EY-0060E-SG-0201

Programming RSX-11M in MACRO

A Self-Paced Course

Volume II

digital

EY-0060E-SG-0201

Programming RSX-11M in MACRO

A Self-Paced Course

Student Workbook
Volume II

Prepared by Educational Services
of
Digital Equipment Corporation

Printed in U.S.A.

9/82/14

CONTENTS

VOLUME I

SG STUDENT GUIDE

1 USING SYSTEM SERVICES

2 DIRECTIVES

3 USING THE QIO DIRECTIVE

4 USING DIRECTIVES FOR INTERTASK COMMUNICATION

5 MEMORY MANAGEMENT CONCEPTS

6 OVERLAYS

VOLUME II

7 STATIC REGIONS

8 DYNAMIC REGIONS

9 FILE I/O

10 FILE CONTROL SERVICES

AP APPENDICES

FIGURES

TABLES

EXAMPLES

STATIC REGIONS

7

INTRODUCTION

Logical address space in a task is composed of regions. There are three basic types of regions, task regions, static regions, and dynamic regions. Task regions, into which tasks are loaded, are created using information set up by the Task Builder. Static and dynamic regions are generally used to share code or data among several tasks. Static regions are created using the Task Builder; dynamic regions are created during task execution, using executive directives.

This module discusses static regions. You can use these static regions to:

- Create memory areas containing code which is shared among tasks
- Create memory-resident data areas which can be used for communication between tasks or successive invocations of the same task
- Communicate directly with a peripheral device through the I/O page.

OBJECTIVES

1. To create and use a resident common region
2. To create and use a resident library
3. To determine whether a position independent or an absolute shared region should be used in a given situation
4. To create and use a device common.

RESOURCE

- RSX-11M/M-PLUS Task Builder Manual, Chapter 5

TYPES OF STATIC REGIONS

Static regions, also called shared regions, are areas of memory which are shared among tasks. They allow tasks to share data or code with very little overhead. Unlike send and receive directives, no executive directives are needed, and the area's size is limited only by virtual address and possibly physical memory limitations. The virtual addressing limit must be met for both the region itself and any tasks which use the region. For a task which uses the region, the total applies to all regions used plus the task's code.

Static regions offer very quick access, since the area is loaded before the tasks which use it are run. Once loaded, it is available directly in memory. Therefore, it offers much faster access than disk-resident data.

Table 7-1 summarizes the types of shared regions available on an RSX-11M system. A resident common contains data. The data can be accessed by several different tasks, each with read only access or with read/write access.

A resident library contains reentrant subroutines, which can be called by several different tasks. A single copy of each subroutine can be shared, thus reducing the total memory requirements of the tasks. The term resident is used because the shared region is task-built, installed, and loaded into memory separately from the tasks which access it.

A third type of shared region is a device common, a special type of resident common. It occupies physical addresses on the I/O page, which correspond to I/O device registers instead of physical memory. Therefore, this kind of common allows a task to reference an I/O device directly. Unlike other resident commons, a device common has no true contents because it has no physical memory associated with it.

Table 7-1 Types of Static Regions Available on RSX-11M

Type of Region	Contents	Advantages
Resident Common	Data accessed by two or more tasks	Serves as communications link Serves as memory-resident data base
Resident Library	Reentrant routines, used by two or more tasks	One copy of common routines shared in memory
Device Common	No true "contents" Region is a range of physical addresses within I/O page	Nonprivileged task can directly access an I/O device without being mapped to the Executive

MEMORY ALLOCATION

Memory is allocated independently to the shared region and to the individual tasks which use it. We will call the tasks which use the region referencing tasks. On an RSX-11M system, the shared region must reside in a dedicated common type partition. The name of the partition must be the same as the name of the region. The partition can be created at SYSGEN time or later by the system manager or a privileged user. Once the region is installed and loaded into the partition, it cannot be checkpointed.

MAPPING

Shared regions can be written and task-built as either position independent regions or as absolute regions. On a mapped system, position independent regions can be placed anywhere in a referencing task's virtual address space. This means that the virtual addresses used to map to the region can correspond to any available APR.

Figure 7-1 shows a position independent region, POSIND, and three referencing tasks. The region is loaded into memory into the partition POSIND; the partition name must be the same as the name of the region. Recall that a virtual address window for mapping must begin with a base address for an APR on a 4K word boundary. Because the region is 5K words in length and each APR can only map at most 4K words, two APRs are needed to map the region.

Task A maps the shared region using APRs 6 and 7, starting at virtual address 140000(8). It could in fact use APRs 5 and 6, beginning at virtual address 120000(8) or APRs 4 and 5, beginning at virtual address 100000(8).

Task B maps the shared region at the first available APR above the task code, using APRs 2 and 3, beginning at virtual address 40000(8). It could use APRs 3 and 4, 4 and 5, 5 and 6, or 6 and 7 as well.

Task C maps the shared region using APRs 6 and 7, starting with virtual address 140000(8). There is no other possible way for Task C to map the shared region because APR 6 is the first available APR.

When you task-build a referencing task, you can specify which APR to use in mapping the region. If you do not specify an APR, the Task Builder selects the highest set of available APRs. When task A and task C were built, the user either did not specify an APR, or specified APR 6. When task B was built, the user specified APR 2.

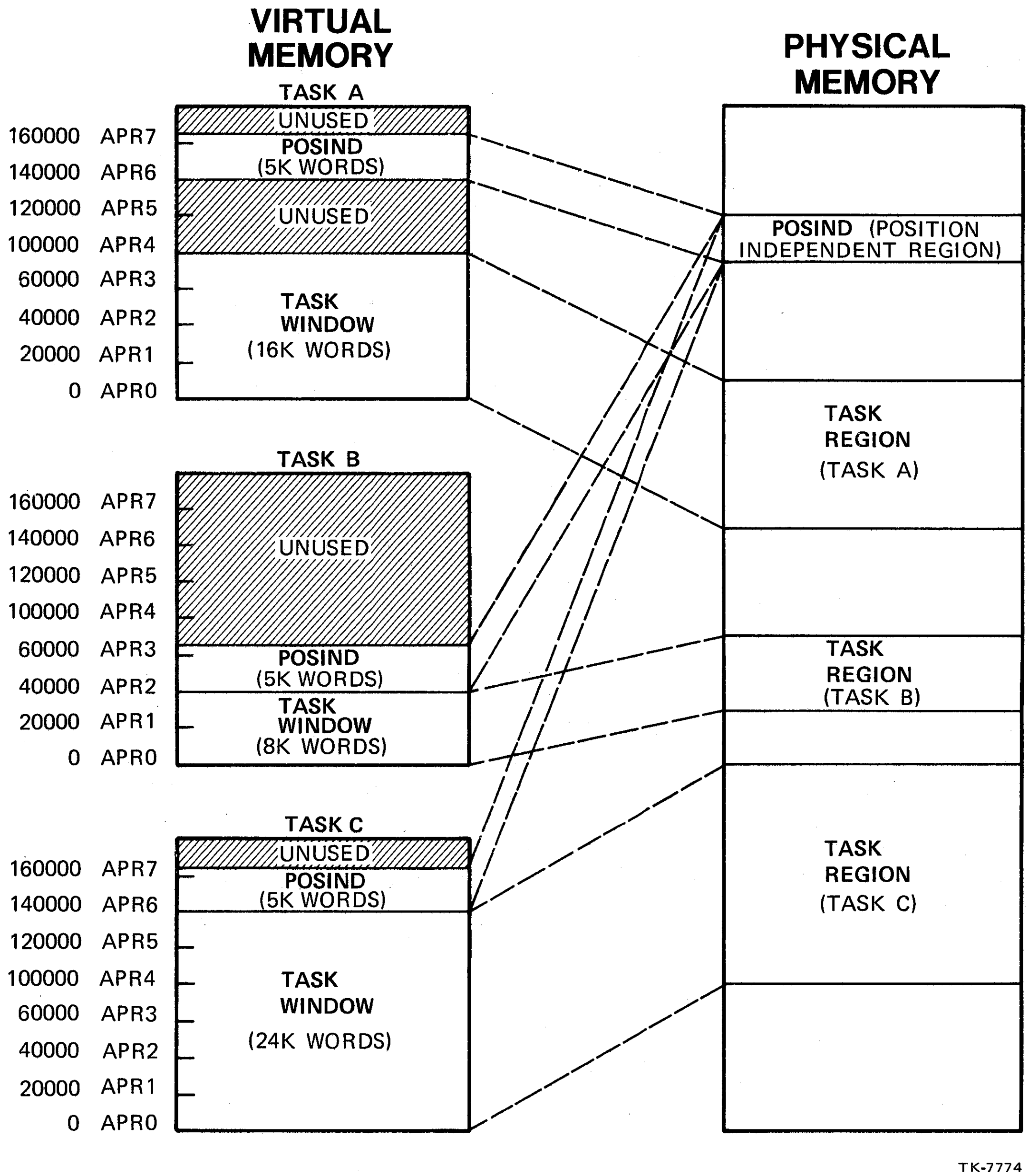


Figure 7-1 Tasks Using a Position Independent Shared Region

An absolute shared region has its virtual addresses fixed when it is task-built. All tasks which reference it must use those virtual addresses, and the corresponding APRs, to map to the region. Figure 7-2 shows another region ABSOLU and three referencing tasks, A, B and C. The shared region ABSOLU was built to use virtual addresses 120000(8)-147777(8) (6K words) with APRs 5 and 6. All referencing tasks must map to the region using these APRs. Therefore, task A and task B can both map to the region, since APRs 5 and 6 are available. Task C, on the other hand, cannot reference ABSOLU, since APR 5 is already used by its task code.

You may think that there is no reason to ever limit yourself by making a region absolute. However, there are code restrictions for position independent regions due to the fact that a shared region is task-built separately from any of its referencing tasks.

When the region is task-built, all code within it is set. The code has to be written using special position independent coding techniques to allow it to be placed at possibly different virtual addresses in the various referencing tasks. This is only a problem for data if the data is not position independent; for example, a jump table.

The starting virtual address of each routine, defined by its label, is assigned when the referencing task is task-built. This address may vary depending on which base APR is used to map the region. The address of a given routine may vary from one referencing task to another. But the address placed in the table itself was already fixed when the region was task-built, and does not change for each referencing task. Further, that address may not match any of the addresses assigned in referencing tasks. For example, consider the following jump table and routines W, X, and Y:

```
JMPTAB:  .WORD W
         .WORD X
         .WORD Y
     W:    .
           .
           .

     X:    .
           .
           .

     Y:    .
           .
           .
```

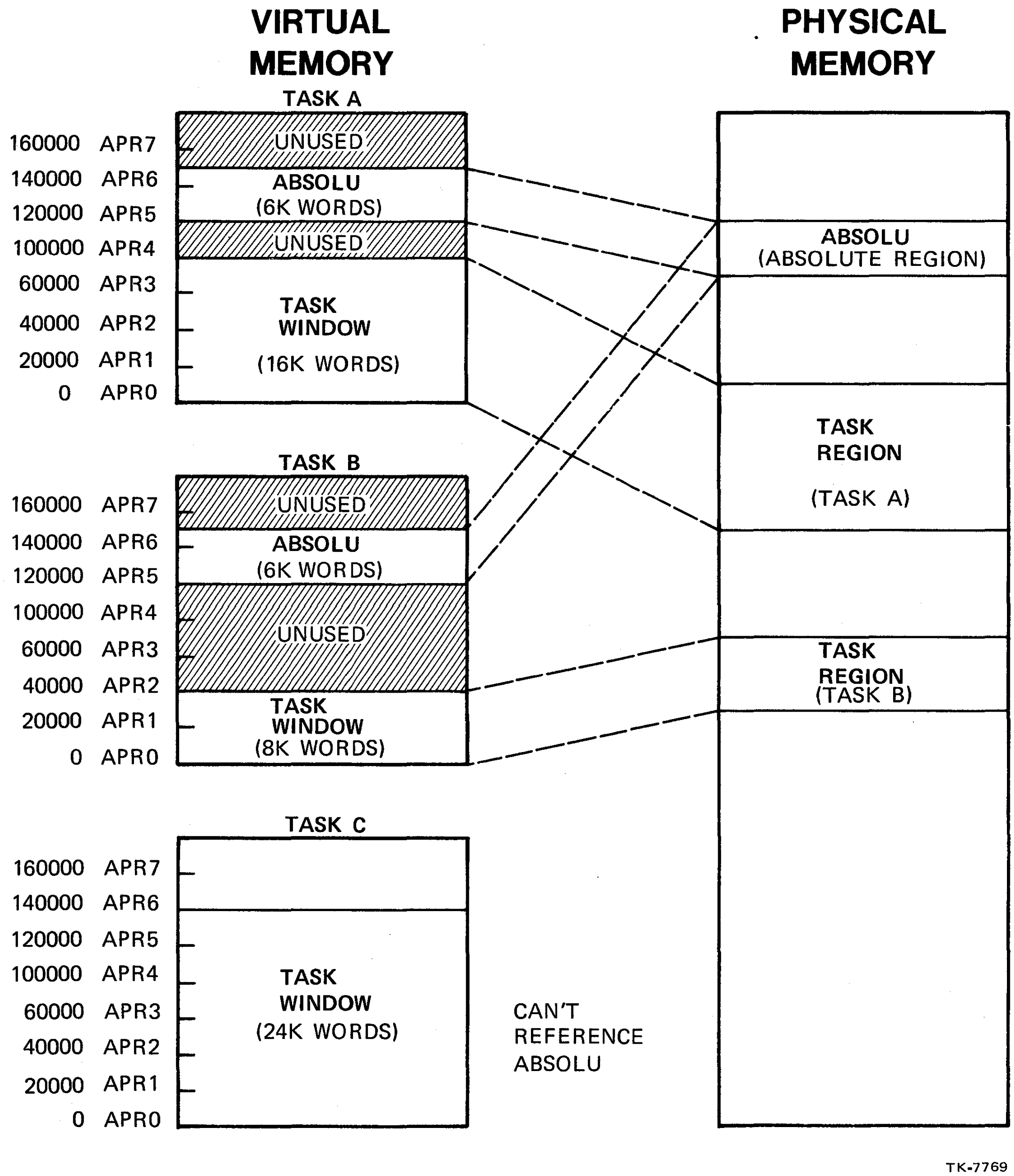


Figure 7-2 Tasks Using an Absolute Shared Region

The addresses resulting from the .WORD directives are fixed when the region is task-built; e.g., at W = 1500(8), X = 1540(B), and Y = 1626(8). If the referencing task places the actual addresses W, X and Y at those virtual addresses, everything will work fine. But if it starts mapping at APR 4 (virtual address 120000), the labels themselves will be assigned addresses 120000(8) + 1500(8) = 121500(8), 120000(8) + 1540(8) = 121540(8), and 120000(8) + 1626(8) = 121626(8).

However, the values in the table are already set at 1500(8), 1540(8), and 1626(8), and they no longer address the correct locations. A jump or call by way of the table to routine W will result in a transfer to location 1500(8) in the referencing task, and definitely not to routine W. To avoid this problem, jump tables should be included in the referencing task code instead of in the shared region.

Instructions in shared regions are even trickier to program. All references which are relative to the current PC, for example, eight bytes from here, work fine. But a reference to an actual virtual address, for example, virtual address 4260(8) or @#A, only works if 4260(8) or A remains set at that virtual address. For a discussion of position independent code and how to write it, see Appendix H of the <u>IAS/RSX-11 MACRO-11 Reference Manual</u>.

All of this means that in general, the decision about whether to create a position independent or an absolute shared region is based on the code restrictions, rather than the flexibility. In general, resident commons, containing data, are created position independent; and resident libraries, containing code, are created absolute.

Figure 7-3 shows the program development process for creating a shared region and a referencing task. Specific steps for each process are discussed later in this module. Assemble and task-build the shared region separate from the referencing task, and before task-building the referencing task.

Since it is not an executable task, certain task-build switches are used to create a task image with no header and no stack. An additional file, called a symbol definition file, is also created at task-build time. This file contains information about the symbols defined in the region, which the Task Builder will use when it builds the referencing task to set up linkage to the region.

After task-building the shared region, task-build the referencing task. It can be written and assembled earlier, if desired. The name of the region is specified to the Task Builder so that it can access the symbol definition file and set up the linkage to the shared region. The shared region must be installed (causing it to be loaded into memory as well) before any referencing task is run.

REFERENCES TO A SHARED REGION

The following kinds of references are made to a shared region by a referencing task:

- The task retrieves data from, or stores data in, a resident common.
- The task calls or jumps to a routine in a resident library.

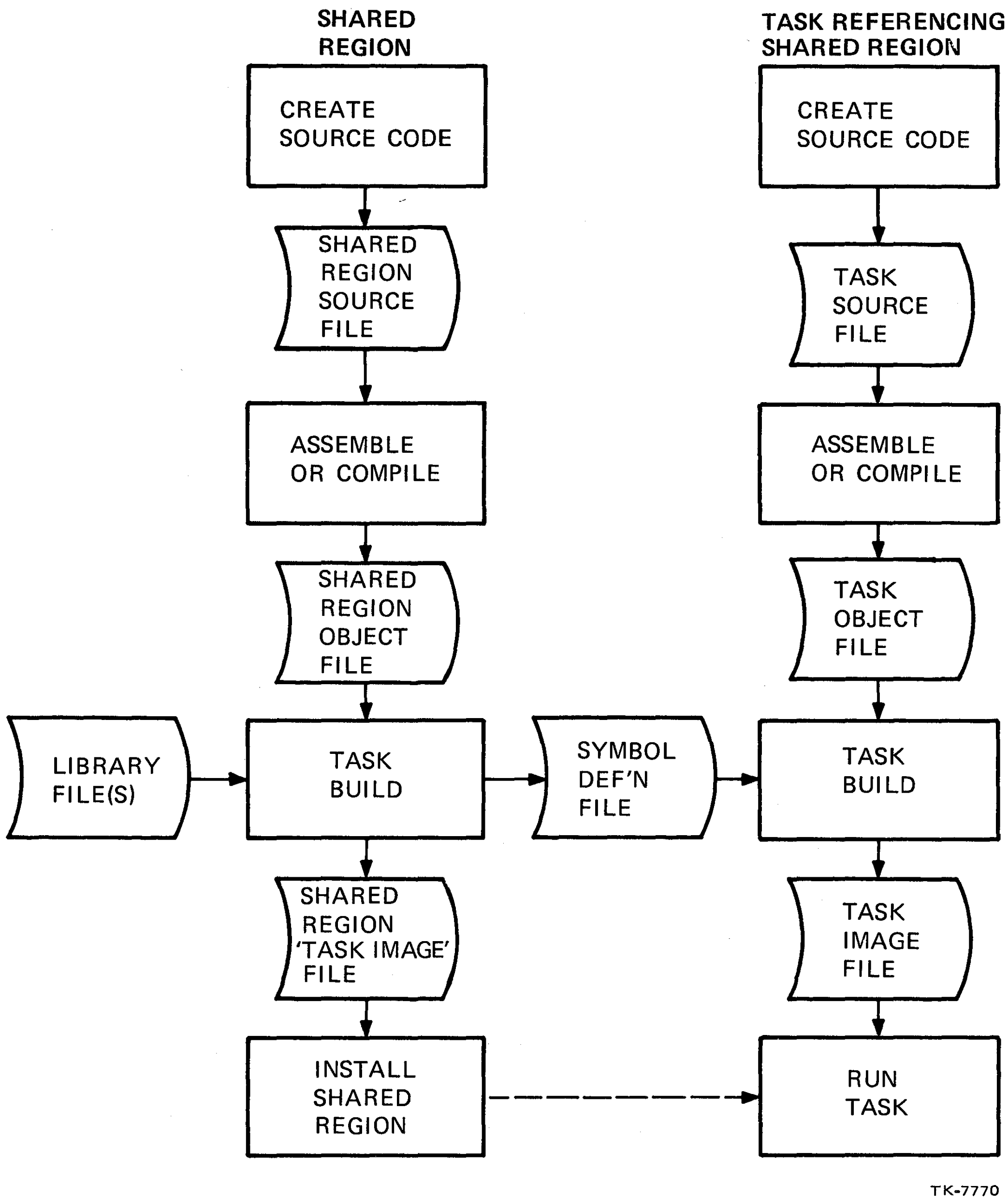


Figure 7-3 Program Development for Shared Regions

Techniques of Referencing

When you write the code for the shared region and the referencing task, you must pick a technique to resolve the references to the shared region. Some of the techniques are the same as the ones we used in Chapter 6 for referencing data in the root from an overlay segment.

Using Overlaid Psects (Data Only)

This technique is similar to the one for overlays. An example appears below. This time the Overlaid (OVR) Psect is defined in the shared region, then the same Psect is specified in the referencing tasks. The Task Builder, as usual, combines the different occurrences of each Psect. Because the shared region is built first, the Psect MYDATA is placed there. Later, when the referencing task is built, the new occurrence of MYDATA is combined with the one in the shared region. The OVR attribute tells the Task Builder to start the allocation at the same location as the allocation already there, causing the addresses to be overlaid. This, in effect, just sets up the addressing so that M references the first word of the region, the 3.

Shared region:

```
        .PSECT   MYDATA D,GBL,OVR ; Defaults: REL,RW
        .WORD    3.,4.,5.
        .END
```

Referencing task:

```
        .PSECT   MYDATA D,GBL,OVR ; PSECT in shared
                                  ;  region
M=.                         ; Addr of start of region
        .PSECT              ; Back to blank Psect
          .
          .
          .
START:  CMP      M,#5       ; Check value
        BGT      FIFTY      ; Branch if greater
```

Using Global Symbols (Data or Subroutines)

This technique is also the same as the one used in overlays. An example for data and for a subroutine appear below. In both cases, the label or labels are defined as global symbols. The referencing task uses the same global symbols to access the data or to call the subroutine. The possibly needed Psect name will be discussed later in the module.

For Data

Shared region:

```
; Possibly needed Psect name
;       .PSECT  ZZZ
        M::     .WORD   3.,4.       ; Define data and symbols
        N::     .WORD   5.
                .END
```

Referencing task:

```
        CMP     M,#5        ; Check value
        BGT     FIFTY       ; Branch if greater
```

For Subroutine Names

Shared region:

```
; Subroutine
AADD::          .
                .
                .
        RETURN              ; Return
```

Referencing task:

```
; Set up arguments
                .
                .
                .
        CALL    AADD        ; Call subroutine
```

Using Virtual Addresses (Data Only)

This technique is not available with overlays. If the shared region is built absolute, the starting virtual address of the region is fixed when the region is task-built. In the example below, it is assumed that the region is task-built absolute to begin at virtual address 160000(8). The referencing task can access the data by using the actual virtual address where the data is mapped.

If the region is built position independent, it can be mapped at a specific base virtual address (or base APR) by specifying a base APR in the task-build command for the referencing task. In this example, specifying APR 7 would set the base virtual address for the region at 160000(8).

Shared region:

```
; Possibly needed Psect name
;       .PSECT  ZZZ
        .WORD   3.,4.,5.
        .END
```

Referencing task:

```
; Shared region must be task-built either:
;
;       absolute starting at V.A. 160000
;
;       position independent and referencing task is
;       task-built to force region to start at
;       V.A. 160000

M = 160000                     ; Addr of start of region
        CMP     M,#5           ; Check value
        BGT     FIFTY          ; Branch if greater
```

Table 7-2 summarizes the techniques for referencing a shared region. When you task-build the shared region, you can specify whether or not you want the Psect names placed in the symbol definition file (.STB file). They must be there if you use the technique of overlaid Psects to reference the region. Use the /SHAREABLE:COMMON qualifier (/CO in MCR) to include the Psect names.

If global symbols or virtual addresses are used, it is best to exclude the Psect names in the .STB file. Use the /SHAREABLE:LIBRARY qualifier (/LI in MCR) to exclude Psect names. This avoids possible task-build errors due to Psect conflicts.

If Psect names are kept in the .STB file, each Psect defined in the region, including the default blank (. BLK.) Psect is there. The Task Builder tries to collect allocations together if a matching Psect name appears in the referencing task. However, it can't add to the allocation in the region, since the region is already built. Therefore, if the Psect results in additional allocation to the Psect (always true if the Psect has the concatenate (CON) attribute), then a task-build error "LOAD ADDRESS OUT OF BOUNDS" results. This is because the new allocation can't be added to the already built shared region.

Therefore, if Psect names are placed in the .STB file, the Psect names in the referencing task must not match any in the shared region, including the default blank Psect. Avoiding this is especially difficult if the shared region is a system resident library like FCSRES or FORRES, which was written by DIGITAL. In this case, you may not know what Psect names were used in the original source code.

As a general rule, place Psect names in the .STB file only if you use overlaid Psects to reference the region. Table 7-3 shows the interaction of three task-builder switches or qualifiers. They are: /CODE:PIC, for position independent or not; /SHAREABLE:COMMON, for placing Psect names in the .STB file; and /SHAREABLE:LIBRARY, for not placing Psect names in the .STB file.

The name COMMON is used for keeping Psect names because overlaid Psects can only be used for data references, therefore they are generally used in resident commons. The name LIBRARY is used for not keeping Psect names because global symbols are generally used to reference subroutines in a resident library. In fact, if a resident common is referenced using global symbols or virtual addresses, it is also built /SHAREABLE:LIBRARY to avoid Psect conflicts.

Table 7-2 Techniques of Referencing a Shared Region

Technique	Shared Region	Referencing Task
Overlaid Psects*	Data only	
MACRO-11	Define offsets within Psect	Use same overlaid Psect
Global symbols**	Subroutines or data	
MACRO-11	Labels defined as global symbols	Data - use global symbol Subroutine - CALL or use JSR to global symbol
Virtual addresses**	Data only	
MACRO-11	Use offsets within any Psect	Use same offsets from a base VA If shared region PI, specify APR for that base VA If shared region absolute, use same base VA

* Must keep Psect names in .STB file
** Possible Psect conflicts

Table 7-3 Effect of /CODE:PIC, /SHAREABLE:COMMON, and /SHAREABLE:LIBRARY on a Shared Region's STB

	Position Independent /CODE:PIC (/PI)*	Absolute by Default (/-PI, MCR only)
/SHAREABLE:COMMON (/CO)	Required for overlaid Psects or FORTRAN COMMONs	Required for overlaid Psects or FORTRAN COMMONs
	Psect declarations are maintained as declared in the source code (default if /PI specified)	Psect names saved but all are flagged as absolute (ABS)
/SHAREABLE:LIBRARY (/LI)	Avoids Psect conflicts	Avoids Psect conflicts
	A single Psect is declared name is same as first object file in TKB command Psect is relocatable (REL)	A single Psect named . ABS. is used and is absolute
Default	/SHAREABLE:COMMON	/SHAREABLE:LIBRARY

* Switches in parentheses are for MCR format (with the TKB command).

PROCEDURE FOR CREATING SHARED REGIONS AND REFERENCING TASKS

Creating a Resident Common or Resident Library

1. Code your shared region.

   - Set up for an appropriate referencing technique.

     - Choose either overlaid Psects, global symbols, or virtual addresses for a resident common.

     - Use global symbols for a resident library.

   - Choose position independent or absolute.

     - The decision is based mainly on the coding techniques used.

     - If the code is position independent, build position independent (typical for resident common).

     - If the code is not position independent, build absolute (typical for resident library).

   - Resident common - reserve space, plus you may initialize locations.

   - Resident library - code must be reentrant. See the section on Reentrancy in Chapter 5 of the PDP-11 Processor Handbook for more information about reentrant code.

2. Assemble the shared region.

3. If not already done, create the common type partition.

   - Name must be the same as the name of the region.

   - Best done when the system is SYSGENed.

   - Use the SET PARTITION (SET/MAIN in MCR) command to create a partition.

   - Use the SET NOPARTITION (SET/NOMAIN in MCR) command to eliminate a partition.

- Examples:

  ```
  >SET PARTITION:MYCOM/BASE:7114/SIZE:200-
  ->/COMMON
  ```

  Creates the common type partition MYCOM with base physical address 711400(8) and size 20000(8) bytes. No other partition may use this space at the same time.

  ```
  >SET NOPARTITION:MYCOM
  ```

  Eliminates the partition MYCOM.

  NOTE

  Before you create or eliminate any partitions on your system, check with your system manager to find out what area of memory you may use.

4. Task-build the shared region.

   - Symbol definition file (.STB) required.
   - Build position independent or absolute (see Table 7-3).
   - Keep or do not keep Psect names (see Table 7-3).
   - Use required switches and options (see Table 7-4).

5. Install the shared region in the common type partition before running any referencing task.

   - Not required before task-building the referencing tasks.
   - Use the INSTALL (INS in MCR) command to install the region.
     - This command also loads the region into memory. This is unlike an executable task, which is usually loaded into memory only when it is activated.

- There is no command to remove a region. It is removed by either installing another region or eliminating the partition.

The required switches and options in Table 7-4 are needed for different reasons. No header or stack is needed because this is not an executable task. The referencing tasks each have their own header and stack. The symbol table definition file is needed to allow the Task Builder to link referencing tasks to the region. The partition name specifies the partition into which the region will be loaded.

For an absolute region you must specify a base address. If you specify a nonzero length, the specified value is used as a maximum length. A task-builder error results if the length of the region is longer than the length specified. If you specify a length of zero, the region is set up with the size needed for the code, as long as it doesn't exceed the 32K word virtual addressing limit.

Table 7-4 Required Switches and Options for Building a Shared Region

Switch/Option in DCL (MCR)	Effect	Defaults	Notes
/NOHEADER (/-HD)	No task header	/HEADER	
/SYMBOL_TABLE (Specify third output file)	Create a .STB file	No .STB file	Needed for task-building referencing task
STACK = Ø	No space for stack in .TSK file	STACK = 256(1Ø) words	
PAR = par[:base:len]	Specify partition name (set base virtual address - required if absolute; must also specify length, zero or maximum)	PAR = GEN If base and length are not specified, information is taken from partition on the system	Partition name must be same as name of the .TSK and .STB files For PI regions, if specifying base and length, use base = Ø, length = Ø or maximum

Example 7-1 has the source code for a resident common COMWP and a referencing task COMGP. Overlaid Psects are used for referencing the region. The following procedure is used to create the resident common.

1. Code the shared region.

   See COMWP.MAC in Example 7-1. The following notes are keyed to the example.

   1 The code is placed in an OVR Psect named MYDATA. This same Psect is used in the referencing task.

   2 This series of assembler directives is equivalent to 128(1Ø) .WORD 3 assembler directives. It initializes the first 128(1Ø) words in the region to 3.

   3 This series of assembler directives initializes the next 128(1Ø) words in the region to 6.

2. Assemble the shared region.

   ```
   >MACRO/LIST COMWP
   ```

3. If necessary, create the common type partition.

   We will make a partition COMWP, eight blocks = 1ØØØ(8) bytes long. If the partition TSTPAR already exists on your system, you may be able to eliminate it and then set up your partition. Be sure to check with your system manager before doing this and also be sure to put TSTPAR back when you are finished.

   ```
   !  Check current partitions on the system
   >SHOW PARTITIONS
   !Record base address and length of TSTPAR and the type
   !of partition. Convert the values to blocks by
   !dropping the last two zeroes. (For example,
   !base address 1234ØØ(8) = 1234 blocks,
   !length = 2ØØØØ(8) bytes = 2ØØ(8) blocks)
   !  Eliminate the partition TSTPAR
   >SET NOPARTITION:TSTPAR
   !  Create the partition COMWP
   >SET PARTITION:COMWP/BASE:1234/SIZE:1Ø/COMMON
   !  Check to see if this worked correctly
   >SHOW PARTITIONS
   ```

Later, to eliminate the partition and to replace TSTPAR, use the commands:

```
>SET NOPARTITION:COMWP
>SET PARTITION:TSTPAR/BASE:1234/SIZE:200/TASK
```

4. Task-build the shared region.

To build position independent:

```
>LINK/OPTIONS/MAP/SHAREABLE:COMMON/NOHEADER -
->/SYMBOL_TABLE/CODE:PIC COMWP
Option? STACK=0
Option? PAR=COMWP
Option? <RET>
```

The /OPTIONS switch allows you to enter options. /MAP indicates that you want a map file. /SHAREABLE:COMMON indicates that Psect names are to be placed in the .STB file (required to reference the shared region using overlaid Psects). /NOHEADER means do not include a task header in the task image because this is not an executable task. /SYMBOL_TABLE means make a .STB file (COMWP.STB). /CODE:PIC means position independent code for a position independent region. STACK = 0 means no stack space is needed because this is not an executable task. PAR = COMWP means the partition is COMWP. The Task Builder gets the length (for a maximum check) from the partition on the system.

To build absolute:

```
>LINK/OPTIONS/MAP/SHAREABLE:COMMON/NOHEADER -
->/SYMBOL_TABLE COMWP
Option? STACK=0
Option? PAR=COMWP:160000:20000
Option? <RET>
```

Only changes:

1. Omit /CODE:PIC.

2. Specify a base virtual address and a maximum length. The base virtual address must correspond to a base virtual address for an APR (e.g., 2, 20000(8), 40000(8), 60000(8), 100000(8), 120000(8), 140000(8), or 160000(8)). The APRs used must be available in all referencing tasks.

5. Install the region.

   ```
   >INSTALL COMWP
   ```

   Installs the region and also loads it into memory. Note that this is different from an executable task, which usually isn't loaded until it is requested.

```
        .TITLE COMWP
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;+
; File COMWP.MAC
;
; Program which creates and initializes a common region
; which will be referenced using overlaid Psects.
;
; Task-build instructions: Must include /SHAREABLE:COMMON
; and /NOHEADER switches; STACK=0 and PAR=COMWP options.
; Must create .STB file. May be /CODE:PIC or absolute
; (default).
;
; The code is placed in a Psect named MYDATA
;-
        .PSECT  MYDATA D,GBL,OVR ; Defaults REL,RW
        .REPT   128.                ; Repeat count
        .WORD   3.                  ; Word value of 3.
        .ENDR                       ; End repeat range
        .REPT   128.                ; Repeat count
        .WORD   6.                  ; Word value of 6.
        .ENDR                       ; End repeat range
        .END
```

```
        .TITLE  COMGP
        .IDENT /01/
        .ENABL  LC                  ; Enable lower case
;+
; FILE COMGP.MAC
;
; This task gets the values from the static common
; region COMWP. It uses the technique of overlaid Psects
; to reference the region.
;
; Task-build instructions:
;
;       >LINK/MAP/OPTION COMGP
;       Option? RESCOM=COMWP/RO
;       Option? <RET>
;-
        .MCALL  QIOW$S,EXIT$S      ; System macros
        .PSECT  MYDATA D,GBL,OVR ; Psect used in COMWP
M=.                                 ; local symbol for start
                                    ;  of region
        .PSECT                      ; Back to blank Psect
IOSB:   .BLKW   2                   ; I/O status block
ARG:    .BLKW   1                   ; Argument block for
                                    ;  error code
BUFF:   .BLKB   100.                ; Output buffer
FMT:    .ASCIZ  /%8D/               ; Format string for
                                    ;  output of data
FERR1:  .ASCIZ  /DIR ERROR ON QIO. DSW = %D/ ; Directive
                                    ;  error message
FERR2:  .ASCIZ  !I/O ERROR ON QIO. CODE = %D! ; I/O error
                                    ;       message
```

Example 7-1 Resident Common Referenced With Overlaid Psects (Sheet 1 of 3)

```
        N=32.                           ; Loop count - 32. lines,
                                        ;  8 #s per line
        .EVEN
;
START:  MOV     #M,R2                   ; Starting addr of data
                                        ;  in the region
        MOV     #N,R5                   ; Loop count
LOOP:   MOV     #BUFF,R0                ; Output buffer
        MOV     #FMT,R1                 ; Format string
        CALL    $EDMSG                  ; Edit message
        QIOW$S  #IO.WVB,#5,#1,,#IOSB,,<#BUFF,R1,#40>
        BCS     ERROR                   ; Check for dir error
        TSTB    IOSB                    ; Check for I/O error
        BLT     ERROR1                  ; Branch on I/O error
; Stay here for good write
        SOB     R5,LOOP                 ; Decrement counter, loop
                                        ;  back if not yet done
        EXIT$S                          ; Exit
; Error code
ERROR:  MOV     $DSW,ARG                ; Move DSW to arg block
        MOV     #FERR1,R1               ; Addr of format string
        BR      SETUP                   ; Branch to $EDMSG code
ERROR1: MOVB    IOSB,R0                 ; Extend sign on I/O
        MOV     R0,ARG                  ;  status and place in
                                        ;  arg block
        MOV     #FERR2,R1               ; Addr of format string
SETUP:  MOV     #BUFF,R0                ; Addr of output buffer
        MOV     #ARG,R2                 ; Addr of argument block
        CALL    $EDMSG                  ; Edit message
        QIOW$S  #IO.WVB,#5,#1,,,,<#BUFF,R1,#40> ; Write
                                        ;  message
        EXIT$S                          ; Exit
        .END    START
```

Example 7-1 Resident Common Referenced With Overlaid Psects
(Sheet 2 of 3)

```
Run Session

>INS COMWP
>RUN COMGP
3       3       3       3       3       3       3       3
3       3       3       3       3       3       3       3
                                .
                                .
                                .
3       3       3       3       3       3       3       3
6       6       6       6       6       6       6       6
6       6       6       6       6       6       6       6
                                .
                                .
                                .
6       6       6       6       6       6       6       6
>
```

Example 7-1 Resident Common Referenced With Overlaid Psects
(Sheet 3 of 3)

Creating a Referencing Task

1. Code the task, using the corresponding referencing technique.

   - If Psect names are kept in the .STB file of the referencing task, avoid Psect conflicts.

2. Assemble the task.

3. Task-build the task.

   - Specify shared regions using one of the following options:

     COMMON=common name for a system resident common (.STB and .TSK files must be in LB:[1,1]).

     LIBR=common name for a system resident library (.STB and .TSK files must be in LB:[1,1]).

RESCOM=common name for a user resident common (.STB and .TSK files in any device and any UFD using normal defaults).

RESLIB=common name for a user resident library (any device and any UFD using normal defaults).

- Append :RO if read-only access is desired.
  :RW if read-write access is desired.
  Use "/" instead of ":" for RESCOM and RESLIB.

- Only if the shared region is position independent, can you specify the base APR to be used to map the region. If not specified, the highest available APR or set of APRs is used, as needed.

4. After installing the shared region, install and/or run the task.

If the shared region is to be a system shared region, the .STB file and the .TSK file should be placed in LB:[1,1]. Otherwise, they can reside on any device under any UFD, as long as both files are in the same UFD on the same device.

Read-only or read/write access affects the way the access bits in the page descriptor registers (PDRs) in the APRs are set up. A memory protect violation occurs if a task attempts to write to a region when it has read-only access.

COMGP.MAC in Example 7-1 contains the source code for a task to reference the shared region COMWP. Use the following procedure to create the task.

1. Code the task.

   See COMGP.MAC in Example 7-1. The following notes are keyed to the example.

   1 The same Psect, MYDATA, is used here as in COMWP.MAC to set up referencing. M marks the beginning of the region. No initialization of the Psect can be performed in the referencing task.

   2 The main code is in the blank (. BLK.) Psect.

   3 Move the starting address of the region to R2.

   4 We use $EDMSG to set up each line of the display. We loop through once for each line, editing and displaying the values.

   5 The format string for $EDMSG. %8D means convert eight words to signed decimal, with a tab between values.

2. Assemble the task.

3. Task-build the task.

   ```
   >LINK/OPTION/MAP COMGP
   Option?  RESCOM = COMWP/RO
   Option?  <RET>
   ```

   Link task to resident common COMWP. COMWP.TSK and CONWP.STB are in the current UFD on SY:. Set up read-only access. Use the highest available APR, APR 7, if the region was built position independent.

4. After installing the shared region, install and/or run the task.

   To do a temporary install, run, remove:

   ```
   >RUN COMGP
   ```

   To install and then run:

   ```
   >INSTALL COMGP
   >RUN COMGP
   ```

Deciding whether read-only or read/write access to a region is required is usually straightforward. If a task moves data into the region or changes a value in the region, read-write access is required. If a task moves data out of the region or just reads values in the region, just read-only access is required.

However, when QIOs are issued and the buffer is in the shared region, things get a little tricky. Obviously, to do a read (e.g., from a terminal) into a buffer in the shared region requires write access. A write (e.g., to a terminal) from a buffer in the region should only require read access. However, because the Executive is designed for very fast, real-time applications, it does not check the function code for a QIO directive to see whether it is a read or a write. Instead it assumes the worst case - that all QIOs involving a buffer in a shared region are reads (from a peripheral device) into a buffer in the region, and that therefore, all QIOs require read/write access. This condition causes an I/O error (IE.SPR) for illegal user buffer. This condition does not affect Example 7-1 because $EDMSG creates the output string in a buffer within the referencing task area, and the QIOs do the writes from the referencing task area.

In an example in a later module, you will see this problem come up. One solution is to get read/write access to the shared region. Another solution is to move the data from the shared region to a buffer in the referencing task area, and then use that buffer for the QIOs. A third solution is to build the task as a privileged task.

Privileged tasks, similar to privileged terminals, are granted certain extra access to the system which nonprivileged tasks don't have. Some privileged tasks just gain these extra access rights, others map to the Executive as well. Normally, the Task Builder builds a task as a nonprivileged task. For a discussion of privileged tasks and how to task-build them, see Appendix D.

Example 7-2 shows a shared region (COMNP.MAC) and a referencing task (COMGGS.MAC) using global symbols to reference the shared region. Other than the difference in referencing technique, Example 7-2 is the same as Example 7-1. The following notes are keyed to the example.

1. Because the region is built with the /SHAREABLE:LIBRARY switch, any Psect names used in the file are not placed in the .STB file. Therefore, the code for the referencing task can be placed in the default blank (. BLK.) Psect or any other Psect. If the library were instead built /SHAREABLE:COMMON, the Psect names used in the shared region would all be placed in the .STB file. In that case, using any Psect in the referencing task which is also used in the shared region would cause a Psect conflict, causing a LOAD ADDRESS OUT OF BOUNDS task-builder error.

2. The global symbol K marks the beginning of the shared region.

3. The rest of the code is the same as COMWP.MAC in Example 7-1.

4. Just use the global symbol K to reference the start of the shared region. The Task Builder sets up the linkage to K, as it is defined in the shared region COMNP. The rest of the code is the same as that in COMGP.MAC in Example 7-1.

