Example:

```
.TITLE  TWAIT.MAC
;.TWAIT CAN BE USED IN APPLICATIONS WHERE A PROGRAM MUST BE ONLY
;ACTIVATED PERIODICALLY. THIS EXAMPLE WILL 'WAKE UP' EVERY TEN SECONDS
;TO PERFORM A TASK, AND THEN SLEEP AGAIN. FOR EXAMPLE PURPOSES ONLY A
;COUNT OF TEN CYCLES IS MAXIMUM.
        .MCALL  .TWAIT,.QSET,.EXIT,.PRINT
GO:     .QSET   #QAREA,#7       ;SET UP 7 EXTRA ELEMENTS
        CLR     COUNT           ;MAX COUNT
START:  MOV     #EMTLST,R0      ;SET R0 TO THE ARG. BLOCK
        .TWAIT                  ;GO TO SLEEP FOR 10 SECONDS
        BCS     NOQ             ;NO QUEUE ELEMENT?
        JSR     PC,TASK         ;DO SOMETHING HERE
        INC     COUNT           ;BUMP COUNTER
        CMP     #10.,COUNT      ;AT MAX ?
        BNE     START           ;NO-GO AGAIN
        .EXIT                   ;EXIT
QAREA:  .BLKW   7*7             ;SPACE FOR 7 ELEMENTS
EMTLST: .BYTE   0,24
        .WORD   TIME
TIME:   .WORD   0,10.*60.       ;10 SECOND INTERVALS
TASK:                           ;SOME GENERALIZED USER
                                ;WHERE.

        INC     MPTR
        BIT     #1,MPTR
        BEQ     1$
        .PRINT  #MSG
        RTS     PC
1$:     .PRINT  #MSG1
        RTS     PC
COUNT:  .WORD   0
MPTR:   .WORD   0
MSG:    .ASCIZ  /TICK/
MSG1:   .ASCIZ  /TOCK/
        .EVEN
NOQ:    .EXIT
        .END    GO
```


2.4.61 .WAIT

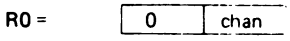
The .WAIT request suspends program execution until all input/output requests on the specified channel are completed. The .WAIT request combined with the .READ/.WRITE requests makes double-buffering a simple process.

.WAIT also conveys information back through its error returns. An error is returned if either the channel is not currently open or if the last I/O operation resulted in a hardware error.

In an FB system, executing a .WAIT when I/O is pending causes that job to be suspended and the other job to run, if possible.

Macro Call: .WAIT chan

Request Format:



Errors:

Code	Explanation
0	Channel specified is not open.
1	Hardware error occurred on the previous I/O operation on this channel.

Example:

For an example of .WAIT used for I/O synchronization, see the examples for the .WRITE/.WRITC/.WRITW requests.

```
.TITLE  WAIT.MAC
;AN EXAMPLE OF THE USE OF .WAIT FOR ERROR DETECTION IS ITS USE IN
;CONJUNCTION WITH .CSIGEN TO DETERMINE WHICH FILE FIELDS IN THE COMMAND
;STRING HAVE BEEN SPECIFIED. FOR EXAMPLE, A PROGRAM SUCH
;AS MACRO MIGHT USE THE FOLLOWING CODE TO DETERMINE IF A LISTING
;FILE IS DESIRED.
        .MCALL  .WAIT,.CSIGEN,.EXIT

START:

        .CSIGEN #DSPACE,#DEXT,#0 ;PROCESS COMMAND STRING
        .WAIT   #0               ;CHECK FOR FILE IN FIRST FIELD
        BCS     NOBINARY         ;NO BINARY DESIRED

NOBINARY:

        .WAIT   #1               ;CHECK FOR LISTING SPECIFICATION
        BCS     NOLISTING        ;NO LISTING DESIRED

NOLISTING:

        .WAIT   #3               ;CHECK FOR INPUT FILE OPEN
        BCS     ERROR            ;NO INPUT FILE

ERROR:  .EXIT

DEXT:   .RAD50  /MAC/
        .RAD50  /OBJ/
        .RAD50  /LST/
        .WORD   0
DSPACE:.
        .END    START
```

.WRITE/.WRITC/.WRITW

2.4.62 .WRITE/.WRITC/.WRITW

Note that in the case of .WRITE and .WRITC, additional queue elements should be allocated for buffered I/O operations (see .QSET).

.WRITE

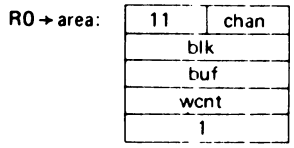
The .WRITE request transfers a specified number of words from memory to the specified channel. Control returns to the user program immediately after the request is queued.

Macro Call: .WRITE area,chan,buf,wcnt,blk

where:

area	is the address of a five-word EMT argument block.
chan	is a channel number in the range 0-377 (octal).
buf	is the address of the memory buffer to be used for output.
wcnt	is the number of words to be written.
blk	is the block number to be written. For a file-structured .LOOKUP or .ENTER, the block number is relative to the start of the file. For a non-file-structured .LOOKUP or .ENTER, the block number is the absolute block number on the device. The user program should normally update blk before it is used again. See Chapter 1 for the significance of the block number for line printers, paper tape readers, etc.

Request Format:



Notes:

See the note following .WRITW.

Errors:

Code	Explanation
0	Attempted to write past end-of-file.
1	Hardware error.
2	Channel was not opened.

Example:

Refer to the examples following .WRITW.

.WRITC

The .WRITC request transfers a specified number of words from memory to a specified channel. Control returns to the user program immediately after the request is queued. Execution of the user program continues until the .WRITC is complete, then control passes to the routine specified in the request. When an RTS PC is encountered in the completion routine, control returns to the user program.

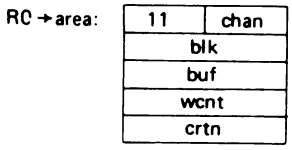
Macro Call: .WRITC area,chan,buf,wcnt,crtn,blk

where:

area	is the address of a five-word EMT argument block.

chan	is a channel number in the range 0 to 377 (octal).
buf	is the address of the memory buffer to be used for output.
wcnt	is the number of words to be written.
crtn	is the address of the completion routine to be entered (see Section 2.2.8).
blk	is the block number to be written. For a file-structured .LOOKUP or .ENTER, the block number is relative to the start of the file. For a non-file-structured .LOOKUP or .ENTER, the block number is the absolute block number on the device. The user program should normally update blk before it is used again. See Chapter 1 for the significance of the block number for line printers, paper tape readers, etc.

Request Format:



Notes:

See the note following .WRITW.

When entering a .WRITC completion routine the following are true:

1. R0 contains the contents of the channel status word for the operation. If bit 0 of R0 is set, a hardware error occurred during the transfer. The data can be unreliable.

2. R1 contains the octal channel number of the operation. This is useful when the same completion routine is to be used for several different transfers.

3. Registers R0 and R1 are available for use by the routine, but all other registers must be saved and restored. Data cannot be passed between the main program and completion routines in any register or on the stack.

Errors:

Code	Explanation
0	End-of-file on output. Tried to write outside limits of file.
1	Hardware error occurred.
2	Specified channel is not open.

Example:

Refer to the examples following .WRITW.

.WRITW

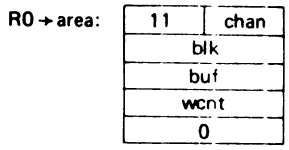
The .WRITW request transfers a specified number of words from memory to the specified channel. Control returns to the user program when the .WRITW is complete.

Macro Call: .WRITW area,chan,buf,wcnt,blk

where:

area	is the address of a five-word EMT argument block.
chan	is a channel number in the range 0-377 (octal).
buf	is the address of the buffer to be used for output.
wcnt	is the number of words to be written. The number must be positive.
blk	is the block number to be written. For a file-structured .LOOKUP or .ENTER, the block number is relative to the start of the file. For a non-file-structured .LOOKUP or .ENTER, the block number is the absolute block number on the device. The user program should normally update blk before it is used again. See Chapter 1 for the significance of the block number for line printers, paper tape readers, etc.

Request Format:



NOTE

Upon return from any .WRITE, .WRITC or .WRITW programmed request, R0 contains the number of words requested if the write is to a sequential-access device (for example, magtape). If the write is to a random-access device (disk or DECtape), R0 contains the number of words that will be written (.WRITE or .WRITC) or have been written (.WRITW). If a request is made to write past the end-of-file on a random-access device, the word count is shortened and an error is returned. The shortened word count is returned in R0. Note that the write is done and a completion routine, if specified, is entered, unless the request cannot be partially filled (shortened word count = 0).

Errors:

Code	Explanation
0	Attempted to write past EOF.
1	Hardware error.
2	Channel was not opened.

Examples:

Each of the following examples is a simple program to duplicate a paper tape. They illustrate RT-11's three types of .READ/.WRITE requests.

In the first example, .READW and .WRITW are used. The I/O is completely synchronous, with each request retaining control until the buffer is filled (or emptied).

```
        .TITLE  READW.MAC
                .MCALL  .FETCH,.READW,.WRITW
                .MCALL  .ENTER,.LOOKUP,.PRINT,.EXIT,.CLOSE,.WAIT
ERRBYT=52
START:  .FETCH  #HSPACE,#TTNAME ;GET TT HANDLER
        BCS     FERR            ;TT NOT AVAILABLE
        MOV     R0,R2           ;R0 HAS NEXT FREE LOCATION
        .FETCH  R2,#PCNAME      ;GET PC HANDLER
        BCS     FERR            ;NOT AVAILABLE
        MOV     #AREA,R5        ;EMT ARGUMENT AREA
        CLR     R4              ;R4 IS OUTPUT CHANNEL; 0
        MOV     #1,R3           ;R3 IS INPUT CHANNEL ;1
        .ENTER  R5,R4,#PCNAME   ;ENTER THE FILE
        BCS     ENERR           ;SOME ERROR IN ENTER
        .LOOKUP R5,R3,#TTNAME   ;LOOKUP FILE ON CHANNEL 1
        BCS     LKERR           ;ERROR IN LOOKUP
        CLR     R1              ;USE R1 AS BLOCK NUMBER
LOOP:   .READW  R5,R3,#BUFF,#256.,R1 ;READ ONE BLOCK
        BCS     RDERR
        .WRITW  R5,R4,#BUFF,#256.,R1 ;WRITE THAT BLOCK
        BCS     WTERR
        INC     R1              ;BUMP BLOCK. NOTE: THIS IS
                                ;NOT NECESSARY FOR NON-FILE
                                ;DEVICES IN GENERAL. IT IS
                                ;USED HERE AS AN EXAMPLE OF
                                ;A GENERAL TECHNIQUE.
        BR      LOOP            ;KEEP GOING
RDERR:  TSTB    @#ERRBYT                ;ERROR. IS IT EOF?
        BEQ     1$              ;YES
        .PRINT  #RDMSG          ;NO, HARD READ ERROR
        .EXIT
1$:     .CLOSE  R3              ;CLOSE INPUT AND OUTPUT
        .CLOSE  R4
        .EXIT                   ;AND EXIT.
WTERR:  .PRINT  #WTMSG
        .EXIT
TTNAME: .RAD50  /TT /           ;NOTE THAT TT NEEDS NO FILE NAME
        .WORD   0               ;FILE NAME NEED ONLY BE 0.
PCNAME: .RAD50  /PC /
        .WORD   0
FERR:   .PRINT  #FMSG           ;ERROR ACTIONS GO HERE. IT IS
        .EXIT                   ;GENERALLY UNDESIRABLE TO
ENERR:  .PRINT  #EMSG           ;EXECUTE A HALT OR RESET
        .EXIT                   ;INSTRUCTION ON ERROR.
LKERR:  .PRINT  #LMSG
        .EXIT
FMSG:   .ASCIZ  /NO DEVICE?/
EMSG:   .ASCIZ  /ENTRY ERROR?/
LMSG:   .ASCIZ  /LOOKUP ERROR?/
RDMSG:  .ASCIZ  /READ ERROR?/
WTMSG:  .ASCIZ  /WRITE ERROR?/
        .EVEN
AREA:   .BLKW   10
BUFF:   .BLKW   256.
HSPACE=.
        .END    START
```

The same routine can be coded using .READ and .WRITE as follows. The .WAIT request is used to determine if the buffer is full or empty prior to its use.

```
.TITLE  READ.MAC
        .MCALL  .FETCH,.READ,.WRITE
        .MCALL  .ENTER,.LOOKUP,.PRINT,.EXIT,.CLOSE,.WAIT

ERRBYT=52

START:  .FETCH  #HSPACE,#TTNAME ;GET TT HANDLER
        BCS     FERR            ;TT NOT AVAILABLE
        MOV     R0,R2           ;R0 HAS NEXT FREE LOCATION
        .FETCH  R2,#PCNAME      ;GET PC HANDLER
        BCS     FERR            ;NOT AVAILABLE
        MOV     #AREA,R5        ;EMT ARGUMENT AREA
        CLR     R4              ;R4 IS OUTPUT CHANNEL; 0
        MOV     #1,R3           ;R3 IS INPUT CHANNEL ;1
        .ENTER  R5,R4,#PCNAME   ;ENTER THE FILE
        BCS     ENERR           ;SOME ERROR IN ENTER
        .LOOKUP R5,R3,#TTNAME   ;LOOKUP FILE ON CHANNEL 1
        BCS     LKERR           ;ERROR IN LOOKUP
        CLR     R1              ;USE R1 AS BLOCK NUMBER
LOOP:   .READ   R5,R3,#BUFF,#256.,R1 ;READ A BUFFER
        BCS     RDERR
        .WAIT   R3              ;WAIT FOR BUFFER
        BCS     IOERR           ;ERROR HERE IS HARD ERROR
        .WRITE  R5,R4,#BUFF,#256.,R1 ;WRITE TTHE BUFFER
        BCS     IOERR           ;I/O ERROR
        INC     R1
        BR      LOOP            ;KEEP GOING
RDERR:  TSTB    @#ERRBYT                ;ERROR. IS IT EOF?
        BNE     IOERR           ;NO, HARD ERROR
        .CLOSE  R3              ;CLOSE INPUT AND OUTPUT
        .CLOSE  R4
        .EXIT                   ;AND EXIT.
IOERR:  .PRINT  #IOMSG          ;NO, HARD READ ERROR
        .EXIT
TTNAME: .RAD50  /TT /           ;NOTE THAT TT NEEDS NO FILE NAME
        .WORD   0               ;FILE NAME NEED ONLY BE 0.
PCNAME: .RAD50  /PC /
        .WORD   0
FERR:   .PRINT  #FMSG           ;ERROR ACTIONS GO HERE. IT IS
        .EXIT                   ;GENERALLY UNDESIRABLE TO
ENERR:  .PRINT  #EMSG           ;EXECUTE A HALT OR RESET
        .EXIT                   ;INSTRUCTION ON ERROR.
LKERR:  .PRINT  #LMSG
        .EXIT
FMSG:   .ASCIZ  /NO DEVICE?/
EMSG:   .ASCIZ  /ENTRY ERROR?/
LMSG:   .ASCIZ  /LOOKUP ERROR?/
IOMSG:  .ASCIZ  "I/O ERROR?"
WTMSG:  .ASCIZ  /WRITE ERROR?/
        .EVEN
AREA:   .BLKW   10
BUFF:   .BLKW   256.
HSPACE=.
        .END    START
```

