The I/O initiation code is executed at priority 0 in system state. This means that no context switch can occur, no completion routines can run, and any traps to 4 and 10 cause a system fatal halt. All registers are available to use in this section. At the end of the section, control is returned to the monitor with an RTS PC. The I/O queue guarantees that transfers will be serialized. Because of this, RT-11 device handlers are not re-entrant. To minimize their size, they are not written as pure code and data segments.

```
000012  012727          MOV     #RKCNT,(PC)+    ;SET ERROR RETRIES
        000010
```

The MOV statement above sets the number of error retries to 8 and moves that value to RETRY:. (The (PC)+ notation points to RETRY:.) At this point, the handler knows that it has a brand new queue element, and that a retry is not in progress.

```
000016  000000  RETRY:  0                       ;HIGH ORDER BIT USED FOR
                                                ;RESET IN PROGRESS FLAG
```

If bit 15 of the word at RETRY: is 1 (that is, if the word is negative), then a retry is in progress.

```
000020  016705          MOV     RKCQE,R5        ;GET Q PARAMETER POINTER
```

RKCQE points to the block number Q.BLKN in the I/O queue element.

```
        177764
000024  011502          MOV     @R5,R2          ;R2 = BLOCK NUMBER
000026  016504          MOV     2(R5),R4        ;R4 = UNIT NUMBER
        000002                                  [The controller requires
                                                the unit number in the top
                                                three bits of the word
                                                loaded into RKDA.]
000032  006204          ASR     R4              ;ISOLATE UNIT BITS IN
                                                ;HIGH 3 BITS
000034  006204          ASR     R4
000036  006204          ASR     R4
000040  000304          SWAB    R4
000042  042704          BIC     #^C<160000>,R4
        017777
000046  000404          BR      2$              ;ENTER COMPUTATION LOOP
```

The device unit number and block number are known; the disk address for a read or write request must be calculated. Once calculated, the disk address is stored in DISKAD in case it must be used

Figure C-10 RK05 Handler Listing(Cont.)

again during retries. The RK disk has 12 blocks per track, and two tracks per cylinder. To find the disk address, the block number is divided by 12, and the quotient and remainder are separated.

```
000050  060204  1$:     ADD     R2,R4           ;ADD 16R TO ADDRESS
000052  006202          ASR     R2              ;R2 = 8R
000054  006202          ASR     R2              ;R2 = 4R
000056  060302          ADD     R3,R2           ;R2 = 4R+S = NEW N
000060  010203  2$:     MOV     R2,R3           ;R3 = N = 16R+S
000062  042703          BIC     #177760,R3      ;R3 = S
        177760
000066  040302          BIC     R3,R2           ;R2 = 16R
000070  001367          BNE     1$              ;LOOP IF R <> 0
000072  022703          CMP     #12.,R3         ;IF S < 12.
        000014
000076  003002          BGT     3$              ;THEN F(S) = S
000100  062703          ADD     #4,R3           ;ELSE F(S)=F(12+S')=16+S'=4+S
        000004
000104  060304  3$:     ADD     R3,R4           ;R4 NOW CONTAINS RK ADDRESS
000106  010467          MOV     R4,DISKAD       ;SAVE DISK ADDRESS
```

The disk address is saved in DISKAD. The significance of the bits in DISKAD, from high order to low order, is as follows: unit, cylinder, track, sector. The next statement points R5 to a queue element, since perhaps this is a retry and R5 is not already set up.

```
        000016
000112  016705  AGAIN:  MOV     RKCQE,R5        ;POINT R5 TO Q ELEMENT
        177672
000116  012703          MOV     #103,R3         ;ASSUME A WRITE
        000103
```

The operation code for a write with interrupt enabled is 103. This information is in the PDP-11 Peripherals Handbook.

```
000122  012704          MOV     #RKDA,R4        ;POINT TO DISK ADDRESS REG
        177412
000126  012714          MOV     (PC)+,@R4       ;PUT IN ADDRESS  UNIT SELECT
```

In the statement above, (PC)+ refers to DISKAD:.

```
000130  000000  DISKAD: 0                       ;SAVED COMPUTED DISK ADDRESS
```

The following statement adds 4 to R5, so that R5 points to Q.BUFF in the queue element.

Figure C-10 RK05 Handler Listing (Cont.)

```
78 000132  022525            CMP    (R5)+,(R5)+     ;ADVANCE TO BUFFER ADDRESS
                                                    ;IN Q ELEMENT
79                     .IFT
80                           MOV    (R5)+,-(R4)     ;PUT IN BUFFER ADDRESS
81                     .IFF
82 000134  004777            JSR    PC,@$MPPTR      ;CONVERT TO PHYSICAL ADDR
```

In the line above, $MPPTR is a pointer to the monitor routine $MPPHY. See Section 1.4.4.5 of this manual for information on the $MPPHY monitor routine. This routine is available for NPR device handlers to use. It converts the virtual buffer address supplied in the queue element into an 18-bit physical address that is returned on the stack. Section 1.4.4.5 explains how to use the routine, and lists the calling conventions, required inputs, and the outputs of the routine.

The monitor supplies the virtual address in two words: Q.PAR and Q.BUFF. This form is used because it can be directly used by character-oriented (non-NPR) devices. NPR devices such as the RK must convert this pair of words into an 18-bit physical address consisting of a 16-bit low part and a two-bit extension part. The extension bits are in positions 4 and 5 for use with UNIBUS controllers. The routine $MPPHY is called through the pointer $MPPTR to do this address conversion. The extension bits must be ORed into the command word being built for RKCS (see statement number 93, below).

```
           000370
83 000140  012644            MOV    (SP)+,-(R4)     ;MOVE LO 16 BITS INTO PLACE
84                     .IFTF
```

The next statement moves the word count Q.WCNT from the queue element into RKWC, the device word count register. (Note that Q.WCNT is a word count.) If the device is character oriented, the word count must be shifted left to change it to a byte count (the same as multiplying it by 2). RT-11 can transfer up to 32767 words per operation. However, it can never transfer an odd number of bytes.

```
85 000142  012544            MOV    (R5)+,-(R4)     ;PUT IN WORD COUNT
86 000144  001406            BEQ    7$              ;0 COUNT => SEEK
87 000146  100402            BMI    5$              ;NEGATIVE => WRITE
```

The RK controller requires that all word counts be negative.

```
88 000150  005414            NEG    @R4             ;POSITIVE => READ.
                                                    ;FIX FOR CONTROLLER
```

Figure C-10 RK05 Handler Listing (Cont.)

```
                 ;       ADD     #2,R3           ;START UP A READ
```

The statement above was replaced by the following statement as a result of a source patch to the V03B handler source file. The following statement converts a write operation code to a read operation code by adding 2 to it. The operation code 105 is for a read operation with interrupt enabled.

```
000152  122323           CMPB    (R3)+,(R3)+     ;CHANGE COMMAND CODE TO READ
000154           5$:
                 .IFF
```

The following operation is necessary for the creation of an 18-bit physical address. The 2-bit extension must be ORed into the command word being built for RKCS.

```
000154  052603           BIS     (SP)+,R3        ;SET IN HI ORDER ADDRESS BITS
                 .IFTF
```

The next statement starts the operation, whatever it is, by moving the operation code to RKCS, the device control and status register.

```
000156  010344   6$:     MOV     R3,-(R4)        ;START THE OPERATION
```

The next statement returns control to the monitor. The I/O transfer continues concurrently.

```
000160  000207           RTS     PC              ;AWAIT INTERRUPT
```

The next statement is reached if the operation is a seek. The operation code for a seek with interrupt enabled is 111.

```
000162  012703   7$:     MOV     #111,R3         ;START UP A SEEK
        000111
                 .IFF
000166  005016           CLR     (SP)            ;NO HI ORDER MEMORY ADDRESS
                                                 ;ON SEEK
                 .IFTF
000170  000771           BR      5$              ;AWAIT INTERRUPT
```

Figure C-10 RK05 Handler Listing (Cont.)

****

The handler Asynchronous Trap Entry Section begins here.

****

The following code is reached when an interrupt occurs.

```
                    ; ASYNCHRONOUS TRAP ENTRY POINT TABLE

                    .NLIST CND
    000172                  .DRAST  RK,5
```

The .DRAST macro generates the following block of code (up to the next .LIST CND directive):

```
                    .GLOBL  $INPTR
    000172  000207  RTS     %7
```

The abort entry point is the word preceding RKINT:. Since no abort entry point was specified in the .DRAST macro, above, RTS PC was generated.

Disks are always allowed to complete an I/O transfer attempt. Aborting them in the middle of an operation is not necessary, and can possibly corrupt the disk. It is not practical to try to stop a disk during an I/O transfer. So, abort requests are ignored by doing an RTS PC. (In contrast, see the corresponding section of the PC handler in Section C.5 of this appendix. The PC handler has an abort entry point because the paper tape reader or punch must be stopped to abort an I/O transfer.)

```
    000174  004577  RKINT:: JSR     %5,@$INPTR
            000344
    000200  000100          .WORD   ^C<5*^040>^0340

                    .LIST CND

```

If the handler is for a system device, the bootstrap fills in vector 220 and the pointers to the fixed offsets in the Resident Monitor. (The bootstrap also relocates the pointers, which are actually set up by defining the values at assembly time.) Otherwise, the information is filled in when the handler is made resident by .FETCH or LOAD.

At interrupt time, the new PC (RKINT:) and new PS (340) are used. The handler calls the monitor through $INPTR in the handler to

Figure C-10 RK05 Handler Listing (Cont.)

$INTEN in the monitor. The monitor lowers priority from 7 to 5, switches to system state, and calls the handler back.

```
                                ; INTERRUPT ENTRY POINT
```

The monitor calls the handler back at this point. Execution is at priority 5 and is in system state. The hardware has now finished the I/O operation, and the handler must determine if the transfer was successful or if there was an error.

```
000202  012705      MOV     #RKER,R5        ;POINT TO ERROR STATUS
                                            ;REGISTER
        177402
000206  012504      MOV     (R5)+,R4        ;SAVE ERRORS IN R4,
                                            ;POINT TO RKCS
```

The value of RETRY is negative if a drive reset was just done. (Bit 15 is the retry flag.)

```
000210  005767      TST     RETRY           ;WERE WE DOING A DRIVE RESET?
        177602
000214  100013      BPL     NORMAL          ;NO-NORMAL OPERATION
000216  005715      TST     @R5             ;YES-ANY ERROR?
```

Bit 15 of RKCS is the error summary bit. If there was an error during a drive reset, it is handled in the same way as an error that occurred during an I/O transfer.

```
000220  100411      BMI     NORMAL          ;YES-HANDLE NORMALLY
```

R5 points to RKCS, the device control and status register.

```
000222  032715      BIT     #20000,@R5      ;RESET COMPLETE?
        020000
```

The RK device interrupts twice during a drive reset. The first interrupt should be ignored.

```
000226  001474      BEQ     RTSPC           ;NO-DISMISS INTERRUPT-RK11
                                            ;WILL INTERRUPT AGAIN
                                            ;WHEN RESET COMPLETE
```

Figure C-10 RK05 Handler Listing (Cont.)