The tape supplied with this course also contains an example using virtual addresses as a referencing technique. The shared region is still COMNP, the same one as in Example 7-2. The referencing task code is in the file COMGVA.MAC. It should be in UFD [2Ø2,3] on your system. Check with your course administrator if you need help locating this example.

```
        .TITLE  COMNP
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; File COMNP.MAC
;
; Program which creates and initializes a common region
; which will be referenced using global symbols or
; actual virtual addresses.
;
; Task-build instuctions: Must include /SHAREABLE:LIBRARY
; and /NOHEADER switches, STACK=0 and PAR=COMNP options.
; Must create a .STB file. May be /CODE:PIC or absolute
; (the default).
;
; This program places the code in the default blank
; (. BLK.) psect. It could be in any psect. Psect
; conflicts are avoided by using the /SHAREABLE:LIBRARY
; switch on the task builder.
;-
; Define K, a global symbol
K::     .REPT   128.                    ; Repeat count
        .WORD   3.                      ; Word value of 3.
        .ENDR                           ; End repeat range
        .REPT   128.                    ; Repeat count
        .WORD   6.                      ; Word value of 6.
        .ENDR                           ; End repeat range
        .END
```

```
        .TITLE  COMGGS
        .IDENT /01/
        .ENABL  LC                      ; Enable lower case
;+
; FILE COMGGS.MAC
;
; This task gets the values from the static common
; region COMNP. It uses a global symbol to reference
; the region.
;
; Task-build notes:
;
;       LINK/MAP/OPTION COMGGS
;       Option? RESCOM=COMNP/RO
;       Option? <RET>
;-
        .MCALL  QIOW$S,EXIT$S   ; External system macros
```

Example 7-2 Resident Common Referenced With Global Symbols (Sheet 1 of 3)

```
;
IOSB:   .BLKW   2                       ; I/O status block
ARG:    .BLKW   1                       ; Argument block for
                                        ;  error code
BUFF:   .BLKB   100.                    ; Output buffer
FMT:    .ASCIZ  /%8D/                   ; Format string for
                                        ;  output of data
FERR1:  .ASCIZ  /DIR ERROR ON QIO. DSW = %D/ ; Directive
                                        ;  error message
FERR2:  .ASCIZ  !I/O ERROR ON QIO. CODE = %D! ; I/O
                                        ;  error message
        N=32.                           ; Loop count - 32. lines,
                                        ;  8 #s per line
        .EVEN
START:  MOV     #K,R2                   ; Starting addrress of
                                        ;  data in the region
        MOV     #N,R5                   ; Loop count
LOOP:   MOV     #BUFF,R0                ; Output buffer
        MOV     #FMT,R1                 ; Format string
        CALL    $EDMSG                  ; Edit message
        QIOW$S  #IO.WVB,#5,#1,,#IOSB,,<#BUFF,R1,#40>
        BCS     ERROR                   ; Check for dir error
        TSTB    IOSB                    ; Check for I/O error
        BLT     ERROR1                  ; Branch on I/O error
; Stay here for good write
        SOB     R5,LOOP                 ; Decrement counter, loop
                                        ;  back if not yet done
        EXIT$S                          ; Exit
; Error code
ERROR:  MOV     $DSW,ARG                ; Move DSW to arg block
        MOV     #FERR1,R1               ; Addr of format string
        BR      SETUP                   ; Branch to $EDMSG code
ERROR1: MOVB    IOSB,R0                 ; Extend sign on I/O
        MOV     R0,ARG                  ;  status and place in
                                        ;  arg block
        MOV     #FERR2,R1               ; Addr of format string
SETUP:  MOV     #BUFF,R0                ; Addr of output buffer
        MOV     #ARG,R2                 ; Addr of argument block
        CALL    $EDMSG                  ; Edit message
        QIOW$S  #IO.WVB,#5,#1,,,,<#BUFF,R1,#40> ; Write
                                        ; error message
        EXIT$S                          ; Exit
        .END    START
```

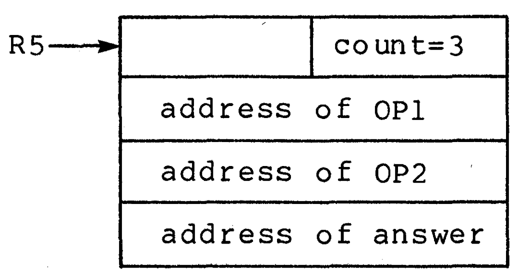
Example 7-2 Resident Common Referenced With Global Symbols (Sheet 2 of 3)

```
Run Session

>INS COMNF
>RUN COMGGS
3       3       3       3       3       3       3       3
3       3       3       3       3       3       3       3
                                .
                                .
                                .
3       3       3       3       3       3       3       3
6       6       6       6       6       6       6       6
6       6       6       6       6       6       6       6
                                .
                                .
                                .
6       6       6       6       6       6       6       6
>
```

Example 7-2 Resident Common Referenced With Global Symbols (Sheet 3 of 3)

Example 7-3 contains a shared library, LIB.MAC, and a referencing task, USELIB.MAC. The shared library contains four simple arithmetic routines to add, subtract, multiply, and divide two numbers. They are all written to be reentrant, plus they can be called from a FORTRAN program with a standard FORTRAN subroutine call. Basically, this means that on entry the arguments are assumed to be set up as follows:



For additional information on the FORTRAN/MACRO-11 interface, see Appendix C. Each subroutine saves and restores all of the registers, using the system library routine $SAVAL. The referencing task, USELIB, calls each of the subroutines once, using the operands 8(10) and 2(10), and displays just the answers for the four operations. The following notes are keyed to Example 7-3.

1. Each subroutine entry point is defined with a global symbol.

2. Each subroutine is in a Psect of the same name as the subroutine. In fact, the Psects are optional since the library is built /SHAREABLE:LIBRARY. The specified Psect names are not placed in the .STB file.

3. For AADD and SUBB, move the first operand to R0, perform the operation in R0, then move the answer to the third operand for return to the caller.

4. For MULL, use R1 instead of R0, so that the product is limited to just R1 (16 bits). If R0 were used instead, a 32-bit product is returned (low-order 16 bits in R1, high-order 16 bits in R0).

5. For DIVV, a 32-bit dividend is assumed in Rn and Rn+1, so here it is R2 and R3 (low-order 16 bits in R3, high-order 16 bits in R2). Therefore, the 16-bit operand is placed in R3 and the high-order word is cleared. The 16-bit quotient, returned in R2, is then moved into the third operand for return to the caller.

6. The two operands.

7. Space allocated for return of the result.

8. FORTRAN type argument block is built on the stack, in reverse order, so that SP points to the start of the block.

9. The address in SP is moved to R5, so R5 points to the start of the argument block.

10. Call each of the routines. Since R5 and the stack are not disturbed between calls, the same argument block can be used for all four calls.

11. Call the subroutine PRINT to edit the output message and display it for all operations.

```
        .TITLE  LIB
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;+
; File LIB.MAC
;
; This file contains the FORTRAN callable subroutines
; AADD, SUBB, MULL, and DIVV, which perform the
; appropriate integer operation.
;
; Calling convention: CALL sub (op1,op2,ans)
;
; Task-build instructions: Must include /SHAREABLE:LIBRARY
; and /NOHEADER switches; STACK=0 and PAR=LIB options.
; Must create .STB file. May be /CODE:PIC or absolute
; (default). Using /SHAREABLE:LIBRARY avoids Psect
; conflicts.
;-
        .PSECT  AADD,RO,I,GBL,REL,CON
AADD::  CALL    $SAVAL              ; Save all registers
        MOV     @2(R5),R0           ; Move 1st operand
        ADD     @4(R5),R0           ; Add 2nd operand
        MOV     R0,@6(R5)           ; Store result
        RETURN                      ; Restore regs and return

        .PSECT  SUBB,RO,I,GBL,REL,CON
SUBB::  CALL    $SAVAL              ; Save all registers
        MOV     @2(R5),R0           ; Move 1st operand
        SUB     @4(R5),R0           ; Subtract 2nd operand
        MOV     R0,@6(R5)           ; Store result
        RETURN                      ; Restore regs and return

        .PSECT  MULL,RO,I,GBL,REL,CON
MULL::  CALL    $SAVAL              ; Save all registers
        MOV     @2(R5),R1           ; Move 1st operand
        MUL     @4(R5),R1           ; Multiply (answer in
                                    ;  just R1)
        MOV     R1,@6(R5)           ; Store result
        RETURN                      ; Restore regs and return

        .PSECT  DIVV,RO,I,GBL,REL,CON
DIVV::  CALL    $SAVAL              ; Save all registers
        MOV     @2(R5),R3           ; Move 1st operand
        CLR     R2                  ; Clear high order 16 bits
        DIV     @4(R5),R2           ; Divide
        MOV     R2,@6(R5)           ; Store result
        RETURN                      ; Restore regs and return
        .END
```

Example 7-3 Shared Library (Sheet 1 of 2)

```
        .TITLE  USELIB
        .IDENT  /01/
        .ENABL  LC                    ; Enable lower case
;+
; File USELIB.MAC
;
; MACRO-11 task to use the resident library LIB
;
; Task-build instructions:
;
;       >LINK/MAP/OPTION USELIB
;       Option? RESLIB=LIB/RO
;       Option? <RET>
;-
        .MCALL  QIOW$S,EXIT$S         ; System macros

OP1:    .WORD   8.                    ; Operand 1
OP2:    .WORD   2                     ; Operand 2
ANS:    .BLKW   1                     ; Result

OUT:    .BLKW   100.                  ; Output buffer
FORMAT: .ASCIZ  /THE ANSWER = %D./ ; Format string
        .EVEN
; Build argument block for subroutine on the stack
START:  MOV     #ANS,-(SP)            ; For result
        MOV     #OP2,-(SP)            ; Operand 2
        MOV     #OP1,-(SP)            ; Operand 1
        MOV     #3,-(SP)              ; Number of arguments
        MOV     SP,R5                 ; R5=> arg block
        CALL    AADD                  ; Add operands
        CALL    PRINT                 ; Print results
        CALL    SUBB                  ; Subtract operands
        CALL    PRINT                 ; Print results
        CALL    MULL                  ; Multiply operands
        CALL    PRINT                 ; Print results
        CALL    DIVV                  ; Divide operands
        CALL    PRINT                 ; Print results
        EXIT$S                        ; Exit

;** PRINT - Prints the results of the operation
PRINT:  MOV     #OUT,R0               ; Set up for $EDMSG
        MOV     #FORMAT,R1            ;
        MOV     #ANS,R2               ;
        CALL    $EDMSG                ; Edit message
        QIOW$S  #IO.WVB,#5,#1,,,,<#OUT,R1,#40> ; Write
                                      ;  message
        RETURN                        ; Return
        .END    START
```

Run Session

```
>INS LIB
>RUN USELIB
THE ANSWER IS 10.
THE ANSWER IS 6.
THE ANSWER IS 16.
THE ANSWER IS 4.
>
```

Example 7-3 Shared Library (Sheet 2 of 2)

DEVICE COMMONS

A device common is a special type of common that occupies physical addresses on the I/O page. Instead of physical memory, the I/O page contains peripheral device registers. Therefore, a device common does not contain data the way a regular resident common does.

A device common is really just a way of setting up addressing to allow a task to manipulate the device registers directly. This might be useful in checking out the proper commands needed to control a device or to check what control status registers (CSRs) are in use on your system (Example 7-4). Obviously, extreme care must be used if you manipulate a device which is also referenced by any system routines (e.g., a system device driver).

Privileged tasks which map to the Executive can also automatically map the I/O page. However, privileged tasks must be written very carefully to avoid causing additional problems for the running system. Device commons allow nonprivileged tasks to manipulate device registers.

Use the procedure outlined below to create a device common and a referencing task. The outline includes the specific steps for Example 7-4. It has a device common, DEVICE.MAC, which covers the entire I/O page. The referencing task, CSR.MAC, checks each address on the I/O page to find out which CSRs are in use. If a nonexistent CSR is found, a nonexistent memory error synchronous system trap (SST) results. Use the following steps to create the device common.

1. Create a device common partition which includes the desired device register addresses.

   - Identify the addresses of the needed device registers, using the PDP-11 Peripherals Handbook or information available from your hardware installation.

   - Determine the base address of the partition at a 1ØØ(8) boundary below the first identified address.

     - Mapping always begins at a 32-word block boundary.

   Example 7-4 covers all of the I/O page. On a PDP-11/7Ø or other PDP-11 with 22-bit addressing, it starts at physical address 1776ØØØØ(8). On a system with 18-bit addressing, it starts at 76ØØØØ(8). On a system with 16-bit addressing, it starts at 16ØØØØ(8).

For a 22-bit system the command is:

```
SET PARTITION:DEVICE/BASE:177600/SIZE:200/DEVICE
```

For a 16-bit or 18-bit system, use the appropriate base address in 32-word blocks.

Note that you don't need to eliminate an existing partition the way you did for resident commons and resident libraries. This is because there isn't any real partition already on the system, because the I/O page does not correspond to physical memory.

You also don't need to create the partition until after you create the shared region. However, you do have to know its base address before you write the code for the device common, so that you can set up the offsets to the locations you plan to reference.

2. Code the shared region.

   - Do not initialize any locations, since there is no physical memory.

   - Set up for an appropriate referencing technique to address the desired registers.

     - Use .=.+n or .BLKB n to get to the first address.

     - Use .BLKB or .BLKW statements to reserve the needed space (or addresses).

   The following note is keyed to DEVICE.MAC in Example 7-4.

   ❶ Because you access the entire I/O page, mark the start of the region with the global symbol FCSR. The .BLKW 4096. directive reserves a full 4K words of addresses for the entire I/O page.

3. Assemble the device common.

```
>MACRO/LIST DEVICE
```

4. Task-build the device common.

```
>LINK/OPTION/MAP/NOHEADER/SHAREABLE:LIBRARY-
->SYMBOL_TABLE DEVICE
Option? STACK=0
Option? PAR=DEVICE:160000:20000
Option? <RET>
```

This command task-builds the region absolute. You can also task-build it position independent.

5. Install the device common before you run the referencing task. Unlike a resident common, a device common is not loaded into memory because it has no real contents.

Use the following steps to create the referencing task.

1. Code the referencing task using the corresponding referencing technique. See the notes which are keyed to the example below.

2. Assemble the task.

```
>MACRO/LIST CSR
```

3. Task-build the task to reference the device common.

```
>LINK/MAP/OPTION CSR
>Option?  RESCOM=CSR/RO
>Option?  <RET>
```

4. After installing the device common, install and/or run the referencing task.

The following notes are keyed to CSR.MAC in Example 7-4.

❶ Use the global symbol FCSR to reference the start of the device common.

❷ SST vector table with one entry for nonexistent memory. NONE is the address of the SST routine. (See Example 2 for an SST example.)

❸ Two words for a range of good CSR addresses. The addresses are offsets into the I/O page (Ø(8) to 17777(8)). FIRST is set initially with Ø; LAST is updated on each read of a CSR. If you ever trap due to nonexistent memory, print the range of addresses, set FIRST for the first address in the next range, and continue.

❹ Set up for SST, using just one table entry.

❺ Count of good addresses in a range. This is used to avoid printing a message if a number of consecutive addresses are not in use.

6 Set first range to start with offset Ø into I/O page.

7 Test (or read) the word and increment R4. Control passes to the next instruction if the CSR is in use; an SST results if it is not in use.

8 Increment count of good addresses.

9 Check to see if you are at the end of the I/O page. Branch back if not.

10 When at the end of the I/O page, display the last range and exit.

11 SST routine, entered for nonexistent memory trap on the TST(R4)+ instruction. Check for some good addresses in this range. If none, do not print a message.

12 Calculate offset to the last good CSR. The last one tested was bad, plus autoincrement incremented R4 by two. Therefore, the current contents of R4 are four bytes higher than the last good CSR. Also, convert the virtual address to an offset from the beginning of the I/O page. Move the last good CSR address to LAST.

13 Edit the range message, and convert the first good and last good addresses to unsigned octal. Then display the message.

14 Set up for the next range and return from the trap. The return picks up at line 49 (INC R5). We want R5 to be zero after it is incremented, so place a -1 in R5. Set up the first good CSR address in FIRST as the offset into the I/O page corresponding to the current address in R4. R4 has already been incremented past the CSR which is not in use. Return from trap at line 49, and continue check of CSRs, unless you have already reached the end of the I/O page.

15 On the Run Session - This command has probably been issued already to create the device partition. It is included here for documentation purposes, and in case it has not been issued previously.

```
        .TITLE  DEVICE
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;
; File DEVICE.MAC
;
; This program sets up a device common for the I/O page
;
; Task-build instructions: Must include /SHAREABLE:LIBRARY
; and /NOHEADER switches, STACK=0, PAR=DEVICE options.
; Must create .STB file. May be /CODE:PIC or absolute
; (the default).
;
; Install and run instructions: DEVICE must be installed
; before running any referencing task.
;
; The code is placed in the default blank Psect. Psect
; conflicts are avoided by using the /SHAREABLE:LIBRARY
; task-builder switch.
;
FCSR::  .BLKW   4096.       ; Set up area 4K words long
        .END
```

```
        .TITLE  CSR
        .IDENT  /01/
;+
; File CSR.MAC
;
; This task displays the CSR addresses that are in use
; on your system. The addresses are listed as offsets
; into the I/O page
;
; Task-build instructions:
;
;       LINK/MAP/OPTION CSR
;       Option? RESCOM=DEVICE/RO
;       Option? <RET>
;
; Install and run instructions: The device common DEVICE
; must be installed before running CSR.
;-
        .MCALL  QIOW$S,EXIT$S,SVTK$C      ; System macros
        .NLIST  BEX                 ; Do not list binary
                                    ;       extensions
; SST vector table
VEC:    .WORD   NONE                ; Nonexistent memory
```

Example 7-4 Creating and Using a Device Common (Sheet 1 of 3)

```
FIRST:  .BLKW   1                   ; First good CSR address
LAST:   .BLKW   1                   ; Last CSR address
                                    ;       before trapping
HDR:    .ASCII  /     **CSR'S IN USE ON SYSTEM:**/<15>
        .ASCII  <12>' (ADDRS ARE OFFSETS INTO I/O PAGE)'
        .ASCII  <15><12><12>        ; Header text
        LHDR    =.-HDR              ; Length of header text
MES:    .ASCIZ  /CSR'S %P THROUGH %P ARE IN USE/ ; Text
                                    ; for each good CSR range
BUFF:   .BLKB   100.                ; Output message buffer
        .EVEN
        .ENABL  LSB

START:  SVTK$C  VEC,1               ; Set up SST vector to
                                    ; handle trap
        QIOW$S  #IO.WVB,#5,#1,,,,<#HDR,#LHDR,#40>
                                    ; Display header text
        MOV     #FCSR,R4            ; Set base address in
                                    ; I/O page
        CLR     R5                  ; Count of addrs found
        CLR     FIRST               ; Offset to first CSR
                                    ;  addr in use
; Test address, causing trap if not in use
1$:     TST     (R4)+               ; Is this a good addr?
        INC     R5                  ; Yes, increment count
        CMP     #<FCSR+17776>,R4 ; At end of I/O page?
        BHIS    1$                  ; Branch back if not yet
; Display last good range and exit
        MOV     #17776,LAST         ; Put last CSR in LAST
        MOV     #BUFF,R0            ; Set up for $EDMSG
        MOV     #MES,R1             ;
        MOV     #FIRST,R2           ;
        CALL    $EDMSG              ; Edit range message
        QIOW$S  #IO.WVB,#5,#1,,,,<#BUFF,R1,#40>
                                    ; Display range message
        EXIT$S

; SST routine for non-existent memory (or CSR not in use)

NONE:   TST     R5                  ; Any good addresses in
                                    ; this range?
        BEQ     OUT                 ; None, nothing to print
        MOV     R4,R3               ; Calculate offset to
                                    ;       last good CSR
        SUB     #<FCSR+4>,R3        ;
        MOV     R3,LAST             ; Put last CSR in LAST
        MOV     #BUFF,R0            ; Set up for $EDMSG
```

Example 7-4 Creating and Using a Device Common (Sheet 2 of 3)

```
            MOV     #MES,R1             ;
            MOV     #FIRST,R2           ;
13          CALL    $EDMSG              ; Edit range message
            QIOW$S  #IO.WVB,#5,#1,,,,<#BUFF,R1,#40>
                                        ; Display range message
    ; Set addresses and counters for continued search
    OUT:    MOV     #-1,R5              ; Initialize count to -1
                                        ; since RTT returns to
                                        ; INC R5 instruction
14          MOV     R4,R3               ; Set up first good CSR
            SUB     #FCSR,R3            ;       in FIRST
            MOV     R3,FIRST            ;
            RTT                         ; Return from trap
            .END    START
```

```
15 >SET PARTITION:DEVICE/BASE:177600/SIZE:200/DEVICE
>INS DEVICE
>RUN CSR
      **CSR'S IN USE ON SYSTEM:**
  (ADDRS ARE OFFSETS INTO I/O PAGE)


CSR'S 000020 THROUGH 000106 ARE IN USE
CSR'S 004200 THROUGH 004236 ARE IN USE
CSR'S 005000 THROUGH 005776 ARE IN USE
CSR'S 010200 THROUGH 010376 ARE IN USE
CSR'S 010500 THROUGH 010526 ARE IN USE
CSR'S 012200 THROUGH 012376 ARE IN USE
CSR'S 012440 THROUGH 012476 ARE IN USE
CSR'S 012516 THROUGH 012516 ARE IN USE
CSR'S 013000 THROUGH 013776 ARE IN USE
CSR'S 016300 THROUGH 016352 ARE IN USE
CSR'S 016400 THROUGH 016452 ARE IN USE
CSR'S 016500 THROUGH 016506 ARE IN USE
CSR'S 016700 THROUGH 016752 ARE IN USE
CSR'S 017170 THROUGH 017176 ARE IN USE
CSR'S 017340 THROUGH 017356 ARE IN USE
CSR'S 017400 THROUGH 017416 ARE IN USE
CSR'S 017440 THROUGH 017476 ARE IN USE
CSR'S 017514 THROUGH 017526 ARE IN USE
CSR'S 017546 THROUGH 017546 ARE IN USE
CSR'S 017560 THROUGH 017676 ARE IN USE
CSR'S 017740 THROUGH 017752 ARE IN USE
CSR'S 017760 THROUGH 017776 ARE IN USE
>
```

Example 7-4 Creating and Using a Device Common (Sheet 3 of 3)

Appendix F contains information about several more advanced shared region topics. It includes a discussion of:

- Overlaid shared regions
- Referencing several shared regions from one referencing task
- Handling interlibrary calls
- Cluster libraries

Most of the techniques discussed are more appropriate for the advanced MACRO-11 programmer who is running into virtual address limitation problems. Cluster libraries are designed to save virtual address space in tasks which use DIGITAL layered products, such as FORTRAN, Forms Management Services (FMS), and File Control Services (FCS). If you write FORTRAN programs which use these products, you may find it useful to read just the last few pages. These cover the procedure for task-building a task which references two or more DIGITAL supplied resident libraries as a set of cluster libraries.

Now do the tests/exercises for this module in the Tests/Exercises book. They are all lab problems. Check your answers against the solutions provided, either the on-line files (should be under UFD [2Ø2,2]) or the printed copies in the Tests/Exercises book.

If you think that you have mastered the material, ask your course administrator to record your progress on your Personal Progress Plotter. You will then be ready to begin a new module.

If you think that you have not yet mastered the material, return to this module for further study.

DYNAMIC REGIONS

8

INTRODUCTION

The last module discussed how to use the Task Builder to create and access static regions. It is also possible to create and access regions while a task is executing. Such regions are called dynamic regions. The memory management directives allow a task to create and access dynamic regions and to access existing static or dynamic regions. In addition, they offer a facility for creating private regions and for allowing other tasks to access these regions.

OBJECTIVES

1. To write tasks which create a dynamic region and access dynamic and/or static regions

2. To write tasks which dynamically control their mapping

3. To write tasks which create a private dynamic region and allow one or more other tasks to access the region.

RESOURCE

- RSX-11M/M-PLUS Executive Reference Manual, Chapter 3 plus specific directives in Chapter 5

SYSTEM FACILITIES

Sometimes a task's needs for memory and for shared regions aren't known until run time, or the needs may change at run time. Examples are:

1. A task (e.g., an editor) needs a temporary work buffer for only part of the time the task is active.

2. A task needs a shared region or work buffer, but its size depends on the needs of the user running the task (e.g., the size of an input file).

3. A task creates a shared region and wants to control access to it by other tasks.

4. A task wants to create a shared region in a system controlled partition (e.g., GEN) instead of in a dedicated common type partition. Then when the shared region isn't needed, the space is automatically available for other system needs (tasks, etc.).

5. A task needs to map to two different shared regions at different times, but has only one 4K word virtual address window available.

Special directives, called memory management directives, are available on mapped systems to allow tasks to perform the following functions.

- Create regions in system controlled partitions

- Attach/detach from a region

- Create/eliminate virtual address windows

- Map/unmap a virtual address window to an attached region

- Obtain information about its mapping from the system.

The memory management directives are a SYSGEN option. Therefore, if users on a system plan to use them, they must be included in the Executive at SYSGEN time. Check with your system manager to find out if they have been included on your system.

Table 8-1 lists the memory management directives which are available on an RSX-11M system.

Table 8-1 Memory Management Directives

Function	MACRO-11
Attach Region	ATRG$
Create Address Window	CRAW$
Create Region	CRRG$
Detach Region	DTRG$
Eliminate Address Window	ELAW$
Get Mapping Context	GMCX$
Map Address Window	MAP$
Receive by Reference	RREF$
Send by Reference	SREF$
Unmap Address Window	UMAP$
Specify Receive by Reference AST	SRRA$

REQUIRED DATA STRUCTURES

Each memory management directive requires that you set up one of two data structures within your task - a region definition block (RDB) or a window definition block (WDB). The RDB and the WDB are the interface between the user task and the Executive. Their contents change dynamically as regions are created and accessed. In general, once the WDB and/or the RDB are set up, the actual memory management directive macro calls are completely straightforward. Their format is either:

```
xxxx$x wdb
```

or

```
xxxx$x rdb
```

where

wdb - the label or address of the WDB
rdb - the label or address of the RDB

Examples:

```
CRAW$C   WDB
CRRG$S   #RDB1
```

As with other executive directives, the $, $C, or $S form of each directive may be used.

Region Definition Block (RDB)

An RDB contains information needed to create a region and/or to attach to a region in a system controlled partition. The RDB is used by the following directives.

1. Attach Region (ATRG$)
2. Create Region (CRRG$)
3. Detach Region (DTRG$)

ARRAY ELEMENT	RDBBK$ ARGUMENT	SYMBOLIC OFFSET	BLOCK FORMAT	BYTE OFFSET
irdb (1)		R.GID	REGION ID	0
irdb (2)	siz	R.GSIZ	SIZE OF REGION (32W BLOCKS)	2
irdb (3)	nam	R.GNAM	NAME OF REGION (RAD50)	4
irdb (4)				6
irdb (5)	par	R.GPAR	REGION'S MAIN PARTITION NAME (RAD50)	10
irdb (6)				12
irdb (7)	sts	R.GSTS	REGION STATUS WORD	14
irdb (8)	pro	R.GPRO	REGION PROTECTION WORD	16

TK-7733

Figure 8-1 The Region Definition Block

Figure 8-1 shows the layout of the RDB along with the symbolic offsets. Use the RDBBK$ macro to create and initialize an RDB. Figure 8-1 also shows the RDBBK$ arguments for the various RDB elements. The meaning of the elements is as follows.

- Region ID - a unique number assigned to a region when your task attaches to a region. The number associates the task with the region and is returned by the Executive after your task attaches to a region.

- Size of Region - the size of a region to be created, in 32-word blocks. It is also returned by the Executive when attaching to an existing region.

- Name of Region - up to six characters. It is assigned when a region is created and used when attaching to a region.

- Region's Main Partition Name - the name of the system controlled partition.

- Region Status Word - used by the user task to send information to the Executive when creating or attaching to a region. Also used by the Executive to return status to the task after a memory management directive is executed. See Table 8-2 for a list of the various bits and their meanings.

- Region Protection Word - analogous to the file protection word, controlling access to regions. As shown below, it is set up with the same format (RWED for read, write, extend, delete) within each category; or: system, owner, group, and world.

World	Group	Owner	System	
DEWR	DEWR	DEWR	DEWR	
1110	1110	0000	0000	= 167000(8)

A '1' means access is denied, a '0' means access is permitted. So the example means that world and group have just read access, and owner and system have all accesses.

Table 8-2 Region Status Word

Symbol	Octal Value	Set By	Definition
RS.CRR	100000	System	Region successfully created
RS.UNM	40000	System	At least one window unmapped on a detach
RS.MDL	200	User	Mark region for deletion on last detach
RS.NDL	100	User	Created region not deleted on last detach
RS.ATT	40	User	Attach to created region
RS.NEX	20	User	Created region not extendable
RS.DEL	10	User	Delete access desired on attach
RS.EXT	4	User	Extend access desired on attach
RS.WRT	2	User	Write access desired on attach
RS.RED	1	User	Read access desired on attach

Creating an RDB in MACRO-11

The format for the RDBBK$ macro call is:

```
RDBBK$  siz,nam,par,sts,pro
```

No argument is provided for the region ID because it is always returned by the Executive and is never specified by the user. See Table 8-2 for a list of the region status word bits, including their symbols and meanings. We will discuss these further when we discuss the individual directives. Any information not filled in at assembly time using the RDBBK$ macro can be filled in using direct MOVs at run time.

Examples:

To create an RDB for use in creating a region with:

Size in 32(10) word blocks = 2
Region name = MYREG
Partition name = GEN
Region to be attached on create
Region to be marked for delete on last detach
Write access desired on attach
Owner to have all privileges and group to have read privileges.

```
RDBBK$    2,MYREG,GEN,<RS.ATT!RS.MDL!RS.WRT>,177017
```

Expansion:

```
.WORD     0                     ; Region ID
.WORD     2                     ; Region size
.RAD50    /MYREG/               ; Region name
.RAD50    /GEN   /              ; Partition name
.WORD     <RS.ATT!RS.MDL!RS.WRT> ; 000242(8) Region status
                                ; word
.WORD     177017                ; Region protection word
```

The example below shows the use of a MOV instruction to set the region size at run time.

To create an RDB for use in creating a region with:

Size in 32(10) word blocks = 1000(8)
Region name = XXXX
Partition name = same as task is installed in
Region status = do not delete, desired access to be filled in before attaching
World to have no privileges, all others to have all privileges

```
RDBBK$   0,XXXX,,RS.NDL,170000
```

Expansion:

```
.WORD    0                    ; Region ID
.WORD    0                    ; Region size
.RAD50   /XXXX  /             ; Region name
.WORD    0,0                  ; Partition name
.WORD    RS.NDL               ; 100(8), Region status word
.WORD    170000               ; Region protection word
   .
   .
   .
 MOV     #1000,RDB+R.GSIZ ; Set region size at run time
```

Window Definition Block (WDB)

A WDB contains information needed to create a virtual address region and to map a virtual address window to an attached region. The WDB is required for the following directives.

1. Create Address Window (CRAW$)

2. Eliminate Address Window (ELAW$)

3. Map Address Window (MAP$)

4. Unmap Address Window (UMAP$)

5. Send by Reference (SREF$)

6. Receive by Reference (RREF$).

Figure 8-2 shows the layout of the WDB along with the symbolic offsets. Use the WDBBK$ macro to create and initialize a WDB. Figure 8-2 also shows the WDBBK$ arguments. The meaning of the elements is as follows.

- Window ID - A number which identifies the window block in the task header which describes the window. Window Ø is used for the task window. Windows 1-7 are used for additional windows set up by the Task Builder, for overlays and static regions, and for windows created dynamically. The window ID is returned by the Executive after a Create Address Window directive.

- Base APR - The base APR to be used in mapping the window, which sets the base virtual address.

- Base Virtual Address - The base virtual address in octal; returned by the Executive after a Create Address Window directive.

- Region ID - The region ID, used to identify the region when mapping a virtual address window to a region. It is returned by the Executive in the RDB after an Attach Region directive. You must move the value returned from the RDB to the WDB before mapping to the region.

NOTE

The Task Builder option WNDWS=n must be used to specify the additional number of window blocks needed for dynamic windows.

ARRAY ELEMENT	WDBBK$ ARGUMENT	SYMBOLIC OFFSET	BLOCK FORMAT		BYTE OFFSET
iwdb (1)	apr	W.NID W.NAPR	BASE APR	WINDOW ID	0
iwdb (2)		W.NBAS	VIRTUAL BASE ADDRESS (BYTES)		2
iwdb (3)	siz	W.NSIZ	WINDOW SIZE (32W BLOCKS)		4
iwdb (4)	rid	W.NRID	REGION ID		6
iwdb (5)	off	W.NOFF	OFFSET IN REGION (32W BLOCKS)		10
iwdb (6)	len	W.NLEN	LENGTH TO MAP (32W BLOCKS)		12
iwdb (7)	sts	W.NSTS	WINDOW STATUS WORD		14
iwdb (8)	srb	W.NSRB	SEND/RECEIVE BUFFER ADDRESS		16

TK-7736

Figure 8-2 The Window Definition Block

- Offset in Region (32-word blocks) - The offset within the region at which mapping is to begin. It allows a task to map to different portions of a region.

- Length to Map (32-word block) - The length within the region to be mapped. It defaults to the shorter of the space remaining in the region and the size of the window.

- Window Status Word - Used by the user task to send information to the Executive when creating and mapping windows. It is also used by the Executive to return status to the user task after a directive is executed. Table 8-3 lists the various bits and their meanings.

- Send/Receive Buffer Address - The address of an 8-word buffer for sending or receiving data as part of the Send by Reference and Receive by Reference directives.

Creating a WDB in MACRO-11

The format of the WDBBK$ macro is:

```
WDBBK$  apr,siz,rid,off,len,sts,srb
```

Note that no argument is provided for either the window ID or the base virtual address, because these elements are always returned by the Executive. Table 8-3 shows a list of the window status word bits, including their symbols and meanings. We will discuss these further when we discuss the individual directives.

Table 8-3 Window Status Word

Symbol	Octal Value	Set By	Definition
WS.CRW	100000	System	Address window successfully created
WS.UNM	40000	System	At least one window unmapped by a CRAW$, MAP$ or UMAP$ directive
WS.ELW	20000	System	At least one window eliminated in a CRAW$ or ELAW$ directive
WS.RRF	10000	System	Reference successfully received
WS.64B	400	User	Defines permitted alignment for offset start within the region 0 for 256-word alignment (8 blocks) 1 for 32-word alignment (1 block)
WS.MAP	200	User	Window to be mapped in a CRAW$ or RREF$ directive
WS.RCX	100	User	Exit if no references
WS.DEL	10	User	Send with delete access
WS.EXT	4	User	Send with extend access
WS.WRT	2	User	Send or map with write access
WS.RED	1	User	Send with read access (map is with read access by default)

Examples:

To create a WDB to describe a window with the following:

APR = 7
Size in 32(1Ø) word blocks = 1ØØ(1Ø)
Region is to be mapped in a CRAW$ or RREF$ directive
Map with read access.

```
WDBBK$     7,1ØØ.,Ø,Ø,1ØØ.,<WS.MAP!WS.RED>
```

Expansion:

```
.BYTE      Ø,7                  ; Window ID, APR
.WORD      Ø                    ; Base virtual address
.WORD      1ØØ.                 ; Length
.WORD      Ø                    ; Region ID
.WORD      Ø                    ; Offset in region
.WORD      1ØØ.                 ; Length in region
.WORD      WS.MAP!WS.RED        ; ØØØ2Ø1(8), window status word
.WORD      Ø                    ; Send/Receive buffer address
```

To create a WDB to describe a window with the following characteristics:

APR = 5
Size in 32(1Ø) word blocks = 2ØØ(8)
Map starting at offset of 5 blocks in region and map 1Ø(1Ø) blocks
Send with delete and write access.

```
WDBBK$     5,2ØØ,Ø,5,1Ø.,<WS.64B!WS.WRT!WS.DEL>
```

Expansion:

```
.BYTE      Ø,5                  ; Window ID, APR
.WORD      Ø                    ; Base virtual address
.WORD      2ØØ                  ; Window length
.WORD      Ø                    ; Region ID
.WORD      5                    ; Offset in region
.WORD      1Ø.                  ; Length in region
.WORD      WS.64B!WS.WRT!WS.DEL ; ØØØ412(8), Window status
                                ; word
.WORD      Ø                    ; Send/Receive buffer address
```

CREATING AND ACCESSING A REGION

Use the following procedure to create and access a region.

1. Create the region (Create Region directive).

2. Attach to the region (Attach Region directive).

3. Move the region ID from the RDB to the WDB.

4. Create a virtual address window (Create Address Window directive).

5. Map the virtual address window to the region (Map Address Window directive).

6. Use the region.

7. Detach from the region (Detach Region directive or task exit).

Steps 1 and 2, and also steps 4 and 5 can each be combined in a single directive call. Step 4 can be performed earlier, if desired. To access an existing region, begin with step 2.

If you don't remember what windows and regions are, or what attaching and mapping mean, look over the sections on Windows and Regions in the last few pages of Module 5 on Memory Management.

The use of each directive in the procedure above is detailed on the following pages. The discussion includes the purpose of the directive, important input and output parameters, and notes about its use. For a complete discussion of each directive, see Chapter 5 of the RSX-11M/M-PLUS Executive Reference Manual. For additional information on the memory management directives, see Chapter 3 of the same manual.

Creating a Region

When you create a region, the Executive allocates space for it in a system controlled partition. Use the Create Region directive (CRRG$) with the following RDB input parameters.

- Size of region (in 32(10) word blocks)
- Name of region (becomes a private region if no name)
- Name of partition (defaults to partition of task)
- Region status word - mark for delete or do not delete (default is mark for delete)
- Region protection word - determines permissible access to region.

The only RDB output parameter is the RS.CRR bit in the region status word. It is set if the region is successfully created, and cleared if not. Normal Executive directive status is returned as well (carry set for error, clear for success; DSW contains directive status word). If the region already exists, success status is returned. Therefore, RS.CRR can be used to tell whether the region was in fact created, or whether it already existed.

Any task which passes the protection test can attach to a named region. For unnamed (private) regions, only tasks which are specifically attached by the creator of the region may attach to it. Therefore, for a private region, the creator completely controls which tasks attach to it and their access rights as well.

By default, or if RS.MDL is set in the region status word, the region is deleted when the last attached task detaches from the region. Named regions are left in existence after the last detach if RS.NDL is set in the region status word when the region is created. Unnamed (private) regions are always marked for delete (deleted on last detach). There is no explicit Delete Region directive.

If the RS.ATT bit is set in the region status word, the Executive also attempts to attach the task to the region. In this case, additional RDB input parameters are required, and additional output parameters are returned. Attaching to a region is discussed after Example 8-1.

Example 8-1 shows how to create a named region which is left in existence on last detach. The following notes are keyed to the example.

1. Set up the RDB. RS.NDL set specifies that the region is to be left in existence.

	World	Group	Owner	System	
	DEWR	DEWR	DEWR	DEWR	
Region protection word =	1111	ØØØØ	ØØØØ	ØØØØ	(2)
	1 7	Ø Ø	Ø	Ø	(8)

- Bit set means access is to be denied.

2. Issue the directive to create the region, specifying the RDB address as the only argument. Here we use the $C form of the directive. Any form is allowed.

3. Check for a directive error.

4. Display message and exit.

```
        .TITLE  CRERG
        .IDENT  /01/
        .ENABL  LC              ; Enable lower case
;+
; File CRERG.MAC
;
; CRERG creates a named region, and exits,
; leaving the region in existence.
;-
        .MCALL  EXIT$S,RDBBK$,CRRG$C ; System macros
        .MCALL  QIOW$C,QIOW$S
RDB:    RDBBK$  100,MYREG,GEN,RS.NDL,170000
;       Define region with:
;               Size            = 100 (32. word blocks)
;               Name            = MYREG
;               Partition       = GEN
;               Protection      = WO:None,SY:RWED
;                                 OW:RWED,GR:RWED
;               Do not mark for delete on last detach
SMES:   .ASCII  /CRERG SUCCESSFULLY CREATED MYREG/
LSMES   =.-SMES
BUFF:   .BLKB   80.             ; $EDMSG buffer
EFMT:   .ASCIZ  /ERROR IN CREATING REGION. DSW = %D./
        .EVEN
;
START:  CRRG$C  RDB             ; Create region
        BCS     ERR             ; Branch on dir error
        QIOW$C  IO.WVB,5,1,,,,<SMES,LSMES,40> ; Write
                                ;  success message
        EXIT$S                  ; Exit
; Error code
ERR:    MOV     #EFMT,R1        ; Set up for $EDMSG
        MOV     #$DSW,R2        ;
        MOV     #BUFF,R0        ;
        CALL    $EDMSG          ; Edit error message
        QIOW$S  #IO.WVB,#5,#1,,,,<#BUFF,R1,#40> ; Write
                                ;  message
        EXIT$S                  ; Exit
        .END    START
```

Run Session

```
>RUN CRERG
CRERG SUCCESSFULLY CREATED MYREG
>
```

Example 8-1 Creating a Named Region

Attaching to a Region

When you attach your task to a region, the Executive creates a logical connection between the two. The region can be either a dynamic region or a static region. Use the Attach Region directive (ATRG$) with the following RDB input parameters:

- Region name
- Region status word (indicating R,W,E,D access)

The following RDB output parameters are returned:

- Region ID
- Region size

The region ID is needed later in order to map a virtual address window to the region. The region size is of interest when attaching to an already existing region whose size may not be known.

Attaching can also be done as part of the Create Region directive (CRRG$), if the RS.ATT bit in the region status word is set when the Create Region directive is issued. In fact, for an unnamed region, attaching must be done as part of the Create Region directive, since there is no region name to be used in a separate Attach Region directive.

A task can detach from a region by using an explicit Detach Region directive (DTRG$) or by exiting (the Executive detaches the task). If a task is changing a region from do not delete to mark for delete, an explicit detach is required with RS.MDL set in the region status word. If the task exits without issuing an explicit detach, the Executive detaches the task but does not mark the region for delete. Once a region is marked for delete, it is deleted when the last attached task detaches from it. Once it is marked for delete, it cannot be changed to "do not delete." If a fixed task exits without issuing an explicit detach, no detach is performed by the Executive.

Creating a Virtual Address Window

When you create a virtual address window for a task, the Executive initializes a window block in the task header. It also checks to ensure that this is the only window that uses the specified range of virtual addresses, unmapping and eliminating any window that overlaps that range. Use the Create Address Window directive (CRAW$) with the following WDB input parameters.

- Base APR number
- Window size (in 32(10) word blocks)

The following WDB output parameters are returned:

- Window ID assigned by the system (1-7)
- Base virtual address

The space for the additional window blocks in the task header must be reserved at task-build time using the WNDWS=n option. N is the number of additional windows needed for windows created at run time. If extra space is not allocated, an address window allocation overflow error (IE.WOV = -85.) results.

The window is also mapped to a region if bit WS.MAP is set in the window status word when the Create Address Window directive is issued. In that case, addition input parameters are needed. See Mapping to a Region in the following section.

The Eliminate Address Window (ELAW$) directive can be used to explicitly eliminate a virtual address window. In general, it is not used, because creating a new window automatically eliminates any overlapping window.

Mapping to a Region

When you map a virtual address window to a region, the Executive creates a logical connection between the virtual address window and the region. Any attached region can be mapped. In the process, the memory management registers are loaded so that references to virtual addresses in the window access the region. This is assuming, of course, that the task keeps control of the CPU. The APRs are reloaded every time a new task takes control of the CPU.

Use the Map Address Window directive (MAP$) to map a window to a region, with the following WDB input parameters.

- Region ID - Returned to RDB by Attach (move from RDB to WDB).

- Offset into Region - in 32-word blocks, used to start mapping at an offset from the start of the region. This must be a multiple of 8(10), unless WS.64B is set in the window status word. If WS.64B is set, any whole number may be specified.

- Length to Map - If specified, must be less than, or equal to, either the length of the window or the length remaining in the region, whichever is shorter. If defaulted, it is set to the shorter of the two.

- Window status word - actual access desired (read-only, or read/write). Read-only is always requested by default.

