
	ACCESS ! USER PDR	12
H.PCBC	COMMON PCB ADDRESS 1	14
	LOW VIRTUAL ADDRESS	16
	HIGH VIRTUAL ADDRESS	20
	ACCESS ! USER PDR	22
	COMMON PCB ADDRESS 2	24
	LOW VIRTUAL ADDRESS	26
	HIGH VIRTUAL ADDRESS	30
	ACCESS ! USER PDR	32
	COMMON PCB ADDRESS 3	34
	LOW VIRTUAL ADDRESS	36
	HIGH VIRTUAL ADDRESS	40
	ACCESS ! USER PDR	42
	0 END OF PCB'S (SENTINEL)	44
H.DSW	SAVE AREA FOR TASK DSW	46
H.FCS	FCS IMPURE POINTER	50
H.FORT	FORTRAN IMPURE POINTER	52
H.OVLY	OVERLAY IMPURE POINTER	54
H.RSVD	RESERVED	56
H.EFLM	EVENT FLAG MASK WORDS	60

```
            --------          ----------
            !       FOR EVENT FLAGS        ! 62
            --------          ----------
            !            1-64              ! 64
            --------         -----------
            !                              ! 66
            --------------------------------
   H.CUIC   !     CURRENT TASK UIC         ! 70
            --------------------------------
   H.DUIC   !     DEFAULT TASK UIC         ! 72
            --------------------------------
   H.IPS    !     INITIAL PS WORD          ! 74
            --------------------------------
   H.IPC    !     INITIAL PC WORD          ! 76
            --------------------------------
   H.ISP    !          INITIAL SP          ! 100
            --------------------------------
   H.ODVA   !     ODT SST VECTOR ADDRESS   ! 102
            --------------------------------
   H.ODVL   !     ODT SST VECTOR LENGTH    ! 104
            --------------------------------
   H.TKVA   !    TASK SST VECTOR ADDRESS   ! 106
            --------------------------------
   H.TKVL   !     TASK SST VECTOR LENGTH   ! 110
            --------------------------------
   H.PFVA   !         POWER FAIL AST       ! 112
            --------------------------------
   H.FPVA   !    FLOATING PNT EXCPN AST    ! 114
            --------------------------------
   H.FPSA   ! FP OR EAE REG SAVE AREA ADDR ! 116
            --------------------------------
            !           REVERSED           ! 120
            --------------------------------
            !           RESERVED           ! 122
            --------------------------------
            !           RESERVED           ! 124
            --------------------------------
   H.GARD   !  ADDRESS OF STACK GUARD WORD ! 126
            --------------------------------
   H.NLUN   !     COUNT OF LOGICAL UNITS   ! 130
            --------------------------------
   H.LUN    !     START OF LOGICAL UNITS   ! 132
            ---   EACH LUN CONSISTS OF  ----
            !     TWO WORDS, UP TO 255     !
            ---   LUNS CAN BE ACCOMODATED --
            !                              !
            ---            .           ----
            /              .               /
            /              .               /
            /              .               /
            !                              !
            --------------------------------
            !                              !
```

'REVERSED' ?! (I used 'RESERVED' here)

```
----------LAST LUN ENTRY---------
!                               !
---------------------------------
!                               !
------FLOATING POINT OR EAE -----
!          SAVE AREA            !
-------- (3 OR 25 WORDS) --------
/                               /
/                               /
/                               /
!                               !
---------------------------------
!          CURRENT PS           !
---------------------------------
!          CURRENT PC           !
---------------------------------
!          CURRENT R5           !
---------------------------------
!          CURRENT R4           !
---------------------------------
!          CURRENT R3           !
---------------------------------
!          CURRENT R2           !
---------------------------------
!          CURRENT R1           !
---------------------------------
!          CURRENT RØ           !
---------------------------------
!       STACK GUARD WORD        !
---------------------------------
```

CHAPTER 3

5.4 I/O DATA STRUCTURES

Of all the control blocks in the I/O data structure, only four are of direct concern to a driver:

1. The I/O Packet;

2. The DCB;

3. The UCB, and

4. The SCB.

Although the data structures contain an abundance of data pertaining to input/output operations, drivers per se are involved only with a subset of this data. Most of the data which is used by a driver is supplied in the data structure source, and is not referenced during driver execution.