.READ and .WRITE are also often used for double-buffered I/O. The basic double-buffering algorithm for input is:

```
          Action                          Explanation

          READ    BUFFER 1      Fill BUFFER 1
LOOP:     WAIT    BUFFER 1      Wait for BUFFER 1 to fill
          READ    BUFFER 2      Start filling BUFFER 2
          USE     BUFFER 1      Process BUFFER 1 while BUFFER 2 fills
          WAIT    BUFFER 2      Wait for BUFFER 2 to fill
          READ    BUFFER 1      Start filling BUFFER 1
          USE     BUFFER 2      Process BUFFER 2 while BUFFER 1 fills
          BR      LOOP
```

Correspondingly, the basic double-buffering algorithm for output is:

```
          Action                          Explanation

          FILL    BUFFER 1      Prepare BUFFER 1 for output
LOOP:     WRITE   BUFFER 1      Start emptying BUFFER 1
          FILL    BUFFER 2      Fill BUFFER 2 while BUFFER 1
                                empties
          WAIT    BUFFER 1      Wait for BUFFER 1 to empty
          WRITE   BUFFER 2      Start emptying BUFFER 2
          FILL    BUFFER 1      Fill BUFFER 1 while BUFFER 2
                                empties
          WAIT    BUFFER 2      Wait for BUFFER 2 to empty
          BR      LOOP
```

The following example duplicates a paper tape by using the .READC and .WRITC requests and completion routines. After the first read, the completion routines control the remaining I/O.

```
.TITLE  WRITC2.MAC
        .MCALL  .FETCH,.READC,.WRITC
        .MCALL  .ENTER,.LOOKUP,.PRINT,.EXIT,.CLOSE,.WAIT

ERRBYT=52

START:  .FETCH  #HSPACE,#TTNAME ;GET TT HANDLER
        BCS     FLNK            ;TT NOT AVAILABLE
        MOV     R0,R2           ;R0 HAS NEXT FREE LOCATION
        .FETCH  R2,#PCNAME      ;GET PC HANDLER
FLNK:   BCS     FERR            ;NOT AVAILABLE
        MOV     #AREA,R5        ;EMT ARGUMENT AREA
        CLR     R4              ;R4 IS OUTPUT CHANNEL; 0
        MOV     #1,R3           ;R3 IS INPUT CHANNEL ;1
        .ENTER  R5,R4,#PCNAME   ;ENTER THE FILE
        BCS     ENERR           ;SOME ERROR IN ENTER
        .LOOKUP R5,R3,#TTNAME   ;LOOKUP FILE ON CHANNEL 1
        BCS     LKERR           ;ERROR IN LOOKUP
        CLR     R1              ;USE R1 AS BLOCK NUMBER
LOOP:   CLR     DFLG            ;CLEAR DONE/ERROR FLAG
        .READC  R5,R3,#BUFF,#256.,#RDCOMP,R1 ;READ ONE BLOCK
        BCS     EOF             ;NO ERROR WILL HAPPEN HERE
1$:     TST     DFLG            ;DONE FLAG SET?
        BEQ     1$              ;NO, WAIT FOR IT TO BE SET.
        BMI     IOERR           ;YES, BUT HARD ERROR OCCURRED
EOF:    .CLOSE  R3              ;CLOSE INPUT AND OUTPUT CHANNELS
        .CLOSE  R4
        .EXIT                   ;ALL DONE
.ENABL  LSB
RDCOMP: RUR     R0              ;IF BIT 0 SET
        BCS     RWERR           ;AN ERROR OCCURRED.
        .WRITC  #AREA,#0,#BUFF,#256.,#WRCOMP,BLKN ;WRITE THAT BLOCK
        BCC     2$              ;ERROR HERE IS HARDWARE
RWERR:  MOV     #-1,DFLG        ;FLAG THE ERROR
2$:     RTS     PC
WRCOMP: ROR     R0
        BCS     RWERR           ;HARDWARE ERROR
        INC     BLKN            ;BUMP BLOCK NUMBER.
        .READC  #AREA,#1,#BUFF,#256.,#RDCOMP,BLKN
        BCC     3$              ;NO ERROR
        TSTB    @#ERRBYT                ;EOF?
        BNE     RWERR           ;NO, HARD  ERROR
        INC     DFLG            ;SAY WE'RE DONE
3$:     RTS     PC
.DSABL  LSB
FERR:   MOV     #FMSG,R0        ;ERROR ACTIONS GO HERE. IT IS
        BR      TYPIT           ;GENERALLY UNDESIRABLE TO
ENERR:  MOV     #EMSG,R0        ;EXECUTE A HALT OR RESET
        BR      TYPIT           ;INSTRUCTION ON ERROR.
IOERR:  MOV     #IOMSG,R0
        BR      TYPIT
LKERR:  MOV     #LMSG,R0
TYPIT:  .PRINT
        .EXIT
.NLIST BEX
FMSG:   .ASCIZ  /NO DEVICE?/
EMSG:   .ASCIZ  /ENTRY ERROR?/
LMSG:   .ASCIZ  /LOOKUP ERROR?/
IOMSG:  .ASCIZ  "I/O ERROR?"
.LIST BEX
.EVEN
DFLG:   .WORD   0
TTNAME: .RAD50  /TT /           ;NOTE THAT TT NEEDS NO FILE NAME
        .WORD   0               ;FILE NAME NEED ONLY BE 0.
PCNAME: .RAD50  /PC /
        .WORD   0
BLKN:   .WORD   0               ;BLOCK NUMBER
AREA:   .BLKW   10
BUFF:   .BLKW   256.
HSPACE=.
        .END    START
```

The following example incorporates the .LOOKUP, .READW, and .CLOSE requests. The program opens the file RT11.MAC on the system device, SY:, for input on channel 0. The first block is read and the file is then closed.

```
.TITLE  WRTT4.MAC
        .MCALL  .CLOSE,.LOOKUP
        .MCALL  .PRINT,.EXIT,.READW,.FETCH

START:  MOV     #LIST,R5        ;EMT ARGUMENT LIST POINTER
        CLR     R4              ;BLOCK NUMBER
        CLR     R3              ;CHANNEL #
        .FETCH  #CORADD,#FPTR   ;FETCH DEVICE HANDLER
        BCC     2$
        MOV     #FETMSG,R0      ;FETCH ERROR
1$:     .PRINT                  ;PRINT ERROR MESSAGE
        .EXIT
2$:     .LOOKUP R5,R3,#FPTR     ;LOOKUP FILE ON CHANNEL 0
        BCC     3$
        MOV     #LKMSG,R0       ;PRINT FAILURE MESSAGE
        BR      1$
3$:     .READW  R5,R3,#BUFF,#256.,R4 ;READ ONE BLOCK
        BCC     4$
        MOV     #RDMSG,R0       ;READ ERROR
        BR      1$
4$:     .CLOSE  R3              ;CLOSE THE CHANNEL
        .EXIT

LIST:   .BLKW   5               ;LIST FOR EMT CALLS
FPTR:   .RAD50  /SY RT11  MAC/  ;RAD50 OF FIEL NAME,DEVICE
FETMSG: .ASCIZ  /FETCH FAILED/  ;ASCII MESSAGES
LKMSG:  .ASCIZ  /LOOKUP FAILED/
RDMSG:  .ASCIZ  /READ FAILED/
        .EVEN
CORADD: .BLKW   2000            ;SPACE FOR LARGEST HANDLERS
BUFF:
        .END    START
```

2.5 CONVERTING VERSION 1 MACRO CALLS TO VERSION 3

As mentioned in the introduction of this chapter, RT-11 Version 3 and later releases support a slightly modified format for system macro calls compared to Version 1. This section details the conversion process from the Version 1 format to Version 3.

2.5.1 Macro Calls Requiring No Conversion

Version 1 macro calls that need no conversion are:

.CSIGN	.RCTLO
.CSISPC	.RELEAS
.DATE	.SETTOP*
.DSTATUS	.SRESET
.EXIT	.TTINR**
.FETCH	.TTOUTR
.HRESET	.TTYIN
.LOCK	.TTYOUT
.PRINT	.UNLOCK
.QSET	

*Provided location 50 is examined for the maximum value.

**Except in FB or XM systems.

2.5.2 Macro Calls That Can Be Converted

The following Version 1 macro calls can be converted:

```
.CLOSE          .RENAME
.DELETE         .REOPEN
.ENTER          .SAVESTATUS
.LOOKUP         .WAIT
.READ           .WRITE
```

The general format of the ..V1.. macro is:

```
.PRGREQ chan,arg1,arg2,...argn
```

In this form, chan is an integer between 0 and 17 (inclusive), and is not a general assembler argument. The channel number is assembled into the EMT instruction itself. The arguments arg1-argn are always pushed either into R0 or on the stack.

The ..V2.. equivalent of the above call is:

```
.PRGREQ area,chan,arg1,....argn
```

In the ..V2.. call, the chan argument can be any legal assembler argument; it need not be in the range 0 to 17 (octal), but should be in the range 0-377 (octal). Area points to a memory list where the arguments arg1...argn will go.

As an example, consider a .READ request in both forms:

```
V1:     .READ 5,#BUFF,#256.,BLOCK

V2:     .READ #AREA,#5,#BUFF,#256.,BLOCK
              .
              .
              .

AREA:   .WORD 0    ;CHANNEL/FUNCTION CODE HERE
        .WORD 0    ;BLOCK NUMBER HERE
        .WORD 0    ;BUFFER ADDRESS HERE
        .WORD 0    ;WORD COUNT HERE
        .WORD 0    ;A 1 GOES HERE.
```

Thus, the difference in the calls is that in Version 2 the channel number becomes a legal assembler argument and the area argument has been added.

Following is a complete list of the conversions necessary for each of the EMT calls. Both the Version 1 and Version 2 formats are given. In Version 3, this function is performed automatically. Note that parameters inside square brackets, [], are optional parameters. Refer to the appropriate section in this chapter for more details on each request.

Version	Programmed Request
V1:	.DELETE chan,dblk
V2:	.DELETE area,chan,dblk[,count]
V1:	.LOOKUP chan,dblk
V2:	.LOOKUP area,chan,dblk[,count]
V1:	.ENTER chan,dblk[,length]
V2:	.ENTER area,chan,dblk[,length[,count]]
V1:	.RENAME chan,dblk
V2:	.RENAME area,chan,dblk
V1:	.SAVESTAT chan,cblk
V2:	.SAVESTAT area,chan,cblk
V1:	.REOPEN chan,cblk
V2:	.REOPEN area,chan,cblk
V1:	.CLOSE chan
V2:	.CLOSE chan
V1:	.READ/.READW chan,buff,wcnt,blk
V2:	.READ/.READW area,chan,buff,wcnt,blk
V1:	.READC chan,buff,wcnt,crtn,blk
V2:	.READC area,chan,buff,wcnt,crtn,blk
V1:	.WRITE/.WRITW chan,buff,wcnt,blk
V2:	.WRITE/.WRITW area,chan,buff,wcnt,blk
V1:	.WRITC chan,buff,wcnt,crtn,blk
V2:	.WRITC area,chan,buff,wcnt,crtn,blk
V1:	.WAIT chan
V2:	.WAIT chan

Important features to keep in mind for Version 3 calls are:

1. Version 3 calls require the area argument, which points to the area where the other arguments will be (unless R0 already points to it and the first word is set up).

2. Enough memory space must be allocated to hold all the required arguments.

3. The chan argument must be a legal assembler argument, not just an integer between 0-17 (octal).

4. Blank fields are permitted in the Version 3 calls. Any field not specified (left blank) is not modified in the argument block.

CHAPTER 3

EXTENDED MEMORY

3.1 INTRODUCTION

The RT-11 operating system is the single-user system for the PDP-11 family of computers. As such, RT-11 has never supported more than 28K words of memory. Extended memory support has been reserved for the multi-tasking systems, since multi-tasking is the usual method for utilizing a large memory space. In such systems, many tasks are run simultaneously, but each task is limited to 32K words or less because of the virtual addressing limitation imposed by the 16-bit word size and the byte addressing capabilities of the PDP-11. However, users of both types of systems encounter the same addressing limitation and have to apply one of several techniques for effectively extending the available logical addressing space.

Two of the standard methods of extending a program are overlaying and chaining. In overlaying, a program is broken into pieces called segments and assembled separately. The segments are then linked together with an overlay handler. When a segment of code is referenced that is not resident, the overlay handler reads the referenced segment into memory, overlaying another segment not currently needed as specified at link time. Communication between segments must be through the root segment of the program, which is never overlayed.

Chaining of programs is most effective when the program can be broken into several completely independent functions that can communicate through a data file. An example of this is the use of a separate program to produce a cross reference listing in RT-11. The MACRO assembler chains to CREF and passes the name of a temporary file containing the necessary symbol data. CREF produces its listing from the file and then chains back to MACRO. These techniques are effective in extending logical addressing space, but they have disadvantages and may not suit a particular application. Overlaying can increase execution time if a great deal of overlaying occurs during program execution. Segmenting may not be applicable. The use of virtual disk arrays can considerably slow down array processing. What is needed is a means of address extension that makes use of the full memory capabilities of the PDP-11.

RT-11 offers as a SYSGEN option the ability to increase the amount of memory it supports from 28K words to 124K words. This optional monitor (extended memory, XM) is a superset of the FB monitor and extends the memory support capability of RT-11 beyond the 28K-word restriction imposed by the 16-bit address size of unmapped PDP-11 processors. The XM monitor is based on the FB monitor and is functionally equivalent to it. The XM monitor offers a set of programmed requests to extend a program's effective logical addressing space that is a subset of similar requests offered on other PDP-11 systems.

The XM monitor software architecture makes it unnecessary for the user to have a detailed knowledge of the PDP-11 memory management hardware. In a mapped system, the user does not need to know where a program resides in physical memory. Mapping, the process of associating program segments with available physical memory, is transparent to the user and is accomplished by the memory management hardware. When a program addresses a location, the memory management unit determines the location's actual physical address in memory. The programmed requests use the memory management hardware to perform address mapping at a higher level that is visible to and controlled by the user. Programs developed on an unmapped system will run on a mapped system. This applies to system programs and user programs. They are called privileged, or compatibility jobs. However, programs that must use the extended memory monitor will not run on an unmapped system. These programs are called virtual jobs. Privileged jobs are not restricted from using the extended memory programmed requests. If they do so, however, they must run on a mapped system under the XM monitor.