The .FORK macro causes the code that follows it to be executed at priority 0 after all interrupts have been serviced, but before any jobs or their completion routines execute. This avoids executing lengthy code in the handler at high processor priority.

```
119 000230                     .FORK   RKFBLK          ;DO  RETRIES AT
                                                       ;FORK LEVEL
    000230  004577             JSR     %5,@$FKPTR
            000312
    000234  000244             .WORD   RKFBLK - .      [PIC]


120 000236  105067  RKRETR: CLRB    RETRY+1         ;YES-CLEAR RESET FLAG
            177555
121 000242  000723             BR      AGAIN           ;AND RETRY OPERATION AT
                                                       ;FORK LEVEL
122
123 000244  022527  NORMAL: CMP     @R5,#310        ;IS THIS FIRST OF TWO
                                                       ;INTERRUPTS CAUSED BY SEEK?
            000310
```

The RK device interrupts twice for a seek operation. The first interrupt should be ignored by the handler. The seek is complete after the second interrupt has occurred.

```
124 000250  001463             BEQ     RTSPC           ;YES-IGNORE IT.RK WILL
                                                       ;INTERRUPT AGAIN
125                                                    ;WHEN SEEK COMPLETE
```

The next statement is reached when I/O is complete or when there is an I/O error. The sign bit (bit 15) of RKCS, the device control and status register, is an error summary bit. If RKCS is negative, there was an error in the I/O transfer.

```
126 000252  005715             TST     @R5             ;ANY ERRORS?
127 000254  100067             BPL     DONE            ;NO-OPERATION COMPLETE
```

The errors are processed at fork level, priority 0.

```
128 000256                     .FORK   RKFBLK          ;PROCESS ERRORS AT FORK LEVEL.
    000256  004577             JSR     %5,@$FKPTR
            000264
    000262  000216             .WORD   RKFBLK - .      [PIC]
```

Figure C-10 RK05 Handler Listing (Cont.)

ADDITIONAL I/O INFORMATION

The following block of code (up to the next .ENDC statement) is generated if the system supports error logging:

```
                                .IF NE ERL$G
```

Register 4 contains errors from RKER, the device error register. Unrecoverable errors that do not indicate hardware faults are not logged.

```
000264  032704          BIT     #62340,R4       ;TEST FOR USER TYPE ERRORS
        062340
000270  001031          BNE     RKERR           ;DON'T LOG THEM
                                                ;SOFT ERROR.
```

The other types of errors are logged:

```
000272  010705          MOV     PC,R5           ;GET ADDRESS TO SAVE
                                                ;SAVE REGISTERS
000274  062705          ADD     #RKRBUF-.,R5    [PIC]
        000214
000300  010502          MOV     R5,R2           ;SAVE ADDRESS IN R2 FOR EL
000302  012703          MOV     #RK$CSR,R3      ;R3 = ADDRESS OF
                                                ;REGISTER TO READ
        177400
000306  012704          MOV     #RKNREG,R4      ;R4 = # OF REGISTERS TO READ
        000007
000312  012325  RKRREG: MOV     (R3)+,(R5)+     ;MOVE REGISTERS TO BUFFER
000314  005304          DEC     R4              ;TEST IF DONE
000316  001375          BNE     RKRREG          ;NO
000320  012703          MOV     #RKNREG,R3      ;R3 = # OF REGISTERS
                                                ;IN LOW BYTE
        000007
000324  062703          ADD     #RKRCNT,R3      ;R3 = TOTAL RETRY COUNT
                                                ;IN HIGH BYTE
        004000
000330  016705          MOV     RKCQE,R5        ;POINT R5 AT 3RD WORD OF Q.
        177454
000334  116704          MOVB    RETRY,R4        ;SET R4=0 IN HIGH BYTE
                                                ;FOR FORK ID
        177456
                                                ;AND RETRY COUNT IN LOW BYTE
000340  005304          DEC     R4              ;RETRY COUNT VALUE AFTER
                                                ;IT IS DECREMENTED
000342  004777          JSR     PC,@$ELPTR      ;CALL ERROR LOGGER.
        000172
000346  012705          MOV     #RKER,R5        ;RESET R5,R4 ON RETURN.
        177402
000352  012504          MOV     (R5)+,R4
                .ENDC
```

Figure C-10 RK05 Handler Listing (Cont.)

The next section of code retries both soft (such as checksum) and hard (hardware malfunction) errors. R5 points to RKCS, the device control and status register.

```
000354  012715  RKERR:  MOV     #1,@R5          ;YES-RESET CONTROL
        000001
```

When the controller is ready, it sets bit 7 of the low byte of RKCS.

```
000360  105715  3$:     TSTB    @R5             ;WAIT
000362  100376          BPL     3$              [loop until ready]
000364  105367          DECB    RETRY           ;DECREASE RETRY COUNT
        177426
000370  001414          BEQ     HERROR          ;NONE LEFT-HARD ERROR
000372  032704          BIT     #110000,R4      ;SEEK INCOMPLETE OR
                                                ;DRIVE ERROR?
        110000
```

Both seek incomplete and drive error require a drive reset before the operation can be retried.

```
                                                ; 100000=DRIVE ERROR
                                                ; 010000=SEEK ERROR
```

Common errors for which the I/O transfer operation should be retried are checksum errors, data late errors, and timing errors.

```
000376  001717          BEQ     RKRETR          ;NO-RETRY OPERATION
```

The next statement is reached if there is a seek incomplete or drive error condition. RKDA was cleared by the controller reset above, but the disk address is saved in DISKAD. The operation code for a drive reset with interrupt enabled is 115.

```
000400  016737          MOV     DISKAD,@#RKDA   ;YES-RESELECT DRIVE
        177524
        177412
000406  012715          MOV     #115,@R5        ;START A DRIVE RESET
        000115
```

Figure C-10 RK05 Handler Listing (Cont.)

The flag in RETRY is set here so that on the next pass the handler will know that a drive reset, and not an I/O transfer, was the last operaton done.

```
000412  052767          BIS     #100000,RETRY   ;SET FLAG
        100000
        177376
```

The next statement returns control to the monitor to wait for the drive reset or seek to finish.

```
000420  000207  RTSPC:  RTS     PC              ;AWAIT INTERRUPT
```

The next statement is reached when there has been an I/O error that has been retried and could not be corrected.

```
000422  016705  HERROR: MOV     RKCQE,R5        ;GET POINTER TO Q ELEMENT
        177362
```

The handler reports the error to the user program by setting bit 0 (the hard error bit) in the channel status word. R5 points to Q.BLKN; R5, decremented by 2, points to the address of the channel status word.

```
000426  052755          BIS     #1,@-(R5)       ;GIVE OUR USER AN
                                                ;ERROR IN CHANNEL
        000001
                .IF NE  ERL$G
000432  000411          BR      RKEXIT          ;HARD ERROR,BR TO EXIT.
```

The following section is reached after a successful transfer. Successful transfers are logged at fork level, priority 0.

```
000434          DONE:   .FORK   RKFBLK          ;CALL ERROR LOG AT FORK
                                                ;LEVEL FOR SUCCESS
000434  004577          JSR     %5,@$FKPTR
        000106
000440  000040          .WORD   RKFBLK - .
000442  012704          MOV     #RKIDS,R4       ;SUCCESSFUL I/O,SET R4=0
                                                ;IN HIGH BYTE FOR RK,
        000377
```

Figure C-10 RK05 Handler Listing (Cont.)

```
                                                   ;-1 IN LOW BYTE FOR SUCESS.
    000446  016705          MOV     RKCQE,R5       ;POINT R5 AT 3RD WORD OF Q.
            177336
    000452  004777          JSR     PC,@$ELPTR     ;CALL ERROR LOGGER.
            000062
                                                   ;ON RETURN EXIT.
                    .IFF
                    DONE:                          [If no error logging]
                    .ENDC
    000456  005067  RKEXIT: CLR     RETRY          ;CLEAR ANY FLAGS
            177334
```

****

The handler I/O Completion Section begins here.

****

```
                    .NLIST CND
    000462                  .DRFIN  RK             ;EXIT TO COMPLETION
```

The .DRFIN macro generates the next block of code (up to the next .LIST CND directive). This section lets the monitor know that the I/O operation is complete so that the queue element can be returned to the free element list. Control returns to the monitor with the JMP statement. The monitor alerts the program if it was waiting for this transfer to finish, or it runs the program's completion routine, if any.

```
                    .GLOBL  RKCQE
    000462  010704          MOV     %7,%4
    000464  062704          ADD     #RKCQE-.,%4    [PIC]
                                                   [Point to address of CQE]
            177324
    000470  013705          MOV     @#^054,%5      [Base of RMON]
            000054
    000474  000175          JMP     @^0270(5)      [Fixed offset in RMON]
                                                   [Go to I/O completion code
                                                   in the monitor]
            000270
                    .LIST CND

                    .ENDC

    000500  000000  RKFBLK: .WORD   0,0,0,0        ;FORK QUEUE BLOCK
    000502  000000
    000504  000000
    000506  000000
```

Figure C-10 RK05 Handler Listing (Cont.)

```
                  .IF NE ERL$G
    000510        RKRBUF: .BLKW   RKNREG          ;ERROR LOG STORAGE
                                                  ;FOR REGISTERS
                  .ENDC

```

****

The handler Termination Section begins here.

****

The .DREND macro generates the block of code up to the .LIST CND directive.

```
                  .NLIST CND
    000526                .DREND  RK
          000000  ...V2=0
          000002  ...V2=...V2+2.
```

If the handler is for a system device, the bootstrap fills in the following table of pointers. Otherwise, it is filled in when the handler is made resident by .FETCH or by LOAD. The pointers are to fixed offsets in the Resident Monitor. Some of the following pointers are optional, and their assembly depends on which system conditionals are defined. See Section C.4 of this appendix for a more detailed explanation of the .DREND macro.

```
    000526  000000  $RLPTR:: .WORD  0
    000530  000000  $MPPTR:: .WORD  0
    000532  000000  $GTBYT:: .WORD  0
    000534  000000  $PTBYT:: .WORD  0
    000536  000000  $PTWRD:: .WORD  0
            000003  ...V2=...V2+1
    000540  000000  $ELPTR:: .WORD  0
            000007  ...V2=...V2+4.
    000542  000000  $TIMIT:: .WORD  0
    000544  000000  $INPTR:: .WORD  0
    000546  000000  $FKPTR:: .WORD  0
                    .GLOBL  RKSTRT
            000550' RKEND == .
    000060          .ASECT
            000060  .=60
    000060  000007          .WORD    ...V2 [Summary of SYSGEN options]
    000550          .CSECT                 [Return to unnamed .PSECT]
                    .LIST CND


            000001  .END
```

Figure C-10 RK05 Handler Listing (Cont.)

The symbol table is generated at the end of the assembly listing.

```
SYMBOL TABLE

AGAIN    000112R     Q.PAR = 000016       RKRBUF   000510R
DISKAD   000130R     Q.UNIT= 000007       RKRCNT= 004000
DLSYS  = 000000 G    Q.WCNT= 000012       RKRETR   000236R
DMSYS  = 000000 G    RETRY    000016R     RKRREG   000312R
DONE     000434R     RFSYS = 000000 G     RKSTRT   000000RG
DPSYS  = 000000 G    RKBA   = 177410      RKSTS  = 100000
DSSYS  = 000000 G    RKCNT  = 000010      RKSYS    000006RG
DTSYS  = 000000 G    RKCQE    000010RG    RKWC   = 177406
DXSYS  = 000000 G    RKCS   = 177404      RK$CSR= 177400 G
DYSYS  = 000000 G    RKDA   = 177412      RK$VEC= 000220 G
ERL$G  = 000001      RKDS   = 177400      RTSPC    000420R
HERROR   000422R     RKDSIZ= 011300       TIM$IT= 000001
MMG$T  = 000001      RKEND  = 000550RG    $ELPTR   000540RG
NORMAL   000244R     RKER   = 177402      $FKPTR   000546RG
Q.BLKN= 000004       RKERR    000354R     $GTBYT   000532RG
Q.BUFF= 000010       RKEXIT   000456R     $INPTR   000544RG
Q.COMP= 000014       RKFBLK   000500R     $MPPTR   000530RG
Q.CSW  = 000002      RKIDEN= 000000       $PTBYT   000534RG
Q.ELGH= 000024       RKIDS  = 000377      $PTWRD   000536RG
Q.FUNC= 000006       RKINT    000174RG    $RLPTR   000526RG
Q.JNUM= 000007       RKLQE    000006RG    $TIMIT   000542RG
Q.LINK= 000000       RKNREG= 000007       ...V2  = 000007

. ABS.   000062      000
         000550      001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  1248 WORDS  ( 5 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  71 PAGES
,RK.LST/L:ME:MEB:TTM=RKCND.MAC,RK.MAC
```

Figure C-10 RK05 Handler Listing (Cont.)