5.4.1 The I/O Packet

Figure 3-1 is a layout of the 18-word I/O Packet which is constructed and placed in the driver I/O queue by QIO directive processing and subsequently delivered to the driver by a call to $GTPKT. Figure 3-2 is the DPB from which the I/O Packet is generated.

```
            ---------------------------------
I.LNK       !   LINK TO NEXT I/O PACKET     ! 0
I.EFN       ---------------------------------
I.PRI       !       EFN      !     PRI      ! 2
            ---------------------------------
I.TCB       !   TCB ADDRESS OF REQUESTER    ! 4
            ---------------------------------
I.LN2       !  ADDRESS OF SECOND LUT WORD   ! 6
            ---------------------------------
I.UCB       ! ADDRESS OF REDIRECT UCB       ! 10
            ---------------------------------
I.FCN       ! FUNCTION CODE !   MODIFIER    ! 12
            ---------------------------------
I.IOSB      !VIRTUAL ADDR OF  I/O STATUS BLK! 14
            ---------------------------------
            !   RELOCATION BIAS OF IOSB     ! 16
            ---------------------------------
            !     REAL ADDRESS OF IOSB      ! 20
            ---------------------------------
I.AST       !VIRTUAL ADDR OF AST SERVICE RTN! 22
            ---------------------------------
I.PRM       !                               !
            -----                       -----
            !                               !
            -----        DEVICE         -----
            !                               !
            -----      PARAMETERS       -----
            !                               !
            -----                       -----
            !                               !
            -----                       -----
            !                               !
            ---------------------------------
            !                               !
            ---------------------------------
            !                               !
            ---------------------------------
```

Figure 3-1 I/O Packet Format

5.4.1.1 I/O Packet Details - The I/O Packet is built dynamically by QIO directive processing. Thus, no static fields exist with respect to a driver. I/O Packets are created dynamically and, therefore, the first parameter does not apply. Fields are classified as:

not referenced,
read-only, or
read/write.

I.LNK

Description:

Links I/O Packets queued for a driver. A zero ends the chain. The listhead is in the SCB (S.LHD).

Initialization/Access:

not referenced.

I.PRI

Description:

Priority copied from the TCB of the requesting task.

Initialization/Access:

not referenced.

PG

I.EFN

Description:

Contains the event flag number as copied by QIO directive processing from the requester's DPB.

Initialization/Access:

not referenced.

I.TCB

Description:

TCB address of the requesting task.

Initialization/Access:

not referenced.

I.LN2

Description:

Contains the address of the second word of the LUT entry in the task header to which the I/O request was directed. For open files on file structured devices, this word contains the address of the window block, otherwise it is zero.

Initialization/Access:

not referenced.

I.UCB

Description:

Contains the address of the Redirect UCB if the starting UCB has been subject to a Redirect MCR command.

Initialization/Access:

not referenced.

I.FCN

Description:

Contains the function code (see table 3-1) for the I/O request.

Initialization/Access:

read-only.

I.IOSB

Description:

I.IOSB contains the virtual address of the I/O Status block (IOSB) if one is specified or zero if not.

I.IOSB+2 and I.IOSB+4 contain the address doubleword for the IOSB (see Appendix A for a detailed description of the address doubleword). On an unmapped system, the first word is zero, the second word is the real address of the IOSB.

In a mapped system, the first word contains the relocation bias of the IOSB; the bias is, in effect, the 32-word block number in which the IOSB starts. This block number is derived by viewing available real memory as a collection of 32-word blocks numbered consecutively, starting with 0. Thus, if the IOSB starts at physical location 3210(8), its block number is 32(8).

The second word is formatted as follows:

Bits 0-5   Displacement In Block (DIB)
Bits 6-12 All zeros
Bits 13-15   6

The displacement in block is the offset from the block base. In the above example where the IOSB started at 3210(8), the DIB is equal to 10(8).

The value 6 in bits 13-15 is constant. It is used to cause an address reference through Kernel Page Address Register 6. Again, see Appendix A for details.

The deferral of a discussion of the address doubleword to an appendix reflects the fact that a writer of a conventional driver has almost no need to concern himself with the contents or format of the address doubleword. Its construction and subsequent manipulation are normally external to the driver; subroutines are provided as Executive services for programmed I/O to render the manipulations of I/O transfers transparent to the driver itself.

Initialization/Access:

not referenced.

I.AST

Description:

Contains the virtual address of the AST service routine to be executed at I/O completion. If no address is specified, the field contains zero.

Initialization/Access:

not referenced.

I.PRM

Description:

Device dependent parameters copied from the DPB.

Initialization/Access:

not initialized, read-only.