The address space extension programmed requests supplied with XM provide the advanced or system programmer with controlled access to extended memory. Through these requests, the program can allocate a region of extended memory for its use and can map selected portions of its virtual address space to portions of that region. A single segment of address space can be mapped into several successive segments of memory, providing an effective extension of the logical address space of the program. The use made of extended memory depends on the application, and can include such uses as resident overlays, buffers, or data arrays.

The remaining sections of this chapter emphasize the use of the programmed requests and their associated parameters, arguments and data structures.

3.2 THE LANGUAGE AND CONCEPTS OF RT-11 EXTENDED MEMORY SUPPORT

Understanding the language and terminology of extended memory is essential to effective program utilization of this feature. Following is a list of terms with their definitions that provides the programmer with the necessary vocabulary:

1. Address Space - The set of addresses available to a program while it is running in a specific processor mode. (RT-11 supports the kernel and user modes of PDP-11 memory management hardware.) The virtual address space is that set of addresses available to a program in a particular mode. The physical address space for the mode is the set of physical addresses to which the virtual addresses are mapped. In general, the kernel and user modes operate in the same virtual address space but possibly in different physical address spaces.

2. Block - A unit of memory. The memory management unit deals in units of 32 words.

3. Dynamic Region - A region in extended memory created by a program at run time through an allocation request.

4. Extended Memory - Memory having a physical address greater than 28K.

5. Kernel Mode - One of the modes of the memory management unit hardware. It is the mapping mode for RMON and the USR. Contrast with user mode.

6. Low Memory - Memory having a physical address in the range 0-28K words.

7. Mapping - The process of associating a virtual address with a physical memory location accomplished by the memory management hardware.

8. Memory Management Fault - An error in an extended memory operation caused by referencing an address not within the program's virtual address space, and indicated by an error message returned by the monitor and displayed at the console terminal.

9. Mode - The memory management unit provides a separate set of relocation registers for use in each of its modes. The mode is specified by bits (15 and 14) in the PS word.

    00 = Kernel
    11 = User

    RT-11 uses kernel mode for monitor and USR operations, and user mode for user programs. The keyboard monitor (KMON) also runs in user mode.

10. Page - A collection of continuous memory addresses mapped by a single relocation register. The 32K word virtual address space is divided into eight 4K word sections, called pages. The lowest address in each page is a whole multiple of 4096. The length of the page is some whole multiple of 32 words ranging from 1 through 128 units. Thus, a page can vary in size from 32 to 4096 words, in 32 word increments.

11. Page Address Register - A memory management unit register containing the base address or relocation constant associated with a page. The memory management unit has 16 page address registers: two groups of eight registers (one register per 4K page). One group is associated with each of the two processor modes (user and kernel).

12. Page Descriptor Register - A memory management unit register containing information associated with a page. This includes the page length, the expansion direction, and the access key. The RT-11 system uses 16 of these registers; eight for user and eight for kernel mode.

13. Physical Address - The hardware address of a specific memory location. The XM monitor supports memory with a physical address between 0 and 124K words.

14. Program Logical Address Space - Program logical address space is the range of effective memory space available to a program. Normally it is limited to the 32K words of virtual address space. It can be extended by overlaying or by using the memory extension capability of the XM monitor.

15. Program Virtual Address Space - Program virtual address space is the 32K (32,768 words) address space accessible to a program determined by the 16-bit word size of the PDP-11 processors.

16. Region - A contiguous segment of physical memory.

17. Static Region - A fixed region of physical memory located in the 0-28K word area. It is created when the program is loaded and it contains the program's base segment. This region cannot be altered by program requests. This region has an identifier of 0.

18. User Mode - One of the modes of the memory management unit hardware. It is the mapping mode for user jobs and KMON. Contrast with kernel mode.

19. Virtual Address - A 16-bit address (0-177777). Under the XM monitor, the memory management unit relocates this address to produce the physical address of the memory location that is to be accessed. (Under the SJ and FB monitors, the virtual address and the physical address of a memory location are the same.)

20. Window - A segment of program virtual address space that begins on a 4K boundary, and can vary in size from 32 to 28K words.

3.3 RT-11 EXTENDED MEMORY FUNCTIONAL DESCRIPTION

The RT-11 software architecture provides programmed requests in the XM monitor that perform the following operations:

1. Divide virtual memory into address windows
2. Allocate regions in extended memory
3. Map the virtual windows to areas within the allocated regions

These three operations are prerequisite to accessing any location in extended memory (above 28K). The first two operations can be performed in any order, but both must be performed before the third operation can take place. A brief description of each operation follows.

3.3.1 Creating Virtual Address Windows

The PDP-11 memory management hardware divides virtual memory into eight pages of 4K (4096) words. The pages are numbered 0 to 7 (see Figure 3-1) corresponding to eight relocation registers. The XM monitor divides virtual memory address space into windows. A window is a segment of address space of any size that must begin on a 4K address boundary. There can be any number of windows up to a maximum of eight (0 to 7). The maximum of eight windows is a compromise between monitor size (seven words per window control block) and allowing enough windows for the user to define eight 4K windows. Windows are similar to overlay segments in that there can be any number of overlay segments, but only one or two are in memory at any given time. Any number of windows can be defined (eight actively defined at a time), but all windows do not have to be mapped at the same time. For example, a multi-user application could segment memory as indicated in Figure 3-2 (example 1). In this figure, the virtual address space is divided linearly. The interpreter remains mapped, but the window containing the user data area is mapped to successive segments of the region. The extended memory region in the example occupies 96K words, which is the largest possible region. If each user is to have a 12K-word data area, as the example shows, there can be up to eight users "sharing" the interpreter at one time. Another example of window usage involves defining several parallel windows of various sizes (see Figure 3-2, example 2) that overlay the same portion of virtual address space.

The size and base of a window is specified by a window definition block supplied by the programmer. Each actively defined window requires a window definition block. The mapping requests must reference the definition block that contains the window specifications, mapping parameters and status information.

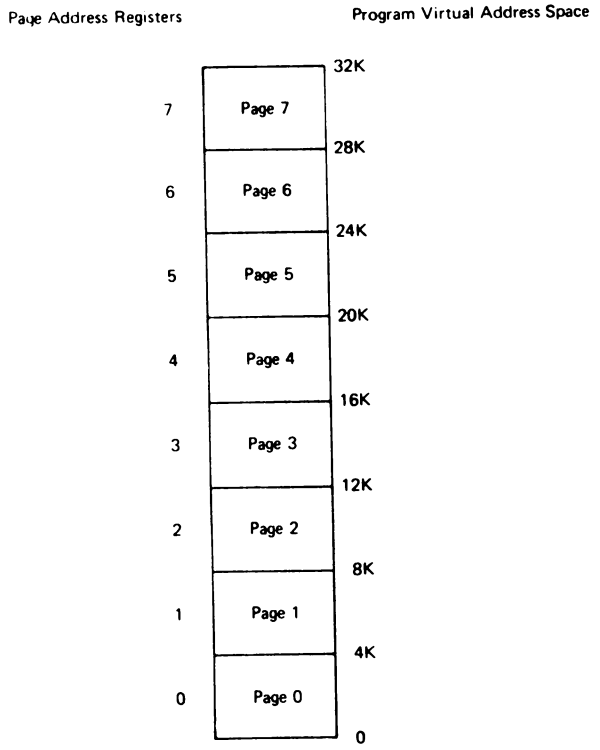
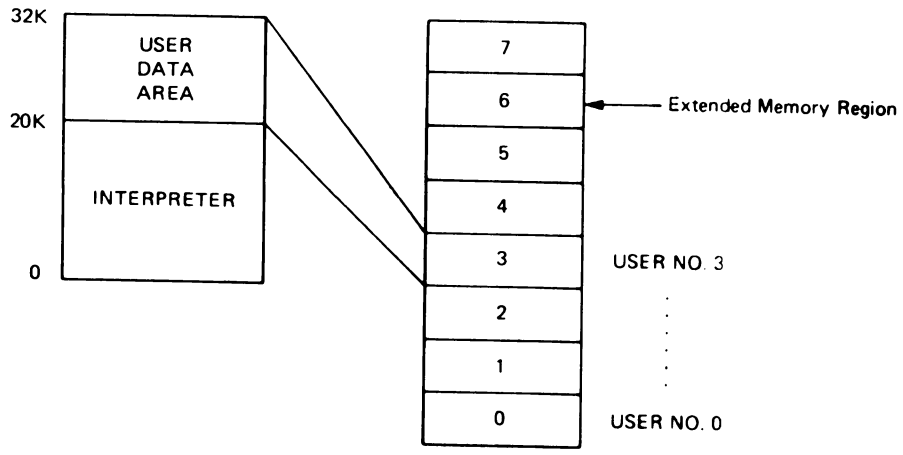
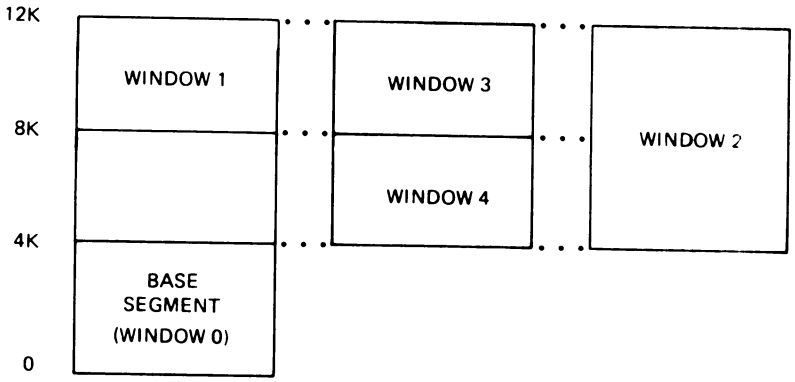


Figure 3-1 Page Address Register Assignments to Program Virtual Address Space Pages



Example 1



Example 2

Figure 3-2 Examples of Window Creation

A window's identification is a number from 0 to 7 that is an index to the window's corresponding window block. The address window identified by 0 is a static window and cannot be changed by programmed requests. This window is automatically created and mapped into the static region by the monitor for virtual programs. Every virtual program contains one static window that maps the program's base segment. The base segment is mapped into its corresponding allocated static region of physical memory when the R or FRUN request is executed.

When a program uses extended memory programmed requests, the program views the relationship between virtual and logical address space in terms of windows and regions. Unless a virtual address is part of an existing address window, the address cannot reference a physical memory location. Similarly, a window can be mapped only to an area that is part of an existing region (see Figure 3-3).

However, privileged jobs (discussed in Section 3.3.4.3) usually have all 32K of virtual address space mapped to the lower 28K and the I/O page. The window 0 concept does not apply to privileged jobs.

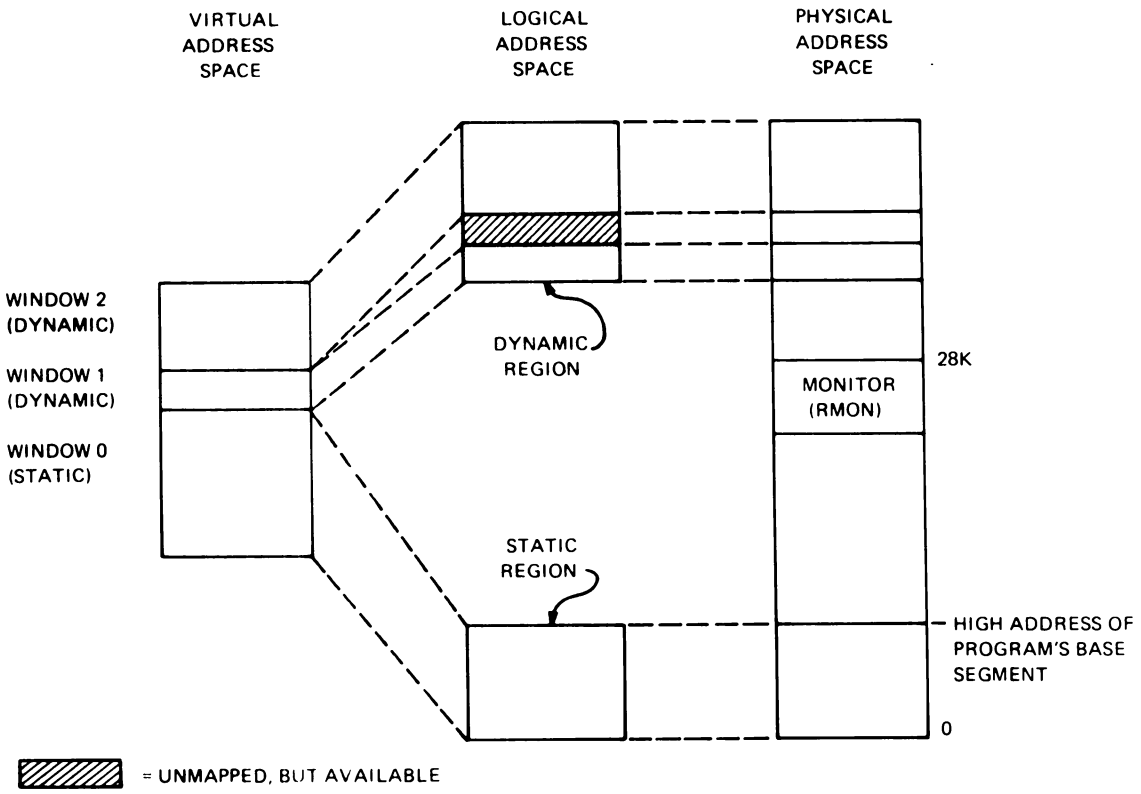


Figure 3-3 Relationship of Windows and Regions

Consider, for example, the case where a program requires two workspace areas (see Figure 3-4): one of 6K words and the other 8K words. The program's base segment requires 8K words. Then, the virtual address space is divided into three windows as follows:

1. Static window, window 0, of 8K words for the base segment
2. Dynamic window of 6K words for workspace area 1
3. Dynamic window of 8K words for workspace area 2

Note that the defined windows overlap page address registers. Window 1 uses page address registers 2 and 3 while window 2 uses registers 4 and 5. Note further that window 1 is only 6K words in size and a discontinuity exists in the program's virtual address space between 14K and 16K. References made to an address in the 14K-16K range cause a memory management fault as long as this discontinuity exists.