C.4 System Device Handlers

The monitor and device handlers reside on the system device. The device must be block-replaceable (random access), and have read/write capability. Writing a device handler for a system device requires very little extra work once the basic device handler is written. (The RK handler in Section C.3 is a good example of a random access device handler.) The programmer simply defines the symbol $SYSDV. The system macros then expand properly, generating all the required code for a system device handler.

C.4.1 Assembling A System Device Handler

The following list shows the steps required to assemble a device handler as a system device handler.

1. The file SYCND.MAC must be edited to set the symbol $xxSYS to 1. For the RK handler, for example, the statement is as follows:

   ```
   $RKSYS = 1
   ```

2. The file SYSDEV.MAC must be included in the assembly. This file contains the single line:

   ```
   $SYSDV = 1
   ```

3. The handler, called MYFILE in this example, should be assembled together with the three system files, as shown:

   ```
   MACRO/LIST xx+SYCND+SYSDEV+MYFILE/OBJECT
   ```

   In the line above, xx represents SJ, FB, or XM. The correct macro source file for the corresponding monitor should be used. The resulting object file is MYFILE.OBJ.

   (To assemble a handler as a data device only, the SYSDEV file should be omitted.)

C.4.2 System Device Handler Requirements

The following list outlines the special requirements for a system device handler. These requirements are filled automatically by the system macros .DRBEG, .DRAST, .DRFIN, and .DREND.

1. Entry points of all current system devices (except for this handler) must be referenced in a global statement, and all must be equated to 0.

2. The handler size must be global, and must be called $SYHSZ.

3. The handler entry point must be tagged xxSYS (xx represents the device name). It must also be global. The xxSYS label is provided by the .DRBEG macro.

4. The handler must be a .PSECT named SYSHND. This .PSECT is defined by the .DRBEG macro.

5. The handler must terminate with a table of pointers to monitor routines. These global routine names are resolved when the handler is linked to the monitor, instead of being filled in by the fetch code at load time. The conditionals that are defined for the handler must match the conditionals defined for the monitor. The .DREND macro provides the table of pointers.

C.4.3 The .DRBEG and .DREND Macros

Figure C-11 shows the .DRBEG and .DREND macros. Appendix B of this manual provides complete listings of all the system macros. In Figure C-11, black ink is used for text and comments. Red ink is used for the actual source listing of the macro files.

```
.MACRO  .DRBEG  NAME,VEC,DSIZ,DSTS,VTBL

.IF NDF $SYSDV          If the handler is not for a system device, the lines
                        up to the .IFF statement are assembled.
.ASECT
. = 52
.GLOBL  NAME'END        This is global so that the handler can be broken
                        into two separately assembled modules.
                        (The RT-11 magtape handler is an example.)
                        .DRBEG can be put in the first module, and
                        .DREND can be put in the last module.

        .WORD   <NAME'END - NAME'STRT>
        .WORD   DSIZ
        .WORD   DSTS
.CSECT
.IFF                    If the handler is for a system device, the next two
                        lines are assembled.

$SYDSZ == DSIZ          This is global because it gets linked into the USR
                        for use by the .DSTATUS request.

.PSECT  SYSHND          The .PSECT is named SYSHND for the system handler.
.ENDC
NAME'STRT::
.IF B VTBL
.GLOBL  NAME'INT        This is for a device with a single vector.

        .WORD   VEC
        .WORD   NAME'INT - .
.IFF
.GLOBL  VTBL,NAME'INT   This is for a device with more than one vector.

        .WORD   <VTBL-.>/2. -1 + ^0100000
        .WORD   NAME'INT - .
.ENDC
        .WORD   ^0340
NAME'SYS::              This is used only by a system handler.
NAME'LQE::      .WORD   0
NAME'CQE::      .WORD   0
.ENDM


.MACRO  .DREND  NAME
...V2=0                 This bit mask is an accumulation of SYSGEN options.
                        As each option is defined, a bit is added to this
                        word.

.IF NE MMG$T            (For XM handler)
...V2=...V2+2
```

Figure C-11 The .DRBEG and .DREND Macros

```
.IF DF $SYSDV
.GLOBL  $RELOC,$MPPHY,$GETBYT,$PUTBYT,$PUTWRD

$RLPTR:: .WORD  $RELOC  These pointers are for use in XM only. The system
$MPPTR:: .WORD  $MPPHY  handler must have this table with these names.
$GTBYT:: .WORD  $GETBYT The boot relocates the pointers appropriately.
$PTBYT:: .WORD  $PUTBYT
$PTWRD:: .WORD  $PUTWRD
.IFF
$RLPTR:: .WORD  0       Handlers for nonsystem devices do not need names
$MPPTR:: .WORD  0       in this table because the .FETCH code sets them
$GTBYT:: .WORD  0       up when the handler is made resident.
$PTBYT:: .WORD  0
$PTWRD:: .WORD  0
.ENDC
.ENDC                   (End of XM conditional)
.IF NE ERL$G
...V2=...V2+1
.IF DF $SYSDV
.GLOBL  $ERLOG
$ELPTR:: .WORD  $ERLOG  Pointer for error logging for system devices.
.IFF
$ELPTR:: .WORD  0       Pointer for error logging for nonsystem devices.
.ENDC
.ENDC
.IF NE TIM$IT
...V2=...V2+4
.IF DF $SYSDV
.GLOBL  $TIMIO
$TIMIT:: .WORD  $TIMIO  Pointer for time-out support for system devices.
.IFF
$TIMIT:: .WORD  0       Pointer for time-out support for nonsystem devices.
.ENDC
.ENDC
.IF DF $SYSDV
.GLOBL  $FORK,$INTEN
$INPTR: .WORD   $INTEN  Pointers for system devices.
$FKPTR:: .WORD  $FORK
.IFF
$INPTR:: .WORD  0       Pointers for nonsystem devices.
$FKPTR:: .WORD  0
.IFTF
.GLOBL  NAME'STRT       These globals allow the handler to be broken into
                        modules.
NAME'END == .
.IFT
$SYHSZ == NAME'END - NAME'STRT  This must be in all system handlers.
                                It defines the size of the handler in bytes.
```

Figure C-11 The .DRBEG and .DREND Macros (Cont.)

```
.IFF
.ASECT
.=60
        .WORD   ...V2      This is the SYSGEN options word. It is placed in
                           location 60 in block 0 of the handler. It must
                           match the SYSGEN fixed offset in RMON. It is used
                           for nonsystem handlers only.
.CSECT
.ENDC
.ENDM
```

Figure C-11 The .DBREG and .DREND Macros (Cont.)

C.5 Study of the PC Handler

Figure C-12 provides detailed comments on a listing of the PC handler. The comments do not duplicate those in the RK handler example; comments are provided only for those features that are different in the PC handler, such as multi-vectored format. Figure C-12 illustrates handler techniques for a serial, character-oriented (non-NPR) device with two vectors. The PC handler can be used for the paper tape reader alone as well as for the combined paper tape reader and punch devices.

In Figure C-12, black ink is used for text and comments. Red ink is used for the actual device handler assembly listing.

```
PC    V03.01  MACRO V03.02B12-SEP-78 15:29:52 PAGE 1

                   ;CONDITIONAL FILE FOR PC HANDLER EXAMPLE
                   ;
                   ;PCCND.MAC
                   ;
                   ;9/1/78 JAD
                   ;
                   ;ASSEMBLE WITH PC.MAC TO TURN ON 18-BIT I/O,
                   ;TIME-OUT SUPPORT, AND ERROR LOGGING FOR
                   ;PC HANDLER
                   ;
            000001 MMG$T   = 1            ;TURN ON 18-BIT I/O
            000001 ERL$G   = 1            ;TURN ON ERROR LOGGING
            000001 TIM$IT  = 1            ;TURN ON TIME-OUT SUPPORT

PC    V03.01  MACRO V03.02B12-SEP-78 15:29:52 PAGE 2

                   .TITLE PC   V03.01
                   .IDENT  /V03.01/
                   ; RT-11 HIGH SPEED PAPER TAPE PUNCH AND READER (PC11) HANDLER
                   ;
                   ; COPYRIGHT (C) 1978
                   ;
                   ; DIGITAL EQUIPMENT CORPORATION
                   ; MAYNARD, MASSACHUSETTS 01754
                   ;
                   ; THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY
                   ; ON A SINGLE COMPUTER SYSTEM AND MAY BE COPIED ONLY WITH
                   ; THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS
                   ; SOFTWARE, OR ANY OTHER COPIES THEREOF, MAY NOT BE
                   ; PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER
                   ; PERSON EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO
                   ; AGREES TO THESE LICENSE TERMS. TITLE TO AND OWNERSHIP
                   ; OF THE SOFTWARE SHALL AT ALL TIMES REMAIN IN DEC.
                   ;
                   ; THE INFORMATION IN THIS SOFTWARE 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.
```

Figure C-12 PC Handler Listing

```
PC   V03.01  MACRO V03.02B12-SEP-78 15:29:52 PAGE 3
```

```
*************************************************
```

The device handler Preamble Section starts here.

```
*************************************************
```

```
                  .MCALL  .DRBEG,.FORK,.DREND,.DRAST,.DRFIN,.QELDF

                  .IIF NDF PR11$X, PR11$X=0          [0=punch and reader;
                                                      1=reader only]
                  .IIF NDF MMG$T, MMG$T=0
                  .IIF NDF ERL$G, ERL$G=0
                  .IIF NDF TIM$IT, TIM$IT=0

                  .NLIST CND
000000            .QELDF
          000000  Q.LINK=0
          000002  Q.CSW=2.
          000004  Q.BLKN=4.
          000006  Q.FUNC=6.
          000007  Q.JNUM=7.
          000007  Q.UNIT=7.
          000010  Q.BUFF=^010
          000012  Q.WCNT=^012
          000014  Q.COMP=^014
          000016  Q.PAR=^016
          000024  Q.ELGH=^024
                  .LIST CND
```

The following three lines are commonly used offsets in the queue element:

```
          177776  CSTAT   = Q.CSW-Q.BLKN
          000006  BYTCNT  = Q.WCNT-Q.BLKN
          000004  BUFF    = Q.BUFF-Q.BLKN

                  ; PAPER TAPE PUNCH CONTROL REGISTERS
          000074  .IIF NDF PC$VEC, PP$VEC == 74          ;PUNCH VECTOR ADDR
                  .IIF NDF PP$CSR, PP$CSR == 177554      ;PUNCH CONTROL
                                                         ;REGISTER
          177556  PPB     = PP$CSR+2               ;PUNCH DATA BUFFER

          000000  PRDSIZ  = 0                      ;PP DEVICE SIZE (0 => NON-
                                                   ;FILE STRUCTURED)
                  .IF EQ PR11$X
          000007  PRSTS   = 7                      ;PP-PR DEVICE STATUS WORD
                  .IFF
                  PRSTS   = 40007                  ;READER ONLY
```

Figure C-12 PC Handler Listing (Cont.)

```
                .ENDC

                ; PAPER TAPE READER CONTROL REGISTERS
                .IIF NDF PR$CSR, PR$CSR == 177550        ;CONTROL REGISTER
        177552  PRB       == PR$CSR+2                     ;DATA REGISTER
                .IIF NDF PR$VEC, PR$VEC == 70            ;READER VECTOR ADDR

        000001  PRGO      = 1                        ;READER ENABLE BIT
        000101  PINT      = 101                      ;INTERRUPT ENABLE BIT
                                                     ;AND GO BIT

                ; CONSTANTS FOR MONITOR COMMUNICATION
        000001  HDERR     = 1                        ;HARD ERROR BIT  [for CSW]
        020000  EOF       = 20000                    ;END OF FILE BIT  [for CSW]
```

****

The device handler Header Section begins here.