The only WDB output parameter generally used is the length actually mapped. If the window is already mapped, it is first unmapped by the Executive. You can also use the Unmap Address Window directive to explicitly unmap a window. Mapping can also be done as part of the Create Address Window directive (CRAW$).

The type of access desired is used here in addition to when you attach to the region, because several different windows in the task may map the same region. Some of the windows may need read-only access, others may need read/write access. In that case, you must attach with read/write access, and then you may map each window with either read-only or read/write access.

Example 8-2 shows how to create a region and place data into it, leaving it in existence upon exit. Example 8-3 shows how to attach to that region, read and display the data, and then detach and mark it for delete. One run session covers both examples. The following notes are keyed to Example 8-2.

1. Task-build with the WNDWS=1 option, allocates space in the task header for one additional window block.

2. We use the $ form of the memory management directives.

3. RDB for region. RS.ATT set means Create Region directive will both create the region and attach to it.

4. WDB for virtual address window. The third argument is for the region ID, which will be filled in at run time, after the task attaches to the region. In the window status word, WS.MAP means that the Create Address Window directive will both create the window and map it to the region. WS.RED is automatic, even though not specified.

5. Create region and attach. Use DIR$ since you are using the $ form of the directive.

6. Move region ID, returned in RDB after attach, into WDB for mapping.

7. Create a virtual address window and map it to the region.

8. The virtual address window begins with APR 7; therefore, the base address in the window is 160000(8), corresponding to the base address in the region.

9. Place a byte count, 400(10), in the first word in the region. This is just one way to communicate this information to other tasks which access the region. The length of the region is returned when a task attaches to the region. You could use this as an alternate way to pass information about the amount of data.

10. Move 100(10) words of ASCII "AB" and 100(10) words of ASCII "12" into the region. This gives you 200(10) words or 400(10) bytes of data.

11. Display a successful creation and initialization message at the terminal.

12. Detach from the region and then exit, leaving the region in existence.

```
        .TITLE  CREURG
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;
; File CREURG.MAC
;
; Program to create a named region (attached on creation),
; create a virtual address window (mapped on creation),
; place ASCII data in to the region, detach from the
; region and exit, leaving the region in existence.
;
; Task-build instructions:
;
;       >LINK/OPTION/MAP CREURG
;       Option? WNDWS=1
;       OPTION? <RET>
;
        .MCALL  EXIT$S,RDBBK$,WDBBK$,CRRG$,CRAW$
        .MCALL  DTRG$,DIR$,QIOW$S,QIOW$C

REG:    CRRG$   RDB       ;DPB for create region
;       Define region with:
;               Size              = 100 (32. word blocks)
;               Name              = MYREG
;               Partition         = GEN
;               Protection        = WO:None,SY:RWED,
;                                   OW:RWED,GR:RWED
;               Do not mark for delete on last detach
;               Attach with read, write and delete access
WSW     = <RS.NDL!RS.DEL!RS.RED!RS.WRT!RS.ATT>
RDB:    RDBBK$  100,MYREG,GEN,WSW,170000
;
WIN:    CRAW$   WDB       ; DPB for create address window
;       Define window with:
;               APR                = 7
;               Size               = 100 (32. word blocks)
;               Offset in region = 0 (32. word blocks)
;               Length in region = 100 (32. word blocks)
;               Map on create with read/write access
WDB:    WDBBK$  7,100,0,0,100,<WS.MAP!WS.WRT>
;
DET:    DTRG$   RDB                 ; DPB for detaching
                                    ;  from region
IOSB:   .BLKW   2                   ; I/O status block
DNMES:  .ASCII  /CREURG HAS CREATED AND INITIALIZED THE/
        .ASCII  / REGION/
LDNMES  =.-DNMES
; Error format strings
FCRRER: .ASCIZ  /ERROR CREATING REGION. DSW = %D./
FCRWER: .ASCIZ  /ERROR CREATING WINDOW. DSW = %D./
FQIODE: .ASCIZ  /DIRECTIVE ERROR ON QIO. DSW = %D./
FQIOIE: .ASCIZ  !I/O ERROR ON QIO. CODE = %D.!
FDETER: .ASCIZ  /ERROR DETACHING FROM REGION. DSW = %D./

BUFF:   .BLKB   80.                 ; Output buffer
```

Example 8-2 Creating a Region and Placing Data in It (Sheet 1 of 2)

```
        .EVEN
START:  DIR$    #REG            ; Create region and attach
        BCS     ERR1            ; Check for error
        MOV     RDB+R.GID,WDB+W.NRID ; Move region ID
                                ;  into WDB
        DIR$    #WIN            ; Create window and map
        BCS     ERR2            ; Check for error
        MOV     #160000,R5      ; Set base addr in region
        MOV     #400.,(R5)+     ; Move byte count to 1st
                                ;  word in region
        MOV     #100.,R0        ; # of words of 'AB' data
LOOP:   MOV     #"AB,(R5)+      ; Move chars to region
        SOB     R0,LOOP         ; Decrement counter and
                                ;  loop until done
        MOV     #100.,R0        ; # of words of '12' data
LOOPB:  MOV     #"12,(R5)+      ; Move chars to region
        SOB     R0,LOOPB        ; Loop until done
        QIOW$C  IO.WVB,5,1,,IOSB,,<DNMES,LDNMES,40>
        BCS     ERR3D           ; Branch on dir error
        TSTB    IOSB            ; Check for I/O error
        BLT     ERR3I           ; Branch on I/O error
        DIR$    #DET            ; Detach from region
        BCS     ERR4            ; Check for error
        EXIT$S
; Error code
ERR1:   MOV     #FCRRER,R1      ; Create region error
                                ;  message
        BR      SHOERR          ; Branch to common code
ERR2:   MOV     #FCRWER,R1      ; Create window message
        BR      SHOERR          ; Branch to common code
ERR3D:  MOV     #FQIODE,R1      ; QIO directive message
        BR      SHOERR          ; Branch to common code
ERR3I:  MOV     IOSB,R0         ; Extend sign on status
        MOV     R0,$DSW         ;  and move to arg block
        MOV     #FQIOIE,R1      ; QIO I/O error
        BR      SHOERR          ; Branch to common code
ERR4:   MOV     #FDETER,R1      ; Detach region message
SHOERR: MOV     #BUFF,R0        ; Set up for $EDMSG
        MOV     #$DSW,R2        ;
        CALL    $EDMSG          ; Edit message
        QIOW$S  #IO.WVB,#5,#1,,,,<#BUFF,R1,#40>
                                ; Display message
        EXIT$S                  ; Exit
        .END    START
```

Run Session

```
>RUN CREURG
 CREURG HAS CREATED AND INITIALIZED THE REGION
>RUN ATTURG
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABAB1212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212
1212121212121212
>
```

Example 8-2 Creating a Region and Placing Data in It (Sheet 2 of 2)

Example 8-3 attaches to the region created by Example 8-2, reads and displays the data, and then detaches from the region and marks it for delete. The following notes are keyed to Example 8-3.

1. Again, task-build with the WNDWS=1 option so that the Task Builder allocates space for the window block in the task header.

2. This example uses all three forms of the directives, for illustration purposes.

3. The RDB for attaching to the region. In fact, the only required information is the region name and the region status word. The partition name and size, although included here, are not needed. RS.MDL set marks the region for delete when we do an explicit detach. You need delete access to mark the region for delete (RS.DEL). Also, attach with read (RS.RED) and write (RS.WRT) access so that you can map with read/write access.

4. The WDB for the virtual address window. We map the entire region (length = 100(8) 32-word blocks), starting from the beginning (offset = 0). WS.MAP means create the address window and map. Map with read (WS.RED) and write (WS.WRT) access.

5. Attach to the region.

6. Move the region ID to the WDB; create the virtual address window and map it to the region.

7. Set base address in region - again 160000(8), because the base APR is APR 7.

8. The first word in the region contains a character or byte count.

9. Number of characters to print on each line, except the last line (if it has less than 64(10) characters).

10 Loop through region, printing 64(10) characters per line. This technique is used to demonstrate how to control the width of the output and make the run session fit on an 8-1/2" by 11" page with margins. If the full terminal buffer width (typically 80(10) or 132(10)) is acceptable, one QIO directive, with the total character count specified for number of characters, would be enough to write the entire region. In that case, the terminal driver will automatically wrap to the next line after a full line is displayed.

11 Detach from the region. An explicit detach is required to mark the region for delete.

NOTE

In Chapter 7, we discussed the fact that a task needs read/write access to a region to issue QIOs to write directly from a region. This also applies to dynamic regions. ATTURG issues QIOs directly from MYREG. Therefore, although it appears that ATTURG only needs read access, it actually needs read/write access. See the discussion following Example 7-1 in Chapter 7 for additonal information.

```
        .TITLE  ATTURG
        .IDENT  /01/
        .ENABL  LC                ; Enable lower case
;+
; File ATTURG.MAC
;
; Program to attach to an existing region, create a
; virtual address window (mapped on creation), read
; ASCII data from the region, detach from the region
; and exit.  The region will be deleted on last detach.
; The first word in the region contains a count of how
; many bytes of data are in the region
;
; Assemble and task-build instructions:
;
;       >MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]ATTURG
;       >LINK/MAP/OPTION ATTURG,LB:[1,1]PROGSUBS/LIBRARY
;       >Option? WNDWS=1
;       >Option? <RET>
;-
        .MCALL  EXIT$S,RDBBK$,WDBBK$,ATRG$C ; System
        .MCALL  CRAW$,DTRG$S,DIR$,QIOW$S    ;  macros
        .MCALL  DIRERR,IOERR      ; Supplied macros
RDB:    RDBBK$  100,MYREG,GEN,<RS.MDL!RS.DEL!RS.RED!RS.WRT>
;       Define region with:
;               Size              = 100 (32. word blocks)
;               Name              = MYREG
;               Partition         = GEN
;               Mark for delete on last detach
;               Attach with read, write and delete access
;
WIN:    CRAW$   WDB     ;DPB for create address window
WDB:    WDBBK$  7,100,0,0,100,<WS.MAP!WS.RED!WS.WRT>
;       Define window with:
;               APR                = 7
;               Size               = 100 (32. word blocks)
;               Offset in region = 0 (32. word blocks)
;               Length in region = 100 (32. word blocks)
;               Map on create with read and write access
;
IOSB:   .BLKW   2                 ; I/O status block
START:  ATRG$C  RDB               ; Attach to region
        BCS     ERR1              ; Check for error
        MOV     RDB+R.GID,WDB+W.NRID ; Move region ID
                                     ; into WDB
        DIR$    #WIN              ; Create window
```

Example 8-3 Attaching to an Existing Region
and Reading Data From It (Sheet 1 of 2)

```
                BCS     ERR2                ; Check for error
                MOV     #160000,R5          ; Set base addr in region
                MOV     (R5)+,R4            ; Get character count
                MOV     #64.,R3             ; Chars per line
        LOOP:   QIOW$S  #IO.WVB,#5,#1,,#IOSB,,<R5,R3,#40>
                                            ; Write data
                BCS     ERR3D               ; Check for dir error
                TSTB    IOSB                ; Check for I/O error
                BLT     ERR3I               ; Branch on error
                SUB     R3,R4               ; Compute chars left
                BLE     DONE                ; Branch if done
                ADD     R3,R5               ; Point to next char
                CMP     R3,R4               ; Check for < 64. chars
                                            ;  left to print
                BLE     LOOP                ; > or =, print next line
                MOV     R4,R3               ; <, print only that many
                                            ;  chars
                BR      LOOP                ; Print the line
        DONE:   DTRG$S  #RDB                ; Detach from region
                BCS     ERR4                ; Check for error
                EXIT$S
        ; Error handling code
        ERR1:   DIRERR  <ERROR ATTACHING TO REGION>
        ERR2:   DIRERR  <ERROR CREATING WINDOW AND MAPPING>
        ERR3D:  DIRERR  <ERROR WRITING DATA>
        ERR3I:  IOERR   #IOSB,<ERROR WRITING DATA>
        ERR4:   DIRERR  <ERROR DETACHING FROM REGION>
                .END    START
```

Example 8-3 Attaching to an Existing Region and Reading Data From It (Sheet 2 of 2)

SEND- AND RECEIVE-BY-REFERENCE

If you create a private (unnamed) region, you have complete control over whether other tasks can have access to it. You specifically attach other tasks to the region by sending a packet containing a reference to the region. When you do that, you can also specify what access they have to the region. At the time, you must be attached with at least that much access yourself. Named regions, on the other hand, can be attached by any task that knows the name and has the access privileges needed to pass the protection check.

Use the Send-by-Reference directive (SREF$) to send a region by reference, with the following input parameters.

Receiver task name
WDB - Region ID
  offset into region - sent unchecked to receiver
  length to map - sent unchecked to receiver
  window status word - determines how receiving task is attached
  address of buffer - 8(10) word buffer which is sent to the receiver
Event flag - if specified, set when the reference is received, not when it is queued up (in the receive-by-reference queue).

The receiver task is attached to the region when the reference is queued up. This avoids the problem of the region being deleted if the sender exits before the receiver receives the region. Remember that private regions are always marked for delete on last detach.

If you are using the event flag for synchronization, note that the flag should be used to notify the sender as to when the receiver receives the region by reference. It is not the same as Send and Receive Data directives, where the flag is set when the reference is queued. That flag should be used to notify the receiver.

The receiver follows a somewhat modified procedure to access the region, as follows.

1. Create window.

2. After reference is queued, receive by reference (fills in region ID in WDB).

3. Map to region.

4. Use region.

5. Detach from region.

Use the Receive-by-Reference directive (RREF$) to receive a reference to a region, with the following WDB input parameters.

Window Status Word - WS.MAP for receive and map; WS.RCX for receive data or exit.

Buffer Address - 10(10) word buffer for sender task name (in Radix-50 format) and data.

The following WDB output parameters are returned, all as set by the sender:

Region ID
Offset into region
Length to map
Window status word - describes how attached

If the WS.MAP bit is set, the Executive maps the window to the region, using the offset, length, and window status word access as sent. If a separate Map directive is used, the receiver can first check and/or modify those parameters before mapping to the region. WS.RCX set tells the Executive to exit the task if there are no packets in the receive-by-reference queue.

Although there are some similarities, Send Data and Receive Data are completely independent from Send-by-Reference and Receive-by-Reference. The receive (data) queue is separate from the receive-by-reference queue.

If you want to use ASTs for synchronization, use the Specify Receive-by-Reference AST directive (SRRA$). This causes the Executive to transfer control to the specified AST routine when a packet is placed in the receive-by-reference queue. Generally, issue this directive when the task starts up.

Examples 8-4 and 8-5 show how to create a pair of tasks, a sender task and a receiver task. The sender, Example 8-4, creates a private region, initializes it, and sends a reference to it to the receiver. The receiver, Example 8-5, in turn receives the reference, displays the data, and then exits. One run session is included for both examples. The following notes are keyed to Example 8-4.

1. This program uses the supplied macro DIRERR to generate directive error messages. Therefore, PROGMACS.MLB must be specified when assembling, and PROGSUBS.OLB when task-building.

2. The RDB for the region. The name is defaulted to create a private region.

3. The WDB for the virtual address window. The length actually mapped will be returned after mapping. Read access is automatic for map, so WS.WRT gets read/write access.

4. Create and attach to region, create virtual address window and map it to the region.

5. Use the base virtual address in the window (returned in the WDB) to set the base address of the region. Since APR 7 is the base APR, this address is 160000(8).

6. Fill the region with ASCII Ms.

7. Send-by-reference to RCVREF (Example 8-4). Event flag 1 will be set when RCVREF actually does a receive-by-reference.

8. Display message that a region was created and sent. Then wait for event flag 1 to be set.

9. Display message saying RCVREF received region, and then exit.

10. Exit. The Executive will detach you from the region. Note that even if SNDREF exits before REVREF receives the region by reference, the region will not be deleted because RCVREF is attached when the reference is queued. The region is deleted only after both SNDREF and RCVREF detach.

```
        .TITLE  SNDREF
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; File SNDREF.MAC
;
; SNDREF creates a 64-word (2 block) unnamed region and
; fills it with ASCII characters. It then sends the
; region to RCVREF, and then waits for RCVREF to receive
; the region. (This is signalled by event flag #1.) It
; then prints a message and exits. Since the area is
; unnamed, it is automatically deleted when the last
; attached task exits.
;
; Assemble and task-build instructions:
;
;       >MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]SNDREF
;       >LINK/MAP/OPTION SNDREF,LB:[1,1]PROGSUBS/LIBRARY
;       Option? WNDWS=1
;
; Install and run instructions: RCVREF must be installed.
; Run SNDREF first, then run RCVREF.
;-
        .MCALL  QIOW$C,QIOW$S,RQST$C ; System macros
        .MCALL  WTSE$C,EXIT$S,RDBBK$,WDBBK$
        .MCALL  CRRG$S,CRAW$S,SREF$C
        .MCALL  DIRERR                  ; Supplied macro
        .NLIST  BEX                     ; SUPPRESS DATA

; Define region with:
;       Size            = 2     32-WORD BLOCKS
;       Name            = none
;       Partition       = GEN
;       Protection      = WO:none,GR:RWED
;                         OW:RWED,SY:none
;       Attach on create
;       Read and write access desired on attach
RPRO    = 170017
RSTAT   = RS.ATT!RS.RED!RS.WRT

RDB:    RDBBK$  2,,GEN,RSTAT,RPRO

; Define window with:
;       APR                 = 7
;       Size                = 2  32-word blocks
;       Offset in region    = 0  32-word blocks
;       Length to map       = 0  32-word blocks (defaults
;                                to smaller of region
;                                size and window length)
;       Map on create with read and write access
WSTAT   = WS.MAP!WS.WRT

WDB:    WDBBK$  7,2,0,0,,WSTAT
```

Example 8-4 Send-by-Reference (Sheet 1 of 2)

```
        ;
        MES1:   .ASCII  / SNDREF HAS CREATED THE REGION AND HAS/
                .ASCII  / SENT IT TO RCVREF./
        LMES1   =.-MES1
        MES2:   .ASCII  / RCVREF HAS RECEIVED IT. SNDREF IS NOW/
                .ASCII  / EXITING./
        LMES2 = . - MES2
                .LIST   BEX             ; Show binary extensions
                .EVEN
                .ENABL  LSB             ; Enable local symbol
                                        ;  blocks
        START:  CRRG$S  #RDB            ; Create and attach to
                                        ;  region
 4              BCS     1$              ; Branch on dir error
                MOV     RDB+R.GID,WDB+W.NRID ; Copy region ID
                                             ;  into WDB
                CRAW$S  #WDB            ; Create and map window
                BCS     2$              ; Branch on dir error
 5              MOV     WDB+W.NBAS,R0   ; base V.A. of region
        ; Fill region with all M's
 6              MOV     #64.,R3         ; count of words to move
        20$:    MOV     #"MM,(R0)+      ; Move in an ASCII M
                SOB     R3,20$          ; Loop through region
        ; Send the region to RCVREF. EF 1 will be set when
        ; RCVREF recieves it
 7              SREF$C  RCVREF,WDB,1    ; Send by reference to
                                        ;  RCVREF
                BCS     3$              ; Branch on dir error
                QIOW$C  IO.WVB,5,2,,,,<MES1,LMES1,40> ; Display
 8                                                     ;  message
                BCS     4$              ; Branch on dir error
                WTSE$C  1               ; Wait for RCVREF to set
                                        ; the region
                BCS     5$              ; Branch on dir error
                QIOW$C  IO.WVB,5,2,,,,<MES2,LMES2,40> ; Display
 9                                                 ;  message
                BCS     6$              ; Branch on dir error
10              EXIT$S                  ; Exit
        ; Error code
        1$:     DIRERR  <ERROR ON CREATE OR ATTACH REGION>
        2$:     DIRERR  <ERROR ON CREATE OR MAP WINDOW>
        3$:     DIRERR  <ERROR ON SEND BY REFERENCE>
        4$:     DIRERR  <ERROR ON 1ST WRITE>
        5$:     DIRERR  <ERROR ON WAIT FOR>
        6$:     DIRERR  <ERROR ON 2ND WRITE>
                .END    START
```

Run Session

```
>INS RCVREF
>SET TERMINAL/WIDTH:64.
>RUN SNDREF
  SNDREF HAS CREATED THE REGION AND HAS SENT IT TO RCVREF.
RUN RCVREF
>
   RCVREF HAS RECEIVED IT. SNDREF IS NOW EXITING.
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
>SET TERMINAL/WIDTH:80.
```

Example 8-4 Send-by-Reference (Sheet 2 of 2)

The receiver, Example 8-5, receives a reference, displays the data, then exits. The following notes are keyed to the example.

1. This program uses the supplied macros DIRERR and IOERR to display directive and I/O error messages.

2. WDB for virtual address window. The size is 200(8) 32-word blocks, a full 4K words. The offset into the region, the length to map, and the access will be filled in on receive. Since the length to map sent by SNDREF is two blocks, '2' will be used in mapping. Note that the window can be more than two blocks long. WS.MAP must be left clear until after the window is created. Otherwise, the Executive will try to map the window to the region, causing an error. See the discussion which follows.

3. Create the virtual address window.

4. Set WS.MAP so that the task will map as part of the receive-by-reference.

5. Receive-by-reference and map.

6. Set base address in region, using base virtual address for APR 7 (160000(8)).

7. Get length actually mapped (two blocks, the same as length of region) and convert from blocks to bytes. Just display that many characters.

8. Display all characters with one QIO directive. Note on the run session that we set the terminal buffer to 64(10) to allow for margins on an 8-1/2" by 11" page.

9. Exit. The Executive will detach the task from the region. When both tasks have detached, the region will be deleted.

The receiver may map after the receive-by-reference or as part of the receive-by-reference. If the receive-by-reference and the map are combined in one directive, issue the Receive-by-Reference directive with the WS.MAP bit set. In that case, the WS.MAP bit must be clear when the window is created, since you can't map until you receive. This is necessary because even though the receiver is attached to the region when the reference is queued up, the region ID isn't filled in the WDB until the receiver executes the Receive-by-Reference directive. Therefore, if you receive and map in one call, issue the Create Address Window directive with the WS.MAP bit clear; then set it before issuing the Receive-by-Reference directive. If you use a separate Map directive, the WS.MAP bit can be left clear.

```
        .TITLE  RCVREF
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;+
; File RCVREF.MAC
;
; Program to receive-by-reference a region from SNDREF,
; map to the region, read ASCII data from the region,
; detach from the region and exit. The region will be
; deleted on last detach.
;
; Assemble and task-build instructions:
;
;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]RCVREF
;       LINK/MAP/OPTIONS RCVREF,LB:[1,1]PROGSUBS/LIBRARY
;       Option? WNDWS=1
;       Option? <RET>
;
; Install and run instructions: RCVREF must be installed
; Run SNDREF first and then run RCVREF
;-
        .MCALL  EXIT$S,WDBBK$,RREF$ ; External system
        .MCALL  QIOW$S,CRAW$,DIR$   ;  macros
        .MCALL  DIRERR,IOERR        ; External supplied
                                    ;  macros
; Define window with:
;               APR                 = 7
;               Size                = 200 (32. word blocks)
;                                         Allow for full APR
;       These are filled in on receive, as set by sender:
;               Offset in region = 0 (32. word blocks)
;               Length in region = 0 (32. word blocks)
;                                       reset when mapped
;               Access              = 0
; Note: Must map after receiving (or as part of receive)
WDB:    WDBBK$  7,200
;
REC:    RREF$   WDB                 ; Set up DPB for RREF$
WIN:    CRAW$   WDB                 ; Set up DPB for CRAW$
IOSB:   .BLKW   2                   ; I/O status block

START:  DIR$    #WIN                ; Create virtual address
                                    ;  window
        BCS     ERR1                ; Branch on error
        BIS     #WS.MAP,WDB+W.NSTS ; Set WDB to map on
                                       ; receive
```

Example 8-5 Receive-by-Reference (Sheet 1 of 2)

```
          DIR$    #REC                ; Receive by reference
                                      ;  and map
          BCS     ERR2                ; Branch on error
          MOV     #160000,R5          ; Set base address in
                                      ;  region
          MOV     WDB+W.NLEN,R3       ; Size of region to R3
          MUL     #64.,R3             ; Convert blocks to bytes
          QIOW$S  #IO.WVB,#5,#1,,#IOSB,,<R5,R3,#40> ; Write
                                                    ;  data
          BCS     ERR3                ; Branch on directive
                                      ;  error
          TSTB    IOSB                ; Check for I/O error
          BLT     ERR4                ; Branch on error
          EXIT$S
 ; Error code
 ERR1:    DIRERR  <ERROR CREATING VIRTUAL ADDRESS WINDOW>
 ERR2:    DIRERR  <ERROR ON RECEIVE AND MAP>
 ERR3:    DIRERR  <ERROR ON WRITE QIO>
 ERR4:    IOERR   #IOSB,<ERROR ON WRITE QIO>
          .END    START
```

Example 8-5 Receive-by-Reference (Sheet 2 of 2)

The Mapped Array Area

If you want to automatically set up a large core resident data area, without using a create region directive, you may use special techniques to set up an area called a mapped array area. Figure 8-3 shows a task using a mapped array area. The Task Builder sets things up so that when the task is initially loaded, the task region is larger than normal, with the mapped array area set aside in memory immediately below the task header.

The task is automatically attached to the region, since it is part of the task region. Therefore, all you have to do is to create a virtual address window and map it to the region. The area may be any size, as long as the task image and the mapped array area fit into the partition. This means that it may be larger than 32K words.

Typically, the virtual address window maps only a portion of the region at a time. In Figure 8-3, the virtual address window maps 4K words at a time.

This technique is used to implement virtual arrays in FORTRAN. Since the area isn't set aside until the task is loaded into memory, any initialization of the area must be performed at run time.

Use the following procedure to create a task which uses the mapped array area.

1. Set up a separate Psect in the source code and reserve space for the virtual address window (using .BLKB or .BLKW statements). Also set up symbols for reference, if desired. Do not initialize any locations.

2. In the code, create a virtual address window.

3. Map the window to a portion of the region.

4. Later, map to other portions of the region by modifying the offset within the region and reissuing the map directive.

5. Task-build with the WNDWS and the VSECT options.

   WNDWS=n allocates space in the task header for the extra window block

   VSECT=psect-name:base:window-length:physical- length

   where

   - psect-name = the name of the psect to be used for the virtual address window
   - base = the base virtual address for the window
   - window-length = the length of the window in bytes
   - physical-length = the length of the mapped array area in 32-word blocks.

   This option sets up virtual addressing for the region and specifies the amount of space to be set aside for the mapped array area.

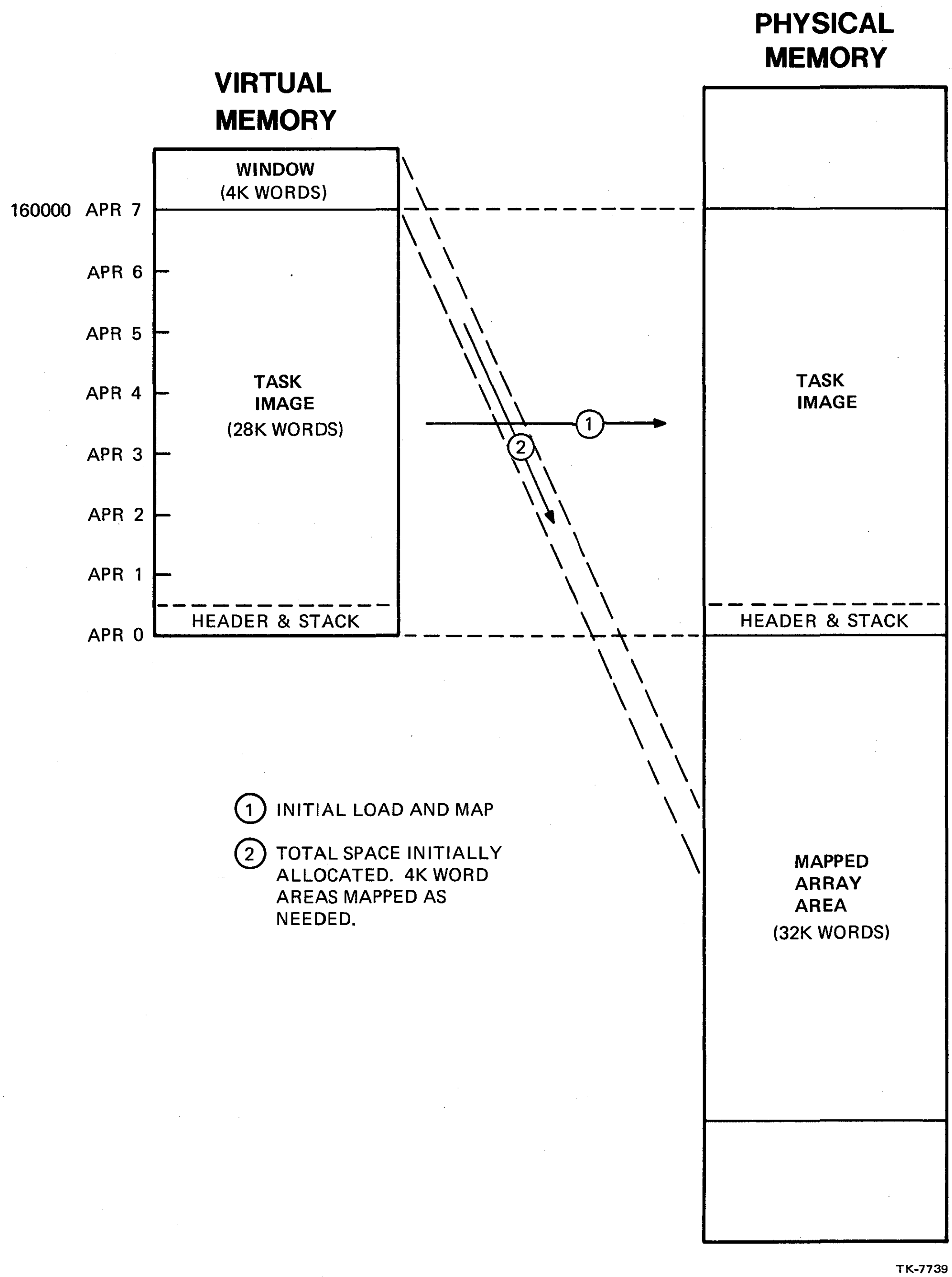


Figure 8-3 The Mapped Array Area

Example 8-6 shows how to create and use a mapped array area. The following notes are keyed to the example.

1. This program uses the supplied macro DIRERR.

2. WNDWS = 1 is needed to reserve space for the extra window block. The VSECT option sets up addressing for Psect VV, beginning at virtual address 160000(8), for a length of 20000(8) bytes or 4K words. The last argument sets up a mapped array area 2000 32-word blocks = 200000(8) bytes long = 32K words.

   Set up Psect VV, which is used for mapping the mapped array area. Symbol A marks the beginning of the window at virtual address 160000(8). The number of bytes reserved must be at least as long as the window size (4K words).

3. Data to be placed in the mapped array area.

4. WDB for the window. The region ID is left '0' because the region is the task region, which always has region ID 0.

5. Create the virtual address window and map starting at offset 0, to the first 4K word area.

6. Move "A1G7" into the first two words of the area.

7. Modify the offset in the region in the WDB (at offset W.NOFF) to 200(8) blocks, so that mapping will begin at that offset within the region.

8. Map to the second 4K word area.

9. Move "B2G7" into the first two words of the second 4K word area.

10. Similarly, map and move "C3G7" to the third 4K word area, and "D4G7" to the fourth 4K word area.

11. Map back to the first 4K word area.

12. Display the first four bytes.

13. Map to the second 4K word area and display the first four bytes.

14. Map to the fourth 4K word area and display the first four bytes.

15. Map to the third 4K word area and display the first four bytes.

The mapping order for displaying the data is different just to show that the order need not match the original order for placing the data into the region.

Now do the Tests/Exercises for this module in the Tests/Exercises book. They are all lab problems. Check your answers against the solutions provided, either the on-line files (which should be under UFD [202,2]) or the printed copies in the Tests/Exercises book.

If you think that you have mastered the material, ask your course administrator to record your progress on your Personal Progress Plotter. You will then be ready to begin a new module.

If you think that you have not yet mastered the material, return to this module for further study.

```
        .TITLE  VS3
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; This program uses the mapped array area.  It places
; data in the first 2 words of each of the first four
; 4K word blocks. It then retrieves the data and prints
; it at the terminal.
;
; Assemble and task-build instructions:
;
;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]VS3
;       LINK/MAP/OPTIONS VS3,PROGSUBS/LIBRARY
;       Option? WNDWS=1
;       Option? VSECT=VV:160000:20000:2000
;       Option? <RET>
;-
        .MCALL  QIOW$S,EXIT$S,WDBBK$,CRAW$S,MAP$S
                                        ; System macros
        .MCALL  DIRERR                  ; Supplied macro
        .PSECT  VV CON,GBL              ; Psect for mapped array
                                        ;  area
A:      .BLKB   20000                   ; Used to reference the
                                        ;  virtual area
        .PSECT                          ; Back to blank Psect
DATA:   .ASCII  /A1/
DATB:   .ASCII  /B2/
DATC:   .ASCII  /C3/
DATD:   .ASCII  /D4/
DATG:   .ASCII  /G7/
; Define window definition block
WDB:    WDBBK$  7,200,0,0,200,<WS.MAP!WS.WRT>,0
START:  CRAW$S  #WDB                    ; Create window and map
                                        ;  to 1st 4KW block
        BCC     A1                      ; Branch on dir ok
        DIRERR  <ERROR CREATING WINDOW OR ON FIRST MAP>
A1:     MOV     DATA,A                  ; Move data to 1st word
        MOV     #2,R5
        MOV     DATG,A(R5)              ; Move data to 2nd word
        MOV     #WDB,R0
        MOV     #200,W.NOFF(R0)         ; Set up next 4KW block
        MAP$S   #WDB                    ; Map 2nd 4KW block
        BCC     A4                      ; Branch on dir ok
        DIRERR  <ERROR ON 1ST MAP TO 2ND 4KW BLOCK>
A4:     MOV     DATB,A
        MOV     DATG,A(R5)
```

Example 8-6 Use of the Mapped Array Area (Sheet 1 of 2)

```
                MOV     #400,W.NOFF(R0) ; Set up 3rd 4K block
                MAP$S   #WDB
                BCC     A7
                DIRERR  <ERROR ON 1ST MAP TO 3RD 4KW BLOCK>
        A7:     MOV     DATC,A
                MOV     DATG,A(R5)
                MOV     #600,W.NOFF(R0) ; Set up 4th 4K block
                MAP$S   #WDB
                BCC     A8
                DIRERR  <ERROR ON 1ST MAP TO 4TH 4KW BLOCK>
        A8:     MOV     DATD,A
                MOV     DATG,A(R5)
                MOV     #0,W.NOFF(R0)   ; Go back to 1st 4K block
                MAP$S   #WDB
                BCC     A9              ; Branch on dir ok
                DIRERR  <ERROR ON 2ND MAP TO 1ST 4KW BLOCK>
        A9:     QIOW$S  #IO.WVB,#5,#1,,,,<#A,#4,#40>
                MOV     #200,W.NOFF(R0) ; Go to 2nd 4K block
                MAP$S   #WDB
                BCC     A10             ; Branch on dir ok
                DIRERR  <ERROR ON 2ND MAP TO 2ND 4KW BLOCK>
        A10:    QIOW$S  #IO.WVB,#5,#1,,,,<#A,#4,#40>
                MOV     #600,W.NOFF(R0) ; Go to 4th 4K block
                MAP$S   #WDB
                BCC     A11             ; Branch on dir ok
                DIRERR  <ERROR ON 2ND MAP TO 4TH 4KW BLOCK>
        A11:    QIOW$S  #IO.WVB,#5,#1,,,,<#A,#4,#40>
                MOV     #400,W.NOFF(R0) ; Go to 3rd 4K block
                MAP$S   #WDB
                BCC     A12             ; Branch on dir ok
                DIRERR  <ERROR ON 2ND MAP TO 3RD 4KW BLOCK>
        A12:    QIOW$S  #IO.WVB,#5,#1,,,,<#A,#4,#40>
                EXIT$S                  ; Exit
                .END    START
```

Run Session

```
>RUN VS3
A1G7
B2G7
D4G7
C3G7
>
```

Example 8-6 Use of the Mapped Array Area (Sheet 2 of 2)

FILE I/O

9

INTRODUCTION

The RSX-11M file system is composed of three parts.

- File structures - the organization and data structures maintained on the mass storage volumes themselves
- Ancillary Control Processors (ACPs) - tasks which maintain the file structures and provide access to them
- File access routines - provide user-written tasks with an interface to ACPs, which provide and maintain organization within files.

This module reviews some basic information about file storage, and provides general information about the RSX-11M primary file structure called FILES-11, and its ACP. This module also presents an overview and comparison of the two supplied file access subsystems, File Control Services (FCS) and Record Management Services (RMS). The following module provides details on programming using FCS, which is the more widely used subsystem.

OBJECTIVES

1. To describe the steps involved in file I/O
2. To describe the FILES-11 structure and how the F11ACP maintains that structure during file I/O
3. To identify the advantages of using either FCS or RMS for file access.

RESOURCES

1. IAS/RSX-11 I/O Operations Reference Manual, Chapters 1 and 5
2. RMS-11 User's Guide

OVERVIEW

Quite often in an application you need to store data on a peripheral device (disk, magtape, etc.) for later retrieval. To write such an application, you must know something about the different devices which are on your system. In addition, you must understand the file structure and its support systems. Once you know that, you can learn the procedure for actually performing I/O operations.

TYPES OF DEVICES

Record-Oriented Devices

Record-oriented devices have the following characteristics.

- Data is handled a record at a time.
- There is no file structure.

Terminals, line printers, and card readers are all record-oriented devices. They are not designed for storage and fast retrieval of data, but are designed instead to support interactive sessions or provide hard copies of reports and other data.

File-Structured Devices

File-structured devices have the following general characteristics. The data they contain:

- Can be handled in files

- Can be stored and retrieved quickly

- Is typically stored on a storage medium which can be moved from one device to another.

Hard disks, floppy disks, and magtape are examples of file-structured devices. The following definitions should prove helpful in our discussion.

a file - a collection of related data; therefore, a logical unit of mass storage.

volume - a physical unit of mass storage consisting of a recording medium and its packaging. Examples are a disk pack, a reel of tape, a diskette, and a DECtape II cartridge.

Types of File-Structured Devices - There are two types of file-structures devices, sequential and random-access. The type is determined by the kind of access to data on it.

Sequential devices have the following characteristics.

- Data is retrieved in the same order as written

- New data is always appended at the logical end of the tape, after the last data written

- data cannot be written in the middle of the volume without losing the data past that point.

Magtape and cassettes are examples of sequential devices. In essence, data is stored in order as written. To access any data, all data before it on the tape must be read first.

Under RSX-11M, the magtape ancillary control processor (MTAACP) supports the ANSI file structure.

The MTAACP supports the following file setups:

- A single file on a single volume
- A single file on multiple volumes
- Multiple files on a single volume
- Multiple files on multiple volumes

Random-access devices, also called block-structured devices or block-replaceable devices, have the following characteristics. They can:

- Store and retrieve data in units called blocks

- Write or read blocks in any order

- Rewrite blocks without interfering with other blocks.

Hard disks (RL01/02, RP06, RM02/03), diskettes (RX11, RX211) and DECtape II are examples of random-access devices.

The FILES-11 file structure, the standard RSX file structure, is supported by the FILES-11 ancillary control processor (F11ACP). F11ACP supports multiple files on a volume, but a file may not extend across volumes. The COPY command (PIP in MCR) maintains the FILES-11 structure during transfers of files within a given device and between FILES-11 devices on a system.

The ANSI file structure is useful for transfers of files between different (possibly non-DIGITAL) systems. FILES-11 is useful between DIGITAL systems under RSX-11M, RSX-11M-PLUS, IAS and VMS if the two systems have a device in common (e.g., both systems have RL02s). The FLX utility is provided to facilitate transfers between RSX and other DIGITAL systems which don't support FILES-11, or between systems which support FILES-11 (even between two RSX-11M systems) which do not have a common FILES-11 device. In that case, the FLX transfer is typically made on magtape, using DOS or RT-11 format.

COMMON CONCEPTS OF FILE I/O

Common Operations

File I/O is often used to perform the following operations.

- Creating a file
- Deleting a file
- Modifying existing data within a file
- Appending new data to a file (or extending the file).

Steps of File I/O

Use the following three basic steps to do file I/O.

1. Open the file.

   Specify a LUN and the file. The ACP connects a task LUN to the file. Specify the access rights desired. The ACP checks against the file protection code. If you are creating a new file, specify the file characteristics (e.g., format and initial length).

2. Perform the I/O operations.

   Use macros to invoke subroutines to store data in the file and/or retrieve data from the file.

3. Close the file.

   Notify the system that the file operations are completed, so that appropriate cleanup work can be performed.

FILES-11

In order to use FILES-11, you need to understand its structure and how to interact with it.

FILES-11 Structure

A block is the smallest unit of storage which is read from, or written to, a FILES-11 device. Typically, the blocks are 256(10) words or 512(10) bytes long. Some devices divide or format their volumes into pieces which are 256(10) words long, and others do not. Therefore, the FILES-11 structure does some converting or mapping so that you work with logical blocks which are all standard size. When the volume is formatted, logical block numbers are assigned to each 256(10) word area on the disk, starting with logical block 0. Generally, the position of data on a FILES-11 volume can be described in three alternate ways, by:

- Physical location
- Logical block number
- Virtual block number

Table 9-1 compares the three ways. Figure 9-1 shows an example of the mapping among the different methods. Typically, you will reference data only within files. The files are referenced by virtual block numbers within the file, starting with 1. Logical block numbers are assigned to the entire disk, starting from 0.

The system converts virtual block number references to logical block number references. For example, if you request a read of virtual block 5, the system looks at the mapping and finds that this corresponds to logical block 1622(8). This logical block, in turn, is mapped to one or more specific sectors on the disk, which are read from the disk.

Table 9-1 Comparison of Physical, Logical and Virtual Blocks

Type of Block Designation	Size	How Designated
Physical	Depends on device	On multi-platter disks, designated by cylinder, track and sector
Logical	256(10) words	Numbered in increments relative to the beginning of the volume, starting with 0
Virtual	256(10) words	Numbered in increments relative to the beginning of a file, starting with 1

Typically, data is accessed as records, units which are not exactly one block or 512(10) bytes long. A record is a unit of user specified size, corresponding, for example, to a single bank account or a single line of text at a terminal.

Figure 9-2 shows how the operating system handles a request to read a record using FCS. The first row shows a FORTRAN READ. The FORTRAN READ instruction is converted by the compiler to a GET$ call to the File Control Services (FCS) to read that record. In MACRO, you will issue the GET$ call yourself. FCS checks to find out which virtual block within the file contains that record and issues the QIO directive for you. The Executive converts the virtual block number to its corresponding logical block number and issues a read logical block QIO. The driver then converts the logical block number to the appropriate physical locations, and reads a block of data into memory. The record itself will then be located within the block of data.

The second row shows a BASIC-PLUS-2 READ under the Record Management Services (RMS). The BASIC-PLUS-2 compiler converts the READ to a RMS $GET call. RMS converts this to a QIO, to read the corresponding virtual block. From that point on, the steps are just like those in the FORTRAN example.