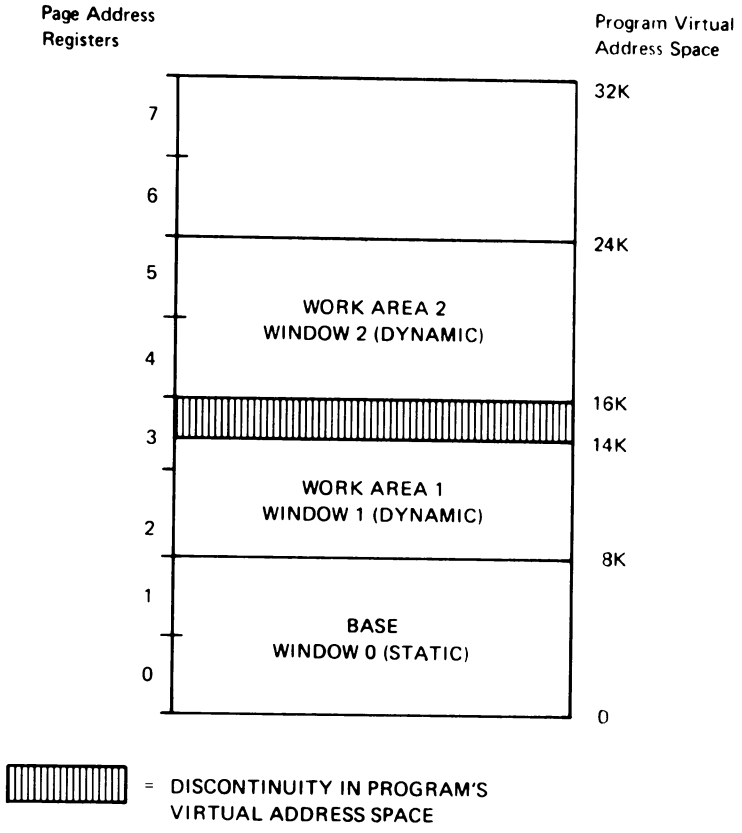


Figure 3-4 Defining Windows for Mapping

This area of undefined virtual address space is produced by the memory management hardware restriction that all windows must begin on a 4K virtual address boundary. In this case, the discontinuity can be avoided by reversing windows 1 and 2. In other situations a linker option can be used to round the window up to a 4K multiple to avoid discontinuities.

Once a program has defined the current windows and regions, it can issue programmed requests to perform operations such as the following:

- Map a window to all or part of a region.
- Unmap a window from one region in order to map it to another region.
- Unmap a window from one part of a region in order to map it to another part of the same region.

3.3.2 Allocating and Deallocating Regions in Extended Memory

Another operation that must be performed before the user can access extended memory is the allocation of dynamic regions. The monitor provides programmed requests that allocate or deallocate dynamic regions. A user program can have up to three of these dynamic regions allocated at any one time. These regions are located in extended memory and do not include the program's base (or static) region

located in the lower 28K of memory. The size of a dynamic region can range up to 96K words in 32 word increments. This convention allows the size to be specified in 16 bits and assures that the regions always begin on a 32-word boundary. When a region is created, a unique region identifier is returned by the monitor and is retained in a 3-word region definition block described later in this chapter. Any subsequent programmed request referring to this region must use the region identification code supplied by the monitor. The current window-to-region mapping assignments determine what part of the program's logical address space can be accessed at any given time. Figure 3-5 illustrates created regions that compose a program's logical address space at a discrete time. Since these are dynamic regions and can be allocated and deallocated several times, the logical address space can increase or decrease in size as a function of the controlling program.

Dynamic region deallocation is also accomplished through programmed requests. When a dynamic region is deallocated (static regions cannot be allocated or deallocated), the extended memory area is returned to the monitor's free list where it can be used by other jobs. At the time a region is deallocated, all windows still mapped to the region are automatically unmapped.

3.3.3 Mapping Windows to Regions

Once the regions and address windows have been defined, the initialization work is complete. The final step in accessing extended memory is to connect the windows in virtual memory to the defined regions of physical memory. This process is referred to as "mapping." As stated earlier, the actual mapping operation is a hardware-implemented function performed by the memory management unit. After software has set up the necessary parameters in descriptor blocks, groups of registers in the memory management hardware relocate the user program address references from virtual to physical memory (see Figure 3-6). It must be understood that the user program cannot directly access extended memory without first mapping a portion of virtual addressing space to the desired portion of physical memory.

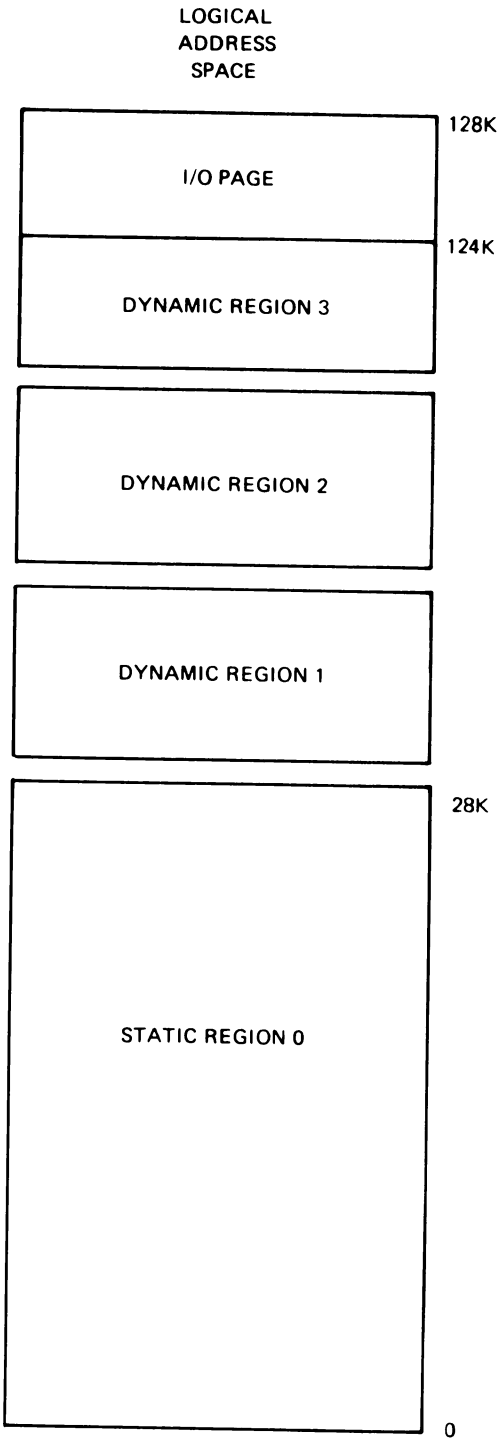


Figure 3-5 Regions Created In Extended Memory

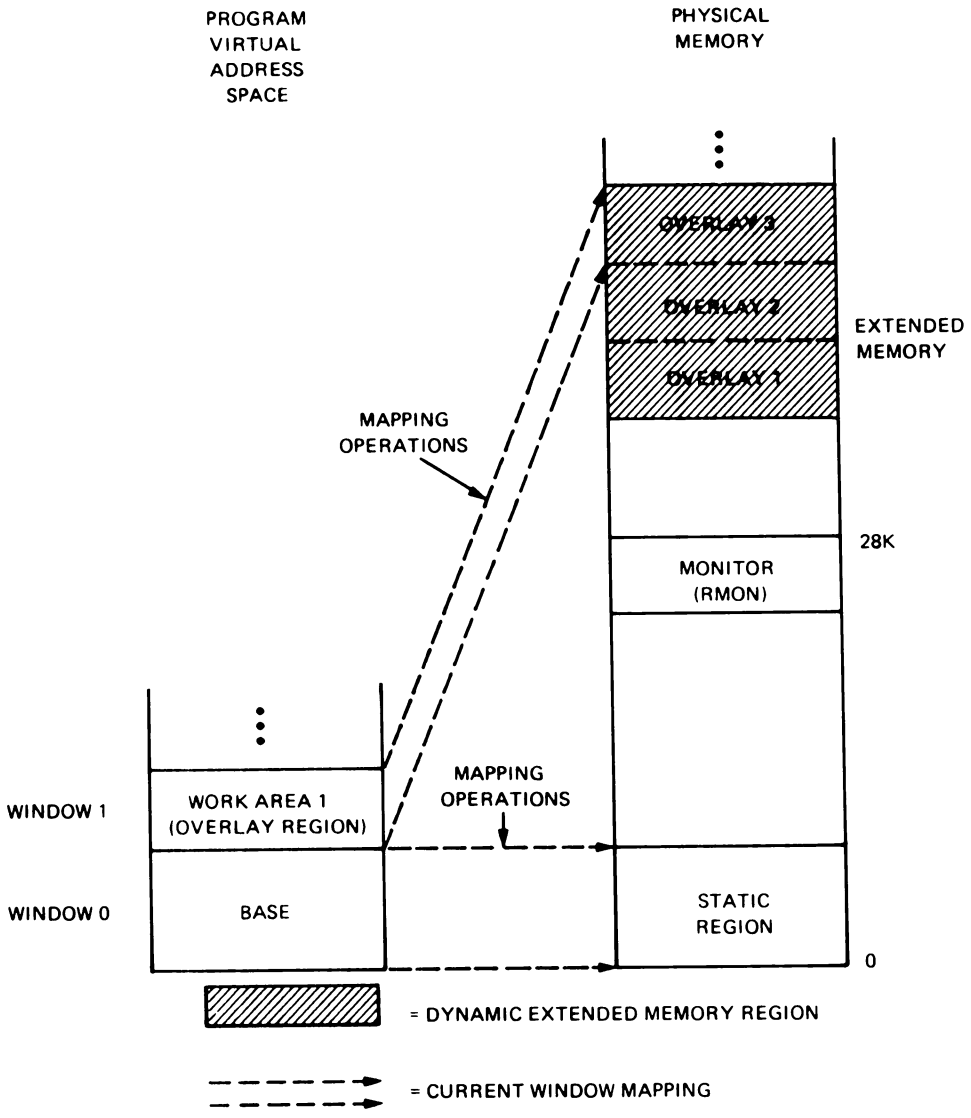


Figure 3-6 Typical Mapping Relationship

The concept of extended memory can be summarized as follows:

1. The user program deals in virtual memory addresses limited to a 16-bit addressing word.

2. A virtual memory address is relocated to an 18-bit physical address capable of accessing 128K words of physical memory.

3. A window of virtual memory can be mapped to successive segments of physical memory by changing offset values through programmed requests.

There are two ways that windows can be mapped to regions. One is to map the window after the creation of that window through a .CRAW (CReate Address Window) programmed request that also performs an implied .MAP programmed request. Under this condition, a window is established and mapped with a single .CRAW request and an additional

programmed request is avoided. However, when mapping previously defined windows, the .MAP programmed request must be used. This request can use the same window definition block that was used in the .CRAW mapping operation to map the associated window into a specified region. An offset into the region must be specified. If the window overlaps the end of the region, the system maps as much of the window as fits in the region.

A window can be unmapped by the .UNMAP programmed request. When a window is unmapped in this manner, for a virtual job, that portion of the program's virtual address space becomes undefined. Further attempts to access this unmapped virtual address space result in a memory management fault.

When a window is unmapped by the .UNMAP programmed request for a privileged job (Section 3.3.4.3) the original mapping arrangment is restored.

For both virtual and privileged jobs, an implicit unmapping operation is performed whenever an existing mapped window is remapped to another region or another part of the same region.

3.3.4 Mapping in the Foreground and Background Modes

Extended memory support is available for foreground and background jobs. Both jobs can use extended memory simultaneously but allocated memory regions are dedicated and cannot be shared by jobs. Memory layout for the XM monitor and the types of mapping that can occur are discussed in the following sections.

3.3.4.1 **Monitor Loading and Memory Layout** - The locations of various system components in the XM monitor are very similar to those in the FB monitor. The monitor is bootstrapped into the high end of the lower 28K of memory (see Figure 3-7). The resident monitor (RMON) executes in kernel mode and maps the lower 28K of memory and the I/O page. The kernel vector space is the lower 256 words of physical memory below the background job. The USR runs mapped in kernel mode and is always memory resident. KMON is a privileged background job and runs in user mode, with the same mapping used by the resident monitor.

3.3.4.2 **Virtual Mapping** - This type of mapping provides the full 32K of virtual space for applications that do not need privileged access to the monitor and I/O page. All of the virtual memory space is available to foreground and background jobs. If the virtual mapping configuration is desired, it must be specified at the time the program is loaded into memory. This is done by setting bit 10 of the JSW in location 44 of the system communication area. The user must do this with an .ASECT at assembly time or with a patch prior to run time. The partition where the job is installed is mapped starting at user virtual address 0. The first 500 bytes are the virtual vector and system communication area for the job. Window 0 maps from virtual 0 to the program's high address. Any remaining address space from the program's high address up to 32K is available for mapping into extended memory. Region 0 is defined to include the area from physical location 0 to program partition high limit.

When a virtual foreground job is loaded, it is installed below the resident monitor. The foreground job is mapped to appear as a virtual background job. The program is linked at a default base of 1000 and the region from 0 to 500 is the system communication area and pseudo vector space for the foreground. The job header (impure area) is located just below the foreground job but is not mapped. Unused address space above the foreground job's high limit can be used to define windows so that the job can access extended memory.

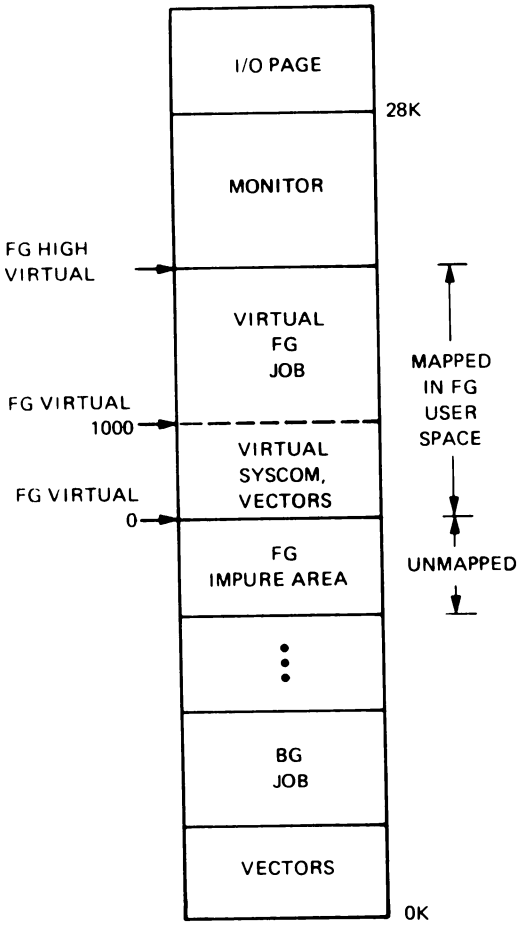


Figure 3-7 Memory Map with Virtual Foreground Job Installed

3.3.4.3 **Privileged or Compatibility Mapping** - This type of mapping is the default mapping that provides compatibility between the FB monitor and the XM monitor (see Figure 3-8). This mapping arrangement gives free access to vectors, monitor and the I/O page. It also is the default mode under which all RT-11 system programs run. In the privileged or compatibility mapping arrangement, the program is normally mapped to the lower 28K of memory plus the I/O page. However, provisions are made through programmed requests for windows to be created and mapped to regions allocated in extended memory so that the effective program space can be increased beyond the virtual address space. In this arrangement, when a window is created and mapped, the default privileged mapping for this space set up by the monitor is temporarily unmapped and the address space is mapped through the window definition block to the new region of memory. Then when the window is unmapped, that address space is returned to the

privileged mapping. This type of mapping is particularly important for the user who desires to include interrupt service routines in the system. Interrupt service routines must run in kernel mode. They depend on privileged mapping being identical to kernel mapping.