****

```
                ; LOAD POINT

                .IF EQ PR11$X                        [If both reader and punch:]
                .NLIST CND
  000000                .DRBEG   PR,PR$VEC,PRDSIZ,PRSTS,PRTAB
```

PRTAB in the line above is the vector table.

```
000000          .ASECT
        000052  . = 52
                .GLOBL  PREND
000052  000334          .WORD   <PREND - PRSTRT>
000054  000000          .WORD   PRDSIZ
000056  000007          .WORD   PRSTS
000000          .CSECT
000000          PRSTRT::
                .GLOBL  PRTAB,PRINT
```

This references the table for a multi-vectored device:

```
000000  100034          .WORD   <PRTAB-.>/2. -1 + ^O100000
000002  000160          .WORD   PRINT - .
000004  000340          .WORD   ^O340
000006          PRSYS::
000006  000000  PRLQE:: .WORD   0
000010  000000  PRCQE:: .WORD   0
```

Figure C-12 PC Handler Listing (Cont.)

```
                  .LIST CND
                  .IFF                                 [If reader only:]
                  .NLIST CND
                      .DRBEG  PR,PR$VEC,PRDSIZ,PRSTS
                  .LIST CND
                  .ENDC

```

****

The device handler I/O Initiation Section begins here.

****

```
                  ; ENTRY POINT

   000012  016704 PP:     MOV     PRCQE,R4        ;R4 POINTS TO CURRENT Q ENTRY
           177772
   000016  006364         ASL     BYTCNT(R4)      ;WORD COUNT TO BYTE COUNT
           000006
   000022  103007         BCC     PR              ;BRANCH => READ
```

The routine for the punch:

```
                  .IF EQ PR11$X                      [Both reader and punch:]
   000024  012767         MOV     #PP$CSR,PRCSR   ;SAVE CSR FOR ABORT.
           177554
           000256
   000032  052737         BIS     #100,@#PP$CSR   ;CAUSES INTERRUPT,
                                                  ;STARTING TRANSFER
           000100
           177554
   000040  000207         RTS     PC
                  .IFF                               [Reader only:]
                          BR      PPERR           ;NO PUNCH,ERROR.
                  .ENDC

```

The routine for the reader:

```
   000042  001505 PR:     BEQ     PRDONE          ;A REQUEST FOR 0 BYTES IS
                                                  ;A SEEK, EXIT.
```

Even though a seek is not a reasonable operation for paper tape, the handler provides for it as part of RT-11's device independence.

Figure C-12 PC Handler Listing (Cont.)

```
000044  012767          MOV     #PR$CSR,PRCSR   ;SAVE CSR FOR ABORT.
        177550
        000236
000052  005737          TST     @#PR$CSR        ;IS READER READY?
        177550
000056  100060          BPL     PRGORD          ;YES, START TRANSFER
000060  052754          BIS     #EOF,@-(R4)     ;IMMEDIATE EOF IF NOT READY
        020000
000064  000474          BR      PRDONE          ;SET EOF BIT,
                                                ;COMPLETE OPERATION

                ; PUNCH-READER VECTOR TABLE

000066          PRTAB:
                .IF EQ PR11$X
000066  000070          .WORD   PR$VEC          ;READER VECTOR
000070  000072          .WORD   PRINT-.         ;READER ISR OFFSET
000072  000340          .WORD   340             ;STATUS
000074  000074          .WORD   PP$VEC          ;PUNCH VECTOR
000076  000000          .WORD   PPINT-.         ;PUNCH ISR OFFSET
000100  000340          .WORD   340             ;STATUS
000102  000000          .WORD   0               ;END OF TABLE
                .ENDC

```

****

The device handler Asynchronous Trap Entry Section begins here.

****

```
                ; PUNCH INTERRUPT SERVICE

                .IF EQ PR11$X
                .NLIST CND
000104                  .DRAST  PP,4,PRDONE
```

PRDONE is the abort entry point. An abort can be requested by any of the following means: typing double CTRL/C, issuing the .HRESET programmed request, any type of I/O error, traps to 4 and 10, and any other condition that causes a MON-F- type of fatal error message to appear. In the event that an abort is requested, is necessary to stop the device. This is not necessary for a disk, but it is important for a character-oriented device like paper tape, in order to prevent a tape runaway condition.

```
                .GLOBL  $INPTR
000104  000464  BR      PRDONE
```

Figure C-12 PC Handler Listing (Cont.)

```
 000106  004577  PPINT:: JSR     %5,@$INPTR
         000216
 000112  000140          .WORD   ^C<4*^040>^0340
                 .LIST CND
 000114  016704          MOV     PRCQE,R4          ;R4 POINTS TO CURRENT Q ENTRY
         177670
 000120  005737          TST     @#PP$CSR          ;ERROR?
```

Bit 15 in PP$CSR is the error bit. The possible errors for paper tape devices are device out of tape, and tape jammed.

```
         177554
 000124  100412          BMI     PPERR             ;YES-PUNCH OUT OF PAPER
 000126  005764          TST     BYTCNT(R4)        ;ANY MORE CHARS TO OUTPUT?
```

The transfer is done if the required number of bytes is transferred without error.

```
         000006
 000132  001451          BEQ     PRDONE            ;NO-TRANSFER DONE
 000134  005264          INC     BYTCNT(R4)        ;DECREMENT BYTE COUNT
                                                   ;(IT IS NEGATIVE)
         000006
                 .IF EQ MMG$T
                         MOVB    @BUFF(R4),@#PPB   ;PUNCH CHARACTER
                         INC     BUFF(R4)          ;BUMP POINTER
                 .IFF
```

$GTBYT is a pointer to the monitor $GETBYT routine. See Section 1.4.4.5 of this manual for a description of the routine.

```
 000140  004777          JSR     PC,@$GTBYT        ;GET A BYTE FROM USER BUFFER
         000152
 000144  112637          MOVB    (SP)+,@#PPB       ;PUNCH IT
         177556
                 .ENDC
 000150  000207          RTS     PC
                 .ENDC
```

Character-oriented devices should check for disabling conditions, such as no power on device or no tape in reader or punch, and set the hard error bit (bit 0) in the channel status word.

Figure C-12 PC Handler Listing (Cont.)

```
 000152  052754  PPERR:  BIS     #HDERR,@-(R4)     ;SET HARD ERROR BIT
         000001
 000156  000437          BR      PRDONE            ;GO TO I/O COMPLETION

                 ; READER INTERRUPT SERVICE

                 .NLIST CND
 000160                  .DRAST  PR,4,PRDONE       ;DEFINE AST ENTRY POINTS
                 .GLOBL  $INPTR
 000160  000436  BR      PRDONE
 000162  004577  PRINT:: JSR     %5,@$INPTR
         000142
 000166  000140          .WORD   ^C<4*^O40>^O340
                 .LIST CND
 000170  016704          MOV     PRCQE,R4          ;R4 POINTS TO Q ENTRY
         177614
                 .IF  EQ MMG$T
                         ADD     #BUFF,R4          ;POINT R4 TO BUFFER ADDRESS
                 .ENDC
 000174  005737          TST     @#PR$CSR          ;ANY ERRORS?
         177550
 000200  100413          BMI     PREOF             ;YES-TREAT AS EOF
                 .IF  EQ MMG$T
                         MOVB    @#PRB,@(R4)       ;PUT CHAR IN BUFFER
                         INC     (R4)+             ;BUMP BUFFER POINTER
                         DEC     @R4               ;DECREASE BYTE COUNT
                 .IFF
 000202  113746          MOVB    @#PRB,-(SP)       ;GET A CHARACTER
         177552
 000206  004777          JSR     PC,@$PTBYT        ;MOVE IT TO USERS BUFFER
         000105
 000212  005364          DEC     BYTCNT(R4)        ;DECREASE BYTE COUNT
         000005
                 .ENDC
 000216  001417          BEQ     PRDONE            ;IF ZERO,WE ARE DONE
 000220  052737  PRGORD: BIS     #PINT,@#PR$CSR    ;ENABLE READER INTERRUPT,
                                                   ;GET A CHARACTER
         000101
         177550
 000226  000207          RTS     PC
```

Stop the device if there are errors or if the end of tape is reached:

```
 000230  005037  PREOF:  CLR     @#PR$CSR          ;DISABLE INTERRUPTS
         177550
 000234                  .FORK   PRFBLK            ;REQUEST SYSTEM PROCESS
 000234  004577          JSR     %5,@$FKPTR
         000072
 000240  000040          .WORD   PRFBLK - .
                 .IF EQ MMG$T
```

Figure C-12 PC Handler Listing (Cont.)

For character-oriented devices, it is necessary to clear the remainder of the user's buffer when end of file is reached (if CTRL/Z is typed on the console terminal, if there is no tape in the reader, etc.). The handler sets the EOF bit in the channel status word the next time the handler is called to do a transfer. This convention makes character-oriented devices appear the same as random access devices, and is in keeping with the RT-11 device independence philosophy.

```
                       PRE01:  CLRB    @(R4)              ;CLEAR REMAINDER OF BUFFER
                               INC     (R4)               ;BUMP BUFFER ADDRESS.
                               DEC     BYTCNT-BUFF(R4)    ;TEST IF DONE.
                               BNE     PRE01              ;BRANCH IF MORE.
                       .IFF
    000242   005046    PRE01:  CLR     -(SP)
    000244   004777            JSR     PC,@$PTBYT         ;CLEAR A BYTE IN USER BUFFER
             000050
    000250   005364            DEC     BYTCNT(R4)         ;DECREMENT BYTE COUNT
             000006
    000254   001372            BNE     PRE01              ;BR IF MORE
                       .ENDC

                       ; OPERATION COMPLETE
```

If the operation is complete or if it cannot complete because of an error, it is necessary to turn off the device:

```
    000256   005077    PRDONE: CLR     @PRCSR             ;TURN OFF THE READER/PUNCH
                                                          ;INTERRUPT
             000026
                                                          ;IN CASE WE GET AN ERROR LATER
                       .NLIST CND
```

****

The handler I/O Completion Section begins here.

****

```
    000262             PRFIN:  .DRFIN  PR                 ;GO TO I/O COMPLETION
                       .GLOBL  PRCQE
    000262   010704            MOV     %7,%4
    000264   062704            ADD     #PRCQE-.,%4
             177524
    000270   013705            MOV     @#^054,%5
             000054
    000274   000175            JMP     @^0270(5)
             000270
```

Figure C-12 PC Handler Listing (Cont.)

```
                      .LIST CND
    000300   000000   PRFBLK: .WORD   0,0,0,0         ;FORK QUEUE BLOCK
    000302   000000
    000304   000000
    000306   000000
    000310   000000   PRCSR:  .WORD   0               ;ADDRESS OF DEVICE TO STOP.
                      .NLIST CND
```

****

The handler Termination Section begins here.