The QIO Directive Parameter Block (DPB) is constructed as follows:

```
---------------------------------
!    LENGTH     !      DIC      ! 0
---------------------------------
!  FUNCT CODE   !   MODIFIER    ! 2
---------------------------------
!   RESERVED    !      LUN      ! 4
---------------------------------
!   PRIORITY    !      EFN      ! 6
---------------------------------
!  I/O STATUS BLOCK ADDRESS     ! 10
---------------------------------
!          AST ADDRESS          ! 12
---------------------------------
!             DEVICE            ! 14
------                     ------
!           DEPENDENT           ! 16
------                     ------
!           PARAMETERS          ! 20
------                     ------
!                               ! 22
------                     ------
!                               ! 24
------                     ------
!                               ! 26
---------------------------------
```

Figure 3-2 QIO DPB

The parameters have the following interpretation.

Length (required):

The length of DPB, which for the RSX-11M QIO directive, is always fixed at twelve words.

DIC (required):

Directive Identification Code. For the QIO directive, the value is a 1.

Function Code (required):

The code of the requested I/O function (0 thru 31.).

Modifier:

Device dependent modifier bits.

Reserved:

Reserved byte and must not be used.

LUN (required):

Logical Unit Number.

Priority:

Request priority. Ignored by RSX-11M, but space must be allocated for RSX-11D compatibility.

EFN (optional):

Event flag number.

I/O Status Block Address (optional):

This word contains a pointer to the I/O status block which is a 2-word device dependent I/O completion data packet formatted as:

Byte 0

I/O status byte.

Byte 1

Augmented data supplied by the driver

Bytes 2 and 3

The contents of these bytes depend on the value of byte 0. If byte 0 = 1, then these bytes usually contain the processed byte count. If byte 0 does not equal zero, then the contents are device dependent.

AST Address (optional):

Address of an AST Service Routine.

Device Dependent Parameters:

Up to six parameters specific to the device and I/O function to be performed. Typically for data transfer functions these are:

Buffer address

Byte count

Carriage control type

Logical block number

Any optional parameters that are not specified should be filled with zeros.

5.4.2 Device Control Block

The device control block (DCB) defines generic information about a device type and the lowest and highest unit numbers. There is at least one DCB for each device type in a system. For example, if there are teletypes in a system, then there is at least one DCB with the device name 'TT'. If part of the teletypes were interfaced via DL11-A's and the remainder via a DH11, then there would be two DCB's. One for all DL11-A interfaced teletypes and one for all DH11 interfaced teletypes.

```
            ----------------------------------
D.LNK       ! LINK TO NEXT DCB  (0=LAST)     ! 0
            ----------------------------------
D.UCB       !         LINK TO FIRST UCB      ! 2
            ----------------------------------
D.NAM       !     GENERIC DEVICE NAME        ! 4
            ----------------------------------
D.UNIT      !HIGHEST UNIT #!LOWEST UNIT #    ! 6
            ----------------------------------
D.UCBL      !           LENGTH OF UCB        ! 10
            ----------------------------------
D.DSP       ! ADDR OF DRIVER DISPATCH TABLE  ! 12
            ----------------------------------
D.MSK       !     LEGAL FCN MSK BITS 0-15.   ! 14
            ----------------------------------
            !   CONTROL FCN MSK BITS 0-15.   ! 16
            ----------------------------------
            ! NO-OP'ED FCN MSK BITS 0-15.    ! 20
            ----------------------------------
            !     ACP FCN MSK BITS 0-15.     ! 24
            ----------------------------------
            ! LEGAL FCN MSK BITS 16.-32.     ! 26
            ----------------------------------
            ! CONTROL FCN MSK BITS 16.-32.   ! 30
            ----------------------------------
            ! NO-OP'ED FCN MSK BITS 16.-32.  ! 32
            ----------------------------------
            !       ACP FCN MSK BITS 16.-32. ! 34
            ----------------------------------
```

Figure 3-3 Device Control Block

5.4.2.1 DCB Details

D.LNK (Link to next DCB)*

----

* Parenthesized contents indicate value to be initialized in the data base source.

Description:

Address link to the next DCB. A zero in this field indicates the last DCB in the chain. The driver writer links his DCB into the system DCB's via the global label $USRTB on his first DCB.

Initialization/Access:

initialized, not referenced.

D.UCB (Pointer to First UCB)

Description:

Address link to the first and possibly the only UCB associated with the DCB. All UCB's, for a given DCB, are in contiguous memory locations and must all be the same length.

Initialization/Access:

initialized, not referenced.

D.NAM (ASCII Device Name)

Description:

Generic device name in ASCII by which device units are mnemonically referenced.

Initialization/Access:

initialized, not referenced.

D.UNIT (Unit Number Range)

Description:

Unit number range for the device. This range covers those logical units available to the user for device assignment. Typically, the lowest number is zero and the highest is n-1, where n is the number of device-units described by the DCB.