The privileged job requiring access to the monitor, vectors, and I/O page is limited in the amount of virtual address space it has available to map to extended memory. The user must select portions of the address space that can be borrowed for memory extension operations. If user interrupt service routines are part of the program, the vectors, I/O page, user interrupt service routines, and possibly the monitor must remain mapped at all times that an interrupt can occur.

3.3.4.4 **Context Switching of Virtual and Privileged Jobs** - The two types of jobs (privileged and virtual) are context switched. When the monitor switches between a virtual and a privileged job, it saves context information about the job it switches out, and restores context information about the job it switches in. When a job is switched out, the contents of the memory management mapping registers for the job are not saved. User programs should not manipulate these registers directly because their contents are lost when context switching occurs. The monitor restores job mapping solely from the window and region definition blocks.

When a virtual job is switched in, the monitor disables all user mapping and scans the job's window definition and mapping data. The monitor maps only that portion of the job's virtual address space that was defined in a window and mapped to a region at the time the job was switched out. Any attempt to access an unmapped address causes a memory management fault. Any unused portions of virtual address space remain unmapped and discontinuities can appear. The virtual job can use the unmapped space by allocating a region in extended memory and mapping to that region.

When a privileged job is switched in, the monitor first sets up the job's user mapping to be identical to kernel mapping (the lower 28K of physical memory and the I/O page). Next, the monitor scans the job's window definition and mapping data. If no windows had been defined at the time the job was switched out, the default kernel mapping remains. If windows had been defined and mapped, those mappings selectively replace the default kernel mapping for the privileged job.

NOTE

User programs should never attempt to access the memory management unit mapping registers directly. These registers should always be addressed through the appropriate programmed requests.

3.3.5 I/O to Extended Memory

The monitor supports I/O within a job's virtual address space regardless of the physical location of the data buffers. However, the buffers must be in a segment of logical space currently mapped at the time a .READ or .WRITE request is issued. The buffers must also be physically contiguous, which implies that they be completely within an

address window. This restriction is necessary because I/O buffer addresses are specified as virtual addresses in the program. The monitor converts the virtual address to an internal physical address representation when the programmed request is executed. This process allows the user program to unmap the buffers on a .READ/.WRITE or .READC/.WRITEC request upon return from the programmed request. Note however, that completion routines must remain mapped until the transfer is complete.

3.4 SUMMARY OF PROGRAMMED REQUESTS

This section briefly describes each of the RT-11 extended memory programmed requests and its associated data structures, arguments and parameters. For convenience, the following requests are ordered by functions and alphabetized within these functional groupings.

Window Requests

.CRAW - Create an address window (3.4.1.3)
.ELAW - Eliminate an address window (3.4.1.4)

Region Requests

.CRRG - Create a region (3.4.2.3)
.ELRG - Eliminate a region (3.4.2.4)

Mapping Requests

.GMCX - Get mapping status (3.4.3.1)
.MAP - Map an address window (3.4.3.2)
.UNMAP - Unmap an address window (3.4.3.3)

The extended memory programmed requests are individually capable of performing a number of separate actions. For example, a single Create an Address Window (.CRAW) request can unmap and eliminate conflicting address windows, create a new window, and then map the new window to a specified region. The complexity of the requests requires a special means of communication between the user program and the RT-11 monitor. The communication is achieved through data structures that:

1. Allow the program to specify which options it wants the monitor to perform.

2. Permit the monitor to provide the job or program with details about the outcome of the requested actions.

Two types of data structures, the region definition block and the window definition block, are used by the requests to provide information to the XM monitor and to receive information from it. Every extended memory programmed request uses one of these structures as its communication area between the job and the monitor. Each issued request includes in the programmed request parameter block a pointer to the appropriate definition block. Values stored by the job in a block define or modify the requested operation. After the monitor has carried out the specified operation, it returns values in various locations within the block to describe the actions taken and to provide the program with information useful for subsequent operations.

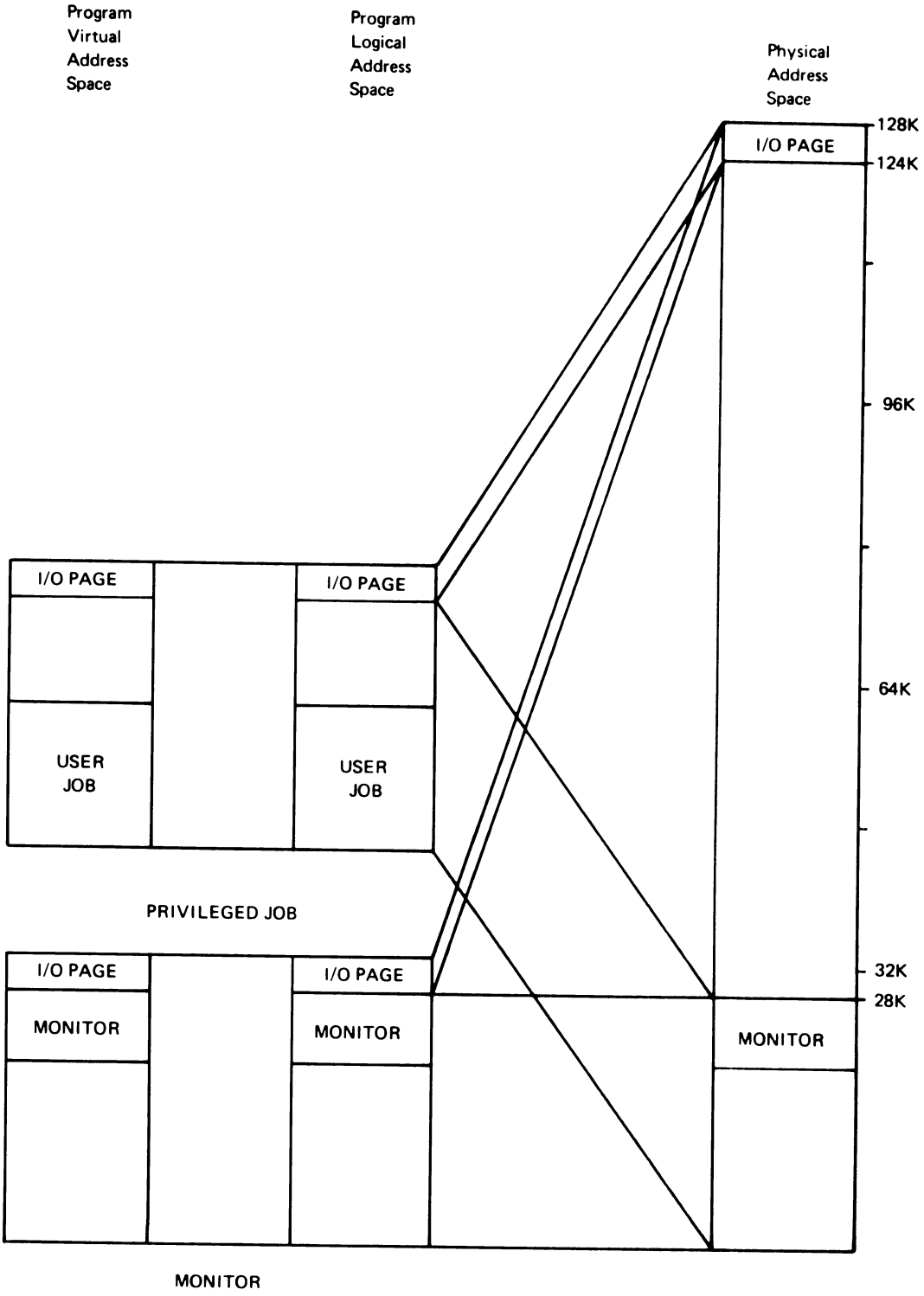


Figure 3-8 RT-11 Privileged Mapping

3.4.1 Programmed Requests to Manipulate Windows

All programmed requests described in this section have a common user data structure, called a window definition block, which is used to store information for the XM monitor and to receive information from it. To use these programmed requests, the window definition block must be defined and set up according to the rules explained in the following section.

3.4.1.1 Window Definition Block - The group of programmed requests to manipulate windows must specify a pointer to the window definition block. The window definition block (see Figure 3-9) is used to define a window and store the returned window identification. It can be created at assembly time by the macro, .WDBBK.

The format of the window definition block is a seven-word block as shown in Figure 3-9.

SYMBOLIC OFFSET	BLOCK FORMAT		BYTE OFFSET
W.NID W.NAPR	BASE PAR	WINDOW ID	0
W.NBAS	VIRTUAL BASE ADDRESS (BYTES)		2
W.NSIZ	WINDOW SIZE (32W BLOCKS)		4
W.NRID	REGION ID		6
W.NOFF	OFFSET IN REGION (32W BLOCKS)		10
W.NLEN	LENGTH TO MAP (32W BLOCKS)		12
W.NSTS	WINDOW STATUS WORD		14

Figure 3-9 Window Definition Block

The first three words are used to establish the window and contain the following information:

W.NID is a one-byte window identifier code returned by the monitor. This identifier must be used in mapping requests involving this window.

W.NAPR is a one-byte value supplied by the user specifying the starting virtual address of the window. Windows must start on a 4K virtual address boundary. The one-byte value is a digit in the range 0 through 7. The digit is the page address register corresponding to the desired 4K virtual address (see Table 3-1). Refer to Figures 3-1 and 3-4, which illustrate the page address registers.

Table 3-1
Virtual Address Boundaries

Starting Virtual Address (Octal)	Page Address Register (W.NAPR)
0 (0K words)	0
20000 (4K words)	1
40000 (8K words)	2
60000 (12K words)	3
100000 (16K words)	4
120000 (20K words)	5
140000 (24K words)	6
160000 (28K words)	7

NOTE

A value of 0 should not be used in W.NAPR since 0 cannot be specified in a .CRAW request.

W.NBAS is the base virtual address of this window, returned by the monitor. This information is redundant with the W.NAPR field, but is provided for user convenience and as a check on the window specification.

W.NSIZ is a one-word value supplied by the user specifying the size of the window in 32-word blocks. If it is not a multiple of 4K words, a discontinuity occurs in the virtual address space, since the next window definition must start on a 4K boundary.

The remaining fields of the window definition block are provided for mapping the window to a region. This same window block can be used with the .MAP request, and since the .CRAW request returns window data in its proper place, an extra operation is avoided. The region-specific data fields returned by the monitor to the window block are as follows:

W.NRID is the region identifier for the region to be mapped, as returned by the .CRRG request.

W.NOFF is the offset into the region at which to start mapping the window, in blocks of 32 words.

W.NLEN is the length of the window to map to the region, in 32-word blocks. If it is 0, the entire window is mapped, or as much as will fit into the region. If W.NLEN is specified, that length portion of the window is mapped. The actual length of the window mapped is returned in W.NLEN.

In addition to creating a window, the .CRAW request is capable of creating a window and then mapping that window to a region by specifying the proper W.NSTS field as described below.

W.NSTS is the window status word.

The window status bits are defined as follows:

Input

WS.MAP Map the window to the specified region after creating it, thus saving an explicit .MAP request.

Output

WS.CRW Address window was successfully created.

WS.UNM One or more windows were unmapped to create and map this window.

WS.ELW One or more windows were eliminated.

3.4.1.2 **Using Macros to Generate Window Definition Blocks** - There are two macros used to generate window definition blocks. The macro .WDBDF defines the offsets and status word bits for the window definition block. The second macro, .WDBBK, actually creates a window definition block. When creating a window definition block with the .WDBBK macro, the offset and status word definitions are automatically supplied because the .WDBBK macro invokes the .WDBDF macro. Hence, the programmer does not need to specify the .WDBDF macro when a .WDBBK is being used. The .WDBBK macro has the following form:

```
.WDBBK wnapr,wnsiz[,wnrid,wnoff,wnlen,wnsts]
```

where:

wnapr is the page address register supplied by the user specifying the starting virtual address of the window (see Table 3-1).

wnsiz specifies the size of the window, in 32-word blocks.

wnrid is the region identifier for the region to be mapped.

wnoff is the offset into the region at which to start mapping the window, in 32-word blocks.

wnlen is the length of the window to map to the region, in 32-word blocks. A value of 0 maps as much of the window as possible.

wnsts is the window status word.

When it is desired only to define the offsets and status bits the .WDBDF macro is invoked by the following call:

```
.WDBDF
```

When this macro is invoked, the following symbols are defined:

1. Window Definition Block Offsets

```
W.NID  = 0
W.NAPR = 1
W.NBAS = 2
W.NSIZ = 4
W.NRID = 6
W.NOFF = 10
W.NLEN = 12
W.NSTS = 14
```

2. Window Definition Block Byte Size

```
W.NLGH = 16
```

3. Window Definition Block Status Word Bits

```
WS.CRW = 100000
WS.UNM =  40000
WS.ELW =  20000
WS.MAP =    400
```

To illustrate the use of these macros to create a window definition block, consider the following example:

A window definition block is to be created defining a window that is 76 decimal blocks long (76 x 32, or 2432 decimal words long) beginning at virtual address 20K, or 120000 octal. Page address register 5 is used.

The defined window is to be mapped to a region beginning 50 decimal blocks (1600 decimal words) from the base of the region. The portion of the region mapped is to be equal to the length of the window or the length remaining in the region, whichever is smaller.

Macro Call: `.WDBBK 5,76.,,50.,,WS.MAP`

Expands to:

```
        .BYTE   0,5     ;window ID = 0, to be
                        ;returned by monitor.
                        ;window starts at 20K,
                        ;uses address register 5.
        .WORD   0       ;base virtual address of
                        ;window, to be returned
                        ;by monitor.
        .WORD   76.     ;window size in 32-word
                        ;blocks.
        .WORD   0       ;region ID, to be returned
                        ;by .CRRG request
                        ;into the region definition block.
        .WORD   50.     ;offset into region, in 32-word
                        ;blocks, at which to start
                        ;mapping the window.
        .WORD   0       ;length of window to map.
                        ;0 = map as much as possible.
                        ;actual length mapped is
                        ;returned here.
        .WORD   400     ;window status word; 400
                        ;causes .CRAW to also map.
```

Note that setting up the window definition block does not in itself create the window. The .CRRG request must be issued to create the region and return the region ID to the region definition block. If the .CRAW request is to perform an implied .MAP, the region ID must be moved from the region definition block to the window definition block. Then the .CRAW request must be issued to create the window.

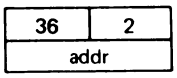
.CRAW

3.4.1.3 **Create an Address Window (.CRAW)** - This request defines a virtual address window and optionally maps it into a physical memory region. Mapping occurs if the user has set the WS.MAP bit in the last word of the window definition block. Since the window must start on a 4K boundary, the program only has to specify the page address register to use and the window size in 32-word increments. If the new window being defined overlaps previously defined windows (except window 0, the static window reserved for the virtual program's base segment), the previously defined windows are eliminated before the new window is created.