****

```
    000312                    .DREND  PR
             000000   ...V2=0
             000002   ...V2=...V2+2.
    000312   000000   $RLPTR:: .WORD  0
    000314   000000   $MPPTR:: .WORD  0
    000316   000000   $GTBYT:: .WORD  0
    000320   000000   $PTBYT:: .WORD  0
    000322   000000   $PTWRD:: .WORD  0
             000003   ...V2=...V2+1
    000324   000000   $ELPTR:: .WORD  0
             000007   ...V2=...V2+4.
    000326   000000   $TIMIT:: .WORD  0
    000330   000000   $INPTR:: .WORD  0
    000332   000000   $FKPTR:: .WORD  0
                      .GLOBL  PRSTRT
             000334'  PREND == .
    000060            .ASECT
             000060   .=60
    000060   000007           .WORD   ...V2
    000334            .CSECT
                      .LIST CND

             000001   .END
```

Figure C-12 PC Handler Listing (Cont.)

```
SYMBOL TABLE

BUFF   = 000004       PREND = 000334RG        Q.CSW = 000002
BYTCNT= 000006        PREOF   000230R         Q.ELGH= 000024
CSTAT = 177776        PREO1   000242R         Q.FUNC= 000006
EOF    = 020000       PRFBLK  000300R         Q.JNUM= 000007
ERL$G = 000001        PRFIN   000262R         Q.LINK= 000000
HDERR = 000001        PRGO   = 000001         Q.PAR = 000016
MMG$T = 000001        PRGORD  000220R         Q.UNIT= 000007
PINT   = 000101       PRINT   000162RG        Q.WCNT= 000012
PP       000012R      PRLQE   000006RG        TIM$IT= 000001
PPB    = 177556       PRSTRT  000000RG        $ELPTR  000324RG
PPERR    000152R      PRSTS = 000007          $FKPTR  000332RG
PPINT    000106RG     PRSYS   000006RG        $GTBYT  000316RG
PP$CSR= 177554 G      PRTAB   000066RG        $INPTR  000330RG
PP$VEC= 000074 G      PR$CSR= 177550 G        $MPPTR  000314RG
PR       000042R      PR$VEC= 000070 G        $PTBYT  000320RG
PRB    = 177552 G     PR11$X= 000000          $PTWRD  000322RG
PRCQE    000010RG     Q.BLKN= 000004          $RLPTR  000312RG
PRCSR    000310R      Q.BUFF= 000010          $TIMIT  000326RG
PRDONE   000256R      Q.COMP= 000014          ...V2 = 000007
PRDSIZ= 000000

. ABS.   000062     000
         000334     001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  1276 WORDS  ( 5 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  71 PAGES
,PC.LST/L:ME:MEB:TTM=PCCND.MAC,PC.MAC
```

Figure C-12 PC Handler Listing (Cont.)

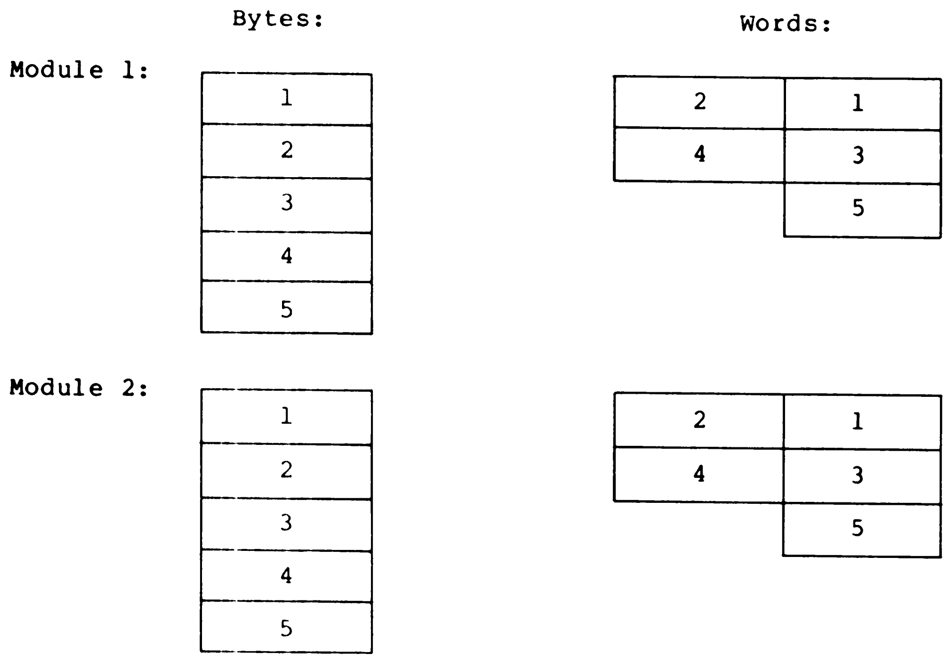
C.6 RT-11 File Formats

C.6.1 Object File Format (OBJ)

An object module is a file containing a program or routine in a binary, relocatable form. Object files normally have an .OBJ file type. In a MACRO program, one module is defined as the unit of code enclosed by the .TITLE and .END pair of MACRO directives. The module name is taken from the .TITLE statement. Object modules are produced by language processors, such as MACRO and FORTRAN, and are processed by the linker to become runnable programs (in SAV, LDA, or REL format, discussed later). Object files can also be processed by the librarian to produce library OBJ files, which are then used by the linker.

Many different object modules can be combined to form one file. Each object module remains complete and independent. However, object modules combined into a library by the librarian are no longer independent. They are concatenated and become part of the library's structure. The modules are concatenated by byte rather than by word in order to save space. For example, suppose a library is to consist of two modules and the first module contains an odd number of bytes. The second module is added to the library behind the first module. The first byte of the second module is positioned as the high order byte of the last word of the first module. The result of this procedure is that one byte is saved in the library.

To understand byte concatenation, it is most helpful to think of the modules as a stream of bytes, rather than as a stream of 2-byte words. Figure C-13 shows how two 5-byte modules would be concatenated. Module 1 and module 2 are shown both as bytes and as words.



Concatenated modules, Module 1 followed by Module 2:

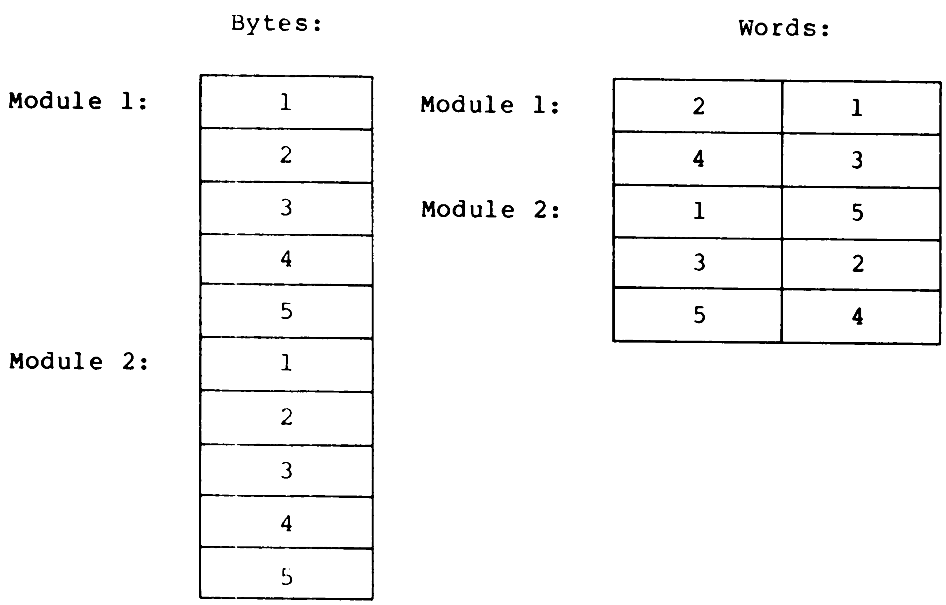


Figure C-13 Modules Concatenated by Byte

If RT-11 is to begin execution of a program within a particular object module of a program, the information on where to start is given as the transfer address. The first even transfer address encountered by the linker is passed to RT-11 as the program's start address. Whenever the resulting program is executed the start address is used to indicate the first executable instruction. If no transfer address is given (if, for example, none is specified with the .END directive in a MACRO program) or if all are odd, the resulting program does not self-start when run.

Object modules are made up of formatted binary blocks. A formatted binary block is a sequence of 8-bit bytes (stored in an RT-11 file, on paper tape, or by some other means) and is arranged as shown in Figure C-14.

Byte containing octal value 1
Byte containing octal value 0
Low order byte of length
High order byte of length
Data bytes
Checksum byte

Figure C-14 Formatted Binary Format

Each formatted binary block has its length stored within it. The length includes all bytes of the block except the checksum byte. The data portion of each formatted binary block contains the actual object module information. The checksum byte is the negative of the sum of all preceding bytes. Formatted binary blocks may be separated by a variable number of null (0) bytes.

If the first two bytes of a formatted binary block (the 1 and 0 bytes) are discarded, and if the checksum byte is discarded, the remainder of the block is compatible with RSX-11M formatted binary blocks. The length bytes indicate the length of the RSX binary block. RT-11 formatted binary blocks are a proper subset of the RSX binary blocks. See Appendix B, "Task Builder Data Formats", in the RSX-11M Task Builder Reference Manual, order number AA-2588D-TC, for detailed information on the many types of formatted binary blocks.

C.6.2 Library File Format (OBJ and MAC)

A library file contains concatenated modules and some additional information. RT-11 supports both object and macro libraries. Object libraries usually have an .OBJ file type; macro libraries usually have a .MAC file type. The modules in a library file are preceded by a Library Header Block and Library Directory, and are followed by the Library End Block, or trailer. Figure C-15 shows the format of a library file.

Library Header
Directory
Concatenated modules (starts on a block boundary)
Library End Trailer Block

Figure C-15 Library File Format

Diagrams of each component in the library file structure are included here. See Chapter 12 of the RT-11 System User's Guide for information on using the librarian.

C.6.2.1 **Library Header Format** - The library header describes the status of the file. There is a different header for object libraries and for macro libraries. The contents of the object library header are shown in Figure C-16. The contents of the macro library header are shown in Figure C-17.

All numeric values shown are octal. The date and time, which are in standard RT-11 format, are the date and time the library was created. This information is displayed when the library is listed.

Offset	Contents	Description
0	1	Library header block code
2	42	
4	7	Librarian code
6	305	Library version number
10	0	Reserved
12		Date in RT-11 format (0 if none)
14		Time expressed in two words
16		
20	0	Reserved
22	0	Reserved
24	0	Reserved
26	10	Directory relative start address
30		Number of bytes in directory
32	0	Reserved
34		Next insert relative block number
36		Next byte within block
40		Directory starts here

Figure C-16 Object Library Header Format

Offset	Contents	Description
0	1001	Library type and ID code
2	305	Library version number
4	0	Reserved
6		Date in RT-11 format (0 if none)
10		Time expressed in two words
12		
14	0	Reserved
16	0	Reserved
20	0	Reserved
22	0	Reserved
24	0	Reserved
26	0	Reserved
30	0	Reserved
32	10	Size of directory entries
34		Directory starting relative block number
36		Number of directory entries allocated (default is 200)
40		Number of directory entries available

Figure C-17 Macro Library Header Format

C.6.2.2 **Library Directories** - There are two kinds of library directories. For object libraries, the directory is an Entry Point Table (EPT). For macro libraries, the directory is a Macro Name Table (MNT).

The directory (see Figure C-18) is composed of 4-word entries that contain information related to all modules in the library file. Note that if the librarian /N option is used for object libraries to include module names, bit 15 of the relative block number word is set to 1. If the librarian is invoked with the keyboard monitor LIBRARY command, module names are never included.

Symbol characters 1-3 (Radix 50)	
Symbol characters 4-6 (Radix 50)	
(bit 15) Block number relative to start of file	
Reserved (7 bits)	Relative byte in block (9 bits)

Figure C-18 Library Directory Format

In the library directory, the symbol characters represent the entry point or macro name. The relative byte maximum is 777 (octal).

The object library directory starts on the first word after the library header, word 40 (octal). The object library directory is only long enough to accomodate the exact number of modules in the library. Space for the object library directory is not pre-allocated. The directory is kept in memory during Librarian operations, and the amount of available memory is the only limiting factor on the maximum size of the directory. Reserved locations, those not used by the directory, are zero-filled. Modules follow the directory. They are stored beginning in the next block after the directory.