Initialization/Access:

initialized, not referenced.

D.UCBL (UCB Length)

Description:

The UCB may have any length to meet the needs of the I/A: DCB must have the same length.

Initialization/Access:

initialized, not referenced.

D.DSP (Dispatch Table Pointer)

Description:

Address of the driver dispatch table.

When the Executive wishes to enter the driver at any of the four entry points contained in the driver dispatch table, it accesses D.DSP, locates the appropriate address in the table, and calls the driver at that address. Thus, null addresses are not permitted. If the driver does not process a given function, then it simply returns. The driver writer must provide a driver dispatch table in the driver source. The label on this table is of the form $nnTBL and must be a global label. The designation nn is the 2-character generic device name for the device. Thus, $TTTBL is the global label on the driver dispatch table for the generic device name TT. This table is an ordered 4-word table containing the following entry points:

I/O Initiator;

Cancel I/O;

Device Timeout, and

Power failure.

When a driver is entered at one of these entry points, entry conditions are as follows:

At Initiator:

If UC.QUE=1
R5 = UCB address
R1 = Address of the I/O Packet

If UC.QUE=0
R5 = UCB address

Interrupts are allowed.

At Cancel I/O:

R5 = UCB address
R4 = SCB address
R3 = Controller index
R1 = Address of TCB of current task
R0 = Address of active I/O packet

Device interrupts are locked out.

At Device Timeout:

R5 = UCB address
R4 = SCB address
R3 = Controller index
RØ = I/O status code IE.DNR (Device Not Ready)

Device interrupts are locked out.

At Power Failure:

R5 = UCB address
R4 = SCB address
R3 = Controller index

Interrupts are allowed.

Initialization/Access:

initialized, not referenced.

D.MSK (Function Masks)

Description:

There are eight words beginning at D.MSK which are of critical importance to the proper functioning of a device driver. The Executive uses these words to validate and dispatch the I/O request specified by a QIO directive. The description which follows applies only to non-file structured devices, as directions for writing drivers for file structured devices (drivers which interface to FCP) are not included in this manual. Four masks, 2-words per mask, are described by the bit configurations established by the driver writer for these words.

1. Legal function mask;

2. Control function mask;

3. No-op'ed function mask, and

4. ACP function mask.

The QIO directive allows for 32 possible I/O functions. The masks, as stated, are filters to determine validity and I/O requirements for the subject driver.

The function value in the I/O request is filtered by the Executive through the four mask words. I/O function codes range from Ø-31. If the function corresponds to a true

condition in a mask word, a bit is set in the mask in the position which numerically corresponds to the function code. Thus, if the function 5 is legal, then bit 5 in the Legal Function Mask is set.

The masks are laid out in memory in two 4-word groups. Each 4-word group covers 16 function codes. The first four words cover the function codes 0-15., the second four words cover codes 16-31. Below is the exact layout used for the driver example in Chapter 5.

```
.WORD   140033      ;LEGAL FUNCTION MASK CODES 0-15.
.WORD   30          ;CONTROL FUNCTION MASK CODES 0-15.
.WORD   140000      ;NO-OP'ED FUNCTION MASK CODES 0-15.
.WORD   0           ;ACP FUNCTION MASK CODES 0-15.
.WORD   5           ;LEGAL FUNCTION MASK CODES 16.-31.
.WORD   0           ;CONTROL FUNCTION MASK CODES 16.-31.
.WORD   1           ;NO-OP'ED FUNCTION MASK CODES 16.-31.
.WORD   4           ;ACP FUNCTION MASK CODES 16.-31.
```

The mask words filter sequentially as follows:

Legal Function Mask:

Legal function values have the corresponding bit position in this word set to 1. Function codes that are not legal are rejected by QIO directive processing by returning IE.IFC in the I/O status block, provided an IOSB was specified.

Control Function Mask:

If any device dependent data exists in the DPB, and this data does not require further checking by the QIO directive processor, the function is considered in the class <control function>. Such a function allows QIO directive processing to copy the DPB device-dependent data directly into the I/O Packet.

No-op'ed Function Mask:

A no-op function is any function that is considered successful as soon as it is issued. If the function is a no-op, QIO directive processing immediately marks the request successful; no additional filtering occurs.

ACP Function Mask:

If a function code is legal, but neither control nor no-op, then it is either an ACP function or a transfer function. If a function code may require intervention of an Ancillary Control Processor (ACP), the corresponding bit in the ACP function mask must be set.