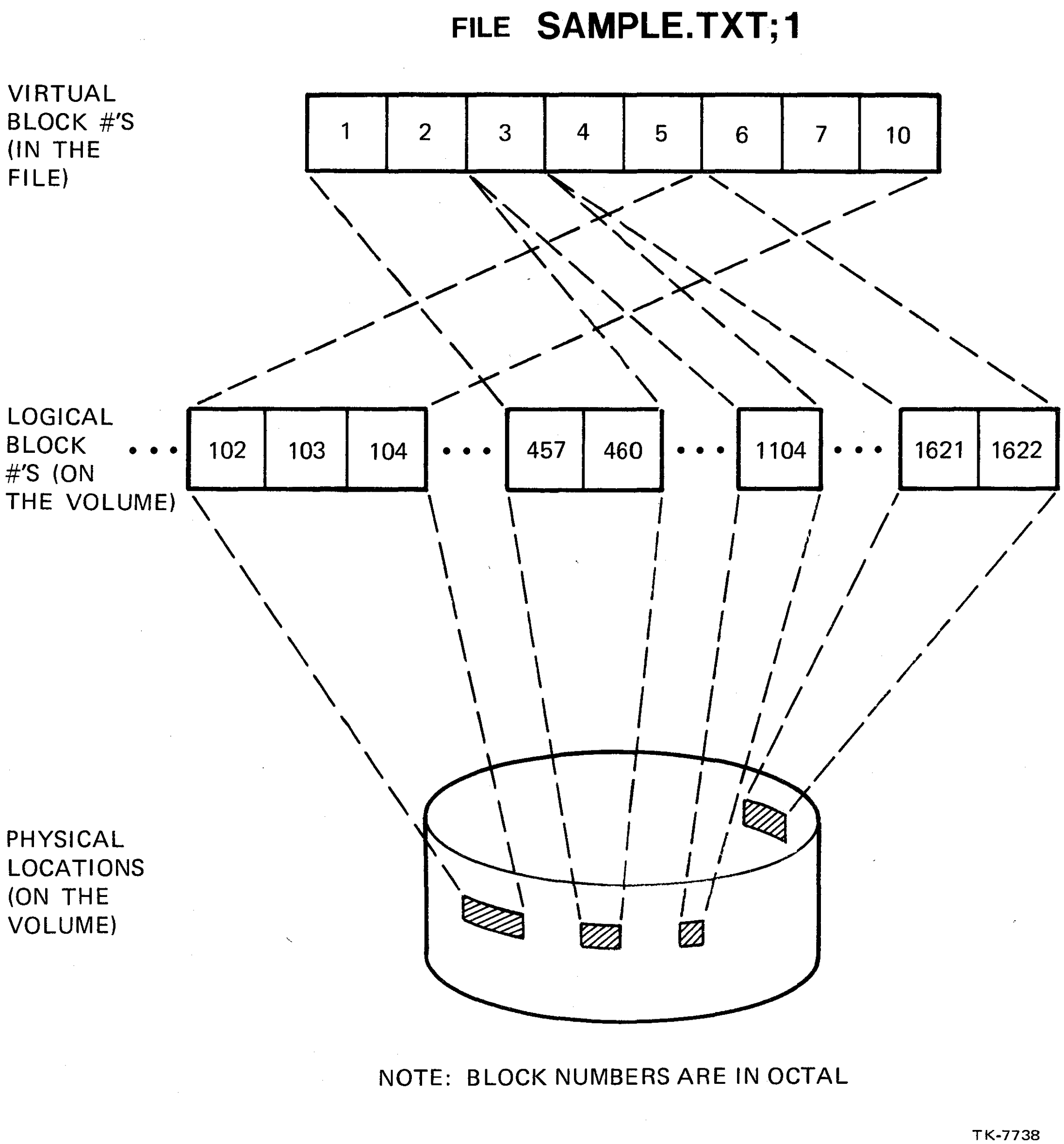


Figure 9-1 Example of Virtual Block to Logical Block, to Physical Location Mapping

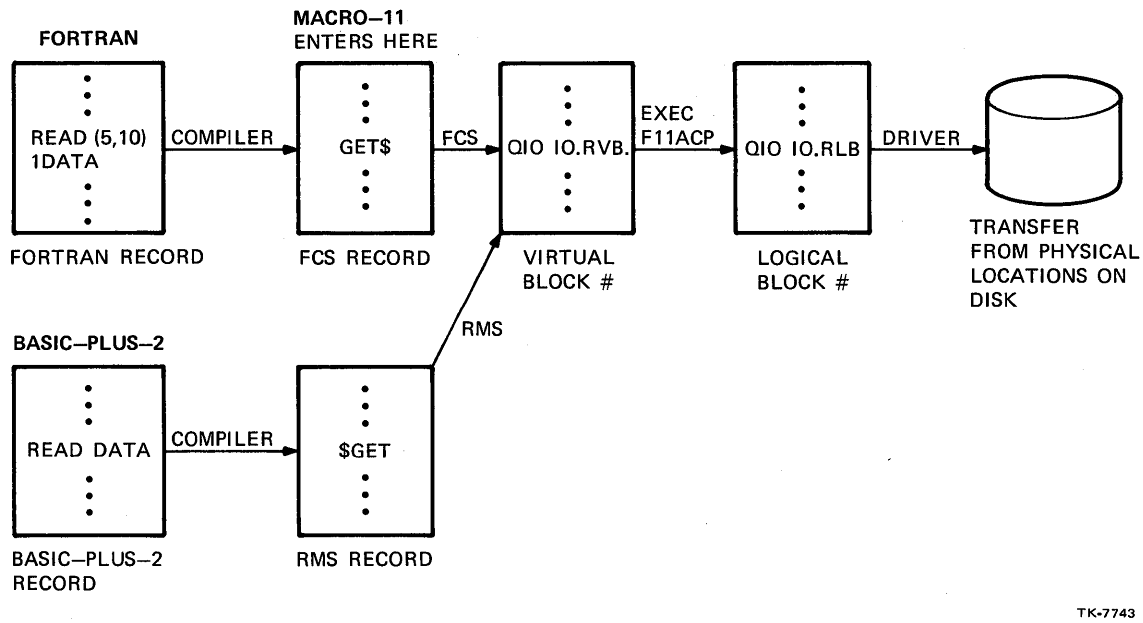


Figure 9-2 How the Operating System Converts Between Virtual, Logical, and Physical Blocks

Figure 9-3 shows the FILES-11 structures which are used to support virtual-to-logical block mapping. Every FILES-11 volume has a number of system files on it, one of which is the Index File (INDEXF.SYS). The Index File contains certain blocks which are for system use, plus a file header block for each file on the volume.

Each file header block contains file retrieval pointers which are used in mapping virtual blocks to logical blocks. Each file retrieval pointer locates a range of contiguous logical blocks. The first byte tells how many contiguous blocks are in the group, and the next three bytes specify the logical block number of the first block in the group. Therefore, in the figure, there are five contiguous blocks, starting with logical block 336851(10). Virtual block 1 = logical block 336851(10), vb 2 = lb 336852(10), vb 3 = lb 336853(10), vb4 = lb 336854(10), and vb 5 = lb 336855(10). The next group of blocks, starting with virtual block 6 has 51(10) blocks and begins at logical block 336900(10) up through logical block 336950(10). The last 17(10) virtual blocks (virtual blocks 57(10) to 73(10)) begin at logical block 337006(10) up through logical block 337022(10). These file retrieval pointers are updated each time a change in block allocation occurs as a result of a file I/O operation.

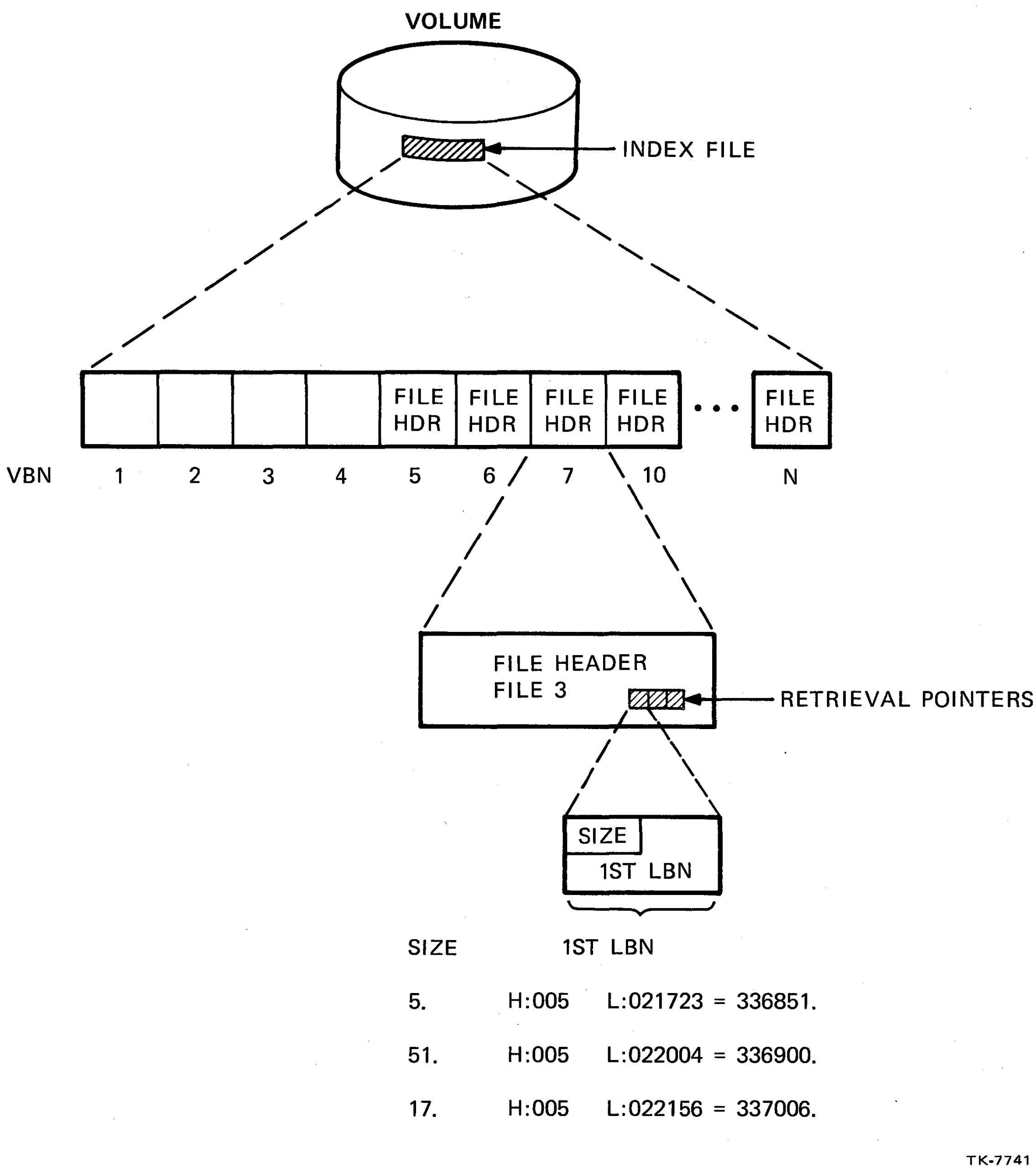


Figure 9-3 FILES-11 Structures Used to Support Virtual-to-Logical Block Mapping

Directories

The operating system identifies files by file IDs, which are used to calculate the location of the file header within the index file. When you need to locate a file, it is difficult to remember where it is on the disk, or even what its file ID is. Instead, you use a file specification, a more English-like way of identifying a file. An example of a file specification is: DR1:[5,6]SAMPLE.TXT;1. Tasks you write also usually identify files with a file specification. Directories are structures set up on a FILES-11 volume that are used to group files together, and to convert file specifications to file IDs.

A directory is a list of files belonging to a single user, or grouped together for other organizational purposes. An example of files grouped together for organization is the libraries in User File Directory (UFD) [1,1] on the system device. On a FILES-11 volume, a directory is a special file containing a list of the files belonging to that user or group. For each file, the list has:

- The file specification: name, type, and version number
- The file ID

The file ID consists of a file number and a sequence number. The file number identifies the offset within the index file to the virtual block containing the file's file header. The sequence number is used to distinguish this file from previously deleted files which used the same file header. There are two levels of directories on a volume, as follows.

- One Master File Directory (MFD) which is directory [Ø,Ø]
- One or more User File Directories (UFDs)

Figure 9-4 shows the relationship between the two levels and the files. The MFD contains a list of the system file, plus one entry for each UFD on the volume. Each UFD file has a name of the form gggmmm.DIR, where [ggg,mmm] is the user identification code (UIC) of the owner. Each UFD contains a list of the files in that directory.

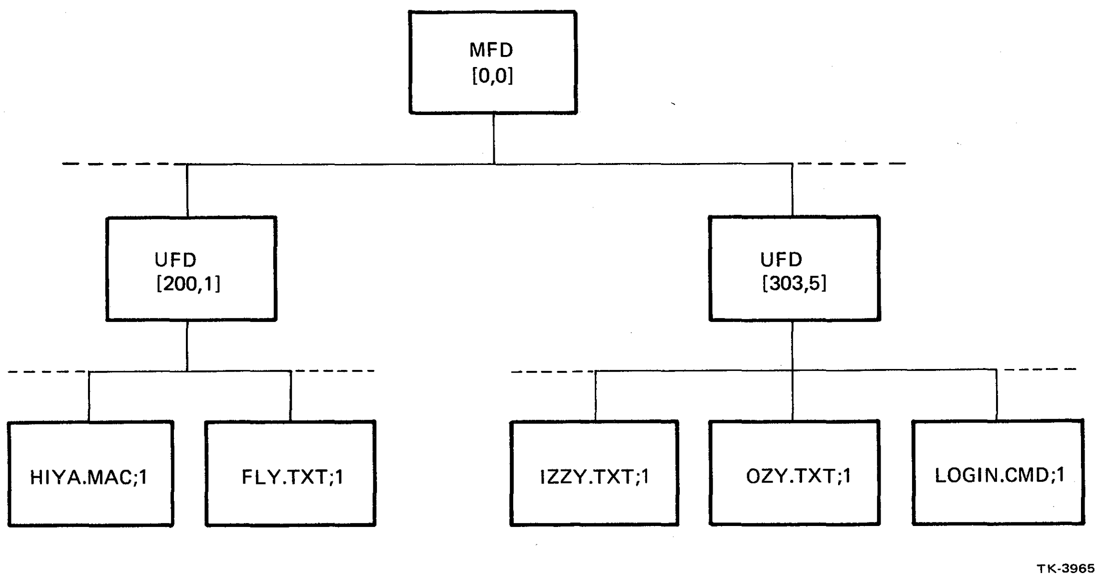


Figure 9-4 Directory and File Organization on a Volume

Figure 9-5 shows the steps used in locating and accessing the blocks of the file DR2:[5,6]SAMPLE.TXT;1. The device name, DR1: tells which device or volume to look on. The operating system reads the MFD file header to find the retrieval pointers for the MFD file itself. It converts the virtual blocks to logical blocks and reads the blocks of the MFD file. It searches through the directory list for the UFD [5,6], namely the file ØØ5ØØ6.DIR.

When it finds that name in the list, it uses the file ID to locate the UFD file header. It reads the retrieval pointers there, converts the virtual blocks to logical blocks, and reads the blocks of directory [5,6]. It looks for an entry SAMPLE.TXT;1. When it finds that entry, it uses the file ID to locate the SAMPLE.TXTs file header. It then reads the retrieval pointers in the file header, converts the virtual blocks to logical blocks, and reads the blocks of the file itself.

If this sounds like a lot of work, it is. Later, you will learn about a way to go directly to the file header using the file ID if it is opened a second time during a task's execution.

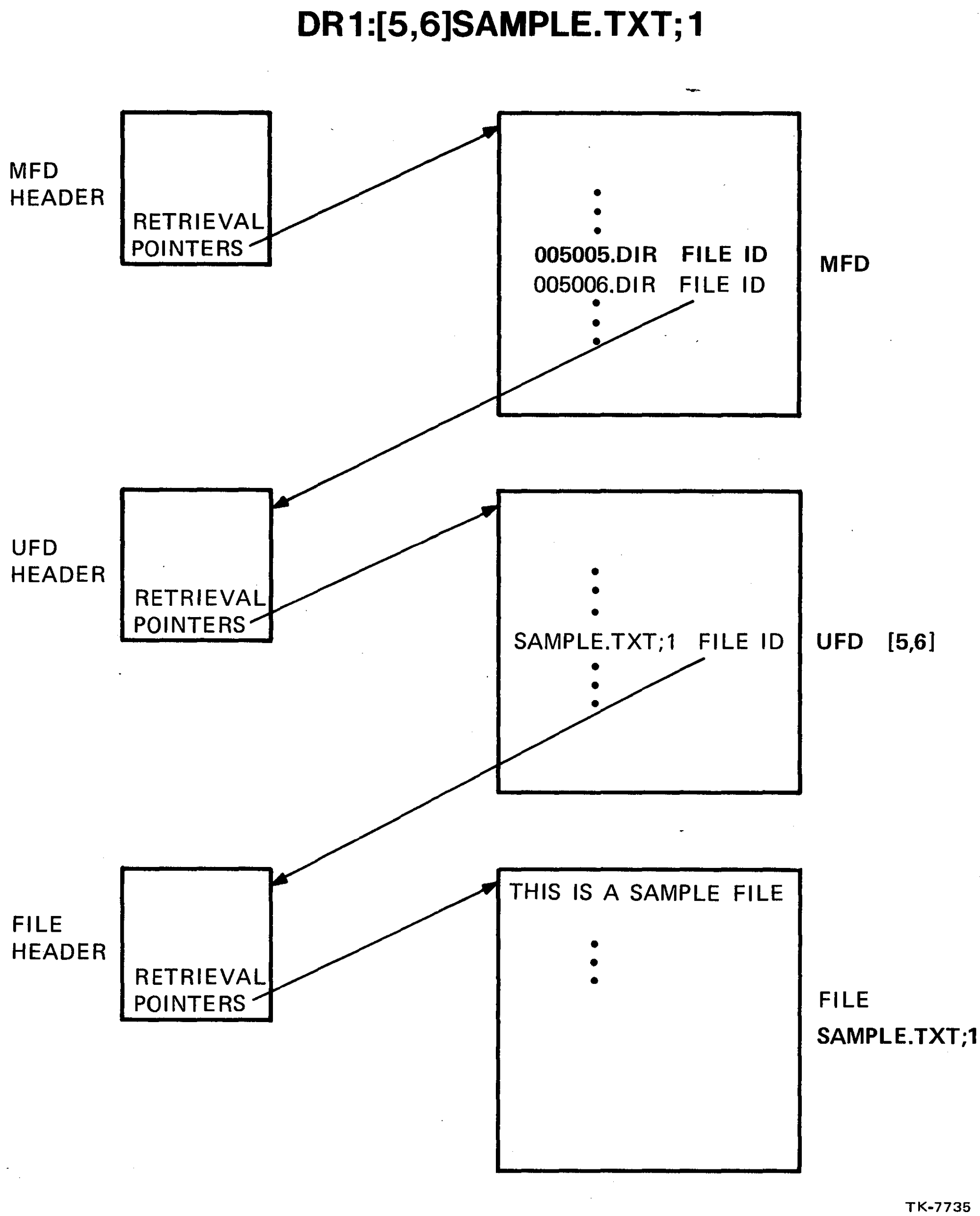


Figure 9-5 Locating a File on a FILES-11 Volume

Five Basic System Files

There are five basic system files found on all FILES-11 volumes. They are all created when the volume is initialized and are all entered in the MFD. Two of these, the Index File and the Master File Directory, have been mentioned previously. The five files and their purposes are as follows.

- The Index File: INDEXF.SYS.
  - Boot block - used when a system volume is bootstrapped
  - Home block - contains volume identification and other information
  - Index file bitmap - a record of which header blocks are in use; used by F11ACP when allocating header blocks to files
  - File header blocks for all files on the volume
- The Storage Map: BITMAP.SYS.
  - A record of which blocks on the volume are in use
  - Used by F11ACP when allocating blocks to files
- The Bad Block File: BADBLK.SYS.
  - A list of blocks on the volume known to be bad
- The Master File Directory: ØØØØØØ.DIR.
  - Entries for the five system files
  - An entry for each UFD file
- The System Checkpoint File: CORIMG.SYS.
  - Space used for checkpointing if the system manager allocates space in it.

Functions of the ACP

The F11ACP maintains the FILES-11 structure on a volume during its use.

The most elementary functions performed by the ACP are as follows.

- Maintaining the File Header Blocks. This includes:
  - Allocating and initializing a file header when a file is created
  - Recovering a file header for reuse when a file is deleted
  - Maintaining file attributes such as protection code, length, etc.
  - Maintaining the file retrieval pointers
- Maintaining directories. This includes:
  - Creating directory entries when a file or UFD is created, or when a file synonym is created (e.g., by the PIP /EN switch)
  - Removing entries from directories when a file is deleted or a file synonym is removed (e.g., by the PIP /RM switch)
- Maintaining block allocation. This includes:

  Allocating blocks to files when a file is created or extended

  Recovering blocks for reuse when a file is deleted or truncated
- Controlling and facilitating task access to files. This includes:

  Checking protection codes to determine access rights

  Connecting a task's LUN to a file to allow virtual block I/O

  Controlling shared access to files.

Table 9-2 shows the F11ACP functions performed when you request some typical file I/O operations.

Table 9-2 Examples of Use of F11ACP Functions

Operation Requested	Functions Performed by F11ACP
Create a new, permanent file and write data to the file.	1. Allocate a header for the file. 2. Allocate blocks to the file, when it is opened and/or when data written requires that extensions be added. 3. Create a directory entry for the file. 4. Assign a LUN to the file. 5. When the file is closed, write the updated file attributes to the file header, deassign the LUN
Read data from an existing file.	1. Assign a LUN to the file.
Delete a file.	1. Remove the directory entry for the file. 2. Deallocate the blocks of the file. 3. Deallocate the header for the file.
Append data to a file.	1. Assign a LUN to the file. 2. Allocate extra blocks to the file.
Create a temporary (scratch) file.	1. When file is opened, allocate a header, allocate blocks, and assign a LUN. (No directory entry is created.) 2. When file is closed, deallocate blocks, deallocate header, and deassign LUN.

Figure 9-6 shows the flow of control during the processing of an I/O request. This figure parallels Figure 9-2, which shows how the operating system converts virtual blocks to logical blocks to physical locations.

The user task issues a read record request which is converted by an FCS routine in the user task to a QIO, to read a virtual block. The Executive converts the virtual block number to a logical block number, using file retrieval pointers in pool. These retrieval pointers are built by F11ACP from the retrieval pointers in the file header. The Executive issues a read logical block request to the driver. The driver converts the logical block number to the actual physical locations and copies the block into the user buffer.

For additional information on the FILES-11 structure, see Chapter 5 of the IAS/RSX-11 I/O Operations Reference Manual.

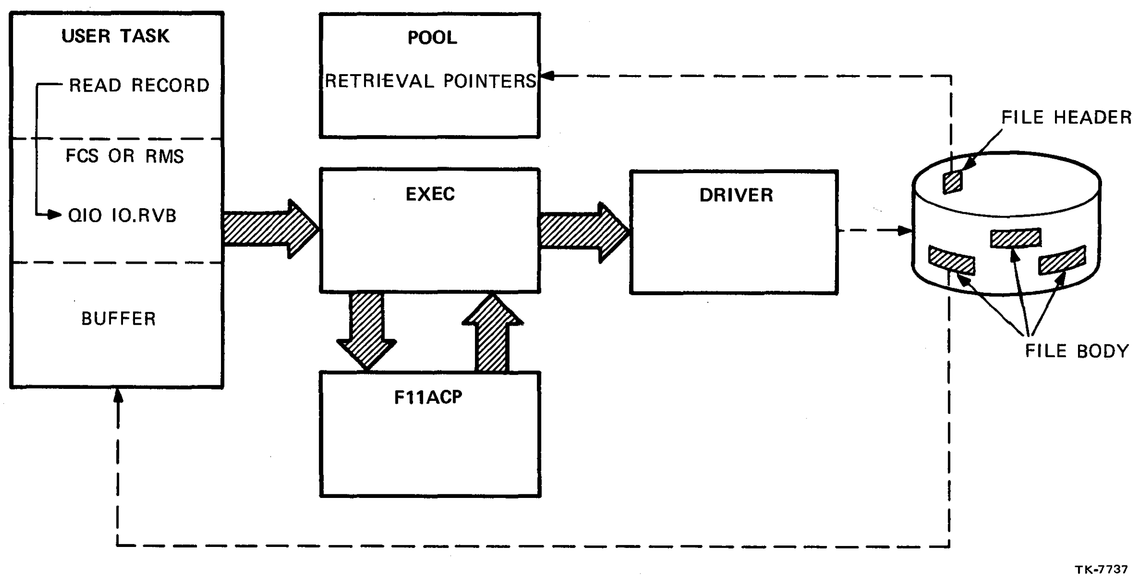


Figure 9-6 Flow of Control During the Processing of an I/O Request

OVERVIEW AND COMPARISON OF FCS AND RMS

Common Functions

The File Control Services (FCS) and the Record Management Services (RMS) both offer easy methods for performing file I/O. The operator or programmer need not be concerned with all the nitty-gritty details, but can instead let FCS or RMS take care of them. Both perform the following functions:

- Serve as an interface to the ACPs
- Allow I/O to the virtual blocks of a file on a block-by-block basis (Block I/O)
- Divide files into logical records and allow I/O to individual records within a file (Record I/O)
- Allow the programmer to process records using one of the following buffers (Figure 9-7)
  - A buffer reserved by the programmer with another buffer transparently used by FCS or RMS (move mode)
  - Directly in the buffer used by FCS or RMS (locate mode)
- Allow device independent I/O - the routines are written to work correctly with terminals, disks, etc.
- Provide mechanisms for controlling shared access to files.

Beyond that, FCS and RMS each offer a variety of file organizations, record types, and access modes. These are described in the following sections.

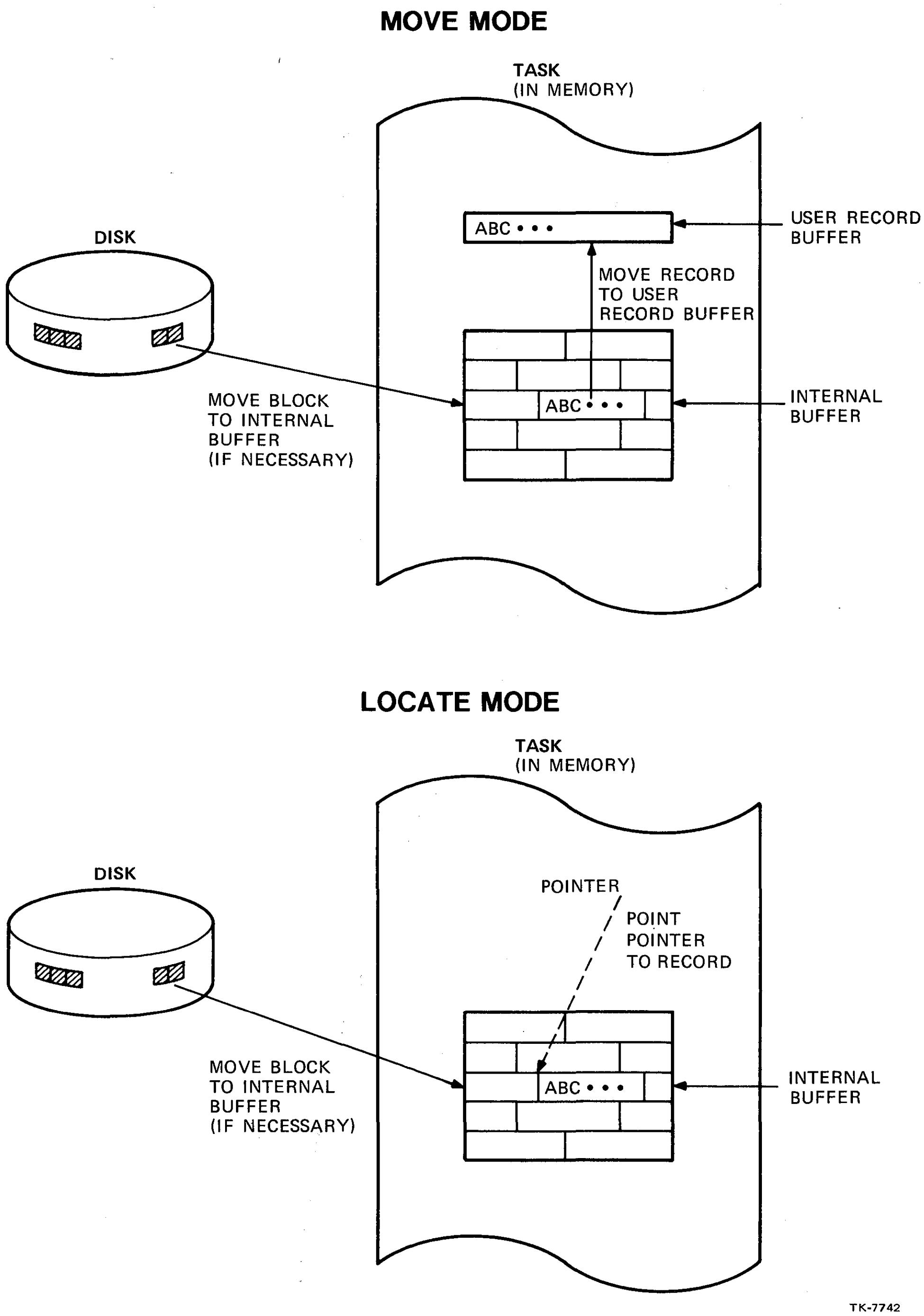
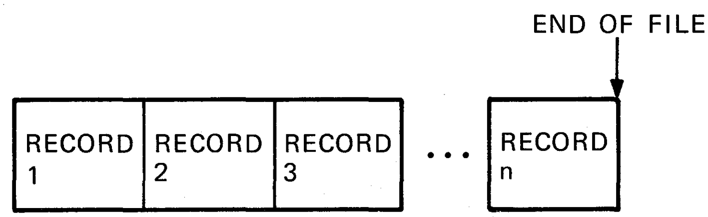


Figure 9-7 Move Mode and Locate Mode

FCS FEATURES

File Organizations

Essentially, all FCS supported files are sequential, meaning that new records are added at the end of the file, and records are stored in the order they are written. Figure 9-8 shows a file with sequential organization.



SEQUENTIAL FILE ORGANIZATION

Figure 9-8 Sequential Files

Supported Record Types

FCS supports two record types, fixed-length records and variable-length records. Variable-length records may be sequenced or nonsequenced. An example of each type of file is shown below with the following three records:

```
12345
123 1234
AAAA BBBB CC D
```

The examples are in DMP format; the six-digit number on the left is the byte count in octal of the first byte in that row. Then 16(1Ø) = 2Ø(8) bytes follow in order in octal. Below each byte in octal is its equivalent in ASCII. An underscore (_) stands for an ASCII blank. Consult the examples as you read the description of each record type which follows.

Examples:

Fixed-Length Records (record length = 17(10))

```
000000 061 062 063 064 065 040 040 040 040 040 040 040 040 040 040 040
        1   2   3   4   5
000020 040 xxx 061 062 063 040 061 062 063 064 040 040 040 040 040 040
           pad  1   2   3        1   2   3   4
000040 040 040 040 xxx 101 101 101 101 040 102 102 102 102 040 103 103
                   pad  A   A   A   A   _   B   B   B   B   _   C   C
000060 040 104 040 040 040 xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx
        _   D   _   _   _  pad
```

Variable-Length Records

```
000000 005 000 061 062 063 064 065 xxx 010 000 061 062 063 040 061 062
                1   2   3   4   5  pad          1   2   3   _   1   2
000020 063 064 016 000 101 101 101 101 040 102 102 102 102 040 103 103
        3   4           A   A   A   A   _   B   B   B   B   _   C   C
000040 040 104 xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx
        _   D
```

Sequenced Variable-Length Records

```
000000 007 000 001 000 061 062 063 064 065 xxx 012 000 002 000 061 062
                        1   2   3   4   5  pad                  1   2
000020 063 040 061 062 063 064 020 000 003 000 101 101 101 101 040 102
        3   _   1   2   3   4                   A   A   A   A   _   B
000040 102 102 102 040 103 103 040 104 xxx xxx xxx xxx xxx xxx xxx xxx
        B   B   B   _   C   C   _   D
```

Fixed-length records all contain the same number of bytes. Therefore, the location of the beginning of any record within the file can be computed from its record number. With all record types, each record begins on an even word boundary. This means that in files with fixed-length records, if each record contains an even number of bytes, the next record begins immediately after it. If, on the other hand, each record contains an odd number of bytes, one byte is unused after each record, and the next record begins at the next word boundary. This unused byte is called a pad byte.

Variable-length records may each have different lengths. For all files with variable-length records, the first word of each record contains a byte count, telling how many bytes are in that record. For variable-length nonsequenced records, this count word is followed by the data itself.

Following this, at the next word boundary, is the byte count for the next record and then its data. To locate a given record within the file, you must first read the byte count for the first record in the file. You can then use the byte count to locate the second record. You then continue reading byte counts and locating successive records until you reach the desired record.

Variable-length sequenced records contain a byte count, a user specified sequence word, and then the data itself. The sequence word can contain the record number or any other user specified value. Variable-length sequenced records are not used much under FCS. They are supported to allow compatibility with RMS variable-with-fixed-control records.

Table 9-3 compares the different FCS record types.

Table 9-3 Comparison of FCS Record Types

Record Type	Characteristics	Overhead in File	Common Applications
Fixed-Length	Record length set when file created	None	Files with similar data in each record
	Records all same length (shorter records padded)		Bank account information, bad credit card lists, etc.
Variable-Length (nonsequenced)	Records may be of different lengths	One word per record (holding record length)	Files with varying contents among records
	First word of each record is a byte count		Files to be printed Source and list files
Variable-length (sequenced)	Variable length records, with an additional word for a user specified sequence number	Two words per record (one for record length, one for sequence field)	Infrequently used, except for compatibility with RMS

Record Access Modes

FCS offers two record access modes, sequential access and random access. Table 9-4 compares the two access modes. The major difference is that with random access, the user can process records in any order (e.g., record 12, then record 4, then record 29). This is possible with fixed length records only, because FCS can calculate the position of each record within the file from the record number and the record size.

With variable-length records, on the other hand, FCS can't locate record 12 unless it reads records 1 through 11 first, using the record length in the first word of each record to calculate the starting position of the next record. Therefore, you must use sequential access with variable length records. You may choose either of the two access modes for fixed length records, depending on how your application processes the records.

Table 9-4 Comparison of Sequential Access I/O and Random Access I/O

Characteristics	Sequential	Random Access
Devices supporting this type of access	All devices	Block-structured devices only
Record types using this type of I/O	All record types	Fixed-length records only
Sequence of records in the file	Determined by the order in which they are written to the file	Usually determined by the order in which they are written to the file
Order of processing records	Usually the same order as in the file (one after another)	In any order, as specified by the user (using the record number)
Overhead if records are processed in same order as they are stored in the file	Low	Low, but not as low as sequential
Overhead involved if records are processed in order different from how they are stored in the file	Much higher than random access I/O	Much lower than sequential I/O

NOTE

With sequential access, special system subroutines allow the user to save pointers to a record for much faster subsequent access.

File Sharing

A task which opens a file may choose one of the following options:

- That no other accessor change any data in the file while it has access ("shared" read, "exclusive" write).
  - If this task desires read access, other accessors may have simultaneous read access, but no other accessor may have simultaneous write access.
  - If this task desires write access, no other accessor may have simultaneous read or write access.
  - Any access request causing a conflict is rejected.
- That other accessors may change the data while it has access ("shared" read/write access).
  - If this task requests read or write access, other accessors may have simultaneous read or write access.
  - Use extreme care - Any precautions against corrupted data are the responsibility of the accessors.
- That no other accessor changes any block within the file which has already been accessed (block locking). Shared access to the file is allowed, but:
  - Each block which is written to is locked for exclusive write access.
  - Each block which is read is locked for shared read access.
  - It is not recommended if accessing a large numbers of blocks, because each block lock uses four words of pool.
  - Any attempt to access a block which causes a conflict, returns an error.

RMS FEATURES

File Organizations

RMS supports three file organizations, sequential, relative and indexed. See Figure 9-9. Sequential files under RMS are the same as sequential files under FCS. A relative file is composed of a series of cells of uniform size. The cell size is greater than or equal to the largest record to be placed in the file. A single record may be written to a cell, or the cell can be empty. The cells may contain variable-length records. Variable-length records within relative files can be accessed randomly because each record is contained within a fixed-length cell. Also, when you read successive records in a relative file, empty records are automatically skipped.

An indexed file is composed of records, plus one or more indexes which are used to access those records. Each index is used to retrieve records according to the contents of a particular field, or key, within the record. The data records themselves are ordered according to a primary key which you declare when you create the file.

Figure 9-9 shows an indexed file with a single key, namely last name. In the example, the data records are in the bottom row, ordered alphabetically by last name. The index for this file contains two other levels, level 1 and level 2 (the root level).

A search for a record begins at the root level. For example, to find the record with key value FRANCIS, search through the root level, checking for the first value which is greater than or equal to FRANCIS. The first such value is SMITH. Go to the next level and again search for the first value greater than or equal to FRANCIS; it is GROSS, the first value. Now go to the next level and search again; this time the value FRANCIS is found. Since this is level Ø, we have found the record.

As new records are added to the file, they are inserted in order at level Ø of the primary index. The primary index structure is adjusted for the new entry at the same time. In addition, any alternate index structures for other keys are adjusted as well. There is always one primary key, and there may be as many as 254(1Ø) alternate keys.

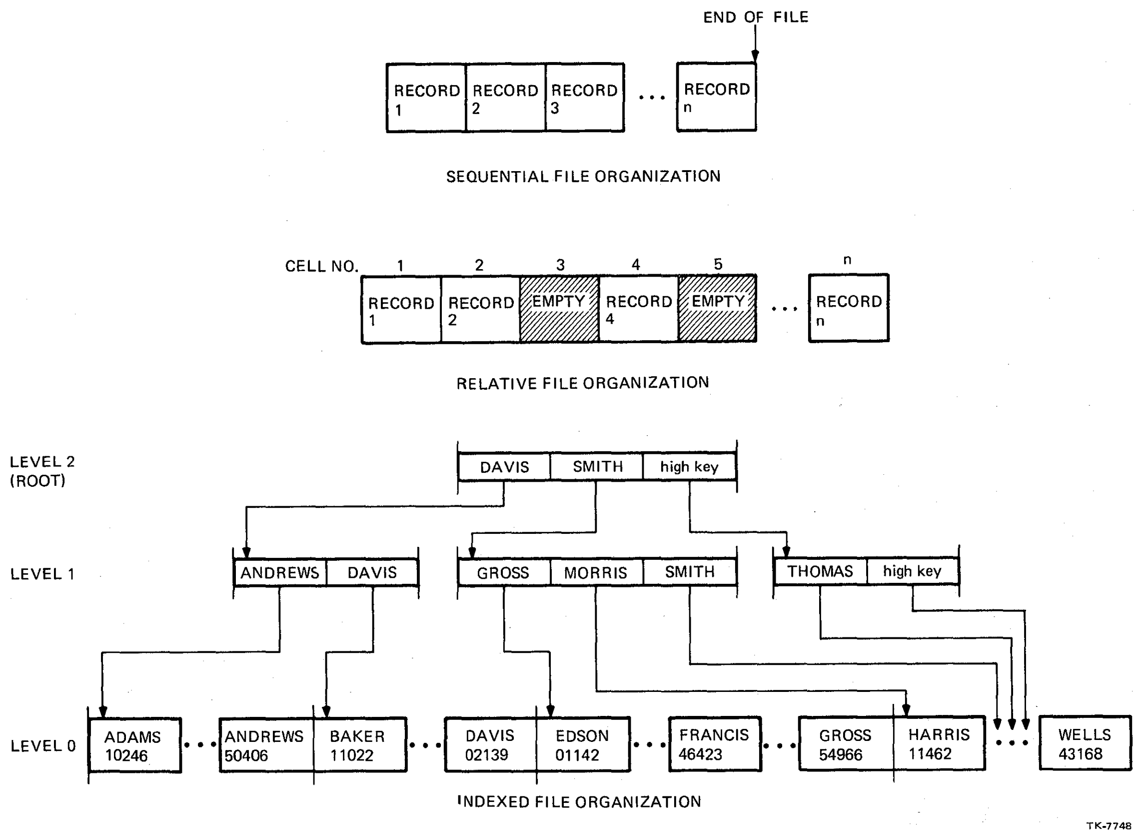


Figure 9-9 RMS File Organizations

Level Ø of the alternate keys contains pointers to the original location of the data record itself. If a data record is ever moved in order to maintain the index structure, a pointer is created and maintained in the records original location, which points the data record's new location.

One specific advantage of an indexed file over a relative file is that an indexed file allows you to search for records using several different key fields, while only the cell number can be used with relative files. Even with a single key, indexed files offer keys consisting of any ASCII characters, in contrast to just a cell number for relative files.

There is, of course, more space overhead required in the file for the index structures. In addition, more execution time is required to insert new records, because the index structures must be updated as well. We are keeping things rather simple in the discussion here. For additional information, see the RMS-11 User's Guide.

Record Formats

RMS supports three record formats; fixed-length records, variable-length records, and variable-length records with fixed control. Fixed-length records and variable-length records are the same as fixed-length records and nonsequenced variable-length records respectively, under FCS. They are both supported under all three file organizations.

Variable-length records with fixed-control (VFC) contain a fixed-length portion, for control, followed by a variable-length portion. The fixed control portion may be up to 255(10) bytes long. A sequenced variable-length record under FCS is the same as a VFC record with a 2-byte (one word) fixed control portion.

An example of the use of VFC records is a bank account file, where some accounts have both savings and checking, and others have just one or the other. The fixed control portion could contain the account number plus an indication of the kinds of accounts contained in it. The variable portion contains the account information for those accounts. The length of this portion varies, depending on how many accounts the person has. VFC records are supported under sequential and relative file organizations only.

Record Access Modes

RMS supports three record access modes: sequential access, random access, and access by Record File Address (RFA). Sequential access and random access are similar to the FCS access modes, except that they are applied differently for indexed files.

For sequential access on an indexed file, the "next" record is the record with the next highest key value using the specified key, not the next record added to the file. For random access, a key value for a certain key is specified, and that record is located and accessed. To access a record-by-record file address, save pointers to the record (called its record file address or RFA) from one access, then use the pointers to subsequently access the record again.

Table 9-5 describes the various access modes supported for each file organization and how they work. For additional information, see the RMS-11 User's Guide.

Table 9-5 File Organization, Record Formats, and Access Modes

	Sequential Files	Relative Files	Indexed Files
Record Formats Supported	Fixed Variable VFC	Fixed Variable VFC	Fixed Variable
Access Modes Supported	Sequential RFA	Sequential Random RFA	Sequential Random RFA
Sequential Access Techniques	Writes and reads subsequent records	Writes to subsequent cells Reads from subsequent cells, skipping empty ones	Accesses cells in ascending order according to user specified key
Random Access Techniques	Not allowed	User specifies cell number of record to be accessed	User specifies key and key value to be used in accessing records
Record File Address Techniques	Task can store RFA of a record for later return	Same as sequential files	Same as sequential files

File Sharing Features

RMS offers more sophisticated file-sharing options than FCS. Sequential files can be shared for read access only. Relative and indexed files can be shared for read and write access. When opening a relative or indexed file, a task indicates one of the following options.