The macro library directory starts on a block boundary, relative block 1 of the library file. Its size is pre-allocated. The default size is two blocks. This can be changed by the Librarian /M option. Unused entries in the directory are filled with -1. Macro files are stored starting on the block boundary after the directory. This is relative block 3 of the library file if the default directory size is used.

Modules in libraries are concatenated by byte. (See Figure C-13 for an example of byte concatenation.) This means that a module can start on an odd address. When this occurs, the linker shifts the module to an even address at link time.

C.6.2.3 **Library End Block Format** - Following all modules in the library is a specially coded Library End Block, or trailer, which signifies the end of the file (see Figure C-19).

	1	Data block header
	10	Data block length
	10	Library End Block code
	0	Reserved, must be 0
	357	Checksum byte

Figure C-19 Library End Block Format

C.6.3 Absolute Binary File Format (LDA)

The linker /L option, or the keyboard monitor LINK command /LDA option, produces output files in a paper tape compatible binary format.

Paper tape format, shown in Figure C-20, is a sequence of data blocks. Each block represents the data to be loaded into a specific portion of memory. The data portion of each block consists of the absolute load address of the block, followed by the absolute data bytes to be loaded into memory beginning at the load address. There can be as many data blocks as necessary in an LDA file. The last block of the file is special: it contains only the program start address, or transfer address, in its data portion. If this address is even, the Absolute Loader passes control to the loaded program at this address. If it is odd (that is, if the program has no transfer address, or the transfer address was specified as a byte boundary), the loader halts upon completion of loading. The final block of the LDA file is recognized by the fact that its length is 6 bytes.

First data block:

1	
0	
BCL	Low order 8 bits of byte count
BCH	High order 8 bits of byte count
ADL	Low order byte of absolute load address of data bytes in the block
ADH	High order byte of load address
Data bytes . . .	
Checksum byte	

Intermediate data blocks:

1	
0	This pattern is repeated for all intermediate blocks
BCL	
BCH	
ADL	
ADH	
Data bytes . . .	
Checksum byte	

Last data block:

1	
0	
6	
0	
JL	Low byte of start address, or odd number
JH	High byte of start address, or odd number
Checksum byte	

Figure C-20 Absolute Binary Format (LDA)

LDA format files are used for down-line loading of programs, for loading stand-alone application programs, and as input to special programs that put code into ROM (Read-Only Memory). The usual procedure for loading a program that will execute in a stand-alone environment is as follows:

1. Toggle the BIN loader into memory.
2. Load the Absolute Loader into memory.
3. Load the LDA file into memory and begin execution.

LSI computer systems have console microcode that makes steps 1 and 2 above unnecessary.

The load module's data blocks contain only absolute binary load data and absolute load addresses. All global references have been resolved and the linker has performed the appropriate relocation.

C.6.4 Save Image File Format (SAV)

Save image format is used for programs that are to be run in the SJ environment, or in the background in the FB and XM environments. Save image files normally have a .SAV file type. This format is essentially an image of the program as it would appear in memory. (Block 0 of the file corresponds to memory locations 0-776, block 1 to locations 1000-1776, and so forth.) See Table C-2 for the contents of block 0. See also Section 11.5.2 of the RT-11 System User's Guide for more information on the load modules created by the linker.

Table C-2
Information in Block 0

Offset	Contents
0	Reserved
2	Reserved
4	Reserved
6	Reserved
10	Reserved
12	Reserved
14	XM BPT trap (XM only)
16	XM BPT trap (XM only)
20	XM IOT trap (XM only)
22	XM IOT trap (XM only)
24	Reserved

(continued on next page)

Table C-2 (Cont.)
Information in Block 0

Offset	Contents
26	Reserved
30	Reserved
32	Reserved
34	Trap vector (TRAP)
36	Trap vector (TRAP)
40	Program's relative start address
42	Initial location of stack pointer (changed by /M option)
44	Job status word
46	USR swap address
50	Program's high limit
52	Size of program's root segment, in bytes (used for REL files only)
54	Stack size, in bytes (changed by /R option) (used for REL files only)
56	Size of overlay region, in bytes (0 if not overlaid) (used for REL files only)
60	REL file ID ("REL" in Radix-50) (used for REL files only)
62	Relative block number for start of relocation information (used for REL files only)
64	Reserved
66	Reserved
.	Reserved
.	Reserved
.	Reserved
360-377	Bitmap area

Locations 360-377 in block 0 of the file are restricted for use by the system. The linker stores the program memory usage bits in these eight words, which are called a bitmap. Each bit represents one 256-word block of memory and is set if the program occupies any part of that block of memory. Bit 7 of byte 360 corresponds to locations 0 through 777; bit 6 of byte 360 corresponds to locations 1000 through 1777, and so on. This information is used by the monitor when loading the program.

The keyboard monitor commands R and RUN cause a program stored in a SAV file to be loaded and started. (The RUN command is actually a combination of the GET and START commands.) First, the Keyboard Monitor reads block 0 of the SAV file into an internal USR buffer. It extracts information from locations 40-64 and 360-377 (the bitmap, described above). Using the protection bitmap (called LOWMAP) which resides in RMON, KMON checks each word in block 0 of the file. Locations that are protected, such as location 54 and the device interrupt vectors, are not loaded. The locations that are not protected are loaded into memory from the USR buffer. Next, KMON sets location 50 to the top of usable memory, or to the top of the user program, whichever is greater.

If the RUN command (or the GET command) was issued, KMON checks the bitmap from locations 360-377 of the SAV file. For each bit that is set, the corresponding block of the SAV file is loaded into memory. However, if KMON is in memory space that the program needs to use, KMON puts the block of the SAV file into a USR buffer, and then moves it to the file SWAP.SYS.

Finally, when it is time to begin execution of the program, KMON transfer control to RMON. The parts of the program, if any, that are stored in SWAP.SYS are read into memory where they overlay KMON and possibly the USR. If the R command was issued, KMON does not check the bitmap to see which blocks of the SAV file to load. Instead, it jumps to RMON and attempts to read all locations above 1000 into memory. (The R command does not use SWAP.SYS.) The monitor keeps track of the fact that KMON and USR are swapped out, and execution of the program begins.

C.6.5 **Relocatable File Format (REL)**

A foreground job is linked using the linker /R option or the keyboard monitor LINK command with the /FOREGROUND option. This causes the linker to produce output in a linked, relocatable format, with a .REL file type.

The object modules used to create a REL file are linked as if they were a background SAV image, with a base of 1000. This permits users to use .ASECT directives to store information in locations 0 through 777 in REL files. All global references have been resolved. The REL file is not relocated at link time; relocation information is included to be used at FRUN time. The relocation information in the file is used to determine which words in the program must be relocated when the job is installed in memory.

There are two types of REL files to consider: those programs with overlay segments, and those without them.

C.6.5.1 **REL Files without Overlays** - A REL file for a program without overlays appears as shown in Figure C-21.

Block 0	Program text	Relocation information

Figure C-21 REL File Without Overlays

Block 0 (relative to the start of the file) contains the information shown in Table C-2. Some of this information is used by the FRUN processor.

In the case of a program without overlays, the FRUN processor performs the following general steps to install a foreground job.

1. Block 0 of the file is read into an internal monitor buffer.

2. The amount of memory required for the job is obtained from location 52 of block 0 of the file, and the space in memory is allocated by moving KMON and the USR down.

3. The program text is read into the allocated space.

4. The relocation information is read into an internal buffer.

5. The locations indicated in the relocation information area are relocated by adding or subtracting the relocation quantity. This quantity is the starting address the job occupies in memory, adjusted by the relocation base of the file. REL files are linked with a base of 1000.

The relocation information consists of a list of addresses relative to the start of the user's program. The monitor scans the list. For each relative address in the list, the monitor computes an actual address. That address is then loaded with its original contents plus or minus the relocation constant. The relocation information is shown in Figure C-22.

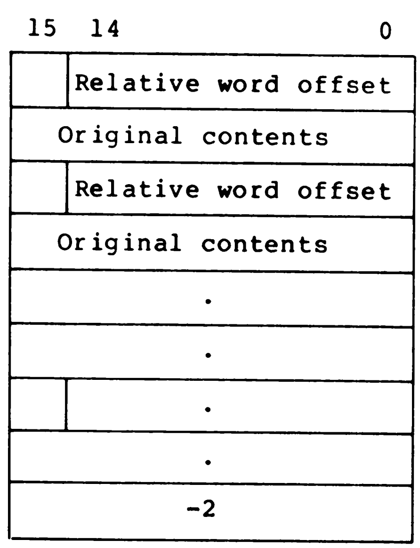


Figure C-22 Relocation Information Format

In Figure C-22, bits 0-14 represent the relative address to relocate divided by 2. This implies that relocation is always done on a word boundary, which is indeed the case. Bit 15 is used to indicate the type of relocation to perform, positive or negative. The relocation constant (which is the load address of the program) is added to or subtracted from the indicated location depending on the sense of bit 15; 0 implies addition, while 1 implies subtraction. The value 177776, or -2, terminates the list of relocation information. The original contents is a full 16-bit word.

C.6.5.2 **REL Files with Overlays** - When overlays are included in a program, the file is similar to that of a nonoverlaid program. However, in addition to the root segment, the overlay segments must also be relocated. Since overlays are not permanently memory resident but are read in from the file as needed, they require an additional operation. FRUN relocates each overlay segment and rewrites it into the file before the program begins execution. Thus, when the overlay is called into memory during program execution, it is correct. This process takes place each time an overlaid file is run with FRUN. The relocation information for overlay files contains both the list of addresses to be modified and the original contents of each location. This allows the file to be executed again after the first usage. It is necessary to preserve the original contents in case some change has occurred in the operating environment. Examples of these changes include using a different monitor version, running on a system with a different amount of memory, and having a different set of device handlers resident in memory. Figure C-23 shows a REL file with overlays.

In the case of a REL file with overlays, location 56 of block 0 of the REL file contains the size in bytes of the overlay region. This size is added to the size of the program base segment (in location 52) to allocate space for the job.

After the program base (root) code has been relocated, each existing overlay is read into the program overlay region in memory, relocated using the overlay relocation information, and then written back into the file.

The root relocation information section is terminated with a -1. This -1 is also an indication that an overlay segment relocation block follows.

The relocation is relative to the start of the program and is interpreted the same as in the file without overlays. (That is, bit 15 indicates the type of relocation, and the displacement is the true displacement divided by 2). Encountering -1 indicates that a new overlay region begins here. A -2 indicates the termination of all relocation information.