In the specific case of read/write virtual functions, the corresponding mask bits may be set at the driver writer's option. If the corresponding mask bits for a read/write virtual function are set, QIO directive processing will recognize that a file-oriented function is being requested to a non-file structured device and convert the request to a read/write logical function.

This conversion is particularly useful. Consider a read/write virtual function to a specific device:

1. If the device is file structured and a file is open on the specified LUN, the block number specified is converted from a virtual block number in the file to a logical block number on the medium and the request is queued to the driver as read/write logical.

2. If the device is file structured and no file is open on the lun, then an error is returned and no further action is taken.

3. If the device is not file structured then the request is simply transformed to read/write logical and queued to the driver. (Specified block number is unchanged).

Transfer Function Processing

Finally, if the function is not an ACP function, then by default, it is a transfer function. All transfer functions cause the QIO directive processor to check the specified buffer for legality (i.e., is it within the address space of the requesting task), and proper alignment (i.e., word or byte). Also, the number of bytes being transfered is checked for proper modulus (i.e., nonzero and a proper multiple).

Initialization/Access:

initialized, not referenced.

Mask Word Creation

The creation of the function mask words involves three steps:

1. Establish the I/O functions available on the device for which driver support is to be provided.

2. Check the standard RSX-11M function code values in table 3-1 for equivalencies. Only function code 0 is mandatory. Function codes 3 and 4, if used, must have the RSX-11M system interpretation. It is suggested that functions having an

RSX-11M system counterpart use the RSX-11M code, but this is not required except in the case where the device is to be used in conjunction with an ACP. From the supported function list, the two legal function mask words can be built.

3. Given the legal function mask,

3a. The Control Function mask is built by asking:

> Does this function carry a standard buffer address and byte count in the first two device dependent parameter words?

If it does not, then it either qualifies as a control function, or the driver itself must effect the checking and conversion of any addresses to the format required by the driver. (Buffer addresses in standard format are automatically converted to Address Doubleword format.)

Control functions are, essentially, those whose DPB's do not contain buffer addresses or counts.

3b. The No-op Function Mask is created by deciding which legal functions are to be no-op'ed. Typically, for File Control Services (FCS) compatibility on non-file structured devices, the file access/deaccess functions are selected as legal functions even though no specific action is required to access or deaccess a non-file structured device; thus, the access/deaccess functions are no-op'ed.

3c. Finally, the ACP functions Write Virtual Block and Read Virtual Block may be included. Other ACP functions that might be included fall into the non-conventional driver classification and are beyond the scope of this document.

3.1.2.2 I/O Function Codes - The filtering process which cascades through the function mask words in the DCB uses the function code byte supplied in the QIO directive DPB as the match value. Table 3-1 contains the function values used for DEC-supplied drivers.

TABLE 3-1
STANDARD I/O FUNCTION CODES

FUNCTION VALUE(8)	EQUATED SYMBOLIC	I/O FUNCTION
0	IO.KIL	CANCEL I/O
1	IO.WLB	WRITE LOGICAL BLOCK
2	IO.RLB	READ LOGICAL BLOCK
3	IO.ATT	ATTACH DEVICE
4	IO.DET	DETACH DEVICE
5		UNUSED
6		UNUSED
7		UNUSED
10		UNUSED
11	IO.FNA	FIND FILE IN DIRECTORY
12		UNUSED
13	IO.RNA	REMOVE FILE FROM DIRECTORY
14	IO.ENA	ENTER FILE IN DIRECTORY
15	IO.ACR	ACCESS FILE FOR READ
16	IO.ACW	ACCESS FILE FOR READ/WRITE
17	IO.ACE	ACCESS FILE FOR READ/WRITE/EXTEND
20	IO.DAC	DEACCESS FILE
21	IO.RVB	READ VIRTUAL BLOCK
22	IO.WVB	WRITE VIRTUAL BLOCK
23	IO.EXT	EXTEND FILE
24	IO.CRE	CREATE FILE
25	IO.DEL	MARK FILE FOR DELETE
26	IO.RAT	READ FILE ATTRIBUTES
27	IO.WAT	WRITE FILE ATTRIBUTES
30		UNUSED
31		UNUSED
32		UNUSED
33		UNUSED
34		UNUSED
35		UNUSED
36		UNUSED
37		UNUSED

Of the function code values listed in Table 3-1, only IO.KIL is mandatory and has a fixed interpretation. However, if IO.ATT and IO.DET are used, they must have the standard meaning. If QIO directive processing encounters a function code of 3 or 4 and the code is not no-op'ed, it will assume they represent Attach device and Detach device, respectively. The other codes are suggested but not mandatory. The driver writer is free to establish all other function code values on non-file structured devices. The mask words must obviously reflect the proper filtering process.