Macro Call: .CRAW area[,addr]

where:

area is the address of a two-word argument block as indicated below.



addr is the address of the window definition block (see Section 3.4.1.1). This argument is optional if the user has filled in the second word of the area argument block with the address pointer.

Errors:

When errors are detected during the execution of this request, the C bit is set after execution is complete and the following error codes are contained in error byte 52.

Codes: 0 - Indicates a window alignment error. The window is too large or W.NAPR is greater than 7.

1 - Indicates that no window control blocks are available. The user should eliminate a window first or redefine the division of virtual space into windows so that no more than seven are required.

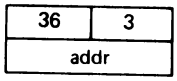
.ELAW

3.4.1.4 **Eliminate an Address Window (ELAW)** - This request eliminates a defined window. An implied unmapping of the window occurs when its definition block is eliminated.

Macro Call: .ELAW area[,addr]

where:

area is the address of a two-word argument block as indicated below:



addr is the address of the window definition block for the window to be eliminated.

Errors:

When errors are detected during the execution of this request, the C bit is set after execution is complete and the following error code is contained in error byte 52 (refer to Section 3.5 for explanation).

Code: 3 - Indicates an illegal window identifier was specified.

3.4.2 Programmed Requests to Manage Extended Memory Regions

As in the case of the programmed requests to manipulate windows (section 3.4.1), all programmed requests in this section also have a common user data structure, the region definition block. To use these programmed requests, the region definition block must be defined and set up according to the rules and syntax explained in the following section.

3.4.2.1 **Region Definition Block** - The programmed requests to manage extended memory regions must specify a pointer to the region definition block. The region definition block is a three-word block describing the region and having the format shown in Figure 3-10.

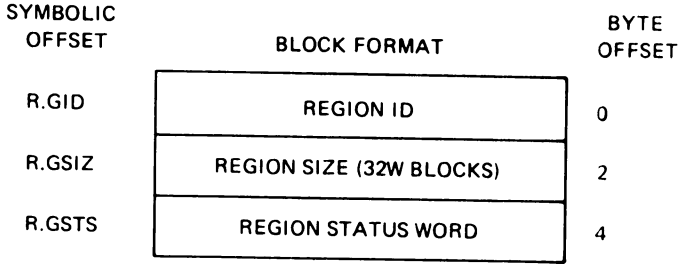


Figure 3-10 Region Definition Block

The words contain the following information:

R.GID — is a unique region identifier returned by the monitor. This identifier must be used when referring to the region in other program requests.

R.GSIZ — is the size of the dynamic region, in 32-word blocks, specified by the user.

R.GSTS — is the region status word. The region status bits are defined as follows:

RS.CRR = 1 — if region was successfully created.

RS.UNM = 1 — if one or more windows were unmapped as a result of eliminating this region.

RS.NAL = 1 — if the region specified was not actually allocated at this time.

3.4.2.2 **Using Macros to Generate Region Definition Blocks** - There are two macros used to generate region definition blocks. The first macro, .RDBDF, defines the offsets and status word bits for the region definition blocks. This macro is invoked with the following call:

```
.RDBDF
```

When the macro is invoked, the following symbols are defined:

1. Region Definition Block Offsets

```
R.GID  = 0
R.GSIZ = 2
R.GSTS = 4
```

2. Region Definition Block Byte Size

```
R.GLGH = 6
```

3. Region Status Word Bits

```
RS.CRR = 100000
RS.UNM =  40000
RS.NAL =  20000
```

The second macro, .RDBBK, actually creates the region definition block. The .RDBBK macro has the following form:

```
.RDBBK    rgsiz
```

where:

rgsiz — is the size of the dynamic region, in 32-block words, specified by the user.

When the region definition block is created with the .RDBBK macro, the region definition block offsets and status word are automatically defined. Therefore, the programmer only needs to specify .RDBBK and this macro automatically invokes .RDBDF.

For example, consider the following case. A region of 102 decimal blocks (3264 decimal words) is to be allocated.

The .RDBBK macro sets up the region definition block.

```
            RGADR:      .RDBBK   #102.

Expands to: RGADR:      .WORD  0     ;region ID=0, to be
                                     ;returned by the
                                     ;monitor.
                        .WORD 102.   ;size of the region
                                     ;in 32-word blocks.
                        .WORD 0      ;region status word.
```

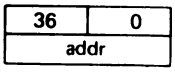
.CRRG

3.4.2.3 **Create a Region (.CRRG)** - The .CRRG request directs the monitor to allocate a dynamic region in physical memory for use by the current requesting program. Symbolically, this request is defined as follows:

Macro Call: .CRRG area [,addr]

where:

area — is the address of a two-word argument block as indicated below:



addr — is the address of the region definition block for the region to be created.

Errors:

When errors are detected during the execution of this request, the C bit is set after execution is complete and the following error codes are contained in error byte 52.

Codes:
- 6 - Indicates no region control blocks available. A region must be eliminated.
- 7 - Indicates a region of the requested size cannot be created. The size of the largest available region is returned in R0.
- 10 - Indicates an illegal region size specification. Requests of 0 size and >96K words are illegal.

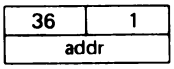
.ELRG

3.4.2.4 **Eliminate a Region (.ELRG)** - The .ELRG request directs the monitor to eliminate a dynamic region in physical memory and return it to the free list where it can be used by the other jobs.

Macro Call: .ELRG area [,addr]

where:

area is the address of a two-word argument block as indicated below:



addr is the address of the region definition block for the region to be eliminated. Windows mapped to this region are unmapped. The static region (region 0) cannot be eliminated.

Errors:

When errors are detected during the execution of this request, the C bit is set after execution is complete and the following error code is contained in error byte 52.

Code: 2 - Indicates an illegal region identifier was specified.

3.4.3 Mapping Requests

The mapping requests explained in this section map virtual address windows into dynamic regions in extended memory. To perform this function, the rules and syntax described in the following sections must be followed.

.GMCX

3.4.3.1 **Mapping Status (.GMCX)** - The .GMCX request returns the mapping status of a specified window. Status is returned in the window definition block, and can be used in a subsequent mapping operation. Since the .CRAW request permits combined window creation and mapping operations, it allows entire windows to be changed by modifying certain fields of the window definition block.

The .GMCX request modifies the following fields of the window definition block:

1. W.NAPR - the base page address register of the window
2. W.NBAS - the window base virtual address
3. W.NSIZ - the window size in 32-word blocks

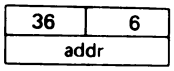
If the window whose status is requested is mapped to a region, the .GMCX request modifies the following additional fields in the window definition block:

1. W.RID - the region identifier
2. W.NOFF - the offset value into the region
3. W.NLEN - the actual length of the mapped window
4. W.NSTS - the state of the WS.MAP bit is set to 1 in the window status word.

Macro Call: .GMCX area[,addr]

where:

area is the address of a two-word argument block as indicated below:



addr is the address of the window definition block where the specified window's status is returned.

Errors:

When errors are detected during the execution of this request, the C bit is set after execution is complete, and the following error code is contained in error byte 52.

Code: 3 - Indicates an illegal window identifier was specified.

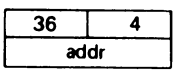
.MAP

3.4.3.2 **Map a Window (.MAP)** - The .MAP request maps a previously defined address window into a dynamic region of extended memory or into the static region in the lower 28K. If a window is already mapped to a region, an implicit unmapping operation is performed.

Macro Call: .MAP area[,addr]

where:

area is the address of a two-word argument block as indicated below:



addr is the address of the window definition block containing a description of the window to be mapped and the region to be mapped to (see Section 3.4.1.1).

Errors:

When errors are detected during the execution of this request, the C bit is set and the following error codes are contained in error byte 52.

Codes: 2 - Indicates an illegal region identifier was specified.

3 - Indicates an illegal window identifier was specified.

4 - Indicates the specified window was not mapped.

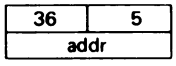
.UNMAP

3.4.3.3 **Unmap a Window (.UNMAP)** - The .UNMAP request unmaps a window and flags that portion of the program's virtual address space as being inaccessible. When an unmap operation is performed for a virtual job, attempts to access the unmapped address space cause a memory management fault. For a privileged job, the default (kernel) mapping is restored when a window is unmapped.

Macro Call: .UNMAP area[,addr]

where:

area is the address of a two-word argument block as indicated below:



addr is the address of the window control block that describes the window to be unmapped.

Errors:

When errors are detected during the execution of this request, the C bit is set and the following error codes are contained in error byte 52.

Codes: 3 - Indicates an illegal window identifier was specified.

5 - Indicates the specified window was not mapped.

3.5 SUMMARY OF STATUS AND ERROR MONITORING

The XM monitor performs error checking and status monitoring. All extended memory programmed requests generate error codes as indicated in Table 3-2. When errors are detected, the C bit is set on return from the program request, and the error code is returned in error byte

52. In addition to the error codes, two status words are provided to log the status of the requested operations. After completing the requested operation, the monitor sets appropriate bits in the region status word or the window status word (depending on the type of request) to indicate what actions were taken. These status words were discussed in conjunction with the window and region definition blocks. Table 3-3 provides a convenient summary of the byte 52 error codes and status word bits.

3.6 USER INTERRUPT SERVICE ROUTINES WITH THE XM MONITOR

There are three restrictions to using user interrupt service routines with the XM monitor. Such routines can only be used within a privileged job, they must be resident in the lower 28K words of memory, and they must be permanently mapped while they are active.

Care must be used in locating buffers and in setting up vectors for these routines. When an interrupt occurs, the interrupt vector is always taken from kernel space. In XM, kernel space always maps the lower 28K words of memory and the I/O page. The contents of the interrupt vector are placed in the PC and PS, causing the interrupt service routine to execute in the mapping mode specified in the PS of the interrupt vector.

It is possible to execute an interrupt service routine in either mode: kernel or user. However, due to protection mechanisms in the mapping hardware, it is impossible to go from user mode to kernel mode when dismissing an interrupt with an RTI instruction. Consequently, if an interrupt service routine is executed in user mode, it is impossible to return to kernel mode. This guarantees a system crash if the interrupt has interrupted the monitor. Therefore, all interrupt service routines must be serviced in kernel mode (that is, the high byte of the second word of the vector pair must be zero). The interrupt service routine will then execute in kernel mode. This is normally no problem, since privileged job mapping defaults to kernel mode. Thus, old programs that ran under RT-11 version 2C or earlier versions should function properly.

Privileged jobs can also use the memory extension programmed requests. However, the portion of user virtual memory mapped to extended memory at the time of the interrupt is not accessible to the interrupt service routine. This is why the interrupt service routine must use addresses that are permanently mapped in the lower 28K words of memory.

Table 3-2
Extended Memory Error Codes

Error Code	REQUESTS							CAUSE
	.CRAW	.CRRG	.ELAW	.ELRG	.GMCX	.MAP	.UNMAP	
0	x							Window alignment error. Window is too large or W.NAPR is greater than 7, or illegally specified window.
1	x							Attempt to create more than eight windows. Unmap a window first or redefine the division of virtual space into windows.
2				x		x		Illegal region identifier specified.
3			x		x	x	x	Illegal window identifier specified.
4						x		Invalid region offset/window size combination.
5							x	Specified window was not mapped.
6		x						Attempt to create more than four regions. A region must be eliminated.
7		x						A region of the requested size cannot be created. The size of the largest available region is returned in R0.
10		x						Illegal region size specification. A size of 0 or a size greater than 96K words was requested.

Table 3-3
Extended Memory Status Words

Status Word	Symbolic Name	Bit Name	Bit Number	Input/Output*	Bit (octal)	Comments/Definition
Region status word	R.GSTS	RS.CRR	15	Output	100000	Set to 1 for successful region allocation.
		RS.UNM	14	Output	40000	Set to 1 if one or more windows were unmapped as a result of eliminating a region.
		RS.NAL	13	Output	20000	Set to 1 if region specified was not allocated at this time.
Window status word	W.NSTS	WS.CRW	15	Output	100000	Set to 1 if address window was successfully created.
		WS.UNM	14	Output	40000	Set to 1 if one or more windows were unmapped by a .CRAW or a .MAP request.
		WS.ELW	13	Output	20000	Set to 1 if one or more windows were eliminated in a .CRAW request.
		WS.MAP	8	Input	400	Set to 1 if a window is to be mapped in a .CRAW request.

* Input by user or output by monitor.