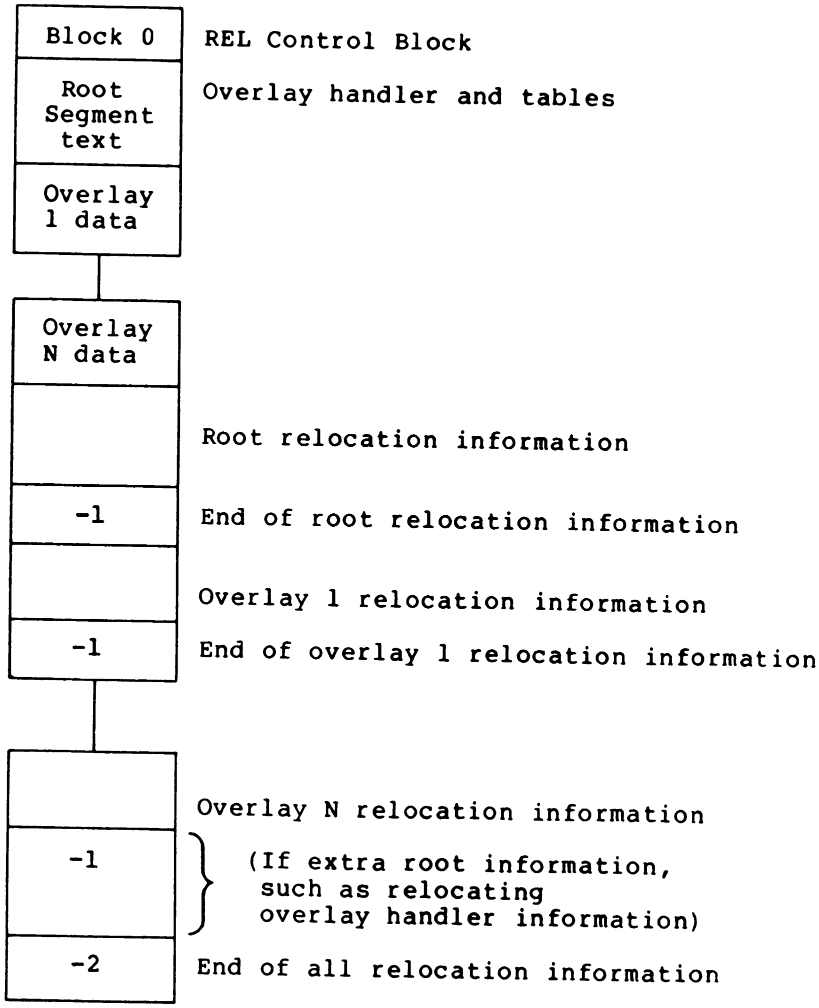


Figure C-23 REL File with Overlays

C.7 The Device Directory

The device directory begins at physical block 6 of any directory-structured device and consists of a series of directory segments that contain the names and lengths of the files on that device. The directory area is variable in length, from 1 to 31 (decimal) directory segments. DUP allows specification of the number of segments when the directory is initialized. The default value varies from device to device. See Chapter 8 of the RT-11 System User's Guide for a table of the default directory segments. Each directory segment is made up of two physical blocks. Thus, a single directory segment is 512 words, or 1024 bytes in length. Figure C-24 shows the general format of the device directory.

5 header words
file entries . . .
End of segment marker

Figure C-24 Device Directory Format

C.7.1 RT-11 File Storage

It is important for users to understand how RT-11 stores files on a device. All RT-11 files must reside on blocks that are contiguous on the device. Because the blocks are located in order, one after the other, the overhead of having pointers in each block to the next block is eliminated. Figure C-25 shows a simplified diagram of a file-structured device with two files stored on it.

file 80 blocks	empty 150 blocks	file 119 blocks	...

Figure C-25 File-Structured Device

When a file is created in RT-11, the size for the file must be allocated in the .ENTER programmed request. If the actual size is not known, as is often the case, the size allocated should be large enough to accomodate all the data possible. There are two special cases for the .ENTER request. A length argument of 0 allocates for the file either one-half the largest space available, or the second largest space, whichever is bigger. A length argument of -1 allocates the largest space possible on the device.

A tentative entry is then created on the device with the length allocated. The tentative entry is always followed by an empty entry. This is in order to account for unused space if the actual data written to the file is smaller than the size originally allocated. Figure C-26 shows an example of a tentative entry whose allocated size is 100 blocks.

file 80 blocks	tentative 100 blocks	empty 50 blocks	file 119 blocks

Figure C-26 Tentative Entry

Suppose, for example, that while the file is being created by one program, another program enters a new file, allocating 25 blocks for it. The device would appear as shown in Figure C-27. Note that every tentative entry must be followed by an empty entry.

file 80 blocks	tentative 100 blocks	empty 0 blocks	tentative 25 blocks	empty 25 blocks	file 119 blocks

Figure C-27 Two Tentative Entries

When a program finishes writing data to the device, it closes the tentative file with the .CLOSE programmed request. The tentative entry is made permanent. Its length is the actual size of the data that was written. The size of the empty entry is its original size plus the difference between the tentative file size and the permanent file size.

Figure C-28 shows the same example after both tentative files were closed. The first file's actual length is 75 blocks, and the second file's length is 10 blocks. Note that the total number of blocks associated with entries in Figure C-28, including empty entries, is equal to the total number of blocks in Figure C-26.

file 80 blocks	permanent 75 blocks	empty 25 blocks	permanent 10 blocks	empty 40 blocks	file 119 blocks

Figure C-28 Permanent Entries

Because of this method of storing files, it is impossible in RT-11 to extend the size of an existing file from within a running program. To make an existing file appear bigger from within a program, it is necessary to read the existing file, allocate a new, larger tentative entry, and then write both the old and the new data to the new file. The old file can then be deleted.

The DUP utility program provides an easy way to extend the size of an existing file. The /T option does this, providing that there exists an empty entry with sufficient space in it immediately after the data file.

C.7.2 Directory Header Format

Each directory segment contains a 5-word header, leaving 507 (decimal) words for directory entries. The contents of the header words are described in Table C-3.

Table C-3
Directory Header Words

Word	Contents
1	The number of segments available for entries. This number can be given to DUP when the device is initialized and must be in the range from 1 to 31 (decimal). Or, DUP can use the default value for the device.
2	Segment number of the next logical directory segment. The directory is a linked list of segments. This word is the link word between logically contiguous segments; if it is equal to 0, there are no more segments in the list. See Section C.7.4 for more details.
3	The highest segment currently open (each time a new segment is created, this number is incremented). This word is updated only in the first segment and is unused in any but the first segment.
4	The number of extra bytes per directory entry. This number can be specified when the device is initialized with DUP. Currently, RT-11 does not allow direct manipulation of information in the extra bytes.
5	Block number on the device where entries (files, tentatives, or empties) in this segment begin.

C.7.3 Directory Entry Format

The remainder of the segment is filled with directory entries. An entry has the format shown in Figure C-29.

<table>
<tr><td colspan="2">Status word</td></tr>
<tr><td colspan="2">Name (chars 1-3)
(in Radix-50)</td></tr>
<tr><td colspan="2">Name (chars 4-6)
(in Radix-50)</td></tr>
<tr><td colspan="2">File type
(1 to 3 characters)
(in Radix-50)</td></tr>
<tr><td colspan="2">Total file length</td></tr>
<tr><td>Job #</td><td>Channel #</td></tr>
<tr><td colspan="2">Date</td></tr>
<tr><td colspan="2">Optional extra words</td></tr>
<tr><td colspan="2">.
.
.</td></tr>
</table>

Figure C-29 Directory Entry Format

C.7.3.1 **Status Word** - The status word is broken down into two bytes of data, as shown in Figure C-30.

Type of entry	Reserved

Figure C-30 Status Word

Table C-4 lists the valid entry types.

Table C-4
Entry Types

Value	Type of Entry
1	Tentative file. (One that has been .ENTERed but not .CLOSEd.) Files of this type are deleted if not eventually .CLOSEd and are listed by DIR as <UNUSED> files.
2	An empty file. The name, file type, and date fields are not used. DIR lists an empty file as <UNUSED> followed by the length of the unused area.
4	A permanent entry. A tentative file that has been .CLOSEd is a permanent file. The name of a permanent file is unique; there can be only one file with a given name and file type. If another exists before the .CLOSE is done, it is deleted by the monitor as part of the .CLOSE operation.
10	End-of-segment marker. RT-11 uses this to determine when the end of the directory segment has been reached during a directory search.

Note that an end-of-segment marker can appear as the 512th word of a segment. It does not have to be followed by a name, type, or other data.

C.7.3.2 **Name and File Type** - These three words, in Radix-50, contain the symbolic name and file type assigned to a file. These words are usually unused for empty entries. However, the DIR utility program /Q option (or the keyboard monitor command DIRECTORY with the /DELETED option) lists the names and file types of deleted files.

C.7.3.3 **Total File Length** - The file length consists of the number of blocks taken up by the entry. Attempts to read or write outside the limits of the file result in an end-of-file error.

C.7.3.4 **Job Number and Channel Number** - A tentative file is associated with a job in one of two ways:

1. In the SJ environment, the sixth word of the entry holds the channel number on which the file is open. This number enables the monitor to locate the correct tentative entry for the channel when the .CLOSE is given. The channel number is loaded into the even byte of the sixth word.

2. In the FB and XM environments, the channel number is put into the even byte of the sixth word. In addition, the number of the job that is opening the file is put into the odd byte of the sixth word. The job number is required to uniquely identify the correct tentative file during the .CLOSE operation. It is also necessary because both jobs can have files open on their respective channels. The job number (0 for background, 2 for foreground) differentiates the tentative files.

NOTE

This sixth word (job number and channel number word) is used only when the file is marked as tentative. Once the entry becomes permanent, the word becomes unused. The function of the sixth word while the entry is permanent permanent is reserved for future use by DIGITAL software.

C.7.3.5 **Date** - When a tentative file is created by means of .ENTER, the system date word is put into the creation date slot for the file. The date word format is shown in Figure C-31. Bit 15 is reserved for future use by DIGITAL. This word is 0 if no date has been entered with the DATE keyboard monitor command.

15	14 13 12 10	9 8 7 6 5	4 3 2 1 0
	Month (1-12) (decimal)	Day (1-31) (decimal)	Year - 110 (octal)

Figure C-31 Date Word

C.7.3.6 **Extra Words** - The number of extra words is determined by specifying an option to DUP at initialization time. This choice is reflected by the number of extra bytes per entry in the header words. Although DUP provides for allocation of extra words, RT-11 provides no direct facility for manipulating this extra information. Any user program that needs to access these words must perform its own direct operations on the RT-11 directory.

Figure C-32 illustrates a typical RT-11 directory segment.

Header block:	4	Four segments available
	0	No next segment
	1	Highest open is #1
	0	No extra words per entry
	16	Files start at block 16 (octal)
File entries:	2000	Permanent entry
	71105	Radix-50 for RKM
	54162	Radix-50 for NFB
	75273	Radix-50 for SYS
	42	File is 42 (octal) blocks long (34 decimal)
	0	Used only for tentative entries
	0	No creation date
	1000	An empty entry
	0	(The name and file type of an
	0	empty entry are not significant.)
	0	
	100	100 (octal) blocks long (64 decimal)
	0	Used only for tentative entries
	0	No creation date
	2000	Permanent entry
	62570	Radix-50 for PIP
	0	Radix-50 for spaces
	50553	Radix-50 for MAC
	11	11 (octal) blocks long (9 decimal)
	0	Used only for tentative entries
	0	No creation date
	400	Tentative file on channel 1
	62570	Radix-50 for PIP
	0	Radix-50 for spaces
	50553	Radix-50 for MAC
	20	20 (octal) blocks long (16 decimal)
	1	Job 0 (BG); channel 1
	0	No creation date
	1000	(Every tentative entry must be
	0	followed by an empty entry.)
	0	
	0	
	1020	1020 (octal) blocks long (528 decimal)
	0	Used only for tentative entries
	0	No creation date
	4000	End of directory segment

Figure C-32 RT-11 Directory Segment

When the tentative file PIP.MAC is closed by the .CLOSE programmed request, the permanent file PIP.MAC is deleted.

To find the starting block of a particular file, first find the directory segment containing the entry for that file. Then take the starting block number given in the fifth word of that directory segment and add to it the length of each file in the directory before the desired file. For example, in Figure C-32, the permanent file PIP.MAC will begin at block number 160 (octal).

C.7.4 Size and Number of Files

The number of files that can be stored on an RT-11 device depends on the number of segments in the device's directory and the number of extra words per entry. The maximum number of directory segments on any RT-11 device is 31 (decimal). The following formula can be used to calculate the theoretical maximum number of directory entries.

$$31 * \frac{512-6}{7 + N} - 2$$

In the formula shown above,

N equals the number of extra information words per entry. If N is 0, the maximum is 2232 (decimal) entries.

Note that all divisions are integer. That is, the remainder should be discarded. No cancelling is valid.