If a driver is being written for a file structured device, the standard function codes of Table 3-1 must be used.

5.4.3 Status Control Block

The status control block (SCB) defines the status of a device controller. There is one SCB for each controller is a system. The SCB is pointed to by unit control blocks. To expand on the teletype example above, each teletype interfaced via a DL11-A would have a SCB since each DL11-A is an independent interface unit. The teletypes interfaced via the DH11 would also have an SCB since the DH11 is a single controller but multiplexes many units in parallel.

```
S.LHD       ------------------------------------
            !         DEVICE I/O QUEUE         ! 0
            -----                          -----
            !            LISTHEAD              ! 2
S.PRI       ------------------------------------
S.VCT       !VECTOR ADDR/4 !DEVICE PRIORITY    ! 4
S.CTM       ------------------------------------
S.ITM       !INT TMEOUT CNT!CURNT TMOUT CNT    ! 6
S.CON       ------------------------------------
S.STS       ! CTRLR STATUS !CONTROLLER #*2     ! 10
            ------------------------------------
S.CSR       ! ADDRESS OF CONTROL STATUS REG    ! 12
            ------------------------------------
S.PKT       !ADDRESS OF CURRENT I/O PACKET     ! 14
            ------------------------------------
S.FRK       !          FORK LINK WORD          ! 16
            ------------------------------------
            !             FORK PC              ! 20
            ------------------------------------
            !             FORK R5              ! 22
            ------------------------------------
            !             FORK R4              ! 24
            ------------------------------------
```

Figure 3-4 Status Control Block

5.4.3.1 SCB Details

S.LHD (first word equals zero; second word points to first)

Description:

Two words which form the I/O queue listhead. The first word points to the first I/O Packet in the queue and the second word points to the last I/O Packet in the queue. If the queue is empty, the first word is zero and the second word points to the first word.

Initialization/Access:

initialized, not referenced.

S.PRI (device priority)

Description:

Contains the priority at which the device interrupts.

Initialization/Access:

initialized, not referenced.

S.VCT (interrupt vector divided by four)

Description:

Interrupt vector address divided by four.

Initialization/Access:

initialized, not referenced.

S.CTM (initialize to zero)

Description:

RSX-11M supports device timeout, which enables a driver to limit the time that elapses between the issuing of an I/O operation and its termination. The current timeout count (in seconds) is initialized by moving S.ITM (initial timeout count) into S.CTM. The Executive clock service will examine active times, decrement them and, if they reach 0, call the driver at its device timeout entry point.

The internal clock count is kept in 1-second increments. Thus, a time count of 1 is not precise, since the internal clocking mechanism is operating asynchronously with driver execution. The only meaningful minimum clock interval is 2 if the programmer intends to treat timeout as a consistently detectable error condition. Note, if the count is 0, that no timeout will occur; it is, in fact, an indication that timeout is not operative. The maximum count is 255. The driver writer is responsible for setting this field. Resetting is at actual timeout or within $FORK.

Initialization/Access:

not initialized, read/write.

S.ITM (set to initial timeout count)

Description:

Contains the initial timeout value.

Initialization/Access:

initialized, read-only.

S.CON (controller number times 2)

Description:

Controller number multiplied by 2. Used by drivers which are written to support more than one controller. S.CON may be used by the driver to index into a controller table created and maintained internal to the driver itself. Indexing the controller table enables the driver to service the correct controller when a device interrupts.

Initialization/Access:

initialized, read-only.

S.STS (initialize to zero)

Description:

Establishes the controller as busy/not busy. This byte is the interlock mechanism for marking a driver as busy for a specific controller. Tested and set by $GTPKT and reset by $IODON.

Initialization/Access:

initialized, not referenced.

S.CSR (Control Status Register address)

Description:

Contains the address of the Control Status Register (CSR) for the device controller. S.CSR is used by the driver to initiate I/O operations and to access, via indexing, other registers related to the device that are located in the I/O page. This address need not be a CSR; it need only be a member of the device's register set. It is accessed at system bootstrap time to determine if the interface exists on the system hosting the Executive. The Executive uses this to set the off-line bit at bootstrap so system software can be interchanged between systems without an intervening system generation. Otherwise, it is only accessed by the driver itself.

Initialization/Access:

initialized, read/only.

S.PKT (Reserve one word of storage)

Description:

Address of the current I/O Packet established by $GTPKT. This field is used to retrieve the I/O Packet address upon the completion of an I/O request.

Initialization/Access:

not initialized, read-only.