- No other accessor can change data in the file while it has access ("shared" read, exclusive "write").
- Other accessors can change data, but subsets of the file are protected at a time, while in use.

Relative and indexed files are divided into units called buckets (of user specified size, each 1 to 32(10) blocks long). In fact, all actual I/O tranfers are performed on full buckets only. In implementing protection of subsets of the file at a time, protection is on a bucket-by-bucket basis (bucket-locking).

A bucket is locked from the time any task with write access accesses a record in a bucket, until that task begins operations on another bucket, or closes the file. This means that records within a given bucket can't be accessed by other tasks while another task with write access is using the bucket. But other tasks may access other buckets in the file during that time.

Summary

Table 9-6 summarizes our comparison of FCS and RMS. The next module discusses the details of how to use FCS in a program.

Table 9-6 Comparison of FCS and RMS

Characteristics	FCS	RMS
Supporting utilities	Standard RSX utilities	Special RMS utilities to define, convert, etc.
Supporting languages	MACRO-11 FORTRAN IV, IV-PLUS, -77, BASIC-11	MACRO-11 FORTRAN IV-PLUS, -77, BASIC-PLUS-2 COBOL
Ease of file design	Relatively simple	Relatively complex
Ease of programming	Relatively simple in high-level languages	Relatively simple in high-level languages, issues of efficiency complex
	Moderate in MACRO-11	Relatively difficult in MACRO-11
Type of data access supported	Virtual block I/O	Virtual block I/O
	Sequential record access	Sequential record access
	Random access by record number with fixed-length records	Random access by cell number in a relative file
		Random access by key field within record, in an indexed file
	Access by record position pointers, saved from previous access of record	Access by record file address, saved from previous access of record

Table 9-6 Comparison of FCS and RMS (Cont)

Characteristics	FCS	RMS
Overhead in file needed to support record structure	Minimal	Minimal for sequential files Moderate for relative files High for indexed files
Execution time overhead to support record access	Low	Low for sequential and relative files Moderate to high for indexed files, depending on file and program design, and file growth
Shared access coordination	System protection on a per-file basis or on an all blocks accessed basis	System protection on per-file or per-bucket basis within a file

Now do the tests/exercises for this module in the Tests/Exercises book. They are all written problems. Check your answers against the provided solutions in the Tests/Exercises book.

If you think that you have mastered the material, ask your course administrator to record your progress on your Personal Progress Plotter. You will then be ready to begin a new module.

If you think that you have not yet mastered the material, return to this module for further study.

FILE CONTROL SERVICES

10

INTRODUCTION

The File Control Services (FCS) subsystem provides the means through which most tasks perform file I/O. You make calls directly to the FCS routines.

This module introduces you to the structure of FCS, the services it offers, and the ways in which you can use those services.

OBJECTIVES

1. To choose file characteristics for a specific application, then create a file with those characteristics
2. To write tasks which read or write data using record I/O or block I/O (MACRO only)
3. To identify and implement methods of optimizing file I/O.

RESOURCE

- IAS/RSX-11 I/O OPERATIONS MANUAL, Chapters 1, 2, and 3 (Additional reading - Chapters 4 and 6)

REVIEW OF FILE I/O

Use the following basic steps to perform file I/O.

1. Open the file.

   - Ask ACP to connect LUN to file.
   - Specify access rights desired (RWED).
   - Specify type of access.
     - Block I/O or record I/O
     - For record I/O only

       Random or sequential access
       Move or locate mode
   - If new file, specify file characteristics.
     - Record type
     - Record attributes
     - File initial size and extend size

2. Perform the actual I/O operations.

3. Close the file.

   - Perform any needed clean-up work.

INTRODUCTORY EXAMPLE

We begin our discussion of FCS with an example. The purpose is to give you a feeling for how to perform the basic steps of file I/O. After that, we will examine the data structures involved, and the specific steps for setting them up and using them to perform file I/O.

Example 10-1 creates a file with variable-length records using sequential access. The records are input from TI: and then placed in the file. The following notes are keyed to the example.

1. The interface with FCS is through system macros.

2. FCSERR is an error message macro supplied with this course. Its source and documentation concerning its use are in Appendix A. It is used here to avoid having to worry about the details of the code.

3. The FSRSZ$ macro reserves space in the user task for a general FCS data area which is called the file storage region (FSR). This macro must be issued in every program that uses FCS.

4. A file descriptor block (FDB) contains data structures for a file opened by FCS. A separate FDB is required for each file which is open at the same time. The FDB and its related data structures can be filled in at assembly time or at run time. In this example, they are set up mostly at assembly time, which is more run time efficient.

5. Open the new file VARI.ASC. Notice that the run-time macro references the label of the FDB. This is necessary in the case of multiple FDBs, for multiple files opened by a single program.

6. Get input record from TI:.

7. Write (PUT$) the record to the file. For variable-length records, specify the record length in bytes.

8. Branch on any FCS error.

9 Get next record. On a ^Z, close the file and exit.

10 On the Dump - A file dump is included for each example in this module which creates a new file. The dumps were created using the DMP utility, and are in octal byte format. Because this file has variable-length records, the first word in each record is a byte count for the record. See the section on FCS File Organizations in the File I/O module for additional information on the dump.

```
        .TITLE  CRESEQ
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;+
; File CRESEQ.MAC
;
; CRESEQ creates a file VARI.ASC of variable-length
; records using sequential access. It reads records from
; TI:, and places them in the file. A ^Z terminates
; input and closes the file.
;
; Assemble and task-build instructions:
;
;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]-
;       ->CRESEQ
;       LINK/MAP CRESEQ,LB:[1,1]PROGSUBS/LIBRARY
;-
        .MCALL  EXST$C,QIOW$C,QIOW$,DIR$ ; System macros
        .MCALL  FSRSZ$,FDBDF$,FDAT$A,FDRC$A,FDOP$A ; System
        .MCALL  NMBLK$,OPEN$W,PUT$,CLOSE$ ;  FCS macros
        .MCALL  DIRERR,IOERR,FCSERR ; Supplied macros
;
        FSRSZ$  1                   ; 1 file for record I/O
; Define file descriptor block for VARI.ASC
FDB:    FDBDF$                      ; Allocate the FDB
        FDAT$A  R.VAR,FD.CR         ; Variable length records,
                                    ;  Listing - implied
                                    ;  <CR>,<LF>
        FDRC$A  ,BUFF               ; Sequential access and
                                    ;   record I/O by
                                    ;   default, BUFF is
                                    ;   user record buffer
        FDOP$A  1,,FNAME            ; Use LUN 1, file spec
                                    ;   at FNAME
FNAME:  NMBLK$  VARI,ASC            ; "VARI.ASC"
BUFF:   .BLKB   80.                 ; User Record Buffer
IOST:   .BLKW   2                   ; I/O status block
        .EVEN
        .ENABL  LSB                 ; Enable local symbol
                                    ; block
; Open file for write, call ERR1 if open fails
START:  OPEN$W  #FDB,,,,,,ERR1
; Get record from terminal, put to file.
10$:    QIOW$C  IO.RVB,5,1,,IOST,,<BUFF,80.>
        BCS     ERR2D               ; Branch on directive
                                    ;   error
        TSTB    IOST                ; Check for I/O error
        BLT     ERR2I               ; Branch on I/O error
```

Example 10-1 Creating a File in MACRO-11 (Sheet 1 of 2)

```
                MOV       IOST+2,R1          ; Number of bytes input
                PUT$      #FDB,,R1           ; Put record to file
                BCS       ERR3               ; Branch on FCS error
                BR        10$                ; Get next record

        EXIT:   CLOSE$    #FDB,ERR4          ; Close file
                EXST$C    EX$SUC             ; Exit with success
                                             ;  status
        ; Error code - Close file if necessary, display error
        ; message and exit
        ERR1:   FCSERR    #FDB,<ERROR OPENING FILE>
        ERR2D:  DIRERR    <DIRECTIVE ERROR ON READ>
        ERR2I:  CMPB      #IE.EOF,IOST       ; Is it ^Z?
                BEQ       EXIT               ; If equal, close file
                                             ;  and exit
                IOERR     #IOST,<ERROR ON READ> ; Display error
                                             ;  message and exit
        ERR3:   CLOSE$    #FDB,ERR4          ; Close file
                FCSERR    #FDB,<ERROR WRITING RECORD>
        ERR4:   FCSERR    #FDB,<ERROR CLOSING FILE>
                .END      START
```

Run Session:

```
>RUN CRESEQ
1111
22222 22
333
JAZZ Jazz JAZZ Jazz
Have you ever seen the sun?
66 66 66 66
^Z
>
```

```
Dump of DB1:[305,301]VARI.ASC;27 - File ID 34772,6,0
                  Virtual block 0,000001 - Size 512. bytes

000000  004 000 061 061 061 061 010 000 062 062 062 062 062 040 062 062
000020  003 000 063 063 063 000 023 000 112 101 132 132 040 112 141 172
000040  172 040 112 101 132 132 040 112 141 172 172 000 033 000 110 141
000060  166 145 040 171 157 165 040 145 166 145 162 040 163 145 145 156
000100  040 164 150 145 040 163 165 156 077 000 013 000 066 066 040 066
000120  066 040 066 066 040 066 066 000 000 000 000 000 000 000 000 000
```

Example 10-1 Creating a File in MACRO-11 (Sheet 2 of 2)

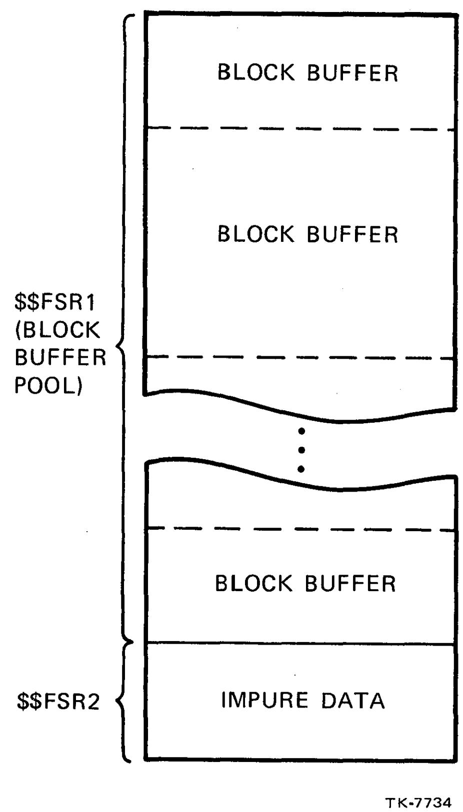


Figure 1Ø-1 The File Storage Region

USING FCS

In this course, we cover many of the options supported by FCS. However, we cannot cover all of the options in detail. Therefore, it is very important that you read the reading references mentioned in the IAS/RSX-11 I/O Operations Reference Manual for further information. This is especially important if you are going to use an option which is not discussed in detail in this course. For a general discussion of FCS and its use, read Chapter 1 of that manual.

Preparing to Open a File

The File Storage Region (FSR) -- The FSR is an area allocated in your task as working storage for FCS operations. The FSR consists of two program sections which are always contiguous to each other. Figure 10-1 shows the layout of the FSR. The program sections and their purposes are as follows.

> $$FSR1 -- contains space for block buffers and the block buffer headers for record I/O operations. You determine the size of this area at assembly time with the FSRSZ$ macro. Block buffers and headers are allocated from this area when a file is opened for record I/O operations. Enough space must be allocated for the greatest need of your task at any one time.
>
> $$FSR2 -- contains impure data which is used and maintained by FCS when performing both record I/O and block I/O operations. The area is set aside at assembly time. Portions of it are initialized at task-build time; other portions are maintained by FCS at run time.

The data flow during record I/O operations for locate mode and move mode is shown in Figure 10-2. Note that blocks of data are transferred directly between the device and the FSR block buffer. In locate mode, you usually access the data directly in the FSR block buffer. In move mode, an additional transfer is made of the specified record between the FSR block buffer and a user specified buffer.

The data flow during block I/O operations is different, as show in Figure 10-3. Blocks of data are transferred directly between the device and a user specified buffer. No FSR block buffer is needed.

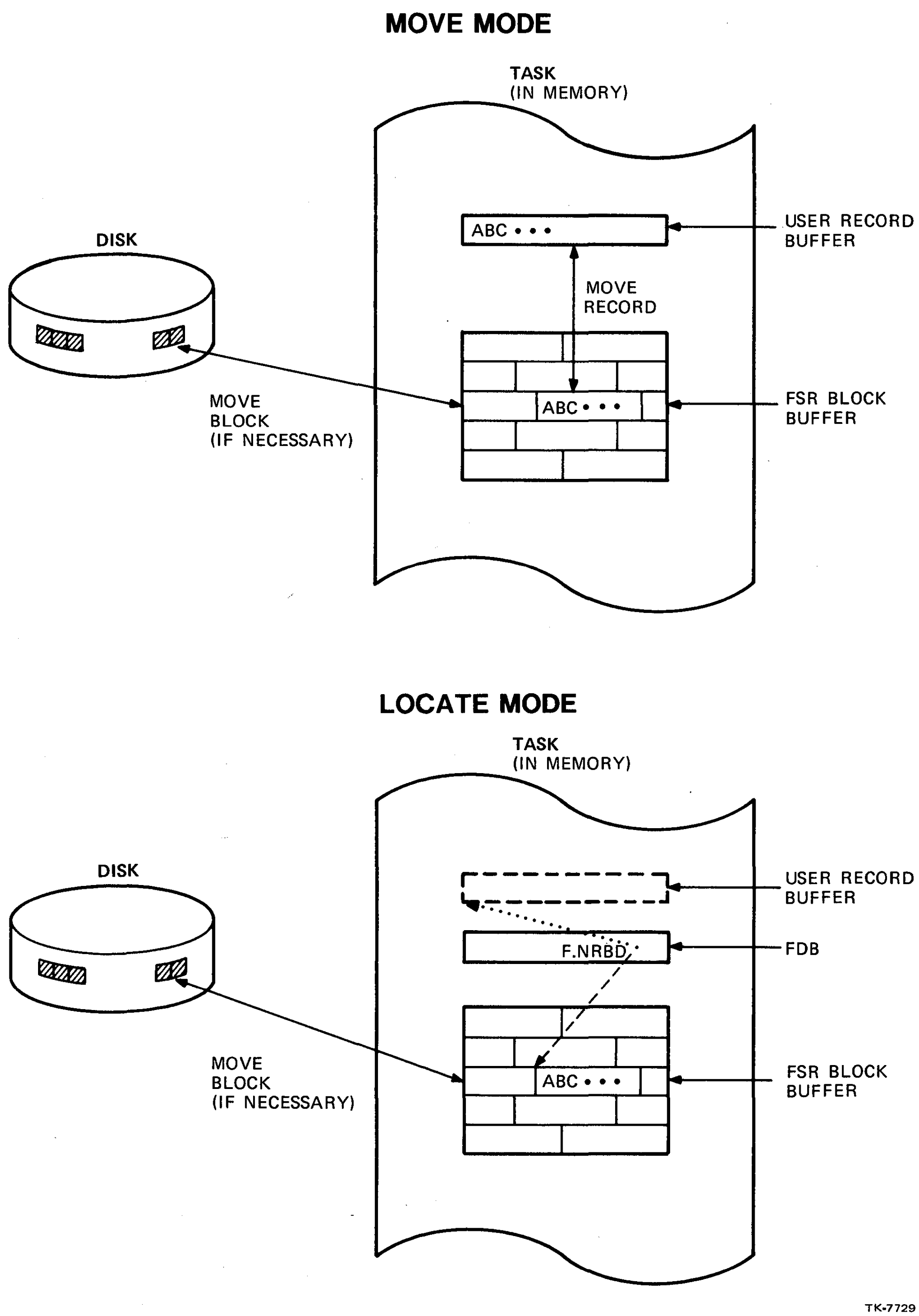


Figure 10-2 Move Mode Versus Locate Mode for Record I/O

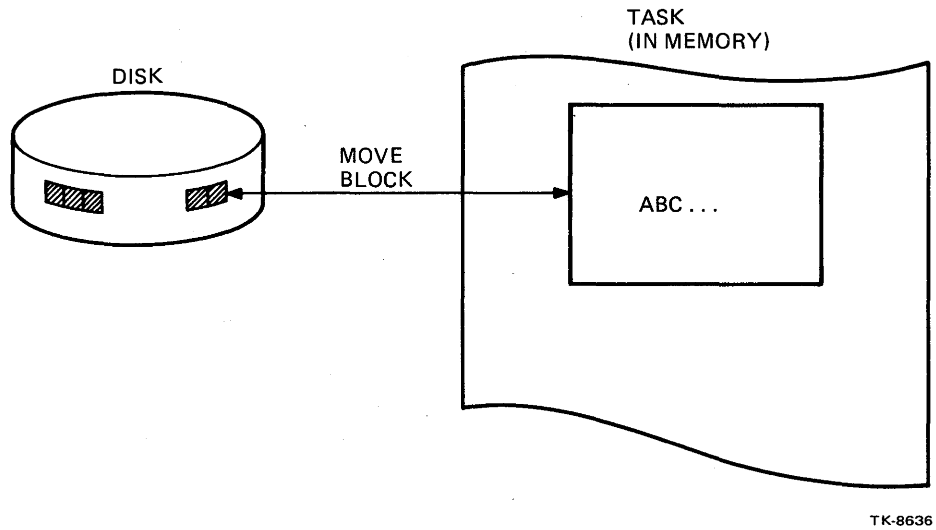


Figure 1Ø-3 Block I/O Operations

Initialization of the FSR

Use the FSRSZ$ macro to establish the size of the FSR at assembly time. This macro must be used in any program using FCS, whether for block I/O or record I/O. The format of the FSRSZ$ macro is as follows.

```
FSRSZ$ fbufs,bufsize,psect
```

fbufs   - for block I/O only, specify Ø

- for record I/O or record and block I/O, maximum number of buffers needed for record I/O

bufsize - total space needed for block buffers (in bytes). Defaults to fbuf*512(1Ø)

psect   - return Psect if other than default.

Examples:

FSRSZ$ Ø

Using FCS for block I/O only. Allocate FSR space for impure data only ($$FSR2).

FRSRSZ$ 2

Using FCS, allocate FSR space for impure data ($$FSR2), and for record I/O block buffers ($$FSR1). Total allocation for block buffers in $$FSR1 is two headers plus 2*512(1Ø) = 1Ø24(1Ø) bytes.

FSRSZ$ 3,2Ø48

Using FCS, allocate FSR space for impure data ($$FSR2), and for record I/O block buffers ($$FSR1). Total allocation for block buffers is three headers plus 2Ø48(1Ø) bytes. For example, two are 512(1Ø) bytes long and the third is 1Ø24(1Ø) bytes long.

The buffer size usually corresponds to a disk block (512(1Ø)) for disks, or the buffer width for terminals. If all record I/O operations use single buffering with the default buffer size of 1 disk block (512(1Ø)), then fbufs should be the maximum number of files open at the same time for record I/O. Bufsize can be defaulted to that number, times 512(1Ø).

If double buffering is used for some record I/O operations, or larger block buffers are desired (to reduce the number of I/O transfers), specify values for fbufs and/or bufsize. This allows for your maximum need for files open at the same time for record I/O.

See section 2.6.1 on FSRSZ$ in the IAS/RSX-11 Operations Reference Manual for a discussion on how to calculate bufsize.

The File Descriptor Block (FDB)

Functions of the FDB - The FDB contains information used by FCS in opening and processing a file. One FDB is required for each file that is open at the same time by your program. An FDB may be reused once the file associated with it is closed. The FDB is used by:

- The task, to pass information to FCS
- FCS, to return information to the task
- FCS, for internal bookkeeping for the file.

You must allocate space for each FDB and initialize specific portions, before opening a file. You may use either assembly-time or run-time macro calls. Figure 10-4 shows an FDB and its different parts.

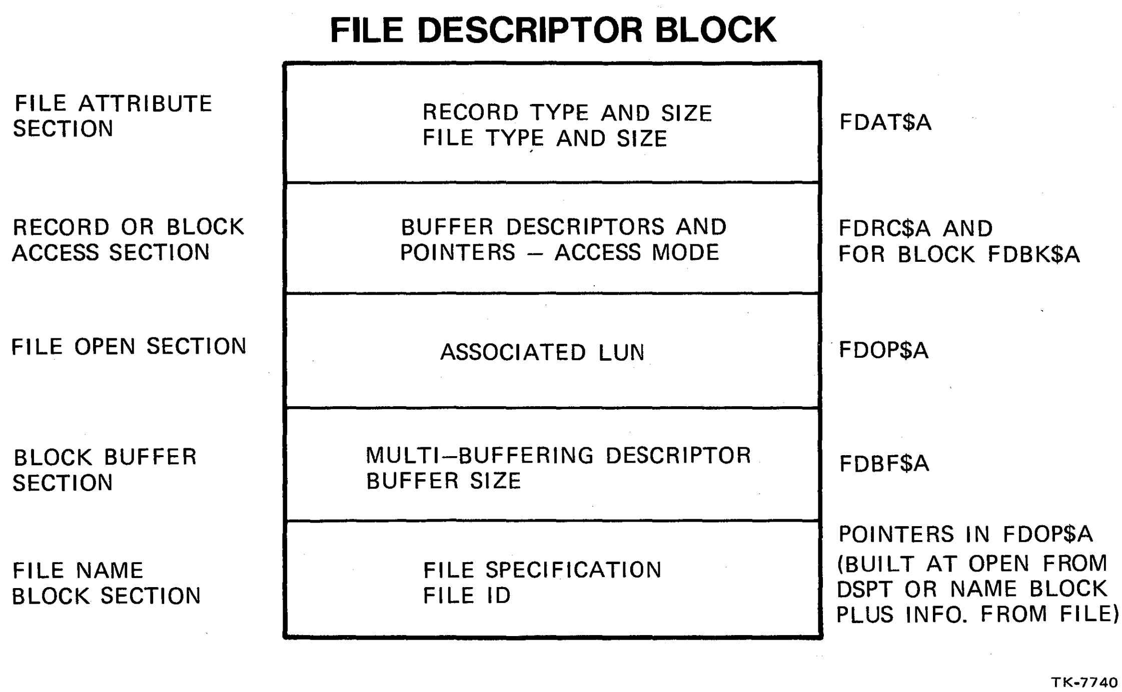


Figure 10-4 The File Descriptor Block

Allocating Space for FDBs - Use the FDBDF$ macro to allocate space for one FDB. The format of the call is:

```
label:   FDBDF$

FDBIN:   FDBDF$
```

The label is used later to refer to a specific FDB.

Initializing an FDB - You can initialize an FDB either at assembly time or at run time. Whenever possible, use the assembly-time macros because they do not need to be executed at run-time. Therefore, your task will be more run-time efficient. With the assembly-time macros, use parameters which are valid source arguments for .WORD or .BYTE assembler directives. Many values have symbolic equivalents which can be used instead of the actual numeric values.

With the run-time macros, use parameters which are valid source arguments for MOV or MOVB instructions. This is similar to the convention for the $ form versus the $S form of the executive directives. At assembly time, use FCS macros which end with $A; at run time, use FCS macros which end with $R. The assembly-time macros must immediately follow the FDBDF$ macro which reserves space for the FDB. The run-time macros have an additional initial argument to specify which FDB they refer to.

Run-time initialization macros override any previous FDB settings. In addition, you can also override the settings in the file open operation or in an I/O operation.

As an aid in referencing a given FDB at run time, all FCS run-time initialization and file-processing macros return the FDB address in RØ. If no FDB pointer is specified in subsequent FCS macro calls, it defaults to RØ. The other registers are saved and restored by all FCS run-time macros.

For additional information on the use of parameters in the different forms of the FCS macro calls, see section 2.2.1 on Assembly-Time FDB Initialization Macros, and section 2.2.2.1 on Run-Time FDB Macro-Call Exceptions in the IAS/RSX-11 I/O Operations Reference Manual.

The following sections describe how to use the different FCS FDB initialization macros to initialize an FDB.

Specifying New File Characteristics

Use either the FDAT$A macro, at assembly time, or the FDAT$R macro, at run time, to specify new file characteristics. These macros are only required where you create a new file. FCS uses the established characteristics for existing files. The format of the FDAT$A macro is:

```
FDAT$A rtyp,ratt,rsiz,cntg,aloc

rtyp - record type
         R.FIX = fixed length
         R.VAR = variable length
         R.SEQ = sequenced

ratt - record attributes

        carriage control
         FD.FTN = FORTRAN type
         FD.CR = list type
         default = no implied carriage control

        spanning of blocks
         FD.BLK = spanning blocks not allowed
         default = spanning blocks is allowed

rsiz - record size

cntg - initial number of blocks for file

aloc - extend size for file.
```

Examples:

1. FDAT$A R.VAR

   File will have variable-length records. Defaults: no implied carriage control, may span block boundaries, initial size of zero blocks, default extend size, on disk, generally five blocks.

2. FDAT$A R.FIX,FD.CR,64.

   File will have fixed-length records, list carriage control, and 64(10) byte records. Defaults: records may span block boundaries, initial size of zero blocks default extend size.

3. FDAT$A R.FIX,RD.FTN!FD.BLK,100.,-15.

   File to have fixed-length records, FORTRAN type carriage control; records may not span block boundaries; 100(10) byte records, initial file size of 15(10) blocks, not necessarily contiguous. Default: default extend size.

4. FDAT$R #FBD1,#R.FIX,#FD.FTN!FD.BLK,#100.,#15.

   The same as the previous example, but using the run-time form.

Note the difference in the format of the parameters in the $A (for assembly-time) and the $R (for run-time) forms. For the $A form, the parameters are symbolic or numeric values, all valid source arguments for .WORD or .BYTE assembler directives. For the $R form, on the other hand, the parameters are all valid source arguments for MOV or MOVB instructions.

If records are allowed to span block boundaries, then a record at the end of a block, which doesn't fit completely within the block, is continued in the next block. If records are not allowed to span block boundaries, a record which doesn't fit completely is started at the beginning of the next block. The space remaining in the current block is unused. This technique uses more file space, but permits quicker I/O operations in locate mode.

Specify one of three possible types of carriage control in the ratt parameter. FD.FTN indicates that the first data byte of each record contains a FORTRAN carriage-control character (e.g., space for single space, 0 for double space). FD.CR indicates that when the record is written to a line printer or a terminal, each record is to be preceded by an <LF> character and followed by a <CR> character. This causes single spacing between records in the printout. If you specify neither FD.FTN nor FD.CR, no carriage control is implied. Any carriage control characters must be imbedded in the data. List (.LST) files are set up with no implied carriage control.

See section 2.2.1.2 on FDAT$A in the <u>IAS/RSX-11 I/O Operations Reference Manual</u> for additional information on the FDAT$A parameters.

Selecting Data Access Methods

First decide whether to use block I/O or record I/O. Normally use block I/O for files with no record structure, and record I/O for record structured files. However, block I/O is faster than record I/O, because no blocking or deblocking of records is required, and transfers are made directly between the device and the user buffer. Therefore, if your operation does not require accessing individual records within the file, e.g., a file copy operation, use block I/O because it is more efficient.

After you select block I/O or record I/O, there are some other considerations. For block I/O, no FSR block buffer is needed. Instead, you must specify a user buffer. Block I/O is asynchronous; set up an event flag or an AST for synchronization. Also, you must use the additional FDBK$A or FDBK$R macro to specify the user buffer and the synchronization techniques.

For record I/O, choose either sequential access or random access mode. Sequential access can be performed on files with either variable-length records or fixed-length records. Successive PUT$ or GET$ operations in sequential access mode access successive records in the file. This is useful if you need to process all records in the file in order. It is required if the file has variable-length records.

Random access can be performed only in files with fixed-length records. With random access, your program can access records randomly by specifying a record number in each PUT$ or GET$ call. Random access is desirable if you want to access records in an order which is different from their order in the file.

With sequential access, you can use FCS routine to save pointers to an accessed record, and later return to that record. This offers you a limited ability to access records in a random order, or at least an ability to back up to a certain point in the file and continue from there. The actual subroutines are discussed later in this module under Performing I/O.

For record I/O, an FSR block buffer is used for the actual I/O transfers. Blocking and deblocking of records is done transparently for you by FCS. When FCS blocks a record on output, it places it into one or more virtual blocks as needed. When FCS deblocks a record on input, it takes one or more virtual blocks and constructs a logical record. Because GET$/PUT$ operations, used for record I/O, process records which are contained in virtual blocks, not all I/O operations cause an actual I/O transfer. Generally, an I/O transfer is needed only when the end of a block is reached.

You may choose either move mode or locate mode. Figure 10-2 compares the two. In move mode, you always access records in a user specified buffer, sometimes called a user record buffer. Move mode is simple to program, but every PUT$ or GET$ operation requires an extra transfer of the record between the user record buffer and the FSR block buffer. A user record buffer is required.

In locate mode, as long as complete records are located totally within an FSR block buffer, you access the record directly in the FSR block buffer. FCS returns information in the FDB about the location and the size of the record. If records are allowed to span block boundaries and the last record in a block does span the block boundary, then the full record cannot be accessed until the next virtual block is read (in the case of a GET$ operation), or until the current virtual block is written (in the case of a PUT$ operation). In that special case, the record is accessed in a user specified buffer. Therefore, a user record buffer is required in locate mode only if one or more records actually span block boundaries. Table 10-1 summarizes the situations when a user record buffer is needed.

Record I/O operations are synchronous. All synchronization is handled for you by FCS. Control is returned to your program only after the requested PUT$ or GET$ operation is completed.

Table 10-1 When the User Record Buffer Is Needed

Mode	I/O Operation	If Records Span Block Boundaries	If Records Do Not Span Block Boundaries
Move	GET$	Needed	Needed
	PUT$	Needed	Needed
Locate	GET$	Needed	Not needed
	PUT$	Needed	Not needed

Specifying Data Access Methods

Use the FDRC$A or the FDRC$R macro to specify data access methods.

```
FDRC$A racc,urba,urbs

racc - type of access

      methods
        FD.RWM = block mode
        FD.RAN = record mode, random I/O
        default = record mode, sequential I/O

     file truncation
        FD.INS = PUT$ in middle of file does not truncate
                 file
        default = does truncate file

        move or locate
          FD.PLC = locate mode
          default = move mode

urba - user record buffer address (Table 10-1)

urbs - user record buffer size (in bytes).
```

Examples:

1. FDRC$A ,BUFF,80.

   Defaults to record I/O, sequential access in move mode. User record buffer at BUFF, 80. bytes long.

2. FDRC$A FD.RWM

   Block I/O. buffer is specified in FDBK$A macro or in open, READ$, or WRITE$ macros.

3. FDRC$R #FDB4,#FD.RAN!FD.PLC,#BUFF,#100.

   Record I/O, random access in locate mode. User record buffer at BUFF, 100. bytes long. This is a run-time macro which initializes the FDB at FDB4.

If FD.INS is not specified, a PUT$ in the middle of the file places the logical end-of-file right after that record, which truncates the file. If FD.INS is specified, a PUT$ in the middle of the file does not change the logical end-of-file. See section 2.2.1.3 on FDRC$A in the IAS/RSX-11 I/O Operations Manual for additional information.

Additional Initialization of the FDB for Record I/O

Normally, no further initialization is needed for record I/O. However, if you wish to override one or more of the defaults, use the FDBF$A or the FDBF$R macro. The defaults are included in the list of parameters below. The format of the FDBF$A call is:

```
FDBF$A efn,ovbs,mbct,mbfg
```

efn - event flag used internally for synchronization (default is 32(10))

ovbs - override FSR block buffer size (in bytes) (default is standard block size for device)

mbct - multiple buffer count (default generally 1, or single buffering)

mbfg - multiple buffering type (only for multiple buffering)

FD.RAH = read ahead operations
FD.WRB = write behind operations
(default - FD.RAH if file opened for read only, FD.WRB if file opened for a write operation)

Examples:

1. FDBF$A ,,2

   Use double buffering. Defaults: event flag 32(10), FSR block buffer size standard for device (e.g., 512(10) bytes for disk). Multiple buffering type - read ahead if file is opened for read only, write behind if it is opened for a write operation.

2. FDBF$A 12.,2048.

   Use event flag 12(10) and an FSR block buffer size of 2048(10) bytes. This is the standard size for ANSI magtape. It can also be used for disks to cut down on the number of I/O transfers. Default: single buffering.

In the second example, you must reserve enough space in the FSR using the FSRZ$ macro. See section 2.2.1.6 on FDBF$A in the IAS/RSX-11 I/O Operations Reference Manual for further information.

Additional Initialization for Block I/O

For block I/O, you only specify the access method in the FDRC$A or FDRC$R macro. You must use the FDBK$A or FDBK$R macro to set up the user buffer and your synchronization methods. The format of the FDBK$A macro is as follows.

```
FDBK$A bkda,bkds,bkvb,bkef,bkst,bkdn
```

bkda - user buffer address

bkds - user buffer size (in bytes)

bkvb - address of two-word virtual block number

bkef - event flag for synchronization (default = 32(10))

bkst - I/O status block address (must be specified for FCS to return I/O status)

bkdn - AST service routine address

NOTE

Bkvb must be specified after the file is opened using the $R form, or in a READ$ or WRITE$ call.

Example:

```
FDBK$A MYBUF,1024.,,20.,IOST
```

User buffer at MYBUF, size is 1024(10) bytes. Use event flag 20(10), the I/O status block is at IOST. No AST routine.

Bkvb is the address of a two-word data block containing the first virtual block number for a block I/O operation. This data block is copied into the FDB and then used to locate the starting block for the I/O operation.

However, the virtual block number in the FDB is always initialized to '1' when a file is opened. Therefore, this parameter must be specified after the file is opened if you wish to start I/O operations with a block other than virtual block 1. Do this using either a FDBK$R, a READ$, or a WRITE$ call.

The parameter should be left null if you use the $A form. It is present in the $A form only for compatibility with the $R form.

Bkst is the address of an I/O status block. Unlike record I/O, where FCS sets up its own internal I/O status block, block I/O requires that you specify an IOSB in order to get I/O status reports. FCS issues QIOs for you. With record I/O, FCS reports both directive errors and I/O errors automatically. With block I/O, I/O errors are reported only if you specify an IOSB address in a FDRK$A or FDBK$R call.

Initializing the File-Open Section of the FDB

You must also initialize the file-open section of the FDB before opening a file. It contains information about the file to be opened. You must set up data structures so that FCS can build a file specification for the file. In addition, you must specify the LUN to be assigned to the file and the kind of access rights you need (read, write, extend or delete). You can do all of this with an assembly or run-time macro, or in the actual open macro call.

Setting Up the File Specification in the FDB -- At run time, FCS constructs a standard file specification in the filename block in the FDB using the following, in order:

1. The dataset descriptor
2. The default filename block
3. Other defaults of the task or system

FCS first uses any information which is set up in the dataset descriptor. Any non-null data is translated from ASCII to Radix-50 format, and stored in the appropriate offsets in the filename block. If any pieces of the file specification are not specified in the dataset descriptor, FCS next checks the default filename block for any of the missing pieces. Any missing pieces which are found there are filled in next.

If the device or the UFD is still not filled in, normal system defaults are used. The device defaults to the current LUN assignment of the LUN to be used to access the file. The UFD defaults to the default UIC of the task, which is typically the default UIC of the user who runs the task. If the file name or the file type are still not filled in, a file open failure occurs.

If only a dataset descriptor or a default filename block is specified, and not both, the missing structure is skipped. Typically, the dataset descriptor is used for building file specifications at run time. Several routines (get command line (GCML), command string interpreter (CSI), etc.) are available for prompting for input, getting a command line, and then filling in a dataset descriptor. Typically, the default filename block is used to default any fields not specified in the dataset descriptor, or to completely set up a file specification at assembly time. However, one or both structures may be set up at assembly time, if desired.

If you want to have FCS perform I/O to a terminal, just build a file spec with the device TTnn: or TI:. If the specified device is a terminal, FCS just issues QIOs to the terminal. The advantage of this technique over issuing QIOs yourself is that the same I/O routines work correctly with file-oriented devices and terminals. You do not have to rewrite the I/O code to change between device types. The system utility PIP uses FCS calls for all of its I/O operations.

Setting Up the Dataset Descriptor

The dataset descriptor is a six-word data area in your program containing the sizes and the addresses of the ASCII data strings that together make up a file specification. The format of the data area and the ASCII strings is:

```
label:     .WORD    ldev,adrdev
           .WORD    lufd,adrufd
           .WORD    lnam,adrnam
             .
             .
             .
adrdev:    .ASCII   /dev/
ldev       =.-adrdev
adrufd:    .ASCII   /ufd/
lufd       =.-adrufd
adrnam     .ASCII   /full name/
lnam       =.-adrnam
```

Example for file DB1:[202,1]SAMPLE.MAC:

```
DSPT:   .WORD LDEV, DEV
        .WORD LUFD, UFD
        .WORD LNAM, NAM
DEV:    .ASCII /DB1:/
LDEV    =.-DEV
UFD     .ASCII /[202,1]/
LUFD    =.-UFD
NAM:    .ASCII/SAMPLE.MAC/
LNAM    =.-LNAM
```

This example sets up the dataset descriptor and all of its file specification pieces at assembly time. This can also be done at run time. As shown above, FCS builds the file spec DB1:[202,1]SAMPLE.MAC. If no default filename block is specified, the version number takes the normal system default, the latest version for an existing file, and the latest version, plus one, for a new file. See section 2.4.1 on the Dataset Descriptor of the IAS/RSX-11 I/O Operations Reference Manual for additional information.

Setting Up the Default Filename Block

The default filename block is an area within your program containing the various elements of a file specification. Use the NMBLK$ macro call to both reserve space for this area, and to initialize it at assembly time. The format of the NMBLK$ call is:

```
label:  NMBLK$ fnam,ftyp,fver,dvnm,unit
```

Example for file DB1:SAMPLE.MAC:

```
NMBLK$ SAMPLE,MAC,DB,1
```

Notice that you divide the file specification into pieces in the macro call. Also note that you cannot specify a UFD in the default filename block. It can be specified using a dataset descriptor. Otherwise, it is usually taken from the default UIC of the task.

See section 2.4.2 (on Default Filename Block - NMBLK$ Macro Call) for additional information on the default filename block. It also explains how to manually define or override data in the default filename block.

Initializing the File-Open Section Prior to Opening the File

Use the FDOP$A or the FDOP$R macro call. The format of the FDOP$A call is as follows.

FDOP$A lun,dspt,dfnb,facc,actl

lun - LUN for I/O requests

dspt - pointer to dataset descriptor

dfnb - pointer to default name block

facc - type of file access (Table 1Ø-2)

actl - access control

The type of file access indicates the kind of activity that you will perform on the file. Table 1Ø-2 lists these types. Note that you do not specify read, write, extend, or delete; but instead write, read, append, modify, or update. Each implies a request for a particular set of access rights. The meanings of the types are:

write - Write (create) a new file.

read - Read an existing file.

append - Append (add) data to the end of an existing file.

modify - Modify an existing file without changing its length.

update - Update an existing file, extending its length if necessary.

In all cases, the file can also be read.

The actl parameter is used to override the defaults for certain FDB control information, namely:

- Initial magnetic tape position - default depends on file operation.
- Locking of a disk file opened for write if it is not properly closed, e.g., if the task is aborted. Default is that the file is locked.
- The number of retrieval pointers in pool for a disk file window. Default is volume default.
- Enable or disable block locking. Default is disable block locking.

See section 2.2.1.5 on FDOP$A in the IAS/RSX-11 I/O Operation Reference Manual for an explanation of the defaults, and the arguments to override them. This section also covers additional information on the FDOP$A and the FDOP$R macros.

If desired, you can specify all of the FDOP$A or FDOP$R parameters, except actl, in the open macro call instead. The following examples show the use of the FDOP$A call, dataset descriptors, and default filename blocks.

Table 1Ø-2 Types of Access

Class	FDOP$A facc Value	Suffix to Open Call	New or Old File	Access Rights Requested	Default Location of First I/O Operation
Write	FO.WRT	W	New	Read Write Extend	Beginning
Read	FO.RD	R	Old	Read	Beginning
Append	FO.APD	A	Old	Read Write Extend	End (sequential access) Beginning (random access)
Modify	FO.MFY	M	Old	Read Write	Beginning
Update	FO.UPD	U	Old	Read Write Extend	Beginning

Examples:

1.

```
        FDOP$A  1,,DFNB
          .
          .
          .
DFNB:   NMBLK$  MYFILE,DAT,,DB,Ø
```

Use LUN 1, build the file spec in the FDB with the default filename block (since there is no dataset descriptor). The file spec will be DBØ:MYFILE.DAT. The UFD will be taken from the default UIC of the task; the version number takes the normal default.

2.

```
        FDOP$A  2,DSPT
          .
          .
          .
DSPT:   .WORD   Ø,Ø
        .WORD   LUFD,ADRUFD
        .WORD   LNAM,ADRNAM
ADRUFD: .ASCII  /[15,12]/
LUFD    =.-ADRUFD
ADRNAM: .ASCII  /MYFILE.FFF;3/
LNAM    =.-ADRNAM
```

Use LUN 2, build the file spec first with the dataset descriptor, then go to task and system defaults (since there is no default filename block). The File spec will be [15,12]MYFILE.FFF;3. The device will be defaulted to the current LUN assignment and to SY: if not currently assigned.

3.

```
        FDOP$A  1,DSPT1,DFNB1,FO.WRT
          .
          .
          .
DFNB1:  NMBLK$  ANY,FIL
DSPT1:  .WORD   LDEV,DEV
        .WORD   Ø,Ø
        .WORD   LNAM,NAM
DEV:    .ASCII  /DK2:/
LDEV    =.-DEV
DNAM:   .ASCII /MINE/
LNAM    =.-NAM
```

Use LUN 1; open the file for write (create a new file). Build the file spec first from the dataset descriptor, then fill in any missing information from the default filename block. The resulting file spec will be DK2:MINE.FIL. The UFD and version number take normal system defaults. The filename is MINE because the dataset descriptor is used first. Since the name is then filled in, the default filename block is not checked for a name.

Examples of Setting up an FDB

The following examples show the complete process of setting up and initializing FDBs at assembly time before opening a file. Two examples are included for creating a new file, plus two for accessing an existing file. The line comments offer an explanation of the examples.