3.7 EXAMPLE PROGRAM

This section provides a complete and detailed MACRO listing of a sample program that uses all the RT-11 extended memory programmed requests.

```
        .TITLE  XMCOPY
        .NLIST  BEX
        .MCALL  .UNMAP,.ELRG,.ELAW
        .MCALL  .CRRG,.CRAW,.MAP,.PRINT,.EXIT
        .MCALL  .RDBBK,.WDBBK,.TTYOUT,.WDBDF,.RDBDF
        .MCALL  .WRITW,.READW,.CLOSE,.CSIGEN
        JSW     =       44
        J.VIRT  =       2000    ;VIRTUAL BIT IN THE JSW
        ERRBYT  =       52
        APR     =       1
        APR1    =       2
        BUF     =       WDB+W.NBAS  ;GET THE VIRTUAL ADDRESS
        BUF1    =       WDB1+W.NBAS ;GET SECOND BUFFER
                                    ; VIRTUAL ADDRESS
        CORSIZ  =       4096.
        PAGSIZ  =       CORSIZ/256.
        WRNID1  =       WDB1    +       W.NRID
        WRNID   =       WDB     +       W.NRID

        .ASECT
      . = JSW
        .WORD   J.VIRT          ;MAKE THIS A VIRTUAL JOB
        .PSECT

;*************************************************
;* XM MONITOR EXAMPLE -- OPENS INPUT FILE AND    *
;* WRITES TO OUTPUT FILE USING 4K BUFFERS IN     *
;* EXTENDED MEMORY.FILES ARE VERIFIED AFTER      *
;* COPYING,TWO 4K BUFFERS IN EXTENDED MEMORY     *
;* ARE USED IN THE VERIFICATION.                 *
;*************************************************
        .WDBDF          ;CREATE WINDOW DEFINITION BLOCK SYMBOLS
        .RDBDF          ;CREATE REGION DEFINITION BLOCK SYMBOLS

START::
5$:     .CSIGEN #ENDCRE,#DEFLT,#0
        BCS     5$
        .CRRG   #CAREA,#RDB     ;CREATE REGION
        BCC     10$
        JSR     PC,ERROR        ;ERROR REPORT IT
10$:    MOV     RDB,WRNID       ;MOVE REGION ID
        .CRAW   #CAREA,#WDB     ;CREATE WINDOW
        BCC     20$             ;NO ERROR
        JSR     PC,ERROR
20$:    .MAP    #CAREA,#WDB-    ;MAP WINDOW
        BCC     30$             ;NO ERROR
        JSR     PC,ERROR        ;REPORT ERROR
30$:    CLR     R1              ;COUNT REG.
READ:   .READW  #RAREA,#3,BUF,#CORSIZ,R1
        BCC     LOOP            ;NO ERROR
        JSR     PC,ERROR
LOOP:   CMP     R0,#CORSIZ      ;SHORT READ
        BNE     CLOSE           ;CLOSE FILE,SHORT READ
WRITE:  .WRITW  #RAREA,#0,BUF,#CORSIZ,R1
        BCC     ADDIT           ;NO ERROR
        JSR     PC,ERROR
ADDIT:: ADD     #PAGSIZ,R1      ;GO GET NEXT BLOCK
        BR      READ            ;READ LOOP
```

```
CLOSE::  MOV     R0,R2              ;SAVE NUMBER OF WORDS
         .WRITW  #RAREA,#0,BUF,R2,R1 ;WRITE LAST BLOCK
         BCC     CHECK              ;NOW VERIFY DATA
         JSR     PC,ERROR           ;ERROR REPORT IT
CHECK::  .CRRG   #CAREA,#RDB1       ;CREATE A REGION
         BCC     35$                ;NO ERROR HERE
         JSR     PC,ERROR           ;REPORT ERROR
35$:     MOV     RDB1,WRNID1        ;GET REGION ID TO WINDOW

;*************************************************
;* EXAMPLE USING THE .CRAW REQUEST DOING AN      *
;* IMPLIED .MAP REQUEST.                         *
;*************************************************
         .CRAW   #CAREA,#WDB1       ;CREATE WINDOW USING
                                    ; IMPLIED .MAP
         BCC     VERIFY             ;CHECK THE DATA
         JSR     PC,ERROR           ;ERROR REPORT IT
VERIFY:: CLR     R1                 ;COUNT REGISTER
GETBLK:  .READW  #RAREA,#0,BUF,#CORSIZ,R1
         BCC     40$
         JSR     PC,ERROR
40$:     .READW  #RAREA,#3,BUF1,#CORSIZ,R1
         BCC     50$                ;NO ERROR
         JSR     PC,ERROR           ;ERROR REPORT IT
50$:     CMP     R0,#CORSIZ         ;IS IT A SHORT READ
         BNE     60$                ;NO IT WASNT
         MOV     #CORSIZ,R4         ;REGULAR BUFFER SIZE
         BR      65$                ;GO VERIFY DATA
60$:     MOVB    #-1,SFLAG          ;SET SHORT BUFFER FLAG
         MOV     R0,R4              ;GET SHORT BUFFER
65$:     MOV     BUF,R2             ;GET BUFFER ADDRESS
         MOV     BUF1,R3            ;GET NEXT BUFFER
70$:     CMP     (R2)+,(R3)+        ;VERIFY DATA
         BEQ     75$                ;DATA IS THE SAME
         JSR     PC,ERRDAT
75$:     DEC     R4                 ;ARE WE FINISHED
         BNE     70$                ;NO WE ARENT
         ADD     #PAGSIZ,R1         ;GET NEXT PAGE
         TSTB    SFLAG              ;HAS SHORT BUFFER BEEN READ
         BMI     ENDIT              ;YES IT HAS
         BR      GETBLK
ENDIT::  .CLOSE  #0                 ;CLOSE CHAN 0
         .CLOSE  #3                 ;CLOSE CHANNEL 3
         .PRINT  #ENDPRG
         .EXIT
;*************************************************
;* EXAMPLES SHOWING THE COMPLEMENTS OF THE .MAP  *
;* .CRRG, AND THE .CRAW REQUESTS.                *
;*************************************************
         .ELRG   #CAREA,#RDB        ;ELIMIMATE A REGION,
                                    ; IMPLIES ELAW AND UNMAP
         .UNMAP  #CAREA,#WDB1       ;UNMAP WINDOW
         .ELAW   #CAREA,#WDB1       ;ELIMINATE A WINDOW
         .ELRG   #CAREA,#RDB1       ;ELIMINATE A REGION
ERROR:   .PRINT  #ERR
         .EXIT
ERRDAT:  .PRINT  #ERRBUF
         .EXIT
RDB:     .RDBBK  CORSIZ/32.         ;DEFINE REGION
WDB:     .WDBBK  APR,CORSIZ/32.
RDB1:    .RDBBK  CORSIZ/32.         ;DEFINE SECOND REGION
WDB1:    .WDBBK  APR1,CORSIZ/32.,,0,0,CORSIZ/32.,,WS.MAP
CAREA:   .BLKW   2
RAREA:   .BLKW   6
```

```
ENDPRG: .ASCIZ  /END OF EXAMPLE/
ERR:    .ASCIZ  /ERROR IN REQUEST/
ERRBUF: .ASCIZ  /ERROR IN DATA/
        .EVEN
DEFLT:  .RAD50  /MAC/
SFLAG:  .BLKB   0               ;SHORT BUFFER FLAG
        .EVEN
ENDCRE  =       .+2
        .END    START
```

3.8 EXTENDED MEMORY RESTRICTIONS

There are some restrictions that the user of RT-11 extended memory support must be aware of. Some restrictions are physical in nature and imposed by hardware limitations. These restrictions are generally discussed in the descriptions of the applicable programmed requests. Other restrictions are on the use of the system facilities and are discussed below:

1. Device handlers to be used under the XM monitor must be loaded into memory through the keyboard monitor LOAD command before they can be used. User interrupt service routines are not supported for virtual jobs.

2. Some programmed requests are restricted when used with the XM monitor. The requests and their restrictions are as follows:

Programmed Request	Restriction
.CDFN	The channel area specified must be entirely in the lower 28K of physical memory.
.QSET	The queue element space specified must lie entirely in the lower 28K of physical memory, and space must be allowed for 10 words per queue element.
.CNTXSW	This request is not available for virtual jobs. There is no need to context switch the system communication area.
.SETTOP	This request returns the high limit for the job. This address is always within the lower 28K of physical memory. .SETTOP does not reflect any mapping to extended memory that may be in effect.

3.9 SUMMARY AND HIGHLIGHTS OF RT-11 EXTENDED MEMORY SUPPORT

This section gives the highlights and summarizes the basic operations of RT-11 extended memory support. Since this is a new and also complex concept, this section is provided as an aid to understanding the material in this chapter. More than one reading of this chapter is necessary to fully understand its contents.

The following material can be used to review the basic operations and features, and subsequent readings of the chapter can be keyed to amplify this abbreviated discussion.

3.9.1 Extended Memory Prerequisites

The following hardware and software components must be incorporated into the RT-11 operating system to utilize the extended memory feature. The system cannot be bootstrapped without these components.

1. Memory management unit
2. XM monitor and handlers
3. Extended instruction set (EIS)

3.9.2 What Is Extended Memory Support?

Extended memory support is the technique of extending the addressing capability of the RT-11 system beyond its limitation of 32K words imposed by the 16-bit PDP-11 processor word.

3.9.3 How Is Extended Memory Support Implemented?

Extended memory support is implemented through hardware and software.

Hardware

1. Memory management unit

Software

1. XM monitor and handlers
2. User Data Structures
   a. Window Definition Block
   b. Region Definition Block
3. Programmed Requests
   a. .CRAW
   b. .ELAW
   c. .CRRG
   d. .ELRG
   e. .MAP
   f. .UNMAP
   g. .GMCX

3.9.4 How To Use Extended Memory Programmed Requests

This section briefly outlines the various steps involved in using the programmed requests and macros to set up extended memory.

1. Create a region definition block by invoking the macro .RDBBK, or define parameters and set up a region definition block by invoking the macro .RDBDF.

2. Create the necessary regions in extended memory by executing the .CRRG request for each region. A region is eliminated by the .ELRG request.

3. Create a window definition block by invoking the macro .WDBBK, or define parameters and set up a window definition block by invoking the macro .WDBDF.

4. For each window to be created, move the region ID (R.GID, returned by the monitor from .CRRG) from the region definition block into the window definition block. (Move it to W.NRID). This procedure links the window and the region together, but does not map the window to the region.

5. Create the necessary windows in the virtual address space, 0-28K, by executing the .CRAW request for each window to be created. A window is eliminated by the .ELAW request.

6. Map the window to the desired region by executing the .CRAW or .MAP request. A window is unmapped by the .UNMAP request or implicitly unmapped by another .MAP request.

3.9.5 Operational Characteristics of Extended Memory Support

1. The two types of user programs are virtual and privileged.

   a. Virtual provides more address space for mapping to extended memory. It is selected by setting a bit of the JSW before program execution.

   b. Privileged is the default mapping that is compatible with SJ and FB monitors. In this mapping arrangement, the low 28K words of memory and the I/O page are mapped to simulate the non-extended memory environment.

2. The two operating modes are kernel and user.

   a. RMON and the USR run in kernel mode.

   b. KMON and user jobs run in user mode.

CHAPTER 4

SYSTEM SUBROUTINE LIBRARY

4.1 INTRODUCTION

The RT-11 FORTRAN system subroutines are a collection of FORTRAN-callable routines that allow a FORTRAN user to utilize various features of RT-11 foreground/background (FB) and single-job (SJ) monitors. There are no FORTRAN routines to manipulate extended memory under the extended memory (XM) monitor. SYSF4 also provides utility functions, a complete character string manipulation package, and a two-word integer support. This collection of routines is usually placed in a default system library, which is an object module library file called SYSLIB.OBJ. This library file is the default library that the linker uses to resolve undefined globals and is resident on the system device (SY:). The concatenated set of FORTRAN-callable routines is in a file called SYSF4.OBJ. Section 4.1.5 describes how to make these routines into a library.

The user of SYSF4 should be familiar with Chapter 2 of this manual. Chapter 4 assumes that FORTRAN users are familiar with the PDP-11 FORTRAN Language Reference Manual and the RT-11/RSTS/E FORTRAN IV User's Guide.

The following are some of the functions provided by SYSF4:

- Complete RT-11 I/O facilities, including synchronous, asynchronous, and event-driven modes of operation. FORTRAN subroutines can be activated upon completion of an input/output operation.
- Timed scheduling of asynchronous subjobs (completion routines). This feature is standard in FB and XM, and optional in the SJ monitor.
- Complete facilities for interjob communication between foreground and background jobs (FB and XM only).
- FORTRAN interrupt service routines.
- Complete timer support facilities, including timed suspension of execution (FB and XM only), conversion of different time formats, and time of day information. These timer facilities support either 50- or 60-cycle clocks.
- All auxiliary input/output functions provided by RT-11, including the capabilities of opening, closing, renaming, creating, and deleting files from any device.
- All monitor-level informational functions, such as job partition parameters, device statistics, and input/output channel statistics.
- Access to the RT-11 Command String Interpreter (CSI) for accepting and parsing standard RT-11 command strings.
- A character string manipulation package supporting variable-length character strings.
- INTEGER*4 support routines that allow two-word integer computations.

SYSF4 allows the FORTRAN user to write almost all application programs completely in FORTRAN with no assembly language coding. Assembly language programs can also utilize SYSF4 routines (see Section 4.1.3).

4.1.1 Conventions and Restrictions

In general, the SYSF4 routines were written for use with RT-11 V2 or later and FORTRAN IV V1B or later versions. The use of this SYSF4 package with prior versions of RT-11 or FORTRAN leads to unpredictable results.

Programs using IPEEK, IPOKE, IPEEKB, IPOKEB, and/or ISPY to access FORTRAN, monitor, hardware, or other system specific addresses are not guaranteed to run under future releases or on different configurations. Suitable care should be taken with this type of coding to document precisely the use of these access functions and to check a referenced location's usage against the current documentation.

The following must be considered when coding a FORTRAN program that uses SYSF4.

1. Various functions in the SYSF4 package return values that are of type integer, real, and double precision. If the user specifies an IMPLICIT statement that changes the defaults for external function typing, he must explicitly declare the type of those SYSF4 functions that return integer or real results. Double precision functions must always be declared to be type DOUBLE PRECISION (or REAL*8). Failure to observe this requirement leads to unpredictable results.

2. All names of subprograms external to the routine being coded that are being passed to scheduling calls (such as ISCHED, ITIMER, IREADF, etc.) must be specified in an EXTERNAL statement in the FORTRAN program unit issuing the call.

3. Certain arguments (noted as such in the individual routine descriptions) to SYSF4 calls must be located in such a manner as to prohibit the RT-11 USR (User Service Routine) from swapping over them at execution time. If the section OTS$I is not 2K words in length, a program using SYSF4 calls can malfunction because the USR can swap over data to be passed to the USR. This should be rare, but if it occurs, making the USR resident through a SET USR NOSWAP command before starting the job or using the linker's /BOUNDARY option to have OTS$O start at 11000 (octal) eliminates the problem.

   FORTRAN IV version 2 uses .PSECTs to collect code and data into appropriate areas of memory. If RT-11 USR is needed and is not resident, it swaps over a FORTRAN program starting at the symbol OTS$I for 2K words of memory.

4. Quoted-string literals are useful as arguments of calls to routines in the SYSF4 package, notably the character string routines. These literals are allowed in subroutine and function calls.

5. Certain restrictions apply to completion or interrupt routines; see Section 4.2.1 for these restrictions.

4.1.2 Calling SYSF4 Subprograms

SYSF4 subprograms are called in the same manner as user-written subroutines. SYSF4 includes both FUNCTION subprograms and SUBROUTINE subprograms. FUNCTION subprograms receive control by means of a function reference, as:

```
i = function name ([arguments])
```

SUBROUTINE subprograms are invoked by means of a CALL statement; that is,

```
CALL subroutine name [(arguments)]
```

All routines in SYSF4 can be called as FUNCTION subprograms if the return value is desired, or as SUBROUTINE subprograms if no return value is desired. For example, the LOCK subroutine can be referenced as either:

```
CALL LOCK

  or

I = LOCK()
```

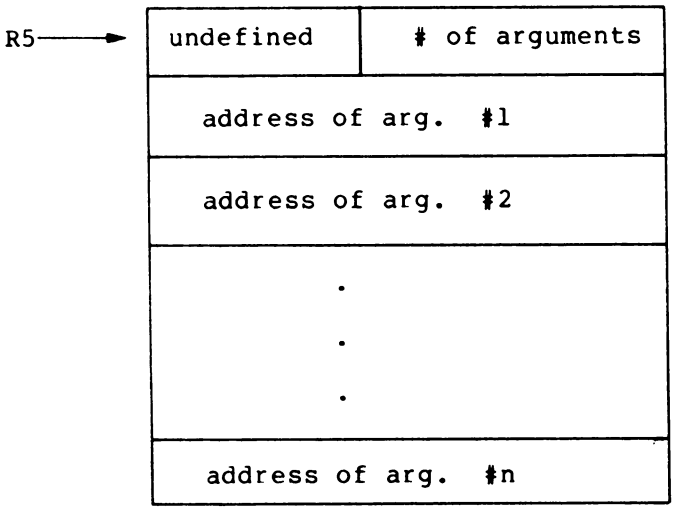
Note that routines that do not explicitly return function results produce meaningless values if they are referenced as functions. In the following descriptions, the more common usage (function or subroutine) is shown.