S.FRK (reserve four words of storage)

Description:

The four words starting at S.FRK are used for fork block storage if and when the driver deems it necessary to establish itself as a Fork process. Fork block storage preserves the state of the driver which is restored when the driver regains control at fork level. This area is automatically used if the driver calls $FORK.

Initialization/Access:

not initialized, not referenced.

5.4.4 Unit Control Block

The unit control block (UCB) defines the status of an individual device unit and is the control block that is pointed to by the first word of an assigned LUN. There is one UCB for each device unit of each DCB. The UCB's associated with a particular DCB are contingous in memory and are pointed to by the DCB. UCB's are variable length between dcb's but are of the same length for a specific DCB. To finish the teletype example above, each unit on both interfaces would have a UCB.

```
          --------------------------------
U.DCB     !     BACK POINTER TO DCB      ! 0
          --------------------------------
U.RED     !    REDIRECT POINTER TO UCB   ! 2
U.CTL     --------------------------------
U.STS     !CONTROL FLAGS ! UNIT STATUS   ! 4
U.ST2     --------------------------------
U.UNIT    ! STATUS EXT   !PHYSICAL UNIT #! 6
          --------------------------------
U.CW1     !     CHARACTERISTICS WORD #1  ! 10
          --------------------------------
U.CW2     !     CHARACTERISTICS WORD #2  ! 12
          --------------------------------
U.CW3     !     CHARACTERISTICS WORD #3  ! 14
          --------------------------------
U.CW4     !     CHARACTERISTICS WORD #4  ! 16
          --------------------------------
U.SCB     !     POINTER TO SCB           ! 20
          --------------------------------
U.ATT     !      TCB OF ATTACHED TASK    ! 22
          --------------------------------
U.BUF     !   BUFFER RELOCATION BIAS     ! 24
          --------------------------------
          !        BUFFER ADDRESS        ! 26
          --------------------------------
U.CNT     !         BYTE COUNT           ! 30
          --------------------------------
          !            DEVICE            !
          ----------           ----------
          !           DEPENDENT          !
          ----------           ----------
                         .
                         .
                         .
          ----------           ----------
          !           STORAGE            !
          --------------------------------
```

I did this section with indenting to match the other block descriptions.

5.4.4.1 UCB Details

U.DCB (pointer to associated DCB)

Description:

Backpointer to the corresponding DCB. Since the UCB is a key control block in the I/O data structure, access to other control blocks usually occurs via links implanted in the UCB.

Initialization/Access:

initialized, not referenced.

U.RED (initialized to point to U.DCB of the UCB)

Description:

Contains a pointer to the UCB to which this device unit has been redirected. this field is updated as the result of an MCR Redirect command. The redirect chain ends when this word points to the UCB itself.

Initialization/Access:

Initialized, not referenced.

U.CTL (set by driver writer)

Description:

U.CTL and the function mask words in the DCB drive QIO directive processing. The driver writer is totally responsible for setting up this bit pattern. Any inaccuracy in the bit setting of U.CTL will produce erroneous I/O processing. Bit symbols and their meaning are as follows:

UC.ALG = Alignment bit.

> If this bit = 0, then byte alignment of data buffers is allowed. If UC.ALG = 1, then buffers must be word aligned.

UC.ATT = Attach/Detach notification.

> If this bit is set, then the driver will be called when an Attach/Detach I/O function is processed by $GTPKT. Typically, the driver has no need to obtain control for Attach/Detach requests and the Executive performs the entire function without any assistance from the driver.

UC.KIL = Unconditional Cancel I/O call bit.

If set, the driver is to be called on a Cancel I/O request even if the unit specified is not busy. Typically, the driver is called on Cancel I/O only if an I/O operation is in progress.

UC.QUE - Queue bypass bit.

If set, the QIO directive processor is to call the driver prior to queueing the I/O Packet. Once gaining to-be-queued control, the disposition of the I/O Packet is the driver's responsibility. Typically, an I/O Packet is queued prior to a call to the driver, which later retrieves it by a call to $GTPKT.

UC.PWF - Unconditional call on power failure bit.

If set, the driver is always to be called when power is restored after a power failure occurs. Typically, the driver is called on power restoration only when an I/O operation is in progress.

UC.NPR - NPR device bit.

If set, the device is an NPR device. This bit determines the format of the two word address in U.BUF (details given under the discussion of U.BUF).

UC.LGH - Buffer size mask bits (2-bits).

These two bits are used to check if the byte count specified in an I/O request is a legal buffer modulus.

00 - any buffer modulus valid
01 - must have word alignment modulus
11 - must have double word alignment modulus
10 - combination invalid.