In the formula shown above, the -2 is required for two reasons. First, in order to enter a file, the tentative entry must be followed by an empty entry. Second, an end-of-segment entry must exist. Note that on a disk squeezed by DUP, the end-of-segment entry might not be a full entry, but may contain just the status word.

If files are added sequentially (that is, one immediately after another) without deleting any files, roughly one-half the total number of entries will fit on the device before a directory overflow occurs. This situation results from the way filled directory segments are handled.

When a directory segment becomes full and it is necessary to open a new segment, approximately one half the entries of the filled segment are moved to the newly-opened segment. Thus, when the final segment is full, all previous segments have approximately one half their total capacity.

If files are continually added to a device and the SQUEEZE keyboard monitor command is not issued, the maximum number of entries can be computed from the following formula:

$$(M-1) * \frac{S}{2} + S$$

In the formula shown above,

M equals the number of directory segments

S can be computed from the following formula:

$$S = \frac{512 - 5}{7 + N} - 2$$

N equals the number of extra information words per entry.

The theoretical total of directory entries (see the first formula, above) can be realized by compressing the device (by using the DUP /S option or the monitor SQUEEZE command) when the directory fills up. DUP packs the directory segments as well as the physical device.

C.7.5 Directory Segment Extensions

RT-11 allows a maximum of 31 (decimal) directory segments. This section covers the processing of a directory segment. For illustrative purposes, the following symbols are used:

```
n !.    This represents a directory segment with some
  !     directory entries. The segment number is shown as n.
  !
  !


n !.    This represents a segment that is full. That is, no more
  !.    entries will fit in the segment.
  !.
  !.
```

The directory starts out with entries entered into segment 1:

```
1  !.
   !
   !
   !
```

As entries are added, segment 1 fills:

```
1  !.
   !.
   !.
   !.
```

When this occurs and an attempt is made to add another entry to the directory, the system must open another directory segment. If another segment is available, the following occurs:

1. One half of the entries from the filled segment are put into the next available segment and the header words of the new segment are filled with the correct information.

2. The shortened segment is rewritten to the disk.

3. The directory segment links are set.

4. The file is entered in either the shortened or the newly created segment, depending on which segment has the an emtpy entry of the required size.

NOTE

If the last segment becomes full and an attempt is made to enter another file, a fatal error occurs and an error message is generated.

Thus, in the normal case, the segment appears as:

```
       1  !.   Before extension. Segment 1 is full.
          !.
          !.
          !.

       1  !.   After extension. Half the entries are in the
          !.   new segment, half in the old. Segment 1 is
          !    linked to segment 2.
  1       !
Links
to 2
       2  !.
          !.
          !
          !
```

If many more files are entered, they fill up the second segment and overflow into the third segment, if it is available.

```
       1  !.
          !.
          !
  1       !
Links
to 2
       2  !.
          !.
          !
  2       !
Links
to 3
       3  !.
          !.
          !
          !
```

In this case, the segments are contiguous. However, the links between them are still required by the USR. The links are also required when the segments are not contiguous. For example, if a large file were deleted from segment 2 and many small files were entered, it would then be possible to overflow segment 2 again. If this occurred and a fourth segment existed, the directory would appear as follows:

```
       1  !.   In this case, segment 2 overflows into
          !.   segment 4 and the links are used to link
  1       !    logical pieces rather than physical
Links          pieces.
to 2
       2  !.
          !.
          !
  2       !
Links
to 4
       3  !.
          !.
          !
  4       !
Links
to 3
       4  !.   Segment 4 is linked to segment 3 because segment
          !.   2 was previously linked to 3.
          !
          !
```

C.8 Magtape Structure

This section covers the magtape file structure as implemented in RT-11 V03 and V03B. RT-11 V03 and V03B can read magtapes created under RT-11 V02C. RT-11 magtapes use a subset of the VOL1, HDR1, and EOF1 ANSI standard labels. RT-11 automatically writes magtapes with ANSI standard labels. RT-11 magtape implementation includes the following restrictions:

1. There is no EOV (end-of-volume) support. This means that no file can continue from the end of one tape volume over onto another volume.

2. RT-11 does not ignore noise blocks on input.

3. RT-11 assumes that data is written in records of 512 characters per block. The logical record size equals the physical record size.

   Note that the hardware magtape handler (as opposed to the file-structured magtape handler) can read data in any format at all. Or, users can make use of .SPFUN programmed requests and the file-structured magtape handler to read tapes whose data is in a non-standard format. The RT-11 utility programs, such as PIP, DUP, and DIR, can only read and write tapes in the standard RT-11 format of 512-character blocks.

4. RT-11 provides no volume protection by checking access fields.

In the diagrams shown below, an asterisk (*) represents a tape mark. The actual tape mark itself depends on the encoding scheme that the hardware uses. A typical nine channel NRZ tape mark consists of one tape character (octal 23) followed by seven blank spaces and an LRCC (octal 23). Programmers should consult the hardware manual for their particular tape devices if the format of the tape mark is important to them.

A file stored on magtape has the following structure:

```
HDR1 * data * EOF1 *
```

A volume containing a single file has the following format:

```
VOL1 HDR1 * data * EOF1 * * *
```

A volume containing two files has the following format:

```
VOL1 HDR1 * data * EOF1 * HDR1 * data * EOF1 * * *
```

A double tape mark following an EOF1 * label indicates logical end of tape. (Note that the EOF1 label is considered to consist of the actual EOF1 information plus a single tape mark.)

A magtape that has been initialized has the following format:

```
VOL1 HDR1 * * EOF1 * * *
```

A bootable magtape is a multi-file volume that has the following format:

```
VOL1 BOOT HDR1 * data * EOF1 * * *
```

To create an RT-11 bootable magtape, the file MBOOT.BOT must be used to copy the primary bootstrap. The primary bootstrap is represented by BOOT in the diagram above. It occupies a 256-word physical block. The first real file on the tape must be the secondary bootstrap, the file MSBOOT.BOT. If the tape is designed to allow another user to create another bootable magtape, the file MBOOT.BOT should be copied to the tape, as a file. (This is in addition to copying it into the boot block at the beginning of the tape.) Instructions for building bootable magtapes are in the RT-11 System Generation Manual.

Each label on the tape, as shown in the diagrams above, occupies the first 80 bytes of a 256-word physical block, and each byte in the label contains an ASCII character. (That is, if the content of a byte is listed as '1', the byte contains the ASCII code and not the octal code for '1'.) Table C-5 shows the contents of the first 80 bytes in the three labels. Note that the VOL1, HDR1, and EOF1 occupy a full 256-word block each, of which only the first 80 bytes are meaningful.

The meanings of the table headings for Table C-5 are as follows:

CP:	Character position in label
Field Name:	Reference name of field
L:	Length of field in bytes
Content:	Content of field
(space):	ASCII space character

Table C-5
ANSI Magtape Labels in RT-11

Volume Header Label (VOL1)			
CP	Field Name	L	Content
1-3	Label identifier	3	VOL
4	Label number	1	1
5-10	Volume identifier	6	Volume Label. If no volume ID is specified by the user at initialization time, the default is RT11A(space)
11	Accessibility	1	(Space)
12-37	Reserved	26	(Spaces)
38-50	Owner identifier	13	CP38 = D This means tape CP39 = % was written by CP40 = B DEC PDP-11 CP40-50 = Owner Name. Maximum is ten characters; default is (spaces)
51	DEC standard version	1	1
52-79	Reserved	28	(Spaces)
80	Label standard version	1	3
File Header label (HDR1)			
CP	Field Name	L	Content

(continued on next page)

Table C-5 (Cont.)
ANSI Magtape Labels in RT-11

1-3	Label identifier	3	HDR
4	Label number	1	1
5-21	File identifier	17	The 6-character ASCII file name (spaces can be used to pad the file name to six characters; the dot can be written without the padding), dot, 3-character file type. This field is left-justified and followed by spaces.
22-27	File set identifier	6	RT11A(space)
28-31	File section number	4	0001
32-35	File sequence number	4	First file on tape has 0001. This value is incremented by 1 for each succeeding file. On a newly initialized tape, this value is 0000.
36-39	Generation number	4	0001
40-41	Generation version	2	00
42-47	Creation date	6	(Space) followed by (year*1000) + day in ASCII; (space) followed by 00000 if no date. For example, 2/1/75 is stored as (space)75032.
48-53	Expiration date	6	(Space) followed by 00000 indicates an expired file.
54	Accessibility	1	(Space)
55-60	Block count	6	000000
61-73	System code	13	DECRT11A(space) followed by spaces.
74-80	Reserved	7	(Spaces)
First End-of-File Label (EOF1)			
This label is the same as the HDR1 label, with the following exceptions:			
CP	Field Name	L	Content
1-3	Label identifier	3	EOF
55-60	Block count	6	Number of data blocks since the preceding HDR1 label, unless a .SPFUN operation is done. If .SPFUNs are issued, the block count is 0. However, if only 256-word .SPFUN writes are done, block count is accurate.

C.9 Cassette Structure

A blank, newly initialized TU60 cassette appears in the format shown in Figure C-33.

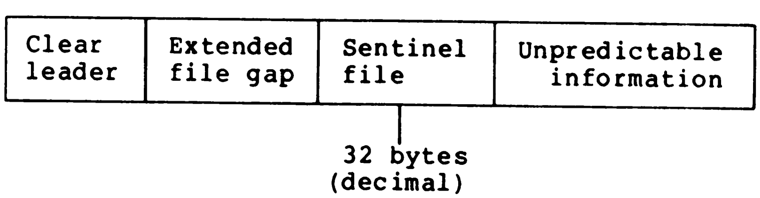


Figure C-33 Initialized Cassette Format

A cassette with a file on it appears as shown in Figure C-34.

Clear leader	Extended file gap	Header block	Block gap	Data block	Block gap	Data block	File gap	Sentinel file
		32 bytes (decimal)		128 bytes (decimal)				

Figure C-34 Cassette With Data

Files normally have data written in 128-byte (decimal) blocks. This can be altered by writing cassettes while in hardware mode. In hardware mode, the user program must handle the processing of any headers and sentinel files. In software mode, the handler automatically does this.

Figure C-34 illustrates a file terminated in the usual manner, by a sentinel file. However, the physical end of cassette can occur before the actual end of the file. This format appears as shown in Figure C-35.

Block gap	Data block	Block gap	Clear trailer

or as:

Block gap	Data block	Block gap	Data block	Clear trailer
			(Partially written)	

Figure C-35 Physical End of Cassette

In the latter case, for multi-volume processing, the partially written block must be rewritten as the first data block of the next volume.

The file header is a 32-byte (decimal) block that is the first block of any data file on a cassette. If the first byte of the header is null (000), the header is interpreted as a sentinel file, which is an indication of logical end of cassette. The format of the header is illustrated in Table C-6. The data in Table C-6 is binary (that is, 0 equals a byte of 0) unless it is specified to be ASCII.

Table C-6
Cassette File Header Format

Byte Number	Contents
0-5	File name in ASCII characters (ASCII is assumed to imply a 7-bit code).
6-8	File type in ASCII characters
9	Data type (0 for RT-11)
10-11	Block length of 128 (decimal), 200 (octal). Byte 10 = 0, high order; byte 11 = 200, low order.
12	File sequence number. (0 for single volume file or the first volume of a multi-volume file; successive numbers are used for continuations.
13	Level 1; this byte is a 1. This byte must be changed to 0 if CAPS-11 will be used to load files. See the RT-11 System Generation Manual for details.
14-19	Date of file creation (six ASCII digits representing day (0-31); month (0-12); and last two digits of the year; 0 or 40 (octal) in first byte means no date present)
20-21	0
22	Record attributes (0 is RT-11 cassette)
23-28	Reserved
29-31	Reserved for user

INDEX