Creating a New File:

1.

```
        FSRSZ$   1                ; 1 file will be open for
          .                       ; record I/O
          .
          .
FDB1:   FDBDF$
        FDAT$A   R.VAR,RD.CR      ; Variable length records,
                                  ; "list" carriage control
        FDRC$A   ,BUFF,80.        ; URB at BUFF, length 80.
                                  ; bytes. Defaults: sequential
                                  ; access, move mode
        FDOP$A   2,,DFNB          ; Use LUN 2, file spec from
                                  ; Default Name Block
DFNB:   NMBLK$   VARIABLE,ASC     ; File Spec VARIABLE.ASC
```

2.

```
        FSRSZ$  1
          .
          .
          .
FDB1:   FDBDF$
        FDAT$A  R.FIX,FD.FTN,80,    ; Fixed length records,
                                    ; FORTRAN carriage control,
                                    ; 80. byte records
        FDRC$A  RD.RAN,BUFF,80.     ; Random access, URB at BUFF,
                                    ; length is 80. bytes
        FDOP$A  1,DSPT,,FO.WRT      ; Use LUN 1, build file spec
                                    ; from dataset descriptor,
                                    ; open a new file for write
DSPT:   .WORD   0,0                 ; Use default device
        .WORD   0,0                 ; Use default UFD
        .WORD   LNAM,NAM            ; Pointer to file spec
NAM:    .ASCII  /MINE.FIL;2/        ; File name
LNAM    =.-NAM                      ; Length of file name
```

Accessing an Existing File:

1.

```
        FSRSZ$    1
          .
          .
          .
FDB1:   FDBDF$
        FDRC$A    ,URB,25.          ; URB at URB, length = 25.
                                    ; bytes. Defaults: sequential
                                    ; access, move mode
        FDOP$A    3,,DFNB           ; Use LUN 3, build file spec
                                    ; from Default Name Block
```

2.

```
        FSRSZ$   Ø                  ; Only block I/O
          .
          .
          .
FDB1:   FDBDF$
        FDRC$A   FD.RWM             ; Block I/O, no URB needed
        FDBK$A   BUFF,512.          ; For block I/O - sets up
                                    ; buffer at BUFF, length =
                                    ; 512. bytes
        FDOP$A   2,,DFNB            ; Use LUN 2, build file spec
                                    ; from Default Name Block
```

LEARNING ACTIVITY 1Ø-1

The example below shows two FDBs. The second FDB is filled in to display a file at a terminal. Fill in the first FDB for a file YOURS.MAC, with variable length records which will be read and displayed. Use sequential access in locate mode.

```
        FSRSZ$   2               ; 2 "Files" open for record I/O
          .
          .
          .
FDBI:                            ; To be filled in by the student


FDBO:   FDBDF$
        FDAT$A   R.VAR,RD.CR     ; Variable length records,
                                 ; implied carriage return,
                                 ; line feed
        FDRC$A   ,BUFF,8Ø.       ; Sequential I/O, move mode,
                                 ; URB at BUFF, length = 8Ø.
                                 ; bytes
        FDOP$A   2,DSPTO         ; Use LUN 2, override LUN
                                 ; assignment. Build file spec
                                 ; using dataset descriptor
DSPTO:  .WORD    LDEV,DEV        ; pointers to ASCII data
        .WORD    Ø,Ø
        .WORD    Ø,Ø
DEV:    .ASCII   /TI:/           ; Device is TI:
LDEV    =.-DEV
```

Opening a File

Whether or not you set the file access parameter with an FDOP$A or FDOP$R macro call, you can use the general OPEN$ macro call to open the file. If the access parameter is not already specified, specify it in the OPEN$ call. You can also use a number of other open macro calls, which have a single letter suffix to specify the file access. See Table 10-2 for the suffixes and their meanings. With file open macros, you can choose:

- Whether shared access is allowed
- Whether a file is permanent or temporary (deleted when closed)
- Which FCS object modules are used to open the file.

The following list shows all of the possible open macros.

OPEN$ fdb,facc,lun,dspt,dfnb,racc,urba,urbs,err

- General form
- File access specified in facc or previously using FDOP$A or FDOP$R

OPEN$x* fdb,lun,dspt,racc,urba,urbs,err

- Used for most applications
- Requests exclusive write access, shared read access

OPNS$x fdb,lun,dspt,racc,urba,urbs,err

- Allows shared access

OPNT$D fdb,lun,dspt,racc,urba,urbs,err

- Opens temporary file, deletes when closed

OFID$x fdb,lun,dspt,racc,urba,urbs,err

- Opens file by file ID

OFNB$x fdb,lun,dspt,racc,urba,urbs,err

- Specifies file by file name block.

* The "x" in the macro name represents one of the suffixes listed in Table 10-2.

Examples:

```
OPEN$     #FDB1,#FO.WRT,,,,,,ERR1
```

Open the file using the FDB at FDB1 for write access (create a new file). Call ERR1 on an error. All other information is already in the FDB.

```
OPEN$W    #FDB1,,,,,ERR1
```

The same as the last example, only using the other form of the call.

```
OPNT$D    #FDB3,,,,,ERR2
```

Open a new file as a temporary file using the FDB at FDB3. Call ERR2 on an error.

```
OPNS$U     RØ,#3,,,,ERR5
```

Open the file for update using the FDB whose address is in RØ. Allow shared access. Use LUN 3. Call ERR3 on an error.

```
OPEN$      RØ,#FO.UPD!FA.SHR,#3,,,,,ERR5
```

The same as the last example, using the OPEN$ form of the call.

```
OPEN$
```

Open the file using the FDB pointed to by RØ. All information is already in the FDB. The user should check the carry flag for an error.

There is no difference in functionality between the OPEN$ macro with the facc argument filled in, and the OPEN$x, OPNS$x, or OPNT$D forms. Use the form which is most convenient.

OFNB$x uses information already in the filename block of the FDB to open the file. When this occurs, FCS does not build a file spec prior to the open call. This is more efficient if the filename block is still intact, or has been restored after a previous open and close of the file. However, the OFNB$X call causes the Task Builder to include different object modules in your task, thus increasing your task's size. These will be additional modules unless OFNB$x is already used in your program. The same run-time savings can be achieved if you first fill in the filename block and then use an OPEN$, OPEN$x, or OPNS$x call, with no additional object modules added.

OPFNB$x is useful only in overlay situations, or when OFNB$X is already included. Note that the Get Command Line routine (GCML) uses OFNB$X.

As shown in the last module, accessing a file-by-file specification involves a minimum of six disk reads (see Figure 9-5). If you know the file ID of a file, opening the file-by-file ID reduces the number of file accesses to two. This is possible if you reopen a file for a second time or use other FCS routines to obtain the file ID. This is because the file ID allows direct access to the file header of the file.

Any time the file ID field in the FDB is filled in, any open macro call automatically opens the file-by-file ID. The OFID$x call performs the same function, but like the OFNB$x call, it causes the Task Builder to include different object modules in your task, thus increasing its size. Therefore, fill in the file ID and use the regular open macros to open a file-by-file ID. Only use the OFID$x call in an overlay situation, or if OFID$x has already been included in your task.

ERROR CHECKING

If an error condition is detected during any of the file processing operations, the FCS routines set the carry bit in the processor status word (PSW), and return the error code and the type of error to FDB offset locations F.ERR and F.ERR+1.

The run-time FDB initialization macros are an exception to this convention. They do not return any error indications because they involve only moves into FDB locations. The FCS file-processing routines issue appropriate QIOs for you.

As with regular QIOs you issue yourself, directive errors or I/O errors can occur. For record I/O, FCS returns the error codes to the offset F.ERR of the FDB for you so that you don't have to check the I/O status block and the directive status word (DSW) directly yourself. The error codes are always returned as byte values. Since some of the error code values for directive and I/O errors overlap, another byte, offset location F.ERR+1 in the FDB, contains an indicator, whether the error was a directive or an I/O error. A value of 'Ø' in F.ERR+1 indicates an I/O error, a negative value indicates a directive error.

Therefore, to check for errors, check the carry bit on return from each file-processing FCS macro call. If there is an error, use a TSTB to check offset location F.ERR+1 to distinguish whether it is an I/O error or a directive error. Then, check and display the error code value. The following section of code shows a technique for doing this.

Example of Error Checking and Processing

```
BUFF:   .BLKB   80                  ; Output buffer
ARG:    .BLKW   1                   ; Argument block for $EDMSG
EDIR:   .ASCIZ  /FCS DIRECTIVE ERROR. ERROR CODE = %D./
EIO:    .ASCIZ  ?FCC I/O ERROR. ERROR CODE = %D.?
        .EVEN
           .
           .
           .
        OPEN$W  #FDB                ; Open file
        BCS     ERR1                ; Branch on FCS error
           .
           .
           .
;Error Processing
ERR1:   TSTB    F.ERR+1(R0)         ; Directive error or I/O
                                    ;  error?
        BEQ     IO                  ; Branch on I/O error
        MOV     #EDIR,R1            ; Addr of directive error text
                                    ;  string for $EDMSG
        BR      FINSET              ; Branch to common code
IO:     MOV     #EIO,R1             ; Addr of I/O error text string
                                    ;  for $EDMSG signs
FNSET:  MOVB    F.ERR(R0),R0        ; Sign extend FCS error
        MOV     R0,ARG              ;  code and place
        MOV     #ARG,R2             ;  in arg block
        MOV     #BUFF,R0            ; Output buffer
        CALL    $EDMSG              ; Edit error message
        QIOW$S #IO.WVB,#5,#1,,,,<#BUFF,R1,#40> ; Display message
        BCS     ERRQIO              ; Branch on directive error
        EXIT$S                      ; Exit
ERRQIO:   .
          .                         ; Directive error code
          .
```

Using the READ$ and WRITE$ macros, directive errors are returned normally by FCS. Unlike record I/O, with block I/O FCS does not set up an internal IOSB for you. Therefore, you will not get I/O success or failure indications if you do not set up and specify an IOSB.

The error code is sign extended because $EDMSG works only on word values, not on byte values. The error codes and their meanings are listed in Appendix I of the IAS/RSX-11 I/O Operations Reference Manual. They are also in the RSX-11M Mini Reference. Just the directive error codes are in Appendix B of the Executive Reference Manual, and just the I/O error codes are in Appendix B of the RSX-11M/M-PLUS I/O Driver's Reference Manual.

You can also specify the address of your own error-handling routine, and specify it as the last macro call parameter. A JSR PC instruction to the specified user routine is generated. This takes the place of the BCS, and causes a call to the error-handling subroutine in the case of an FCS error. Note that it is a JSR PC which places the return address on the stack. You must clear off the stack for a nonfatal error if you do not use a return at the end of the error routine.

PERFORMING RECORD I/O

Different Forms of PUT$ and GET$

The three different forms of the PUT$ and GET$ macros are:

- GET$ and PUT$
  - Used for sequential access
  - Can also be used for random access if either:

    Records are actually accessed in sequence

    Program manually changes record number field in the FDB
- GET$S and PUT$S
  - Used for sequential access only
  - Takes less space than GET$ and PUT$
  - Used only to optimize space in an overlaid task
- GET$R and PUT$R
  - Used for random access only.

The formats of the macro calls are:

```
GET$      fdb,urba,urbs,err

GET$S     fdb,urba,urbs,err

GET$R     fdb,urba,urbs,lrcnm,hrcnm,err
```

urba and urbs override any previous URB setups

lrcnm and hrcnm - the low word and high word of the record number (random I/O only)

```
PUT$      fdb,nrba,nrbs,err

PUT$S     fdb,nrba,nrbs,err

PUT$R     fdb,nrba,nrbs,lrcnm,hrcnm,err
```

nrba and nrbs override the previous settings for the next record buffer (NRB)

Examples:

PUT$ #FDB1,,,ERR

Write the record pointed to by the next record buffer pointer into the file at the current location. Use the FDB at FDB1.

GET$R ,#MYBUF,#64.,#93.

Read record 93(10) into the buffer MYBUF. The buffer length is 64(10). R0 contains the FDB address.

Sequential Access

For sequential access, use PUT$ and GET$, or PUT$S and GET$S. FCS uses internal pointers to identify the record to be operated on next. The initial pointer location is at the beginning of the file unless the file is opened for Append. In that case, the original pointer location is at the end of the file. Each PUT$ or GET$ operation sets the pointers to the record after the record just accessed. This means that a series of PUT$s or a series of GET$s work on successive records.

To update a record in place, you cannot use a GET$, then update the record, and then use a PUT$. With that seqeunce, the GET$ updates the record after the one you read. Instead, use two special file control routines, .MARK and .POINT, which allow you to save and reset the internal pointers. Use a .MARK before you do the GET$, and save the returned pointers to the record. Then do a GET$ and update the record. Use a .POINT to reset the internal pointers. Finally, issue a PUT$ to update the record. See sections 4.10.1 on .POINT, and 4.10.3 on .MARK, in the IAS/RSX-11 I/O Operations Reference Manual for details on how to use these routines.

After all GET$ operations, the next record buffer (NRB) descriptors identify the address and length of the record just read. The address is located at offset F.NRBD+2 in the FDB, and the length is at offset F.NRBD.

For all PUT$ operations, the NRB descriptors identify the record to be written. Depending on whether you use move or locate mode, as described in the following paragraphs, you may or may not need to use the NRB descriptors.

In move mode (Figure 1Ø-2), GET$ operations always move the record read to the user specified record buffer. Therefore, in general, specify the URB address and size once (in a FDRC$A, FDRC$R, or a file open call). Once these are set up, do not specify them again unless you want to use a different buffer. After each GET$, access the record directly in the URB, which has a known address. If you specify a different URB in a GET$ call, that becomes the URB for later GET$ calls, unless another URB is specified.

PUT$ operations in move mode (Figure 1Ø-2) assume that the record has been built at the location set up in the NRB descriptor. This defaults to the URB. Therefore, the easiest method is to specify the URB once, and then build all records in the URB. Then issue PUT$s without specifying an NRB.

If you want to build your records in a different buffer, you must specify an NRB in the first PUT$ call. After that, for successive PUT$s, build all records in the NRB so that you won't have to respecify an NRB. If however, you mix GET$s and PUT$s, you must specify your NRB in each PUT$ call, because each GET$ call updates the NRB descriptors to point to the record just read (specifically the NRB pointer points to the URB).

In locate mode (Figure 1Ø-2), you generally access records directly in the FSR block buffer. The only time a user record is needed is if a record spans block boundaries. Set up a URB only if this is a possibility.

For GET$ operations, the NRB descriptors identify the record just read. Access the record at the NRB address (offset F.NRBD+2). This pointer points directly into the FSR block buffer if the record does not span block buffer boundaries.

For records which span block buffer boundaries, FCS moves the record to the URB and the NRB pointer points to the URB instead of a location within the FSR block buffer. Do not specify a new URB unless you want to use a different URB for records that span block boundaries.

For PUT$ operations in locate mode (Figure 1Ø-2), build the record at the NRB address. This assumes that the NRB descriptors have already been updated to point to the record to be built, either by the file open macro, or the previous PUT$ or GET$. Once the record is built, use a PUT$ to allow FCS to do some internal bookkeeping and update its internal pointers for the next operation.

In locate mode, be very careful when you write to a file, because you are working directly in the FSR block buffer. If you build a record in the wrong location by mistake, you cannot easily recover any record which gets overwritten. In move mode, on the other hand, since you work in a separate URB buffer, a mistake discovered before issuing a PUT$ does not update the FSR block buffer.

For both move and locate modes, you can also use the .POINT routine to return to the beginning of a file, or the .MARK and .POINT routines to save and later return to a record previously accessed. This allows a very limited form of random access.

Random Access

For random access, use PUT$ and GET$ or PUT$R and GET$R. PUT$R and GET$R are easier to use because you can specify the record number in the macro call. For random operations, on each PUT$ or GET$ call, the record number field in the FDB (offsets F.RCNM, high-order word, and F.RCNM+2, low-order word) is used to calculate the position of the record to be operated on.

When the file is opened, the record number is always initialized to '1', even if the file is opened for Append. After each PUT$ or GET$ operation, the record number is set to one more than the last record accessed. You can override this default by specifying a record number in a PUT$R or GET$R call, or by manually placing the record number directly into the FDB before a PUT$ or GET$ call.

For move mode, the URB and NRB mechanics are exactly the same as for sequential access. For locate mode, GET$ operations are the same as for sequential access.

PUT$ operations are very similar. For PUT$ operations in locate mode, build the record directly at the NRB address. After each PUT$ operation, the NRB pointer is updated to point to the record after the record written. Therefore, if you are updating a record other than the next record, use either a dummy GET$R call or the .POSRC routine to set the NRB pointer to the record to be built. See section 4.10.2 on .POSRC in the IAS/RSX-11 I/O Operation Reference Manual for details on how to use that routine.

For all types of access, as you do PUT$s to a file, FCS transparently extends the file as necessary.

NOTE

FCS updates the logical end-of-file information in the FDB, but not in the file header. Close the file using the CLOSE$ macro to write the end-of-file information out to the file header. See the next section, on closing the file, for additional information.

See sections 3.9 (on GET$) through 3.14 (on PUT$) in the IAS/RSx-11 I/O Operation Reference Manual for additional information on performing record I/O.

Closing the File

Use the CLOSE$ macro to explicitly close a file, specifying the address of the FDB. CLOSE$ performs appropriate cleanup work which involves I/O transfers to the file.

- Waiting for I/O in progress to complete (multiple buffered record I/O only)
- Performing any needed write of the FSR block buffer (record I/O only)
- Updating the file header (high block, end of file block, first free byte).

Since CLOSE$ performs I/O transfers to the file, always check for errors on return to ensure that the transfers were successfully performed. If a CLOSE$ is not issued before a task exits, the Executive closes the file. If the file was opened with write access (write, modify, append, or update), the Executive locks the file unless you specify "no lock of files" in the FDOP$A or FDOP$R call.

Examples of Record I/O

This section contains several examples which show how to use the various FCS services discussed previously for record I/O. Also, look back at Example 1Ø-1, our introductory example. It shows how to create a file with variable-length records using sequential I/O. Examples 1Ø-2 and 1Ø-3 show how to create a file with fixed-length records using sequential I/O. Example 1Ø-2 uses the assembly-time FDB initialization macros, and Example 1Ø-3 uses the run-time FDB initialization macros. Examples 1Ø-1 to 1Ø-3 all use move mode. Example 1Ø-4 shows how to read records from an existing file using locate mode. Example 1Ø-5 shows how to read records from an existing file using random access in move mode.

Example 1Ø-2 creates the file FIXED.ASC. It takes records input at TI: and places them in the file, terminating input and closing the file when a ^Z is typed. The following notes are keyed to Example 1Ø-2.

1. Symbol for record size. Allows easy modification of the record size.

2. The user record buffer (URB). Input from TI: is read into this buffer and then written to the file using PUT$s.

3. Output buffer, argument block, and format strings for generating error messages using $EDMSG. This is for both QIO errors and FCS errors.

4. Allocate FSR space; one FSR block buffer for record I/O. The default size is 512(1Ø) bytes.

5. Assembly-time initialization of FDB.

6. Open file for write. All FDB parameters are already set. The extra ERR1 argument causes a call to the subroutine ERR1 in the case of an open error.

7. Fill the URB with blanks before each read to avoid garbage from a previous read, because you will be reusing the buffer.

8. Issue read. Check for directive and I/O errors. Display an error message and exit on either type of error using common code at SHOERR, except for a ^Z. In that case, branch to a common exit routine which closes the file and exits.

9 Use PUT$ to write the record to the file. FCS takes the record at NRB (in this case the same as URB by default), writes it to the file and updates its internal pointers for the next PUT$. Call the subroutine ERR2 in the case of an error.

10 Branch back, clear the URB and read the next input.

11 After a ^Z, close the file, check for errors and exit. Use BCS here instead of an additional CLOSE$ argument, to show that this technique is also possible. The two forms are similar. The only difference is that BCS does not affect the stack, while the additional argument form uses a JSR PC, which pushes the return address onto the stack.

12 Error processing, as discussed in the section on Error Checking. This code always issues an explicit CLOSE so that the file is unlocked. No error occurs if a file which is not yet opened, is closed. This code does not distinguish whether the error was caused by the OPEN$R, a PUT$, or the CLOSE$ call. Additional code can be added to tell which call caused the error.

```
        .TITLE  CREFXA
        .IDENT  /01/
        .ENABL  LC              ; Enable lower case
;+
; File CREFXA.MAC
;
; CREFXA opens FIXED.ASC for write, inputs records
; from TI: and puts them sequentially to the file.
; A ^z terminates input and closes the file.
;-
        .MCALL  EXST$C,QIOW$C,QIOW$,DIR$ ; System macros
        .MCALL  FSRSZ$,FDBDF$,NMBLK$    ; System FCS
        .MCALL  FDRC$A,FDAT$A,FDOP$A    ;  macros
        .MCALL  OPEN$W,GET$,PUT$,CLOSE$ ;
        .NLIST  BEX             ; Suppress ASCII
RSIZ    = 30.                   ; Record size (bytes)
IOST:   .BLKW   2               ; QIO status block
PRINT:  QIOW$   IO.WVB,5,1,,,,<OBUFF,0,40>
BUFF:   .BLKB   RSIZ            ; User record buffer
OBUFF:  .BLKB   80.             ; Output buffer for
                                ;  error messages
ARG:    .BLKW   1               ; Argument block for
                                ;  $EDMSG
EFDQIO: .ASCIZ  /DIRECTIVE ERROR ON QIO. ERROR CODE = %D./
EFIQIO: .ASCIZ  ?I/O ERROR ON QIO. ERROR CODE = %D.?
EFCDIR: .ASCIZ  /FCS DIRECTIVE ERROR. ERROR CODE = %D./
EFCSIO: .ASCIZ  ?FCS I/O ERROR. ERROR CODE = %D.?
        .EVEN
        .LIST   BEX             ; Show offsets

        FSRSZ$  1               ; 1 file for record I/O
FDB:    FDBDF$                  ; File descriptor block
        FDRC$A  ,BUFF,RSIZ      ; User buffer and size,
                                ;  default is record I/O
                                ;  with sequential access
        FDAT$A  R.FIX,FD.CR,RSIZ ; Fixed length records,
                                ; implied <CR><LF>
        FDOP$A  1,,FILE         ; use LUN 1
FILE:   NMBLK$  FIXED,ASC       ; FIXED.ASC

START:  OPEN$W  #FDB,,,,,,ERR1  ; OPEN; if open fails,
                                ; CALL ERR1
CLRBUF: MOV     #RSIZ,R1        ; Size of URB
        MOV     #BUFF,R2        ; Addr of URB
LOOP:   MOVB    #' ,(R2)+       ; Blank fill record
        SOB     R1,loop         ;  so no garbage fill
```

Example 1Ø-2 Creating a File of Fixed Length Records, Initializing FDB at Assembly Time (Sheet 1 of 3)

```
           QIOW$C  IO.RVB,5,1,,IOST,,<BUFF,RSIZ>; Read a
                                      ;  line from TI:
           BCC     DIROK              ; Branch on Directive ok
           MOV     #EFDQIO,R1         ; Set up for $EDMSG
           MOV     #$DSW,R2           ;
           BR      SHOERR             ; Branch to show error
                                      ;  and exit
   DIROK:  TSTB    IOST               ; Check for I/O error
8          BGT     OKIO               ; Branch if I/O ok
           CMPB    #IE.EOF,IOST       ; Check for EOF
           BEQ     EXIT               ; If EQ, close and exit
           MOVB    IOST,R0            ; I/O status is sign
                                      ;  extended and placed
                                      ;  in argument block
           MOV     R0,ARG             ;  for $EDMSG call
           MOV     #ARG,R2            ; Set up for $EDMSG call
           MOV     #EFIQIO,R1         ;
           BR      SHOERR             ; Branch to show error
                                      ;  and exit
9  OKIO:   PUT$    #FDB,,,ERR2        ; Write next record
10         BR      CLRBUF             ; Get next record

   EXIT:   CLOSE$  #FDB               ; Close file
11         BCS     ERR3               ; Branch on FCS error
           EXST$C  EX$SUC             ; Exit with status of 1

   ; Error  Processing
   ERR1:
   ERR2:
   ERR3:   TSTB    F.ERR+1(R0)        ; Directive error or I/O
                                      ;  error
           BEQ     IO                 ; Branch on I/O error
           MOV     #EFCDIR,R1         ; Set up for $EDMSG,
                                      ;  directive error
           BR      FINSET             ; Branch to finish setup
   IO:     MOV     #EFCSIO,R1         ; Set up for $EDMSG, I/O
12                                    ;  error
   FINSET: MOVB    F.ERR(R0),R0       ; FCS error code
           MOV     R0,ARG             ;  is sign extended and
           MOV     #ARG,R2            ;  placed in arg block
                                      ; $EDMSG argument block
   SHOERR: MOV     #OBUFF,R0          ; Output buffer
           CALL    $EDMSG             ; Format error message
           MOV     R1,PRINT+Q.IOPL+2  ; Size of message
           DIR$    #PRINT             ; Print error message
           CLOSE$  #FDB               ; Close file
           EXST$C  EX$ERR             ; Exit with status of 2
           .END    START
```

Example 10-2 Creating a File of Fixed Length Records, Initializing FDB at Assembly Time (Sheet 2 of 3)

Run Session:

```
>RUN CREFXA
11111
2222
333333
44
Where did you go?
6666 66
^Z
>
```

```
Dump of DB1:[305,301]FIXED.ASC;4 - File ID 23746,13,0
                 Virtual block 0,000001 - Size 512. bytes

000000  061 061 061 061 061 040 040 040 040 040 040 040 040 040 040 040
000020  040 040 040 040 040 040 040 040 040 040 040 040 040 040 062 062
000040  062 062 040 040 040 040 040 040 040 040 040 040 040 040 040 040
000060  040 040 040 040 040 040 040 040 040 040 040 040 063 063 063 063
000100  063 063 040 040 040 040 040 040 040 040 040 040 040 040 040 040
000120  040 040 040 040 040 040 040 040 040 040 064 064 040 040 040 040
000140  040 040 040 040 040 040 040 040 040 040 040 040 040 040 040 040
000160  040 040 040 040 040 040 040 040 127 150 145 162 145 040 144 151
000200  144 040 171 157 165 040 147 157 077 040 040 040 040 040 040 040
000220  040 040 040 040 040 040 066 066 066 066 040 066 066 040 040 040
000240  040 040 040 040 040 040 040 040 040 040 040 040 040 040 040 040
000260  040 040 040 040 000 000 000 000 000 000 000 000 000 000 000 000
000300  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

Example 1Ø-2 Creating a File of Fixed Length Records, Initializing FDB at Assembly Time (Sheet 3 of 3)

Example 1Ø-3 performs the same function as Example 1Ø-2, but it uses the run-time FDB initialization macros. The following notes are keyed to Example 1Ø-3.

1. Include the run-time ($R) macros.

2. At assembly time, simply allocate space for the FDB and initialize the default filename block.

3. Issue the run-time FDB initialization macros, specifying the FDB address in the first call. The first call returns the FDB address in RØ. The subsequent calls default the FDB address to RØ.

```
            .TITLE   CREFXR
            .IDENT   /01/
            .ENABL   LC                   ; Enable lower case
      ;+
      ; File CREFXR.MAC
      ;
      ; CREFXR opens FIXED.ASC for write, inputs records
      ; from TI: and puts them sequentially to the file.
      ; A ^Z terminates input and closes the file.
      ;
      ; This program uses the $R macros at run time to
      ; initialize the FDB
      ;-
            .MCALL   EXST$C,QIOW$C,QIOW$,DIR$ ; System macros
            .MCALL   FSRSZ$,FDBDF$,NMBLK$     ; System FCS
(1)         .MCALL   FDRC$R,FDAT$R,FDOP$R     ;  macros
            .MCALL   OPEN$W,GET$,PUT$,CLOSE$ ;
            .NLIST   BEX                  ; Suppress ASCII
      RSIZ  = 30.                         ; Record size (bytes)
      IOST:    .BLKW    2                 ; QIO status block
      PRINT:   QIOW$    IO.WVB,5,1,,,,<OBUFF,0,40>
      BUFF:    .BLKB    RSIZ              ; User Record Buffer
      OBUFF:   .BLKB    80.               ; Output buffer for
                                          ;  error messages
      ARG:     .BLKW    1                 ; Argument block for
                                          ;  $EDMSG
      EFDQIO: .ASCIZ  /DIRECTIVE ERROR ON QIO. ERROR CODE = %D./
      EFIQIO: .ASCIZ  ?I/O ERROR ON QIO. ERROR CODE = %D.?
      EFCDIR: .ASCIZ  /FCS DIRECTIVE ERROR. ERROR CODE = %D./
      EFCSIO: .ASCIZ  ?FCS I/O ERROR. ERROR CODE = %D.?
            .EVEN
            .LIST    BEX                  ; Show offsets

            FSRSZ$   1                    ; 1 file for record I/O
(2)   FDB:     FDBDF$                     ; Allocate space for FDB
      DFILE:   NMBLK$   FIXED,ASC         ; Default Name Block,
                                          ;  for 'FIXED.ASC'

      START:   FDAT$R   #FDB,#R.FIX,#FD.CR,#RSIZ ; Fixed length
                                          ;  records, implied <CR>
                                          ;  <LF>
               FDRC$R   ,,#BUFF,#RSIZ     ; User buffer addr and
(3)                                       ;  size, move mode and
                                          ;  sequential access by
                                          ;  default
               FDOP$R   ,#1,,#DFILE       ; Use LUN 1, Default
                                          ;  Name Block
```

Example 10-3 Creating a File of Fixed Length Records, Initializing FDB at Run Time (Sheet 1 of 3)

```
          OPEN$W   ,,,,,,ERR1          ; OPEN - if open fails,
                                       ;  CALL ERR1
CLRBUF:   MOV      #RSIZ,R1            ; Size of URB
          MOV      #BUFF,R2            ; Addr of URB
LOOP:     MOVB     #' ,(R2)+           ; Blank fill record
          SOB      R1,LOOP             ;  so no garbage fill
          QIOW$C   IO.RVB,5,1,,IOST,,<BUFF,30.> ; Read a
                                       ; line from TI:
          BCC      DIROK               ; Branch on Directive ok
          MOV      #EFDQIO,R1          ; Set up for $EDMSG
          MOV      #$DSW,R2
          BR       SHOERR              ; Branch to show error
                                       ;  and exit
DIROK:    TSTB     IOST                ; Check for I/O error
          BGT      OKIO                ; Branch if I/O ok
          CMPB     #IE.EOF,IOST        ; Check for EOF
          BEQ      EXIT                ; If EQ, close and exit
          MOVB     IOST,R0             ; I/O status is sign
                                       ;  extended and placed
                                       ;  in argument block
          MOV      R0,ARG              ;  for $EDMSG call
          MOV      #ARG,R2             ; Set up for $EDMSG call
          MOV      #EFIQIO,R1          ;
          BR       SHOERR              ; Branch to show error
                                       ;  and exit

OKIO:     PUT$     #FDB,,,ERR2         ; Write next record
          BR       CLRBUF              ; Get next record

EXIT:     CLOSE$   #FDB                ; Close file
          BCS      ERR3                ; Branch on FCS error
          EXST$C   EX$SUC              ; Exit with status of 1

; Error  Processing
ERR1:
ERR2:
ERR3:     TSTB     F.ERR+1(R0)         ; Directive error or I/O
                                       ;  error
          BEQ      IO                  ; Branch on I/O error
          MOV      #EFCDIR,R1          ; Set up for $EDMSG,
                                       ;  directive error
          BR       FINSET              ; Branch to finish setup
IO:       MOV      #EFCSIO,R1          ; Set up for $EDMSG, I/O
                                       ;  error
FINSET:   MOVB     F.ERR(R0),R0        ; FCS error code
          MOV      R0,ARG              ;  is sign extended and
          MOV      #ARG,R2             ;  placed in arg block
                                       ;  for $EDMSG
```

Example 10-3 Creating a File of Fixed Length Records, Initializing FDB at Run Time (Sheet 2 of 3)

```
SHOERR: MOV     #OBUFF,R0         ; Output buffer
        CALL    $EDMSG            ; Format error message
        MOV     R1,PRINT+Q.IOPL+2 ; Size of message
        DIR$    #PRINT            ; Print error message
        CLOSE$  #FDB              ; Close file
        EXST$C  EX$ERR            ; Exit with status of 2
        .END    START
```

Run Session

```
>RUN CREFXR
11111
2222
333333
44
Where did you go?
6666 66
^Z
>
```

```
Dump of DB1:[305,301]FIXED.ASC;5 - File ID 24564,6,0
                  Virtual block 0,000001 - Size 512. bytes

000000  061 061 061 061 061 040 040 040 040 040 040 040 040 040 040 040
000020  040 040 040 040 040 040 040 040 040 040 040 040 040 040 062 062
000040  062 062 040 040 040 040 040 040 040 040 040 040 040 040 040 040
000060  040 040 040 040 040 040 040 040 040 040 040 040 063 063 063 063
000100  063 063 040 040 040 040 040 040 040 040 040 040 040 040 040 040
000120  040 040 040 040 040 040 040 040 040 040 064 064 040 040 040 040
000140  040 040 040 040 040 040 040 040 040 040 040 040 040 040 040 040
000160  040 040 040 040 040 040 040 040 127 150 145 162 145 040 144 151
000200  144 040 171 157 165 040 147 157 077 040 040 040 040 040 040 040
000220  040 040 040 040 040 040 066 066 066 066 040 066 066 040 040 040
000240  040 040 040 040 040 040 040 040 040 040 040 040 040 040 040 040
000260  040 040 040 040 000 000 000 000 000 000 000 000 000 000 000 000
000300  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

Example 10-3 Creating a File of Fixed Length Records, Initializing FDB at Run Time (Sheet 3 of 3)

Example 1Ø-4 reads the first five records of the file VAR1.ASC (which is created using Example 1Ø-1) and displays them at TI:. It uses sequential I/O in locate mode. The following notes are keyed to Example 1Ø-4.

1. FDB allocation and initialization. Specify locate mode; default is sequential access. No FDAT$A or FDAT$R macro is needed because the file already exists. No user record buffer is needed because none of the first five records spans block boundaries. None span block boundaries because the maximum input record for Example 1Ø-1 is 8Ø(1Ø) bytes. Specify a URB if records can span block boundaries.

2. Set up loop counter to read five records.

3. GET$ a record. The FDB pointer is returned in RØ after the OPEN$R call.

4. Write the record at TI:. The pointer to the record is at offset F.NRBD+2 in FDB; size is at offset F.NRBD in FDB. No '#' is used because we want to use the contents of those locations as arguments.

5. Decrement the counter and loop back until done. When done, close the file and exit.

```
        .TITLE  READLC
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; File READLC.MAC
;
; This task reads the first 5 records from the file
; VARI.ASC and displays them at the terminal. It uses
; locate mode.
;-
        .MCALL  OPEN$R,GET$,QIOW$S,NMBLK$,FDOP$A
        .MCALL  CLOSE$,EXIT$S,FDBDF$,FDRC$A,FSRSZ$

        FSRSZ$  1                       ; 1 FSR block buffer
FDB:    FDBDF$
        FDRC$A  FD.PLC                  ; Locate mode
        FDOP$A  1,,DFNB                 ; LUN 1, default file
                                        ; name block
DFNB:   NMBLK$  VARI,ASC                ; File FIXED.ASC

```

Example 1Ø-4 Accessing a File in Locate Mode (Sheet 1 of 2)

```
OBUFF:  .BLKW   80.             ; Error message buffer
ARG:    .BLKW   1               ; $EDMSG argument block
EFCDIR: .ASCIZ  /FCS DIRECTIVE ERROR. ERROR CODE = %D./
EFCSIO: .ASCIZ  'FCS I/O ERROR. ERROR CODE = %D.'
        .EVEN
START:  OPEN$R  #FDB            ; Open file for read
        BCS     ERRO            ; Branch on FCS error
        MOV     #5,R2           ; Loop counter
LOOP:   GET$                    ; Get record
        BCS     ERRR            ; Branch on error
        QIOW$S  #IO.WVB,#5,#1,,,,<FDB+F.NRBD+2,FDB+F.NRBD,#40>
        SOB     R2,LOOP         ; Decrement loop counter
        CLOSE$                  ; Close file
        BCS     ERRC            ; Branch on error
        EXIT$S                  ; Exit
; Error code
ERRC:
ERRR:
ERRO:   TSTB    F.ERR+1(R0)     ; Directive error or I/O
                                ;  error?
        BEQ     IOERR           ; Branch on I/O error
        MOV     #EFCDIR,R1      ; Set up for $EDMSG
        BR      FINSET          ; Branch to display code
IOERR:  MOV     #EFCSIO,R1      ; Set up for $EDMSG
FINSET: MOVB    F.ERR(R0),R0    ; Sign extend FCS error
        MOV     R0,ARG          ;  code and place in
        MOV     #ARG,R2         ;  argument block
        MOV     #OBUFF,R0       ; Output buffer
        CALL    $EDMSG          ; Format error message
        QIOW$S  #IO.WVB,#5,#1,,,,<#OBUFF,R1,#40> ; Write
                                ;   message
        CLOSE$  #FDB            ; Close file
        EXIT$S                  ; Exit
        .END    START
```

Run Session

```
>RUN READLC
1111
22222 22
333
JAZZ Jazz JAZZ Jazz
Have you ever seen the sun?
>
```

Example 10-4 Accessing a File in Locate Mode (Sheet 2 of 2)

Example 1Ø-5 uses random access to read records from the file FIXED.ASC, which is created using either Example 1Ø-2 or Example 1Ø-3. It prompts at TI: for a record number and displays that record at TI:. The following notes are keyed to Example 1Ø-5.

1. FCSMC$ is a macro containing .MCALLs for most of the FCS macros. Using it can avoid having to specify all of the FCS macros in .MCALL statements. Note that GET$R is not included in FCSMC$, so it is specified separately.

2. Using the $ form of all QIOs. With the exception of the QIO at PRINT, all parameters are set at assembly time. Therefore, any of the three forms ($, $C, or $S) can be used for the QIOs at ECHO, PROMPT, and WARN. Either the $ or the $S can be used for the QIO setup at PRINT.

3. The record number is input in ASCII (up to two digits).

4. FDB for file. Use random access I/O.

5. Open file for read. Prompt for and read record number.

6. On a directive error, use the supplied macro DIRERR to display an error message and the DSW, and then exit.

7. Check for I/O error. If the error was a ^Z, branch to EXIT to close the file and exit. If any other error occurred, use the supplied macro IOERR to display an error message and the IOSB, and then exit.

8. Set up and call .DD2CT to convert the record number from ASCII to double-precision binary. During setup, make sure that at least one digit was input. If none were input, prompt again. Double precision is used only to show how to take advantage of the full range of record numbers (up to 31 bits in two words).

9. Move the low-order 16 bits of the record number to R2, and use GET$R to read the record. Call the subroutine ERR2 on any FCS error. Only the low-order 16 bits are needed, because we know the record number is less than or equal to 99(1Ø).

10. Display the record and prompt for the next record number. Always display 3Ø(1Ø) characters, because the file has fixed-length records.

11 On FCS errors, first check for end-of-file. If it is not, branch to common FCS error code at ERR3. If the error is read past end-of-file, display a warning message and prompt for the next input. Because this routine is entered with a JSR PC instruction, you must clear the return address off the stack before using a BR instruction.

Notice that the supplied macros DIRERR and IOERR are used for QIO errors, but not for FCS errors. This is because FCS returns error status to the FDB. In fact, DIRERR would work correctly because the DSW is returned by the Executive, and FCS moves the DSW value into the FDB. IOERR however, does not work correctly because FCS sets up its own internal I/O status block, and its address is not available to the user task. The supplied macro FCSERR is set up to handle both types of FCS errors.

```
        .TITLE  RANDOM
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; File RANDOM.MAC
;
;  RANDOM uses direct access to a file, FIXED.ASC, which
;  contains fixed length records. This task prompts at
;  TI: for a record number, and displays it at TI:
;  It exits when a control Z is input
;
; Assemble and task-build instructions:
;
;       >MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]-
;       ->RANDOM
;       >LINK/MAP RANDOM,LB:[1,1]PROGSUBS/LIBRARY
;-
        .MCALL  QIOW$,DIR$,EXIT$S,GET$R ; System macros
        .MCALL  FCSMC$                  ; Macro to get most
                                        ;  system FCS macros
        .MCALL  DIRERR,IOERR            ; Supplied macros
        FCSMC$                          ; Get most FCS macros
        .NLIST  BEX                     ; Supress ASCII
; LOCAL DATA
        RSIZ    =30.                    ; Record size
ECHO:   QIOW$   IO.WVB,5,1,,,,<BUFF,30.,40>
PRMPT:  QIOW$   IO.RPR,5,1,,IOST,,<AREC,2,,MES1,SIZ1,'$>
WARN:   QIOW$   IO.WVB,5,1,,,,<MES3,SIZ3,40>
PRINT:  QIOW$   IO.WVB,5,1,,,,<OUT,0,40>

BUFF:   .BLKB   RSIZ                    ; User record buffer
IOST:   .BLKW   2                       ; I/O status block
AREC:   .BLKW   1                       ; Record number in ASCII
REC:    .BLKW   2                       ; Record number in binary
OUT:    .BLKB   100.                    ; $EDMSG output buffer

MES1:   .ASCII  /RECORD NUMBER?/
        SIZ1 =  .-MES1
MES2:   .ASCIZ  /FCS ERROR. ERROR CODE = %D./
MES3:   .ASCII  /***PAST END OF FILE***/
        SIZ3 =  .-MES3
EFCDIR: .ASCIZ  /FCS DIRECTIVE ERROR. CODE = %D./
EFCSIO: .ASCIZ  'FCS I/O ERROR. CODE = %D.'
        .EVEN
```

Example 1Ø-5 Accessing a File in Random Mode
(Sheet 1 of 3)

```
            FSRSZ$  1                     ; 1 file opened for
                                          ;  record I/O
    FDB:    FDBDF$                        ; File descriptor block
            FDRC$A  FD.RAN,BUFF,RSIZ      ; Random access, URB
                                          ;  addr and size
(4)         FDOP$A  1,,FILE               ; Use LUN 1, default
                                          ;  name block at FILE
    FILE:   NMBLK$  FIXED,ASC             ; Default name -
                                          ;  FIXED.ASC
            .ENABL LSB

    START:  OPEN$R  #FDB,,,,,,ERR1        ; Open file for read
(5)                                       ;  CALL ERR1 on open
                                          ;  error
    10$:    DIR$    #PRMPT                ; Prompt for record
                                          ;  number
            BCC     DIROK                 ; Branch on dir ok
(6)         DIRERR  <ERROR ON QIO>
(7) DIROK:  TSTB    IOST                  ; Check for I/O error
            BLT     ERR1I                 ; Branch on error
    ; Convert ASCII record number to double-worded decimal
            MOV     IOST+2,R4             ; # of characters to
                                          ;  convert
            BEQ     10$                   ; If no characters,
                                          ;  prompt again
(8)         MOV     #AREC,R5              ; Address of ASCII
                                          ;  characters
            MOV     #REC,R3               ; Buffer to store
                                          ;  converted number
            CALL    .DD2CT                ; Convert ASCII to
                                          ;  decimal
(9)         MOV     REC+2,R2              ; Move low order 16 bits
            GET$R   #FDB,,,R2,,ERR2       ; Get specified record
(10)        DIR$    #ECHO                 ; Print it on TI:
            BR      10$                   ; Prompt for next input

    ; ERROR ROUTINES
    ERR1I:  CMPB    #IE.EOF,IOST          ; ^Z?
(7)         BEQ     EXIT                  ; If yes, branch to EXIT
            IOERR   #IOST,<ERROR ON QIO>
    ; Here for errors on GET$
    ERR2:   CMPB    #IE.EOF,F.ERR(R0)     ; Was the error an EOF?
            BNE     ERR1                  ; No, it is another
                                          ;  error, branch to ERR1
(11); Just display a warning for end of file
            DIR$    #WARN                 ; Display EOF message
            TST     (SP)+                 ; Clean off return addr
                                          ;  from stack
            BR      10$                   ; Prompt for next input
```

Example 1Ø-5 Accessing a File in Random Mode
(Sheet 2 of 3)

```
ERR3:
ERR1:   MOVB    F.ERR(R0),R5      ; Extend sign on error
        MOV     R5,IOST           ;  code and move into
        MOV     #IOST,R2          ;  argument block
        TSTB    F.ERR+1(R0)       ; I/O or directive error?
        BEQ     IOERR             ; Branch on I/O error
        MOV     #EFCDIR,R1        ; Directive error message
        BR      DSPERR            ; Branch to display code
IOERR:  MOV     #EFCSIO,R1        ; I/O error message
DSPERR: MOV     #OUT,R0           ; Output buffer
        CALL    $EDMSG            ; Edit ouput message
        MOV     R1,PRINT+Q.IOPL+2 ; Length of error
                                  ;   message
        DIR$    #PRINT            ; Print error message
        CLOSE$  #FDB              ; Close file
        EXIT$S                    ; EXIT
EXIT:   CLOSE$  #FDB,ERR3         ; Close file
        EXIT$S                    ; Exit
        .END    START
```

Run Session:

```
>RUN RANDOM
RECORD NUMBER?1
11111
RECORD NUMBER?3
333333
RECORD NUMBER?9
***PAST END OF FILE***
RECORD NUMBER?5
Where did you go?
^Z
>
```

Example 10-5 Accessing a File in Random Mode
(Sheet 3 of 3)

PERFORMING BLOCK I/O

READ$ and WRITE$ Calls

The formats of the READ$ and WRITE$ calls are:

```
READ$   fdb,bkda,bkds,bkvb,bkef,bkst,bkdn,err
WRITE$  fdb,bkda,bkds,bkvb,bkef,bkst,bkdn,err
```

All parameters except fdb and err override any previous FDB settings. Always use a user specified buffer, which can be specified in a FDBK$A, or FDBK$R call, an open call, or in a READ$ or WRITE$ call.