UC.ALG and UC.LGH are independent settings.

UC.ATT, UC.KIL, UC.QUE, and UC.PWF will usually be zero, especially for conventional drivers.

Every driver must, however, be concerned with its particular values for UC.ALG, UC.NPR, and UC.LGH.

The driver writer is totally responsible for the values in these bits, and erroneous values are likely to produce unpredictable results.

Initialization/Access:

initialized, not referenced.

U.STS (initialize to zero)

Description:

This byte contains device-independent status information. The bit meanings are as follows:

US.BSY - If set, device-unit is busy.

US.MNT - If set, volume is not MOUnted.

US.FOR - If set, volume is foreign

US.MDM - If set, device is marked for dismount.

The unused bits in U.STS are reserved for system use and expansion. US.MDM, US.MNT, and US.FOR apply only to MOUntable devices.

Initialization/Access:

Initialized, not referenced.

U.UNIT (unit number)

Description:

This byte contains the physical unit number of the device-unit. If the controller for the device supports only a single unit, the unit number is always zero.

Initialization/Access:

initialized, read-only.

U.ST2 (initialize to zero)

Description:

This byte contains additional device-independent status information. The bit meanings are as follows:

US.OFL - If set, the device is offline (that is, not in the configuration).

The remaining bits are reserved for system use and expansion.

Initialization/Access:

initialized, not referenced.

U.CW1 (set by driver writer)

Description:

The first of a 4-word contiguous cluster of device characteristics information. U.CW1 and U.CW4 are device independent. U.CW2 and U.CW3 are device dependent. The four characteristic words are retrieved from the UCB and placed in the requester's buffer on issuance of a GLUN$ Executive directive (Get LUN Information). It is the responsibility of the driver writer to supply the contents of these four words in the assembly source of the driver data structure.

U.CW1 is defined as follows:

```
DV.REC  Bit 0--Record-Oriented Device(1=yes)
DV.CCL  Bit 1--Carriage Control Device(1=yes)
DV.TTY  Bit 2--Terminal Device(1=yes)
DV.DIR  Bit 3--Directory Device(1=yes)
DV.SDI  Bit 4--Single Directory Device(1=yes)
DV.SQD  Bit 5--Sequential Device(1=yes)
DV.PSE  Bit 12--Pseudo Device(1=yes)
DV.COM  Bit 13--Device Mountable as a
                Communications Channel(1=yes)
DV.F11  Bit 14--Device mountable as a FILES-11
                device(1=Yes)
DV.MNT  Bit 15--Device mountable(1=yes)
```

Initialization/Access:

initialized, not referenced.

U.CW2 (initialize to zero)

Description:

Specific to a given device driver (available for working storage or constants).

Initialization/Access:

initialized, read/write.

U.CW3 (initialize to zero)

Description:

Specific to a given device driver (available for working storage or constants).

Initialization/Access:

initialized, read/write.

U.CW4
4 (set by driver writer)

Description:

Default buffer size.

Initialization/Access:

initialized, read-only.

U.SCB
B (SCB pointer)

Description:

This field contains a pointer to the SCB for this UCB. In general, R4 on entry to the driver via the driver dispatch table will contain the value in this word, since the SCB is frequently referenced by service routines.

Initialization/Access:

initialized, read-only.

U.ATT
TT (initialize to zero)

Description:

If a task has attached itself to a device-unit, this field contains its TCB address.

Initialization/Access:

initialized, not referenced

U.BUF
UF (reserve two words of storage)

Description:

U.BUF labels two consecutive words which serve as a communication region between $GTPKT and the driver. If a non transfer function is indicated, then U.BUF, U.BUF+2, and U.CNT receive the first three parameter words from the I/O Packet.

For transfer operations, the format of these two words depends on the setting of UC.NPR in U.CTL. The driver does not format the words; all formatting is completed prior to the driver receiving control. For unmapped systems, the first word is zero and the second word is the physical address of the buffer. For mapped systems, the format is determined by the UC.NPR bit which is set for an NPR device

This page seems to be missing

U.CNT (reserve one word of storage)

Description:

Contains the byte count of the buffer described by U.BUF. The driver will use this field in constructing the actual device request.

U.BUF and U.CNT are used to keep track of the current data item in the buffer for the current transfer (except for NPR transfers). Since this field is being altered dynamically, the I/O packet may be needed to reissue an I/O operation.

Initialization/Access:

not initialized, read/write.

Device Dependent Words:

Description:

The field is variable in length and is established by the driver writer to suit driver-specific requirements.

Initialization/Access:

not initialized, read/write.