4.1.3 Using SYSF4 with MACRO

The calling sequence is standard for all subroutines, including user-written FORTRAN subprograms and assembly language subprograms. SYSF4 routines can be used with MACRO programs by passing control to the SYSF4 routine with the following instruction:

```
        JSR     PC,routine
```

Register five points to an argument list having the following format:



Control is returned to the calling program by use of the instruction:

```
RTS     PC
```

The following is an example of calling a SYSF4 function from an assembly language routine.

```
        .GLOBL JMUL         ;GLOBAL FOR JMUL

        .
        .
        .

        MOV #LIST,R5        ;POINT R5 TO ARG LIST
        JSR PC,JMUL         ;CALL JMUL
        CMP #-2,R0          ;CHECK FOR OVERFLOW
        BEQ OVRFL           ;BRANCH IF ERROR

        .
        .
        .

LIST:   .WORD 3             ;ARG LIST,3 ARGS
        .WORD OPR1          ;ADDR OF 1ST ARG
        .WORD OPR2          ;ADDR OF 2ND ARG
        .WORD RESULT        ;ADDR OF 3RD ARG
OPR1:   .WORD 100           ;LOW-ORDER VALUE OF 1ST ARG
        .WORD 0             ;HIGH-ORDER VALUE OF 1ST ARG
OPR2:   .WORD 10            ;LOW-ORDER VALUE OF 2ND ARG
        .WORD 10            ;HIGH-ORDER VALUE OF 2ND ARG
RESULT: .BLKW 2             ;2-WORD RESULT (LOW ORDER, HIGH ORDER)
        .END
```

The following routines can be used only with FORTRAN:

GETSTR
IASIGN
ICDFN
IFETCH
IFREEC
IGETC
IGETSP
ILUN
INTSET
IQSET
IRCVDF
IREADF
ISCHED
ISDATF
ISPFNF
ITIMER
IWRITF
PUTSTR
SECNDS

User-written assembly language programs that call SYSF4 subprograms must preserve any pertinent registers before calling the SYSF4 routine and restore the registers, if necessary, upon return.

Function subprograms return a single result in the registers. The register assignments for returning the different variable types are:

Integer, Logical functions - result in R0

Real functions - high-order result in R0,
low-order result in R1

Double Precision functions - result in R0-R3, lowest order result in R3

Complex functions - high-order real result in R0,
low-order real result in R1,
high-order imaginary result in R2,
low-order imaginary result in R3

User-written assembly language routines that interface to the FORTRAN Object Time System (OTS) must be aware of the location of the RT-11 USR (User Service Routine). If a user routine requests a USR function (such as IENTER or LOOKUP), or if the USR is invoked by the FORTRAN OTS, the USR is swapped into memory if it is nonresident. The FORTRAN OTS is designed so that the USR can swap over it. User routines must be written to allow the USR to swap over them or must be located outside the region of memory into which the USR swaps. User interrupt service routines and completion routines, because of their asynchronous nature, must be further restricted to be located where the USR will not swap. The USR, if in a swapping state, will swap at the address specified in location 46 of the system communication area. If location 46 is 0, the USR will swap at the default USR swap location (shown in Figure 1-1). The USR occupies 2K words. Interrupt and completion routines (and their data areas) must not be located in this area. The best way to accomplish this is to examine the link map, determine whether the USR will swap over an assembly language or FORTRAN asynchronous routine, and, if so, change the order of object modules and libraries as specified to the linker. Continue this process until a suitable arrangement is obtained.

The order in which program sections are allocated in the executable program is controlled by the order in which they are first presented to the LINK utility. Applications that are sensitive to this ordering typically separate those sections that contain read-only information (such as executable code and pure data) from impure sections containing variables.

The main program unit of a FORTRAN program (normally the first object program in sequence presented to LINK) declares the following PSECT ordering:

Section Name	Attributes
OTS$I	RW, I, LCL, REL, CON
OTS$P	RW, D, GBL, REL, OVR
SYS$I	RW, I, LCL, REL, CON
USER$I	RW, I, LCL, REL, CON
$CODE	RW, I, LCL, REL, CON
OTS$O	RW, I, LCL, REL, CON
SYS$O	RW, I, LCL, REL, CON
$DATAP	RW, D, LCL, REL, CON
OTS$D	RW, D, LCL, REL, CON
OTS$S	RW, D, LCL, REL, CON
SYS$S	RW, D, LCL, REL, CON
$DATA	RW, D, LCL, REL, CON
USER$D	RW, D, LCL, REL, CON
.$$$$.	RW, D, GBL, REL, OVR
Other COMMON Blocks	RW, D, GBL, REL, OVR

The User Service Routine (USR) can swap over pure code, but must not be loaded over constants or impure data that can be passed as arguments to it.

The above ordering collects all pure sections before impure data in memory. The USR can safely swap over sections OTSI, OTSP, SYS$I, USER$I, and $CODE.

Assembly-language routines used in applications sensitive to PSECT ordering should use the same program sections as output by the compiler for this purpose. This is, the programmer should place pure code and read-only data in section USER$I, and all impure storage in section USER$D. This ensures that the assembly-language routines will participate in the separation of code and data.

Note that the ordering of PSECTs in an overlay program follows the guidelines herein for each overlay segment (that is, the root segment will contain pure sections followed by impure, and each overlay segment will have a similar separation of pure and impure internal to its structure).

See the RT-11/RSTS/E FORTRAN IV User's Guide for more information.

To remove these restrictions, the user must make the USR resident either by specifying the /NOSWAP option to the FORTRAN command (when compiling a program to be run in the background of FB or XM, or under SJ) or by issuing the SET USR NOSWAP command before executing the program.

4.1.4 Running a FORTRAN Program in the Foreground

The FRUN monitor command must be modified to include various SYSF4 functions. The following formula allocates the needed space when running a FORTRAN program as a foreground job.

```
x = [1/2[44@+(33*N)+(R-136)+A*512)]]
```

The variables are defined as follows:

A = The number of files open at one time. If double buffering is used, A should be multiplied by 2.

N = The number of channels (logical unit numbers).

R = Maximum record length. The default is 136 characters.

This formula must be modified for SYSF4 functions as follows:

The IQSET function requires the formula to include additional space for queue elements (qleng) to be added to the queue:

```
x = [1/2[440+(33*N)+(R-136)+A*512)]]+[10*qleng]
```

The ICDFN function requires the formula to include additional space for the integer number of channels (num) to be allocated.

```
x = [1/2[440+(33*N)+(R-136)+A*512)]]+[6*num]
```

The INTSET function requires the formula to include additional space for the number of INTSET calls (INTSET) issued in the program.

```
x = [1/2[440+(33*N)+(R-136)+A*512]]+[25*INTSET]
```

Any SYSF4 calls, including INTSET, that invoke completion routines must include 64(decimal) words plus the number of words needed to allocate the second record buffer (default is 68(decimal) words). The length of the record buffer is controlled by the /R option to the FORTRAN compiler. If the /R option is not used, the allocation in the formula must be 136(decimal) words.

```
x = [1/2[440+(33*N)+(R-136)+A*512]]+[64+R/2]
```

If the /N option does not allocate enough space in the foreground on the initial call to a completion routine, the following message appears:

```
?ERR 0, NON-FORTRAN ERROR CALL
```

This message also appears if there is not enough free memory for the background job or if a completion routine in the single-job monitor is activated during another completion routine. In the latter case, the job aborts. The FB monitor should be used for multiple active completion routines.

4.1.5 Linking with SYSF4

SYSF4 is provided on the distribution media as a file of concatenated object modules (SYSF4.OBJ). If this file is linked directly with the FORTRAN program, all SYSF4 modules are included whether they are used or not. For example:

```
.LINK PROG,SYSF4
```

A library can be created by using the librarian to transform SYSF4 into a library file (SYSLIB.OBJ) as follows:

```
.LIBRARY/CREATE SYSLIB SYSF4
```

Normally the default system library file (SYSLIB.OBJ) also includes the appropriate FORTRAN runtime system routine.

When a library is used, only the modules called are linked with the program. For example:

```
.LINK PROG
```

To add the SYSF4 modules to the default library SYSLIB.OBJ, the following command should be used:

```
.LIBRARY/INSERT SYSLIB SYSF4
```

The following example links the object module EXAMPL.OBJ into a single memory image file EXAMPL.SAV and produces a load map file on LP:. The default system library (SYSLIB.OBJ), which contains the FORTRAN OTS routine, is searched for along with any routines that are not found in other object modules.

```
.LINK/MAP EXAMPL
```

4.2 TYPES OF SYSF4 SERVICES

Ten types of services are available to the user through SYSF4. These are:

1. File-oriented operations
2. Data transfer operations
3. Channel-oriented operations
4. Device and file specifications
5. Timer support operations
6. RT-11 service operations
7. INTEGER*4 support functions
8. Character string functions
9. Radix-50 conversion operations
10. Miscellaneous services

Table 4-1 alphabetically summarizes the SYSF4 subprograms in each of these categories. Those marked with an asterisk (*) are allowed only in a foreground/background environment, under either the FB or XM monitor.

Table 4-1
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
File-Oriented Operations		
CLOSEC	4.3.3	Closes the specified channel.
IDELET	4.3.20	Deletes a file from the specified device.
IENTER	4.3.23	Creates a new file for output.
IRENAM	4.3.41	Changes the name of the indicated file to a new name.
LOOKUP	4.3.70	Opens an existing file for input and/or output via the specified channel.
Data Transfer Functions		
GTLIN	4.3.11	Transfers a line of input from the console terminal or indirect file (if active) to the user program.
*IRCVD *IRCVDC *IRCVDF *IRCVDW	4.3.39	Receives data. Allows a job to read messages or data sent by another job in an FB environment. The four modes correspond to the IREAD, IREADC, IREADF, and IREADW modes.

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
Data Transfer Functions (cont.)		
IREAD	4.3.40	Transfers data via the specified channel to a memory buffer and returns control to the user program when the transfer request is entered in the I/O queue. No special action is taken upon completion of I/O.
IREADC	4.3.40	Transfers data via the specified channel to a memory buffer and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the read, control transfers to the assembly language routine specified in the IREADC function call.
IREADF	4.3.40	Transfers data via the specified channel to a memory buffer and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the read, control transfers to the FORTRAN subroutine specified in the IREADF function call.
IREADW	4.3.40	Transfers data via the specified channel to a memory buffer and returns control to the program only after the transfer is complete.
*ISDAT *ISDATC *ISDATF *ISDATW	4.3.45	Allows the user to send messages or data to the other job in an FB environment. The four modes correspond to the IWRITE, IWRITC, IWRITF, and IWRITW modes.
ITTINR	4.3.51	Inputs one character from the console keyboard.
ITTOUR	4.3.52	Transfers one character to the console terminal.
IWAIT	4.3.55	Waits for completion of all I/O on a specified channel. (Commonly used with the IREAD and IWRITE functions.)
IWRITC	4.3.56	Transfers data via the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the write, control transfers to the assembly language routine specified in the IWRITC function call.

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
Data Transfer Functions (cont.)		
IWRITE	4.3.56	Transfers data via the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue. No special action is taken upon completion of the I/O.
IWRITF	4.3.56	Transfers data via the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the write, control transfers to the FORTRAN subroutine specified in the IWRITF function call.
IWRITW	4.3.56	Transfers data via the specified channel to a device and returns control to the user program only after the transfer is complete.
*MTATCH	4.3.72	Attaches a particular terminal in a multi-terminal environment
*MTDTCH	4.3.73	Detaches a particular terminal in a multi-terminal environment
*MTGET	4.3.74	Provides information about a particular terminal in a multi-terminal system.
*MTIN	4.3.75	Transfers characters from a specific terminal to the user program in a multi-terminal system.
*MTOUT	4.3.76	Transfers characters to a specific terminal in a multi-terminal system.
*MTPRNT	4.3.77	Prints a message to a specific terminal in a multi-terminal system.
*MTRCTO	4.3.78	Enables output to terminal by cancelling the effect of a previously typed CTRL/O.
*MTSET	4.3.79	Sets terminal and line characteristics in a multi-terminal system.
*MWAIT	4.3.80	Waits for messages to be processed.
PRINT	4.3.81	Outputs an ASCII string to the console terminal.

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
Channel-Oriented Operations		
ICDFN	4.3.15	Defines additional I/O channels.
*ICHCPY	4.3.16	Allows access to files currently open in the other job's environment.
*ICSTAT	4.3.19	Returns the status of a specified channel.
IFREEC	4.3.25	Returns the specified RT-11 channel to the available pool of channels for the FORTRAN I/O system.
IGETC	4.3.26	Allocates an RT-11 channel and marks it in use to the FORTRAN I/O system.
ILUN	4.3.29	Returns the RT-11 channel number with which a FORTRAN logical unit is associated.
IREOPN	4.3.42	Restores the parameters stored via an ISAVES function and reopens the channel for I/O.
ISAVES	4.3.43	Stores five words of channel status information into a user-specified array.
PURGE	4.3.82	Deactivates a channel.
Device and File Specifications		
IASIGN	4.3.14	Sets information in the FORTRAN logical unit table.
ICSI	4.3.18	Calls the RT-11 CSI in special mode to decode file specifications and options.
Timer Support Operations		
CVTTIM	4.3.5	Converts a two-word internal format time to hours, minutes, seconds, and ticks.
GTIM	4.3.9	Gets time of day.
ICMKT	4.3.17	Cancels an unexpired ISCHED, ITIMER, or MRKT request. (Valid for SJ monitors with timer support, a SYSGEN option.)

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
Timer Support Operations (cont.)		
ISCHED	4.3.44	Schedules the specified FORTRAN subroutine to be entered at the specified time of day as an asynchronous completion routine. (Valid for SJ monitors with timer support, a SYSGEN option.)
ISLEEP	4.3.46	Suspends main program execution of the running job for a specified amount of time; completion routines continue to run. (Valid for SJ monitors with timer support, a SYSGEN option.)
ITIMER	4.3.49	Schedules the specified FORTRAN subroutine to be entered as an asynchronous completion routine when the time interval specified has elapsed. (Valid for SJ monitors with timer support, a SYSGEN option.)
*ITWAIT	4.3.53	Suspends the running job for a specified amount of time; completion routines continue to run.
*IUNTIL	4.3.54	Suspends the main program execution of the running job until a specified time of day; completion routines continue to run.
JTIME	4.3.67	Converts hours, minutes, seconds, and ticks into 2-word internal format time.
MRKT	4.3.71	Marks time; that is, schedules an assembly language routine to be activated as an asynchronous completion routine after a specified interval. (Valid for SJ monitors with timer support, a SYSGEN option.)
SECNDS	4.3.93	Returns the current system time in seconds past midnight minus the value of a specified argument.
TIMASC	4.3.98	Converts a specified two-word internal format time into an eight-character ASCII string.
TIME	4.3.99	Returns the current system time of day as an 8-character ASCII string.

* FB and XM monitors only.

(continued on next page)