The length of the transfer is controlled by the bkds parameter. The starting virtual block number in the FDB is initially set to 1, unless the file is opened for Append. FCS updates the block number after each operation to point to the block after the last one accessed. To override the default block number, set up a two-word data block for the virtual block number and specify the address of the block in a FDBK$R macro call (after the file is opened), or in a READ$ or WRITE$ macro call.

The following piece of skeletal code shows how to use block 5, then block 12(1Ø), and finally block 13(1Ø).

```
BLCKNM: .WORD   Ø,5                   ; Starting block number
                .
                .
                .
        OPEN$R  #FDB                  ; Open file
        BCS     ERR1
                .
                .
                .
        READ$   #FDB,,,#BLCKNM,,,ERR2 ; Read block 5
                .
                .
                .
        MOV     #12., BLCKNM+2  ; Update block #
        READ$   #FDB,,,#BLCKNM,,,ERR3 ; Read virtual
                                      ; block 12(1Ø)
                .
                .
                .
        READ$    #FDB,,,,,,ERR4  ; Read next virtual block
```

Unlike record I/O, for block I/O each READ$ or WRITE$ causes an I/O transfer between the user buffer and the file.

Synchronization and Error Checking

For block I/O, FCS issues asynchronous QIO directives. You must provide synchronization and check for both directive and I/O errors. Use an event flag or an AST routine for synchronization. Check for errors on return from the READ$ or WRITE$ call (after the QIO directive is issued).

Also check for I/O errors after the I/O operation completes. To get I/O error indications, you must set up and specify an I/O status block. Otherwise, no I/O status conditions are returned, and success must be assumed.

If you use an AST routine for synchronization, check the IOSB directly for I/O errors. If you use an event flag for synchronization, use a WAIT$ call to wait for the flag to be set, rather than a Wait for Single Event Flag (WTSE$) directive. With WAIT$, FCS returns its standard error indications. This means the carry bit is clear for success and set for an I/O error. In addition, for errors, the I/O error code is returned at offset F.ERR in the FDB. If you use the Wait for Single Event Flag directive (WTSE$) instead, standard FCS error codes are not returned. In that case, you must check the IOSB directly yourself.

See sections 3.15 (on READ$), 3.16 (on WRITE$), and 3.17 (on WAIT$) for additional information about block I/O calls, synchronization, and error checking.

Examples 1Ø-6 and 1Ø-7 show how to use block I/O. Example 1Ø-6 creates a file BLOCK.ASC using block I/O. Example 1Ø-7 reads a virtual block from the file BLOCK.ASC and displays it at TI:. The following notes are keyed to Example 1Ø-6.

1. You still need an FSRSZ$ statement to set up an FCS, but no FSR block buffers are needed.

2. FDB setup. FDAT$A and FDOP$A are the same as for record I/O. The only difference here is that a dataset descriptor is used instead of a default filename block. This is done just to show the use of a dataset descriptor. FDRC$A specifies read/write mode (block I/O). FDBK$A specifies the address and size of the user buffer, the event flag for synchronization, and the IOSB address. Don't specify the address of the block number until after opening the file.

3 Other data structures: A two-word block for the virtual block number, initially set to virtual block 1; the user buffer, and the I/O status block.

4 Prompt for and read virtual block number. "Low only" means only a one-word virtual block number, rather than a two-word value.

5 Place a terminating null character at the end of the ASCII virtual block number to set up for a call to $CDTB. Use $CDTB to convert ASCII decimal to binary. There is no error check included, but it can be added.

6 Move the converted virtual block number to the virtual block number block.

7 Prompt for and get a character to place in the block.

8 Fill the user buffer with the character.

9 Open (create) the file.

10 Start the I/O transfer. Specify the address of the virtual block now, since the "open" call initializes the block number to 1.

11 Display the message and wait for the I/O transfer to complete. Using WAIT$, FCS returns its standard error indications. Call subroutine ERR3 in the case of an error.

12 Close the file.

13 Set up to display the number of characters transferred, branch to common code to edit and display the message, and then exit. The code is common to the error message code.

14 Common error code. Set up for an I/O or directive error message. Set up to exit with error status.

15 Common code for displaying a message, closing the file, and exiting. In the case of a successful write, you end up calling CLOSE$ twice. There is no error for closing the file after it is already closed. Use of the common code saves space in the task.

```
        .TITLE  BLOCK1
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; File BLOCK1.MAC
;
; BLOCK1 creates a file BLOCK.ASC and fills the specified
; virtual block of the file with the specified character.
; It uses block I/O.
;-
        .MCALL  QIOW$,DIR$,QIOW$S,EXST$S ; System macros
        .MCALL  FDBDF$,FDRC$A,FDBK$A,FDOP$A,NMBLK$
        .MCALL  FDAT$A,FSRSZ$,OPEN$W,WRITE$,WAIT$,CLOSE$

        .NLIST  BEX
MES1:   .ASCII  /VIRTUAL BLOCK NUMBER (LOW ONLY): /
LEN1    = . - MES1
MES2:   .ASCII  /CHARACTER: /
LEN2=   . - MES2
MES3:   .ASCII  /1 BLOCK BEING WRITTEN TO FILE/
LEN3    = . - MES3
MES4:   .ASCII  /WRITE COMPLETED, %D BYTES WRITTEN TO /
        .ASCIZ  /FILE/
MESD:   .ASCIZ  /FCS DIRECTIVE ERROR, CODE = %D./
MESI:   .ASCIZ  'FCS I/O ERROR, CODE = %D.'

CHAR:   .BLKB   1                       ; Character to write
BUFF:   .BLKB   100.                    ; Buffer for $EDMSG
        .LIST   BEX
        .EVEN
(1)     FSRSZ$  0                       ; No FSR block buffers
                                        ;  needed for block I/O
FDB:    FDBDF$                          ; Reserve FDB space
        FDAT$A  R.VAR,FD.FTN            ; File characteristics
        FDRC$A  FD.RWM                  ; Read/write mode
        FDBK$A  BLOCK,512.,,,1,IOSB ; Adr, size of buffer,
                                        ;  ef 1, IOSB addr
(2)     FDOP$A  1,DSPT                  ; LUN 1, DSPT
DSPT:   .WORD   0,0                     ; Length and addr of device
        .WORD   0,0                     ; Length and addr of UIC
        .WORD   LNAM,NAM                ; Length and addr of name
NAM:    .ASCII  /BLOCK.ASC/             ; File name and type
LNAM    =.-NAM
        .EVEN
VBN:    .WORD   0,1                     ; Default VBN
(3) BLOCK: .BLKW 256.                   ; User buffer
IOSB:   .BLKW   2                       ; I/O status block

TYPE1:  QIOW$   IO.RPR,5,1,,IOSB,,<BUFF,6,,MES1,LEN1,'$>
TYPE2:  QIOW$   IO.RPR,5,1,,,,<CHAR,1,,MES2,LEN2,'$>
TYPE3:  QIOW$   IO.WVB,5,2,,,,<MES3,LEN3,40>
```

Example 10-6 Creating a File With Block I/O (Sheet 1 of 3)

```
; Code
START:  DIR$    #TYPE1          ; Prompt and get VBN
        MOV     IOSB+2,R0       ; Length in R0
        CLRB    BUFF(R0)        ; Put null byte at end
        MOV     #BUFF,R0        ; R0 => ASCII digits
        CALL    $CDTB           ; Convert to binary
        MOV     R1,VBN+2        ; Store as low VBN
        DIR$    #TYPE2          ; Input character
; Fill user buffer with character
        MOV     #BLOCK,R0       ; Get address
        MOV     #512.,R1        ;  and size of user buffer
10$:    MOVB    CHAR,(R0)+      ; Move character
        SOB     R1,10$          ; Loop back until done
; Open file to receive characters, write virtual block
        OPEN$W  #FDB,,,,,,ERR1  ; Open, ERR1 if no good
        WRITE$  ,,,#VBN,,,,ERR2 ; Start transfer
        DIR$    #TYPE3          ; Say transfer started
        WAIT$   ,,,ERR3         ; Wait until it's done
        CLOSE$  ,ERR4           ; Close file
        MOV     #MES4,R1        ; Adr of completion
                                ;  message
        MOV     #EX$SUC,R5      ; Exit status
        BR      FORMAT          ; Branch to common code
                                ;  for message display
                                ;  and exit
ERR1:
ERR2:
ERR3:
ERR4:   TSTB    F.ERR+1(R0)     ; Directive or I/O error?
        BEQ     IOERR           ; Branch on I/O error
        MOV     #MESD,R1        ; => I/O error message
        BR      FCSSET          ; Branch to common code
IOERR:  MOV     #MESI,R1        ; => Dir error message
FCSSET: MOVB    F.ERR(R0),R4    ; Sign extend error code
        MOV     R4,IOSB+2       ; Use I/O status block
                                ;  for arg block
        MOV     #EX$ERR,R5      ; Exit status to R5
; Print message, exit with status in R5
FORMAT:
        MOV     #IOSB+2,R2      ; R2 => I/O status
        MOV     #BUFF,R0        ; R0 => $EDMSG buffer
        CALL    $EDMSG          ; Format the text
        QIOW$S  #IO.WVB,#5,#2,,,,<#BUFF,R1,#40>; And
                                ;  write it out to TI:
        CLOSE$  #FDB            ; Close file and
        EXST$S  R5              ; Exit with status
        .END    START
```

Example 1∅-6 Creating a File With Block I/O (Sheet 2 of 3)

```
Run Session

>RUN BLOCK1
VIRTUAL BLOCK NUMBER (LOW ONLY): 2
CHARACTER: e
1 BLOCK BEING WRITTEN TO FILE
WRITE COMPLETED,    512 BYTES WRITTEN TO FILE
>


Dump of DR2:[305,301]BLOCK.ASC;10 - File ID 37355,2,0

                  Virtual block 0,000001 - Size 512. bytes

        Contains whatever was previously in that block on the disk


Dump of DR2:[305,301]BLOCK.ASC;10 - File ID 37355,2,0

                  Virtual block 0,000002 - Size 512. bytes

000000  145 145 145 145 145 145 145 145 145 145 145 145 145 145 145 145

000020  145 145 145 145 145 145 145 145 145 145 145 145 145 145 145 145

000040  145 145 145 145 145 145 145 145 145 145 145 145 145 145 145 145

                                  .

                                  .

                                  .

000740  145 145 145 145 145 145 145 145 145 145 145 145 145 145 145 145

000760  145 145 145 145 145 145 145 145 145 145 145 145 145 145 145 145

*** EOF ***
```

Example 10-6 Creating a File With Block I/O (Sheet 3 of 3)

Example 1Ø-7 prompts at TI: for a virtual block number, and then reads and displays that block of BLOCK.ASC. The following notes are keyed to the example.

1. This example displays, in addition to the error codes, text error messages which tell the user which FCS call caused the error.

2. No FSR block buffers are needed for block I/O.

3. This is the FDB setup. Use read/write mode. Use FDBK$A to specify the user buffer address and size, the event flag, and the IOSB address.

4. Open the file for read and prompt for the virtual block number.

5. Place a null byte at the end of block number for a call to $CDTB. Use $CDTB to convert the block number from ASCII decimal to binary.

6. Store the result, returned in R1 by $CDTB, in the low byte of the virtual block number block.

7. Issue a READ$ to read the specified block, and use WAIT$ to wait for the I/O operation to complete. READ$ and WAIT$ return standard FCS status and error returns.

8. Display a heading and the virtual block.

9. Close the file and exit.

10. Error code to display the text message, including the error code for each type of error.

```
        .TITLE  BLOCK2
        .IDENT  /01/
        .ENABL  LC              ; Enable lower case
;+
; File BLOCK2.MAC
;
; BLOCK2 prompts at TI: for a virtual block number
; and then reads and displays that block of "BLOCK.ASC"
;-
        .MCALL  QIOW$,DIR$,QIOW$S,EXST$S,EXIT$S
        .MCALL  FDBDF$,FDRC$A,FDBK$A,FDOP$A,NMBLK$
        .MCALL  FSRSZ$,OPEN$R,READ$,WAIT$,CLOSE$

        .NLIST  BEX
CR      = 15
LF      = 12
MES1:   .ASCII  /VIRTUAL BLOCK NUMBER: /
LEN1    = . - MES1
MES2:   .ASCII  <CR><LF>/HERE IS THE BLOCK : /<CR><LF>
LEN2    = . - MES2
MES3I:  .ASCIZ  'I/O ERROR ON OPEN$R, CODE = %D.'
MES3D:  .ASCIZ  /DIRECTIVE ERROR ON OPEN$R, CODE = %D./
MES4I:  .ASCIZ  'I/O ERROR ON READ$, CODE = %D.'
MES4D:  .ASCIZ  /DIRECTIVE ERROR ON READ$, CODE - %D./
MES5I:  .ASCIZ  'I/O ERROR AFTER WAIT$, CODE = %D.'
MES5D:  .ASCIZ  /DIRECTIVE ERROR AFTER WAIT$, CODE = %D./
MES6I:  .ASCIZ  'I/O ERROR ON CLOSE$, CODE = %D.'
MES6D:  .ASCIZ  /DIRECTIVE ERROR ON CLOSE$, CODE = %D./
BUFF:   .BLKB   80.             ; $EDMSG output buffer

        .LIST   BEX
        .EVEN
        FSRSZ$  0               ; No FSR block buffer
                                ;  needed for block I/O
FDB:    FDBDF$                  ; FDB for input file
        FDRC$A  FD.RWM          ; Read/write mode
        FDBK$A  BLOCK,512.,,1,IOSB ; Buffer adr, size,
                                ;  ef 1, iosb adr
        FDOP$A  1,,FILE         ; LUN 1, DFNB
FILE:   NMBLK$  BLOCK,ASC       ; Name is BLOCK.ASC

VBN:    .WORD   0,1             ; Default VBN
BLOCK:  .BLKW   256.            ; User buffer
IOSB:   .BLKW   2               ; IOSB

PROMPT: QIOW$   IO.RPR,5,1,,IOSB,,<BUFF,6,,MES1,LEN1,'$>
                                ; Prompt and get VB #
DONE:   QIOW$   IO.WVB,5,1,,,,<MES2,LEN2,40> ; Done
                                             ;  message
DUMP:   QIOW$   IO.WVB,5,1,,,,<0,64.,40> ; Display of VB
```

Example 10-7 Reading a File With Block I/O (Sheet 1 of 3)

```
; Code
START:
        OPEN$R  #FDB,,,,,,ERR1      ; Open file
        DIR$    #PROMPT             ; Ask for a VBN
        MOV     IOSB+2,R0           ; Put null at end
        CLRB    BUFF(R0)            ;  of digit string
        MOV     #BUFF,R0            ; R0 => VBN
        CALL    $CDTB               ; Convert to binary
        MOV     R1,VBN+2            ; Store as low VBN
        READ$   #FDB,,,#VBN,,,,ERR2 ; Read in the block
        WAIT$   ,,,ERR3             ; Wait until done
        DIR$    #DONE               ; Tell them I/O is done
; Now dump 8 lines of 64. characters each
        MOV     #BLOCK,R0           ; R0 => 1st line to dump
        MOV     #8.,R1              ; # of lines to dump
1$:     MOV     R0,DUMP+Q.IOPL      ; Addr of current line
        DIR$    #DUMP               ; Dump it
        ADD     #64.,R0             ; Point at next line
        SOB     R1,1$               ; Dump all 8. lines
        CLOSE$  #FDB,ERR4           ; Close file
        EXIT$S                      ; Exit
; Error code
ERR1:
        TSTB    F.ERR+1(R0)         ; I/O or directive error?
        BEQ     IOERR1              ; Branch on I/O error
        MOV     #MES3D,R1           ; => Dir error message 3
        BR      FCSERR              ; Branch to common code
IOERR1: MOV     #MES3I,R1           ; => I/O error message 3
        BR      FCSERR              ; Branch to common code
ERR2:
        TSTB    F.ERR+1(R0)         ; I/O or directive error?
        BEQ     IOERR2              ; Branch on I/O error
        MOV     #MES4D,R1           ; => Dir error message 4
        BR      FCSERR              ; Branch to common code
IOERR2: MOV     #MES4I,R1           ; => I/O error message 4
        BR      FCSERR              ; Branch to common code
ERR3:
        TSTB    F.ERR+1(R0)         ; I/O or directive error?
        BEQ     IOERR3              ; Branch on I/O error
        MOV     #MES5D,R1           ; => Dir error message 5
        BR      FCSERR              ; Branch to common code
IOERR3: MOV     #MES5I,R1           ; => I/O error message 5
        BR      FCSERR              ; Branch to common code
ERR4:
        TSTB    F.ERR+1(R0)         ; I/O or directive error?
        BEQ     IOERR4              ; Branch on I/O error
        MOV     #MES6D,R1           ; => Dir error message 6
        BR      FCSERR              ; Branch to common code
IOERR4: MOV     #MES6I,R1           ; => I/O error message 6,
                                    ;  fall into common code
```

Example 10-7 Reading a File With Block I/O (Sheet 2 of 3)

```
FCSERR:
        MOVB    F.ERR(R0),R2        ; Sign extend error code
        MOV     R2,IOSB             ;  and move into IOSB
        MOV     #EX$ERR,R5          ; Exit status in R5
FORMAT:
        MOV     #IOSB,R2            ; Set up for $EDMSG
        MOV     #BUFF,R0            ;
        CALL    $EDMSG              ;
        QIOW$S  #IO.WVB,#5,#1,,,,<#BUFF,R1,#40> ; Display
                                    ;  message
        CLOSE$  #FDB                ; Close the file
        EXST$S  R5                  ; Exit with status
        .END    START
```

```
Run Session

>RUN BLOCK2
VIRTUAL BLOCK: 2

HERE IS THE BLOCK:

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
>
```

Example 10-7 Reading a File With Block I/O (Sheet 3 of 3)

ADDITIONAL TOPICS

Deleting a File

Use the DELET$ macro to delete a file. If the file is open, DELET$ closes the file and then deletes it. If the file is closed, DELET$ just deletes the file. The format of the DELET$ call is:

```
DELET$ fdb,err

DELET$  FDB3
```

NOTE

Unlike the DCL DELETE command, if no version is specified, the latest version of the file is deleted.

File Control Routines

You can use a number of internal FCS routines in your task. Some of the available functions include:

- Filling in an FDB filename block from a dataset descriptor or default filename block.
- Finding, inserting, or deleting a directory entry.
- Marking a place in a file for later return.
- Setting a pointer to a byte within a virtual block, or to the start of a record in a file.
- Renaming, extending or truncating a file.
- Marking a temporary file for deletion, or deleting a file by FDB filename block.

We have already discussed the use of the routines .MARK and .POINT for marking a place in a file so that you can later return to it, and .POSRC for locating a record with random access in locate mode. See Chapter 4 of the IAS/RSX-11 I/O Operations Reference Manual for a description of each of the file control routines, plus information on how to use them.

Command Line Processing

You can use two other collections of routines to facilitate input and processing of command lines, which are useful in general or utility tasks. The routines and their functions are:

- Get Command Line (GCML)
  - Performs command line input operations (issues prompts, gets input)
- Command String Interpreter (CSI)
  - Parses the file specification in a command line from GCML into a dataset descriptor, for use by FCS
  - Parses and processes any switches and switch values in the command line.

See Chapter 6 on Command Line Processing in the IAS/RSX-11 Operations Reference Manual for a description of the command line processing routines. The program CSI.MAC should be available on-line (under UFD [202,1]). It is also listed in Appendix G, and contains an example of the use of GCML and CSI. This example is needed to do optional exercise 6 for this module.

Now do the Tests/Exercises for this module in the Tests and Exercises book. They are all lab problems. Check your answers against the solutions provided, either the on-line files (should be under UFD [202,2]) or the printed copies in the Tests/Exercises book.

If you think that you have mastered the material, ask your course administrator to record your progress on your Personal Progress Plotter. You will then be finished with this course.

If you think that you have not yet mastered the material, return to this module for further study.

APPENDICES

AP

APPENDIX A
SUPPLIED MACROS

The supplied macros are designed for simple invocation. They are intended for use early in the course (before QIOs are taught) to provide easy ways of doing I/O to TI:, and in labs to make writing programs easier for the student. They are also used in some example programs to allow brevity of code and to establish consistency in error checking.

These macros are contained in the macro library PROGMACS.MLB and can be assembled by using the following assembler and task-builder calls:

```
MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]SAMPLE
(in MCR, MAC SAMPLE,SAMPLE=LB:[1,1]PROGMACS/ML,dev:[ufd]SAMPLE)

LINK/MAP SAMPLE,LB:[1,1]PROGSUBS/LIBRARY  ! Needed to include
                                          ! the internal
                                          ! subroutines
(in MCR, TKB SAMPLE,SAMPLE=SAMPLE,LB:[1,1]PROGSUBS/LB)
```

NOTE

If you make copies of PROGMACS.MLB and PROGSUBS.OLB in your UFD (or enter a synonym), then the LB:[1,1] and the dev:[ufd] can be omitted.

This appendix includes directions for using the supplied macros, the MACRO-11 source code for the macros, and any internally-called subroutines.

SIMPLE MESSAGE OUTPUT

Invocations:	TYPE	<message>
	TYPE	<message>,,psect
	TYPE	message-address,message-length

Description: In the first two forms, supply the text of the message; the macro will generate the storage. Use the second form if you are programming in a Psect other than the default; supply the name of the Psect in which you are writing.

In the third form, provide the message address and length using standard addressing modes. The message can be ASCII or ASCIZ. If it is ASCIZ, supply a value of Ø for the message length; if it is ASCII, supply the length in bytes.

Examples:

```
MSG1:   .ASCIZ  /THIS IS MESSAGE #1/
MSG2:   .ASCII  /THIS IS MESSAGE #2/
MSG2LN=.-MSG2
   .
   .
   .
TYPE      #MSG1,#Ø
TYPE      #MSG2,#MSG2LN
TYPE      <THIS IS MESSAGE #3>
```

Outputs: All registers are preserved.

C-bit is set for error;
clear for no error.

Note: Event flag 24(1Ø) is used for synchronization. Avoid using this flag for other purposes in your task.

Task-Building: This macro requires subroutine modules TYPOUT and LENGTH from PROGSUBS.OLB.

SIMPLE MESSAGE INPUT

Invocations:	INPUT buffer,length
Description:	Accepts input data from TI:, into specified buffer. Length is in bytes. Use standard addressing modes for all arguments.
Outputs:	RØ points to the input buffer. R1 contains the byte count from the I/O status block if there is no error. C-bit is set for error; clear for no error. For directive errors, R1 is clear; $DSW contains directive error code. For I/O errors, R1 contains error code from the I/O status block.
Note:	Event flag 24(1Ø) is used for synchronization. Avoid using this flag for other purposes in your task.
Task-Building:	This macro requires subroutine module TYPIN from PROGSUBS.OLB.

ERROR MESSAGE MACROS

Error message macros generate error messages appropriate to Executive directives, I/O operations, and FCS calls.

- All macros have message, length, and Psect arguments whose interpretations are identical to those for the TYPE macro. These are used to specify the user-defined section of the error message.
- The calling program must check for acceptable errors <u>before</u> calling the error message macro, because the error routine aborts the task.
- All macros exit unconditionally with "severe error" status after the error message is printed.
- All macros require the subroutine modules EREXIT, TYPOUT, and LENGTH from PROGSUBS.OLB.

EXECUTIVE DIRECTIVE ERRORS

Invocations:	DIRERR	<message>
	DIRERR	<message>,,psect
	DIRERR	message-address,message-length

Notes: User should check C-bit and dismiss acceptable errors before calling DIRERR.

Format of Message:

DIRECTIVE ERROR
<user-defined message>
DSW = <value>.

I/O ERRORS

Invocations:	IOERR iosb,<message> IOERR iosb,<message>,,psect IOERR iosb,message-address,message-length
Notes:	iosb is a pointer to the I/O status block. User should check the low-order byte of the first word of the I/O status block, and dismiss acceptable errors before calling IOERR.
Format of Message:	I/O ERROR <user-defined message> I/O STATUS BLOCK = <hb>,<lb>/<2nd word> hb is the high byte of the first word. lb is the low byte of the first word.

FCS ERRORS

Invocations:	FCSERR fdb,<message> FCSERR fdb,<message>,,psect FCSERR fdb,message-address,message-length
Notes:	fbd is a pointer to the file descriptor block for the operation which caused the error. User should check the C-bit and/or check F.ERR in the FDB, and dismiss acceptable errors before calling FCSERR.
Format of Message:	FCS ERROR <user-defined message> DSW = <value> or FCS ERROR <user-defined message> I/O ERROR CODE = <value>.

MACRO-11 CODE FOR SUPPLIED MACROS

```
        .MACRO  TYPE    MESSG,LEN,PSCT
        .NLIST
;+
; COPYRIGHT (C) 1981 BY DIGITAL EQUIPMENT CORPORATION
;
; Macro to invoke the "TYPOUT" routine to type a line on
; TI:.
;
; Invoke using one of two forms:
;
;       TYPE    <message>
; or
;       TYPE    address,length
;
; In the first form you specify the text of the message.
; The macro reserves storage for the string.
;
; WARNING: The character   is used as the delimiter in a
; .ASCII directive when you invoke the first form, so
; you may not use this character in your message.
;
; In the second form you must use addressing modes to
; specify the address and length of a string which you
; have reserved in your program. The first argument is
; the address of an ASCII or ASCIZ string.  The second
; argument should have a value of 0 if the string is
; ASCIZ, else should be the length of the ASCII string.
; addressing modes using the stack pointer are not
; allowed.
;
; If you use the first form and are programming in other
; than the blank Psect, you must explicitly provide a
; null "LEN" argument, and specify a third argument
; (psect). This argument must be the name of the Psect
; in which you are programming.
;
; Needed subroutine modules: TYPOUT and LENGTH
;
; symbols SAV.Rn         = -1 if Rn is not saved on the
;                             stack
;                        >= 0 indicates Rn is stored on
;                             the stack at SAV.Rn(SP).
;-
        .LIST
        .GLOBL  TYPOUT            ; Subroutine to issue QIO
                                  ;  directive
        SAV.R0 = -1               ; Assume no need to save
                                  ;  R0
        SAV.R1 = -1               ; Assume no need to save
                                  ;   R1
        SAV.R2 = -1               ; Assume no need to save
                                  ;  R2
```

```
.IF B   LEN                             ; Blank LEN arg means
                                        ;  first form
        .PSECT  MSGTXT                  ; Set up text in Psect
        $$$MES=.                        ;  MSGTXT
        .ASCII  `MESSG`
        $$$LEN=.-$$$MES
        .PSECT  PSCT                    ; Back to original Psect
        MOV     R0,-(SP)                ; Save old R0
        MOV     #$$$MES,R0              ; R0 => message
        MOV     R1,-(SP)                ; Save old R1
        MOV     #$$$LEN,R1              ; R1 = message length
        SAV.R0 = 2                      ; Note R0 saved on stack
        SAV.R1 = 0                      ; Note R1 saved on stack

.IFF                                    ; Second form of
                                        ;  invocation
; If arguments are not already in the correct registers,
; save them temporarily on the stack

        .NTYPE  ADM.A1,MESSG            ; Addressing mode of
                                        ;  MESSG
        .IF     NE,ADM.A1               ; If anything but "R0"
        MOV     MESSG,-(SP)             ; Save argument on the
                                        ;  stack
        SAV.R0 = 0                      ; R0 will be saved here
                                        ;  later
        SAV.R2 = 0                      ; We'll need to save R2
        .ENDC

        .NTYPE  ADM.A2,LEN              ; Addressing mode of LEN
        .IF     NE,ADM.A2-1             ; If anything but "R1"
        MOV     LEN,-(SP)               ; Save argument on the
                                        ;  stack
        SAV.R1 = 0                      ; R1 will be saved here
                                        ;  later
        .IIF GE,SAV.R0, SAV.R0=SAV.R0+2 ; Increase
                                        ;  offset of R0
        SAV.R2 = 0                      ; We'll need to save R2
        .ENDC

; Swap the registers with their argument values which we
; stored on the stack.
        .IF     EQ,SAV.R2               ; If we need to save R2
        MOV     R2,-(SP)                ; Save R2
        .IIF GE,SAV.R0, SAV.R0=SAV.R0+2 ; Increase
                                        ;  offset of R0
        .IIF GE,SAV.R1, SAV.R1=SAV.R1+2 ; Increase
                                        ;  offset of R1
        .ENDC

```

```
        .IF     GE,SAV.R0           ; If R0's arg was put on
                                    ;  stack,
        MOV     R0,R2               ;  swap R0 with its
        MOV     SAV.R0(SP),R0       ;  argument value
        MOV     R2,SAV.R0(SP)
        .ENDC

        .IF     GE,SAV.R1           ; If R1's arg was put on
                                    ;  stack,
        MOV     R1,R2               ; Swap R1 with its
        MOV     SAV.R1(SP),R1       ;  argument value
        MOV     R2,SAV.R1(SP)
        .ENDC

.ENDC                               ; Forms of invocation

        CALL    TYPOUT
; Restore registers
        .IIF GE,SAV.R2, MOV         (SP)+,R2 ; Restore old R2
        .IIF GE,SAV.R1, MOV         (SP)+,R1 ; Restore old R1
        .IIF GE,SAV.R0, MOV         (SP)+,R0 ; Restore old R0
        .ENDM   TYPE
```

```
        .MACRO  INPUT   BUFFER,LEN
;
; COPYRIGHT (C) 1981 BY DIGITAL EQUIPMENT CORPORATION
;
; Macro to invoke the "TYPIN" routine to input data from
; TI:.
;
; Invoke using:
;
;       INPUT   address,length
;
;       where address and length are the address and
;       length of the input buffer.
;
; OUTPUTS:      Data is input synchronously from LUN 5
;
;               R0 => buffer
;
;               C-bit is set for error, clear for no
;               error (for directive or I/O errors)
;
;               If no error, R1 contains byte count from
;               I/O status block.
;
;               If a directive error is encountered, R1
;               is clear, $DSW contains error code
;
;               If an I/O error is encountered, R1
;               contains error code from I/O status block
;
; Needed subroutine module: TYPIN
;
; WARNINGS:     1. ROUTINE USES EVENT FLAG #24 FOR
;               SYNCHRONIZATION
;               2. R0 AND R1 ARE DESTROYED
;
        .GLOBL  TYPIN   ; Subroutine to issue QIO
                        ; directive
        .NTYPE  ADRMOD,BUFFER ; Check addressing mode
.IF     NE,ADRMOD       ; Buffer pointer already in R0?
        MOV     BUFFER,R0 ; No, move it there
.ENDC
        .NTYPE  ADRMOD,LEN ; Check addressing mode
.IF     NE,ADRMOD-1     ; Length already in R1?
        MOV     LEN,R1  ; No, move it there
.ENDC
        CALL    TYPIN   ; Call subroutine to issue QIO
                        ;  directive
        .ENDM
```

```
        .TITLE  TYPOUT  Subroutine to output to TI:
;
; COPYRIGHT (C) 1981 BY DIGITAL EQUIPMENT CORPORATION
;
; TYPOUT provides a simple way for MACRO-11 routines to
; type out a message on TI:.
;
; CALL:           JSR     PC,TYPOUT
;
; INPUTS:         R0  =>  ASCII or ASCIZ message
;                 R1  =   0 if string is ASCIZ
;                     =   n>0 for ASCII string of length n
;
;                 LUN 5 is assumed to be assigned to TI:
;
; OUTPUTS:        Message is output synchronously to LUN 5
;
;                 All registers preserved
;
;                 C-bit is set for error, clear for no
;                 error
;
; WARNING:        Routine uses event flag #24 for
;                 synchronization
;
; The macro "TYPE" can be used to invoke this routine
; in a fairly transparent manner.
;
        .GLOBL  LENGTH          ; Subroutine to determine
                                ;  length of string
        .MCALL  QIOW$S          ; System macro
;
TYPOUT:: TST    R1              ; ASCII input or ASCIZ?
        BNE     1$              ; Branch if ASCII. R1
                                ;  already has length
        CALL    LENGTH          ; Find length of string
                                ;  (returned in R1)
1$:     MOV     R2,-(SP)        ; Save R2 on stack
        SUB     #4,SP           ; Reserve space for IOSB
        MOV     SP,R2           ; R2=>IOSB
; Do a garden variety output to TI:
        QIOW$S  #IO.WVB,#5,#24.,,,R2,,<R0,R1,#40>
        BCC     2$              ; Branch on directive OK
;
        ADD     #4,SP           ; Directive error. Purge
                                ;  IOSB from stack
```

```
        BR      6$              ; Exit (with C-bit set)
;
; Directive succeeded. Record any I/O errors, record
; byte count, purge stack
;
2$:     CMPB    #IS.SUC,(SP)+   ; I/O error?
        BEQ     4$              ; Branch if no error
3$:     SEC                     ; Set C-bit to indicate
                                ;  error
        BR      5$              ; Branch to set I/O count
4$:     CLC                     ; Clear C-bit to indicate
                                ;  no error
5$:     TST     (SP)+           ; Clean up stack
;
; COMMON EXIT
;
6$:     MOV     (SP)+,R2        ; Restore R2
        RETURN                  ; Return
        .END
```

```
        .TITLE  TYPIN   Subroutine to input from TI:
;
; COPYRIGHT (C) 1981 BY DIGITAL EQUIPMENT CORPORATION
;
; TYPIN provides a simple way for MACRO-11 routines to
; input data from TI:.
;
; CALL:           JSR     PC,TYPIN
;
; INPUTS:         R0  =>  Input buffer
;                 R1  =   Length of buffer in bytes
;
;                 LUN 5 is assumed to be assigned to TI:
;
; OUTPUTS:        Data is input synchronously from LUN 5
;
;                 R0 is unchanged
;
;                 C-bit is set for error, clear for no
;                 error (for directive or I/O errors)
;
;                 If no error, R1 contains byte count from
;                 I/O status block.
;
;                 If a directive error is encountered, R1
;                 is clear, $DSW contains error code
;
;                 If an I/O error is encountered, R1
;                 contains error code from I/O status
;                 block
;
; WARNING:        ROUTINE USES EVENT FLAG #24 FOR
;                 SYNCHRONIZATION
;
; The macro "INPUT" can be used to invoke this routine
; in a fairly transparent manner.
;
        .MCALL  QIOW$S              ; System macro
;
TYPIN:: MOV     R2,-(SP)            ; Save R2
        SUB     #4,SP               ; Reserve space for IOSB
        MOV     SP,R2               ; R2=>IOSB
; Do a garden variety input from TI:
        QIOW$S  #IO.RVB,#5,#24.,,R2,,<R0,R1>
        BCC     2$                  ; Branch on directive OK
        ADD     #4,SP               ; Directive error. Purge
                                    ;  IOSB from stack
        CLR     R1                  ; Note directive error
        SEC                         ;
        BR      6$                  ; exit
;
```

```
; Directive succeeded. Check for I/O errors, record
; byte count, purge stack
;
2$:     MOV     (SP)+,R1        ; Get success/error code
        CMPB    #IS.SUC,R1      ; I/O error?
        BEQ     4$              ; Brnch if no error
        TST     (SP)+           ; Error. Forget about
                                ;  I/O count
        SEC                     ; Just return with C-bit
                                ;  set and I/O status in
                                ;  R1
        BR      6$              ; Branch to common exit
4$:     CLC                     ; Clear carry bit
        MOV     (SP)+,R1        ; Return I/O count in R1
;
; COMMON EXIT
;
6$:     MOV     (SP)+,R2        ; Restore R2
        RETURN                  ; Return
        .END
```

```
        .TITLE  LENGTH
;
; COPYRIGHT (C) 1981 BY DIGITAL EQUIPMENT CORPORATION
;
; LENGTH finds the length of an ASCIZ string
;
; CALL:         CALL    LENGTH
;
; INPUT:        R0  =>  ASCIZ string
;
; OUTPUT:       R0 preserved
;               R1  =   Length of string (not including
;                       terminating null)
;
LENGTH:: CLR    R1              ; Clear counter
        MOV     R0,-(SP)        ; Save pointer to
                                ;  beginning of string
1$:     TSTB    (R0)+           ; Real character or null
                                ;  byte?
        BEQ     2$              ; Null means end of
                                ;  string
        INC     R1              ; Count real character
        BR      1$              ; And look at next byte
2$:     MOV     (SP)+,R0        ; Restore original
                                ;  pointer
        RETURN                  ; Return
        .END
```

```
        .MACRO  DIRERR  MESSAG,LEN,PSCT
;
; COPYRIGHT (C) 1981 BY DIGITAL EQUIPMENT CORPORATION
;
; Macro to generate a "directive error" message and
; print out the value of the DSW, plus a user-defined
; message. The task is forced to exit after the message
; is printed.
;
; The form of the resultant error message is:
;
;       DIRECTIVE ERROR
;       <USER-DEFINED MESSAGE>
;       DSW = <VALUE>
;
; It is suggested that the user-defined message identify
; the operation which returned the error.
;
; It is the caller's responsibility to check the c-bit
; prior to invoking DIRERR.  This convention allows the
; user to accept certain types of errors, then invoke
; DIRERR for any other kinds of errors.
;
; Invoke using one of two forms:
;
;       DIRERR  <message>
; or
;       DIRERR  ADDRESS,LENGTH
;
; In the first form you specify the text of the message.
; The macro reserves storage for the string.
;
; In the second form you must use addressing modes to
; specify the address and length of a string which you
; have reserved in your program. The first argument is
; the address of an ASCII or ASCIZ string.  The second
; argument should have a value of 0 if the string is
; ASCIZ, else should be the length of the ASCII string.
;
; If you use the first form and are programming in other
; than the blank Psect, you must explicitly provide a
; null "len" argument, and supply as the third argument
; the name of the Psect to return to.
;
```

```
        .GLOBL  EREXIT          ; Common routine
        .GLOBL  DIRERI          ; "DIRECTIVE ERROR"
                                ;  input string for
                                ;  $EDMSG
        .GLOBL  ERARGS          ; $EDMSG argument block
; Offsets into argument block:
        .GLOBL  E.RUMA          ; User-message address
        .GLOBL  E.RUML          ; User-message length
        .GLOBL  E.RDSW          ; DSW value
;
        MOV     #ERARGS,R2      ; R2=>$EDMSG arg block
.IF B   LEN                     ; Blank len arg means
                                ;  first form
        .PSECT  MSGTXT
$$$MES=.
        .ASCII  /MESSAG/
$$$LEN=.-$$$MES
        .PSECT  PSCT
        MOV     #$$$MES,E.RUMA(R2)  ; Load message addr
        MOV     #$$$LEN,E.RUML(R2)  ;  and message length
                                    ;  into arg block
.IFF
        MOV     MESSAG,E.RUMA(R2)   ; Load message addr
        MOV     LEN,E.RUML(R2)      ;  and message length
                                    ;  into arg block
.ENDC
        MOV     $DSW,E.RDSW(R2) ; Load DSW into arg block
        MOV     #DIRERI,R3      ; R3=>$EDMSG input string
        JMP     EREXIT          ; Jump to common error
                                ;  exit routine
        .ENDM
```

```
        .MACRO  IOERR   IOSB,MESSG,LEN,PSCT
;+
; COPYRIGHT (C) 1981 BY DIGITAL EQUIPMENT CORPORATION
;
; Macro to generate an "I/O error" message and print out
; the value of the I/O status block, plus a user-defined
; message.  The task is forced to exit after the message
; is printed.
;
; The form of the resultant error message is:
;
;       I/O ERROR
;       <user-defined message>
;       I/O STATUS BLOCK = <hb>,<lb>/<2nd word>
;
; where "hb" and "lb" are the high byte and low byte of
; the first word of the I/O status block.
;
; It is suggested that the user-defined message identify
; the operation which returned the error.
;
; It is the caller's responsibility to check the first
; word of the I/O status block prior to invoking IOERR,
; to see whether the operation has been a success or a
; failure. This convention allows the user to accept
; certain types of errors, then invoke IOERR for any
; other kinds of errors.
;
; Invoke using one of two forms:
;
;       IOERR   iosb,<message>
; or
;       IOERR   iosb,address,length
;
; In either form "iosb" is the address of the I/O status
; block, in any addressing mode.
;
; In the first form you specify the text of the message.
; The macro reserves storage for the string.
;
; In the second form you must use addressing modes to
; specify the address and length of a string which you
; have reserved in your program. The second argument is
; the address of an ASCII or ASCIZ string.  The third
; argument should have a value of 0 if the string is
; ASCIZ, else should be the length of the ASCII string.
;
; If you use the first form and are programming in other
; than the blank Psect, you must explicitly provide a
; null "LEN" argument, and supply as the fourth argument
; the name of the Psect to return to.
;-
```

```
        .GLOBL  EREXIT              ; Common routine
        .GLOBL  IOERIN              ; "I/O ERROR" input
                                    ;  string for $EDMSG
        .GLOBL  ERARGS              ; $EDMSG argument block
; Offsets into argument block:
        .GLOBL  E.RUMA              ; User-message address
        .GLOBL  E.RUML              ; User-message length
        .GLOBL  E.RIOS              ; First word of I/O
                                    ;  status block
;
        MOV     #ERARGS,R2          ; R2=>$EDMSG arg block
.IF B   LEN                         ; Blank LEN arg means
                                    ;  first form
        .PSECT  MSGTXT
$$$MES=.
        .ASCII  /MESSG/
$$$LEN=.-$$$MES
        .PSECT  PSCT
        MOV     #$$$MES,E.RUMA(R2)    ; Load message addr
        MOV     #$$$LEN,E.RUML(R2)    ; and message length
;                                       into arg block
.IFF
        MOV     MESSG,E.RUMA(R2)      ; Load message addr
        MOV     LEN,E.RUML(R2)        ; and message length
;                                     ;  into arg block
.ENDC
; Copy I/O status block into $EDMSG arg block
        MOV     IOSB,R1             ; R1 => I/O status block
        MOVB    1(R1),R3            ; Get hi byte of first
;                                   ;  word (and sign-extend
;                                   ;  it)
        MOV     R3,E.RIOS(R2)       ; Copy into arg block
        MOVB    (R1),R3             ; Get lo byte and
                                    ;  sign-extend
        MOV     R3,E.RIOS+2(R2)     ; Copy into arg block
        MOV     2(R1),E.RIOS+4(R2)    ; Copy 2nd word of
                                      ;  IOSB
        MOV     #IOERIN,R3          ; R3 => $EDMSG input
                                    ;  string
        JMP     EREXIT              ; Jump to common error
                                    ;  exit routine
        .ENDM
```

```
        .MACRO  FCSERR  FDB,MESSG,LEN,PSCT
;
; COPYRIGHT (C) 1981 BY DIGITAL EQUIPMENT CORPORATION
;
; Macro to generate an "FCS ERROR" message and print out
; the error code plus a user-defined message.  The task
; is forced to exit after the message is printed.
;
; The form of the resultant error message is:
;
;       FCS ERROR
;       <USER-DEFINED MESSAGE>
;       DSW = <VALUE>
;
;       or
;
;       FCS ERROR
;       <USER-DEFINED MESSAGE>
;       I/O ERROR CODE = <VALUE>
;
; It is suggested that the user-defined message identify
; the operation which returned the error.
;
; It is the caller's responsibility to check F.ERR in
; the FDB prior to invoking FCSERR, to see whether the
; operation has been a success or a failure. This
; convention allows the user to accept certain types of
; errors, then invoke FCSERR for any other kinds of
; errors.
;
; Invoke using one of two forms:
;
;       FCSERR  fdb,<message>
; or
;       FCSERR  fdb,address,length
;
; In either form, "fdb" is the address of the file
; descriptor block for the FCS operation which has
; generated the error.
;
; In the first form you specify the text of the message.
; The macro reserves storage for the string.
;
; In the second form you must use addressing modes to
; specify the address and length of a string which you
; have reserved in your program. The second argument is
; the address of an ASCII or ASCIZ string.  The third
; argument should have a value of 0 if the string is
; ASCIZ, else should be the length of the ASCII string.
;
; If you use the first form and are programming in other
; than the blank Psect, you must explicitly provide a
; null "len" argument, and supply as the fourth argument
; the name of the Psect to return to.
;
```

```
        .GLOBL  EREXIT              ; Common routine
        .GLOBL  FCSDIR,FCSIO        ; $EDMSG input strings
        .GLOBL  ERARGS              ; $EDMSG argument block
; Offsets into argument block:
        .GLOBL  E.RUMA              ; User-message address
        .GLOBL  E.RUML              ; User-message length
        .GLOBL  E.RCOD              ; Error code (DSW value
;                                   ; or I/O error)
;
        .MCALL  FDOF$L
        FDOF$L                      ; Define FDB offsets
;
        MOV     #ERARGS,R2          ; R2=>$EDMSG arg block
.IF B   LEN                         ; Blank len arg means
                                    ;  first form
        .PSECT  MSGTXT
$$$MES=.
        .ASCII  /MESSG/
$$$LEN=.-$$$MES
        .PSECT  PSCT
        MOV     #$$$MES,E.RUMA(R2)    ; Load message addr
        MOV     #$$$LEN,E.RUML(R2)    ;  and message length
                                      ;  into arg block
.IFF
        MOV     MESSG,E.RUMA(R2)      ; Load message addr
        MOV     LEN,E.RUML(R2)        ;  and message length
                                      ;  into arg block
.ENDC
        MOV     FDB,R1              ; R1=> file descriptor
                                    ;  block
        MOVB    F.ERR(R1),R0        ; Get error code
                                    ;  (sign-extend) and
        MOV     R0,E.RCOD(R2)       ;  store into arg block
        TSTB    F.ERR+1(R1)         ; Directive error or I/O
                                    ;  error?
        BEQ     IO                  ; Branch on I/O error
; Directive error:
        MOV     #FCSDIR,R3          ; R3=> "FCS DIRECTIVE
                                    ; ERROR" $EDMSG string
        JMP     EREXIT              ; Jump to common error
                                    ;  exit routine
IO:     MOV     #FCSIO,R3           ; R3=>"FCS I/O ERROR"
                                    ; $EDMSG string
        JMP     EREXIT              ; Jump to common error
                                    ;  exit routine
        .ENDM
```

```
        .TITLE  EREXIT  ERROR EXIT ROUTINE
;+
; COPYRIGHT (C) 1981 BY DIGITAL EQUIPMENT CORPORATION
;
; This is a common exit routine called by the
; error-processing macros DIRERR, IOERR, and FCSERR.  It
; types out an error message and forces the task to exit
; with status "severe error".
;
; Call:         JMP     EREXIT
;
; Inputs:       R2  =>  ERARGS ($EDMSG argument block,
;                               defined in this routine)
;
;                       The argument block has already
;                       been filled in with the
;                       user-message descriptor, and the
;                       system error code (DSW value or
;                       IOSB pointer).  A user-message
;                       length  = 0 means that the
;                       user message is in ASCIZ form.
;
;               R3  =>  One of the $EDMSG input strings
;                       defined in this routine
;
        .GLOBL  $EDMSG
        .GLOBL  LENGTH          ; Computes length of
                                ;  ASCIZ string
        .MCALL  EXST$C          ; System macro
        .MCALL  TYPE            ; Supplied macro
;
        EX$SEV = 4              ; Error exit status
;
; $EDMSG input strings:
;
DIRERI:: .ASCII /%NDIRECTIVE ERROR/
        .ASCII  /%N%VA/
        .ASCIZ  /%NDSW = %D/
;
FCSDIR:: .ASCII /%NFCS ERROR/
        .ASCII  /%N%VA/
        .ASCIZ  /%NDSW = %D/
;
IOERIN:: .ASCII @%NI/O ERROR@
        .ASCII  /%N%VA/
        .ASCIZ  @%NI/O STATUS BLOCK = %D, %D / %D@
;
FCSIO:: .ASCII  @%NFCS ERROR@
        .ASCII  /%N%VA/
        .ASCIZ  @%NI/O ERROR CODE = %D@
;
;
```

```
OUTBUF: .BLKB   200.            ; $EDMSG output buffer
        .EVEN
; $EDMSG argument block
ERARGS::
E.RUML==.-ERARGS
        .WORD                   ; User-message length
E.RUMA==.-ERARGS
        .WORD                   ; User-message address
; Error codes
E.RDSW==.-ERARGS                ; DSW for DIRERR
E.RIOS==.-ERARGS                ; IOSB for IOERR
E.RCOD==.-ERARGS                ; FCS code for FCSERR
        .WORD
        .WORD
        .WORD
;
EREXIT::
        TST     E.RUML(R2)      ; User message ASCII or
                                ;  ASCIZ?
        BNE     1$              ; ASCII
                                ; If ASCIZ, find length
        MOV     E.RUMA(R2),R0   ; R0 => user message
        CALL    LENGTH          ; (returned in R1)
        MOV     R1,E.RUML(R2)   ; Set length field in
;                               ;  argument block
1$:     MOV     #OUTBUF,R0      ; Output buffer for
                                ;  $EDMSG
        MOV     R3,R1           ; Put input string
                                ;  pointer into proper
                                ;  register for $EDMSG
        CALL    $EDMSG          ; (returns length in R1)
        TYPE    #OUTBUF,R1      ; Type out formatted
                                ;  message
        EXST$C  EX$SEV          ; Exit/severe error
        .END
```

APPENDIX B
CONVERSION TABLES

Table B-1 Decimal/Octal, Word/Byte/Block Conversions

Words(1Ø)/Words(8)	Bytes(1Ø)/Bytes(8)	Blocks(1Ø)/Blocks(8)
1/1	2/2	
32/4Ø	64/1ØØ	1/1
1K =1Ø24/2ØØØ	2Ø48/4ØØØ	32/4Ø
2K =2Ø48/4ØØØ	4Ø96/1ØØØØ	64/1ØØ
4K =4Ø96/1ØØØØ	8192/2ØØØØ	128/2ØØ
8K =8192/2ØØØØ	16384/4ØØØØ	256/4ØØ
16K =16384/4ØØØØ	32768/1ØØØØØ	512/1ØØØ
32K =32768/1ØØØØØ	65536/2ØØØØØ	1Ø24/2ØØØ
64K =65536/2ØØØØØ	131Ø72/4ØØØØØ	2Ø48/4ØØØ
128K=131Ø72/4ØØØØØ	262144/1ØØØØØØ	4Ø96/1ØØØØ

Table B-2 APR/Virtual Addresses/Words Conversions

APR	Virtual Addresses	Words
Ø	ØØØØØØ-Ø17776	Ø-4K
1	Ø2ØØØØ-Ø37776	4-8K
2	Ø4ØØØØ-Ø57776	8-12k
3	Ø6ØØØØ-Ø77776	12-16K
4	1ØØØØØ-117776	16-2ØK
5	12ØØØØ-137776	2Ø-24K
6	14ØØØØ-157776	24-28K
7	16ØØØØ-177776	28-32K

APPENDIX C
FORTRAN/MACRO-11 INTERFACE

CALLING A MACRO-11 SUBROUTINE FROM A FORTRAN PROGRAM

FORTRAN Program Call:

```
CALL SUBNAM (I,J,K)
```

MACRO translation:

1. Set up table of arguments.

```
R5 ---->  +----------+---------+
          |          | Count=3 |
          +----------+---------+
          |  Address of I      |
          +--------------------+
          |  Address of J      |
          +--------------------+
          |  Address of K      |
          +--------------------+
```

2. Issue subroutine call.

```
JSR PC,SUBNAM

     or

CALL SUBNAM
```

The FORTRAN Callable MACRO-11 Subroutine

```
; Accessing:
;       Argument count = (R5)
;       Arg1 = @2(R5)
;       Arg2 = @4(R5)
;       Arg3 = @6(R5)
SUBNAM::  .
          .
          .
        RTS PC  ; or RETURN
```

CALLING A FORTRAN PROGRAM FROM A MACRO-11 PROGRAM

In the MACRO program:

```
LINK:   .BYTE   3,Ø
        .WORD   A
        .WORD   B
        .WORD   C
A:      .WORD   2
B:      .WORD   3
C:      .WORD   Ø
          .
          .
          .
        MOV     #LINK,R5
        JSR     PC,SUB
          .
          .
          .
```

In the FORTRAN program:

```
        SUBROUTINE SUB (L,M,N)
        N=L+M
        RETURN
        END
```

NOTE

This method is also used to call a FORTRAN callable subroutine (written in MACRO-11).

Example 7-3 in the Static Regions module shows a shareable library LIB.MAC, which contains FORTRAN callable subroutines. USELIB.MAC, also in Example 7-3, shows a referencing task which calls subroutines in the library.

APPENDIX D
PRIVILEGED TASKS

RSX-11M systems have two classes of tasks, privileged and nonprivileged. The basic difference is that privileged tasks have certain system-access capabilities that nonprivileged tasks do not have. These privileges include one or more of the following:

- Access to Executive routines and data structures
- Automatic mapping to the I/O page
- Bypass of system security features.

NOTE

Privileged tasks may be hazardous to a running system.

Use one of the following qualifiers (switches) to build a privileged task.

1. /PRIVILEGE:Ø qualifier (MCR /PR:Ø)

   This task is built in the same way as a nonprivileged task and does not map to the Executive or the I/O page. It can, however, do the following:

   - Bypass file protection
   - Issue directives which require privileges (e.g., Alter Priority, QIO for Write Logical Break-through)
   - Issue QIOs to write logical blocks to a mounted volume, regardless of who issued the MOUNT or ALLOCATE command.

2. /PRIVILEGE:4 or /PRIVILEGE:5 (MCR /PR:4 or /PR:5)

   This task has the privileges of a /PRIVILEGE:Ø task, plus it maps to the Executive and the I/O page. The user task code is mapped beginning at APR 4 or 5, as specified. The APRs below the one specified are used to map to the Executive, and APR 7 is used to map the I/O page. Use /PRIVILEGE:4 if the Executive is 16K words or less; use /PRIVILEGE:5 if the Executive is between 16K and 2ØK words. If the task code extends beyond the end of the addresses mapped by APR 6, then APR 7 is used to map the excess code, and the task does not map to the I/O page.

Privileged tasks are discussed in detail in the RSX-11M Internals Course. See also Chapter 6 on Privileged Tasks in the RSX-11M/M-PLUS Task Builder Manual.

APPENDIX E
TASK BUILDER USE OF PSECT ATTRIBUTES

The Task Builder collects scattered occurrences of program sections of the same name and combines them in a single area in your task image. The program section attributes control how the Task Builder collects and places each program section.

See Chapter 2 of the RSX-11M/M-PLUS Task Builder Manual for a complete discussion of program section attributes.

Example of allocation code attributes:

CON (concatenate) versus OVR (overlay)

1. A.OBJ has Psect Q,CON - length 1ØØ(1Ø) words

   B.OBJ has Psect Q,CON - length 5Ø(1Ø) words

   When task-built:

   LINK A,B

   Yields 15Ø(1Ø) words in Psect Q
   (first A's 1ØØ(1Ø) words, then B's 5Ø(1Ø) words).

2. A.OBJ has Psect Q,OVR - length 1ØØ(1Ø) words

   B.OBJ has Psect Q,OVR - length 5Ø(1Ø) words

   When task-built:

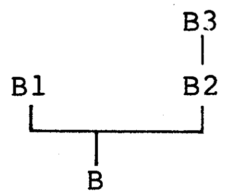
   LINK A,B

   Yields 1ØØ(1Ø) words in Psect Q
   (A's 1ØØ(1Ø) words. B's 5Ø(1Ø) words are the same as A's first 5Ø(1Ø) words).

Example of scope code attributes:

LCL (local) versus GBL (global)

Overlay Tree



B.ODL file:

```
.ROOT B-*!(B1,B2-B3)
.END
```

Task-build command (for all): LINK B/OVERLAY__DESCRIPTION

1. B.OBJ has Psect Q,LCL,CON - length 100(10) words

   B1.OBJ has Psect Q,LCL,CON - length 50(10) words

   When task-built:

   Yields 100(10) words in Psect Q in root segment B
   Yields 50(10) words in Psect Q in overlay segment B1

2. B.OBJ has Psect Q,GBL,CON - length 100(10) words

   B1.OBJ has Psect Q,GBL,CON - length 50(10) words

   When task-built:

   yields 150(10) words in Psect Q in root segment B (in the segment closest to the root); B's 100(10) words, then B1's 50(10) words.

   If GBL,OVR instead, yields 100(10) words in Psect Q in the root segment. B's 100 words, with B1's 50(10) words the same as B's first 50(10) words.

3. B2.OBJ has Psect Q (LCL or GBL) - length 1ØØ(1Ø) words

B3.OBJ has Psect Q (LCL or GBL) - length 5Ø(1Ø) words

When task-built:

If CON, yields 15Ø(1Ø) words in Psect Q in overlay segment B2 (allocation collected, since it is all in the same overlay segment).

If OVR instead, 1ØØ(1Ø) words in Psect Q in overlay segment B2. B3's 5Ø(1Ø) words are the same as B2's first 5Ø(1Ø) words.

LCL and GBL are used only for overlaid tasks. In a non-overlaid task or within an overlay segment in an overlaid task, allocations are collected when either LCL or GBL is specified, as in Example 3.

Example of FORTRAN COMMONs at Psects:

Psect attributes are always: RW,D,GBL,OVR,REL

```
COMMON /RDATA/ I(1ØØ)
```

Macro translation:

```
.PSECT RDATA,RW,D,GBL,OVR,REL
```

APPENDIX F
ADDITIONAL SHARED REGION TOPICS

SHARED REGIONS WITH OVERLAYS

- Can be referenced using a smaller window in referencing task
- Reuse virtual addresses in the referencing task
- Must be memory-resident overlays
- Have overlay structures which are placed in the .STB file and later placed in root segment of referencing task.

BUILDING A RESIDENT LIBRARY WITH OVERLAYS

1. Code and assemble library modules.
2. Write regular .ODL file to define overlay structure.
   - Typical structure has a null root.
3. Task-build as a shared region.
   - Only symbols defined or referenced in the root are included in the .STB file.
   - Force inclusion of global references into root, when necessary, using GLBREF option.

Example .ODL file OVRLIB.ODL (Figure F-1):

```
.NAME   OVRLIB
.ROOT   OVRLIB-*!(H,I-J)
.END
```

Example task-build command:

```
>LINK/NOHEADER/MAP/SYMBOL_TABLE/OPTIONS OVRLIB/OVERLAY-
->_DESCRIPTION
Option? STACK=0
Option? PAR=OVRLIB:140000:40000
Option? GBLREF=H,I,J
Option? <RET>
```

Referencing task is created using regular procedure to reference library OVRLIB.

See section 5.1.4 (on Shared Regions with Memory-Resident Overlays) in the RSX-11M/M PLUS Task Builder Manual for additional information.

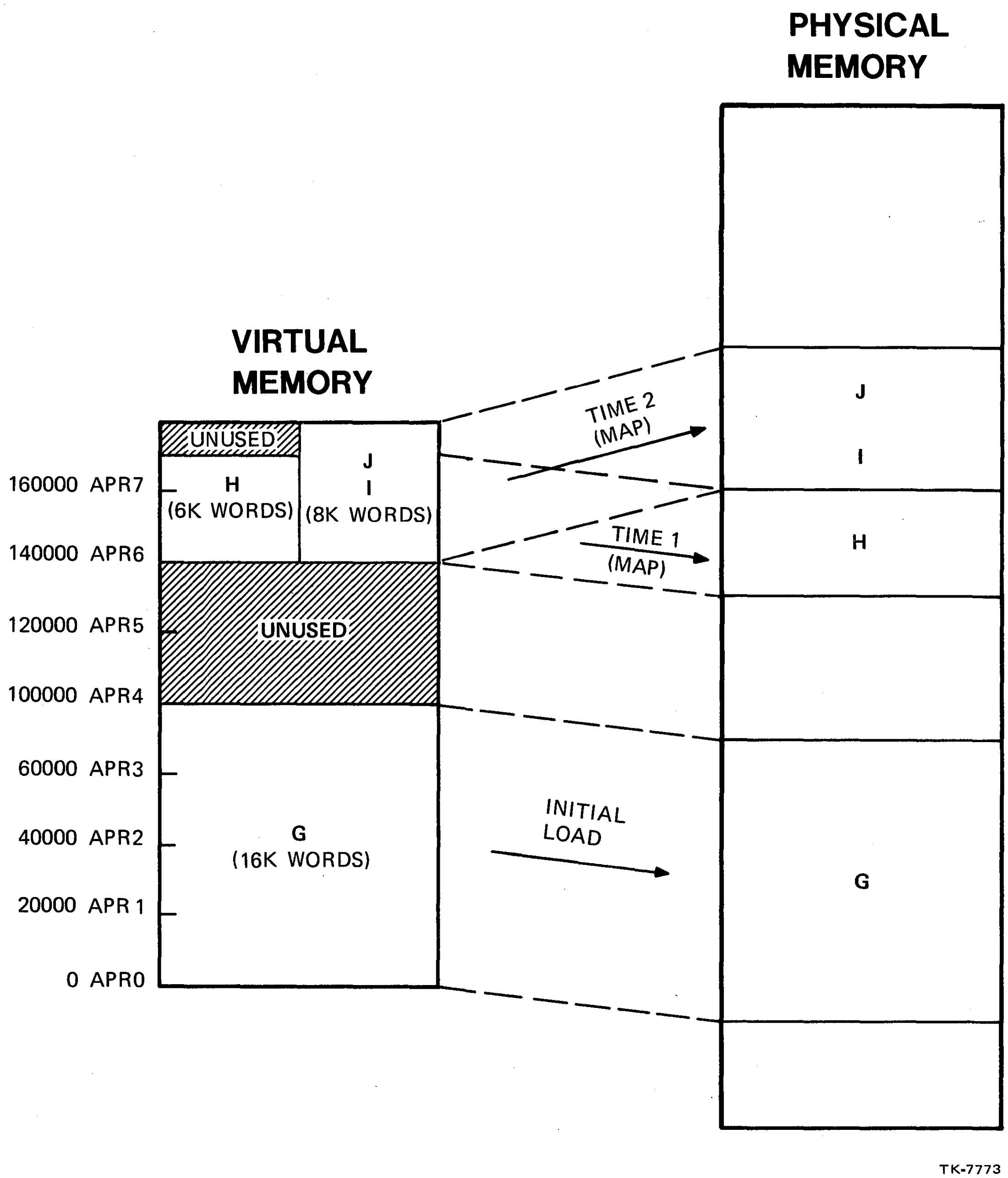


Figure F-1 A Shared Region With Memory-Resident Overlays

REFERENCING MULTIPLE REGIONS IN A TASK

- Use the usual procedure if:

  - The number of available APRs in the referencing task is sufficient

  - Shared regions are logically independent (one library does not call the other library)

- If shared regions are built absolute, APRs (and virtual addresses) cannot overlap.

Example task-build for logically independent libraries (Figure F-2):

```
Libraries: ARES built absolute at V.A. 160000(8); length 4K
                words
           BRES built absolute at V.A. 120000(8); length 6K
                words

Referencing task: REF

>LINK/MAP/OPTIONS REF
Option? RESLIB=ARES/RO
Option? RESLIB=BRES/RO
Option? <RET>
```

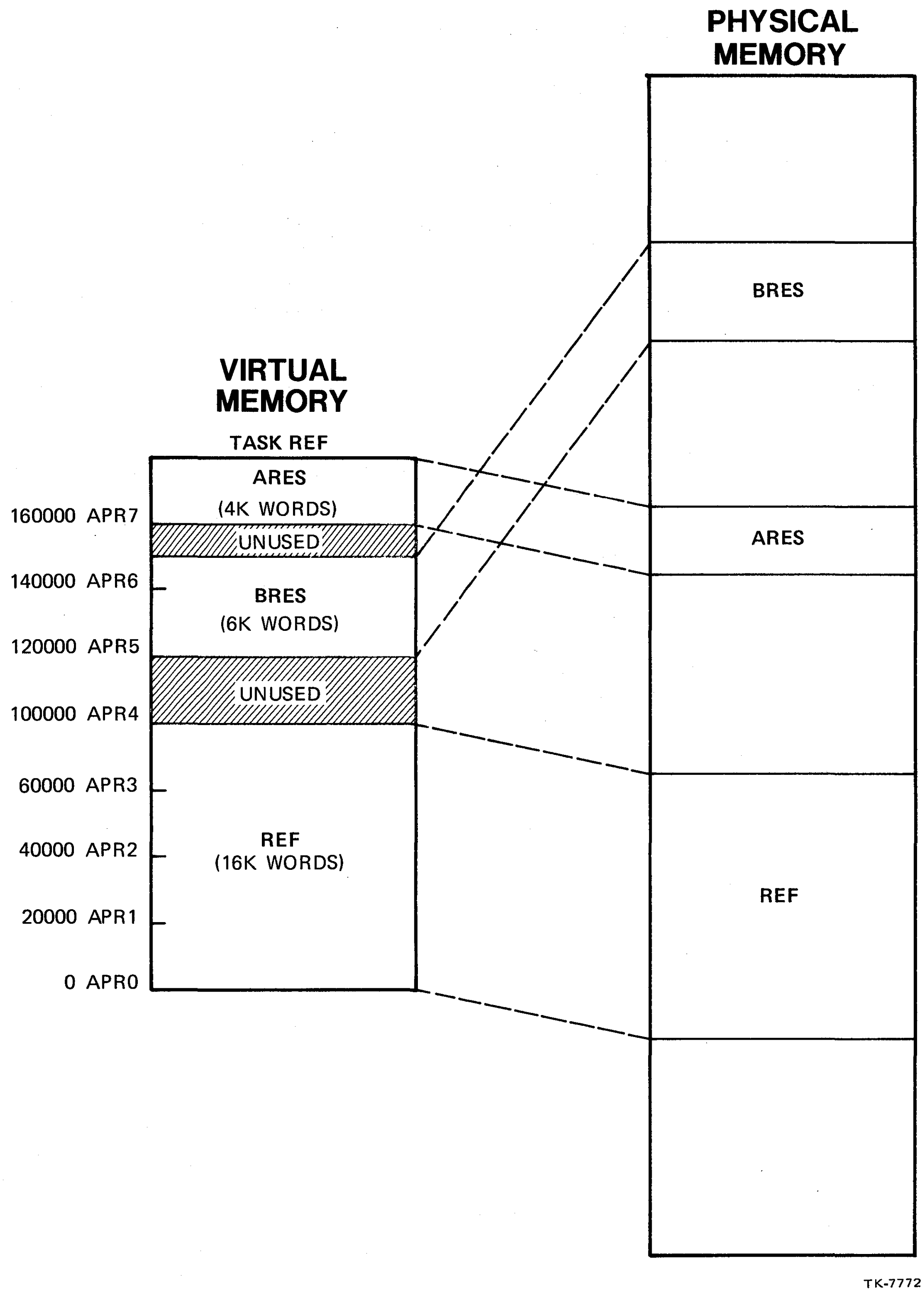


Figure F-2 Referencing Two Resident Libraries

INTERLIBRARY CALLS

One library can call another library

FORRES calls FCSRES

To build libraries with interlibrary calls, use any of these techniques.

- Build as a single combined library, then build referencing task (Figure F-3).
- If referenced library does not contain overlays (Figure F-4):
  - Build referenced library.
  - Build referencing library, specifying referenced library to resolve calls.
  - Build referencing task, specifying only referencing library.
- If referenced library has overlays (Figures F-5 and F-6):
  - You must revector interlibrary calls to allow access to overlay structure and autoload vectors (always in root of referencing task).
  - Once revectoring is included, build shared regions and referencing task as if regions are logically independent.

Example task-build commands for each technique follow.

Example task-build command for combined libraries (Figure F-3):

```
>LINK/MAP/NOHEADER/SHAREABLE:LIBRARY/SYMBOL_TABLE-
->/OPTIONS F4PRES,LB:[1,1]F4POTS/LIBRARY
Option? STACK=Ø
Option? PAR=F4PRES:12ØØØØ:6ØØØØ
Option? <RET>
```

Referencing task is created using normal procedure to reference the library F4PRES.

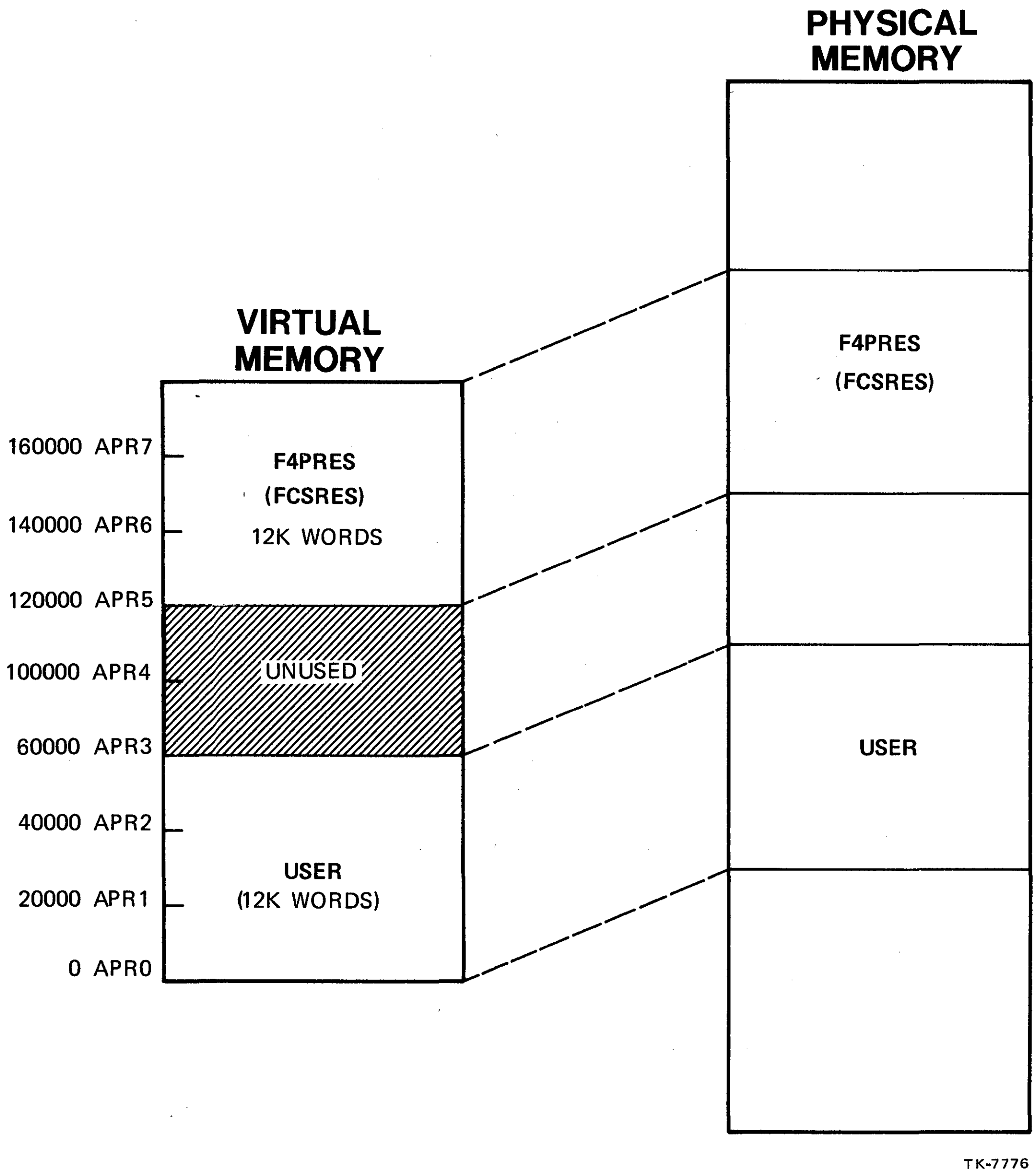


Figure F-3 Referencing Combined Libraries

Example task-build commands for building one library, then building the second (referencing) library (Figure F-4):

```
>LINK/MAP/NOHEADER/SHAREABLE:LIBRARY/SYMBOL_TABLE-
->/OPTIONS/CODE:PIC FCSRES
Option? STACK=Ø
Option? PAR=FCSRES:Ø:2ØØØØ
Option? <RET>

>LINK/MAP/NOHEADER/SHAREABLE:LIBRARY/SYMBOL_TABLE-
->/OPTIONS F4PRES,LB:[1,1]F4POTS/LIBRARY
Option? STACK=Ø
Option? LIBR=FCSRES:RO
Option? PAR=F4PRES:14ØØØØ:4ØØØØ
Option? <RET>
```

Referencing task is created using normal procedure to reference just the library F4PRES. F4PRES must be mapped using APRs 6 and 7 because it is built absolute. FCSRES is mapped at the next available APR, namely APR 5, because it is built position independent.

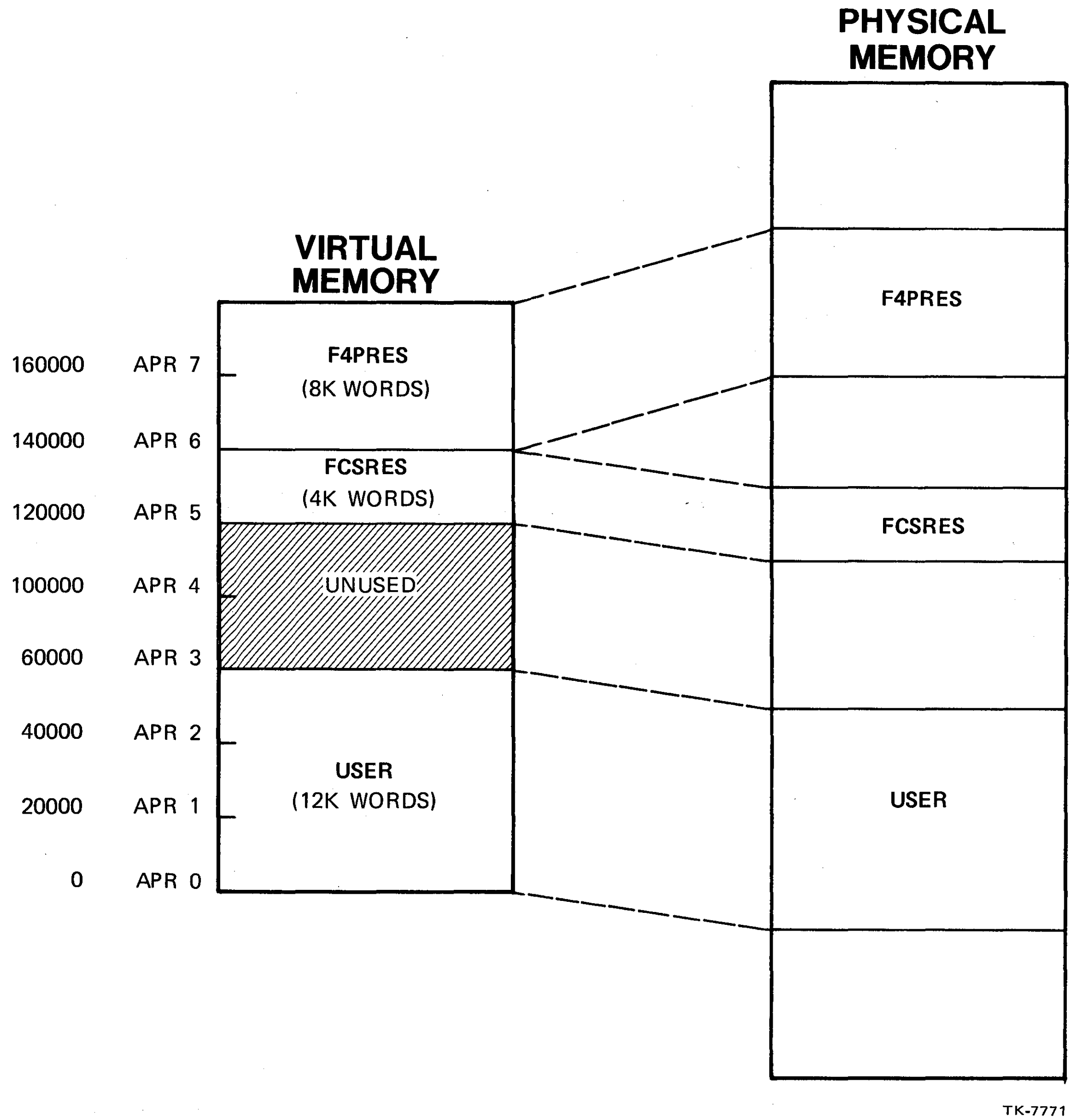


Figure F-4 Building One Library, Then Building a Referencing Library

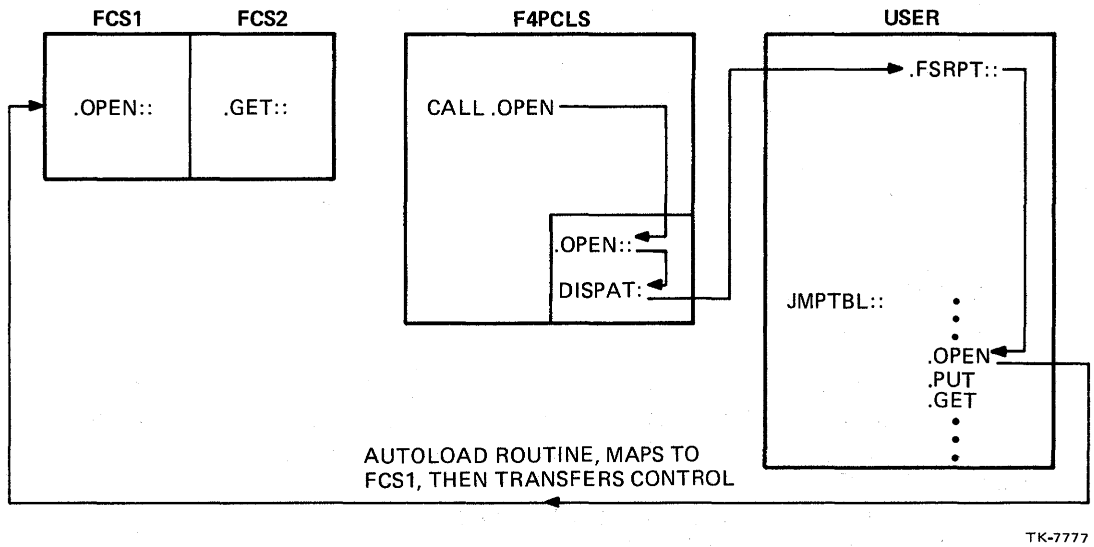


Figure F-5 Revectoring

See Section 5.2.1.3 (on User Task Vectors Indirectly Resolve all Interlibrary References) in the RSX-11M/M-PLUS Task Builder Manual for additional information on revectoring. See also Section 5.2.3 on Examples for commented task-build commands for building libraries with revectoring.

Example task-build commands when revectoring are used (Figure F-6):

```
>LINK/MAP/NOHEADER/SHAREABLE:LIBRARY/SYMBOL_TABLE-
->/OPTIONS/CODE:PIC FCSRES/OVERLAY_DESCRIPTION
Option? STACK=0
Option? PAR=FCSRES:0:20000
Option? GBLREF=.CLOSE
Option? GBLREF=.CSI1
Option? GBLREF=.CSI2
    .
    .
    .
Option? GBLREF=.WAIT
Option? <RET>

>LINK/MAP/NOHEADER/SHAREABLE:LIBRARY/SYMBOL_TABLE:-
->F4PCLS/TASK:F4PCLS/OPTIONS F4PRES,LB:[1,1]F4POTS-
->/LIBRARY,LB:[1,1]SYSLIB/INCLUDE:FCSVEC
Option? STACK=0
Option? PAR=F4PCLS:140000:40000
Option? GBLINC=.FCSJT
Option? GBLXCL=.CLOSE
Option? GBLXCL=.CSI1
Option? GBLXCL=.CSI2
    .
    .
    .
Option? GBLXCL=.WAIT
Option? <RET>
```

Referencing task is created using normal procedure to reference libraries FCSRES and F4PCLS.

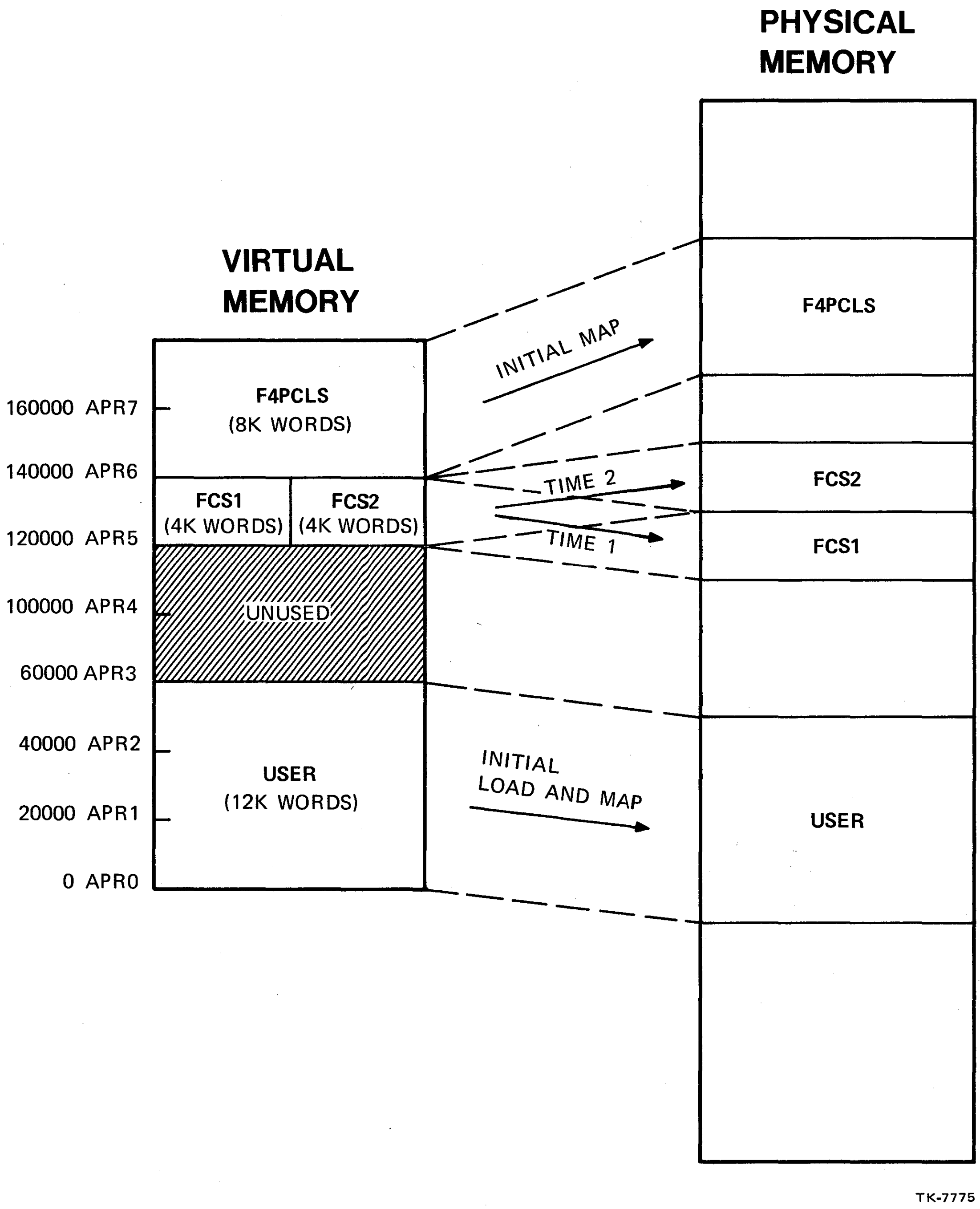


Figure F-6 Using Revectoring When Referencing Library Has Overlays

CLUSTER LIBRARIES

- Allow shared libraries to overlay each other (Figure F-7).
  - Can use one window for several libraries.
  - Only enough virtual address space is needed for largest library.
- One library can call another.
  - Generally moving in one direction only.
  - First library in cluster is initially mapped (no autoload).
  - When a call is made to another library in cluster:

    Autoload routines save mapping context and map called library for a call.

    Original library is remapped for return from subroutine.
- Revectoring is necessary for interlibrary calls (Figure F-5).
  - Special coding must be included in the resident libraries.
- Some special rules must be followed when building the resident libraries.
- Are useful for FORTRAN tasks using the resident object time system (FORRES, F4PRES, or F77RES), plus layered products.

See Section 5.2 on Cluster Libraries in the RSX-11M/M-PLUS Task Builder Manual for additional information.

Example of task-build command:

```
>LINK/MAP/OPTIONS/CODE:FPP CLSDEM,LB:[1,1]HLLFOR,-
->LB:[1,1]F4POTS/LB,LB:[1,1]FDVLIB/LB
Option? CLSTR=F4PCLS,FMSCLS,FCSRES:RO
Option? <RET>
```

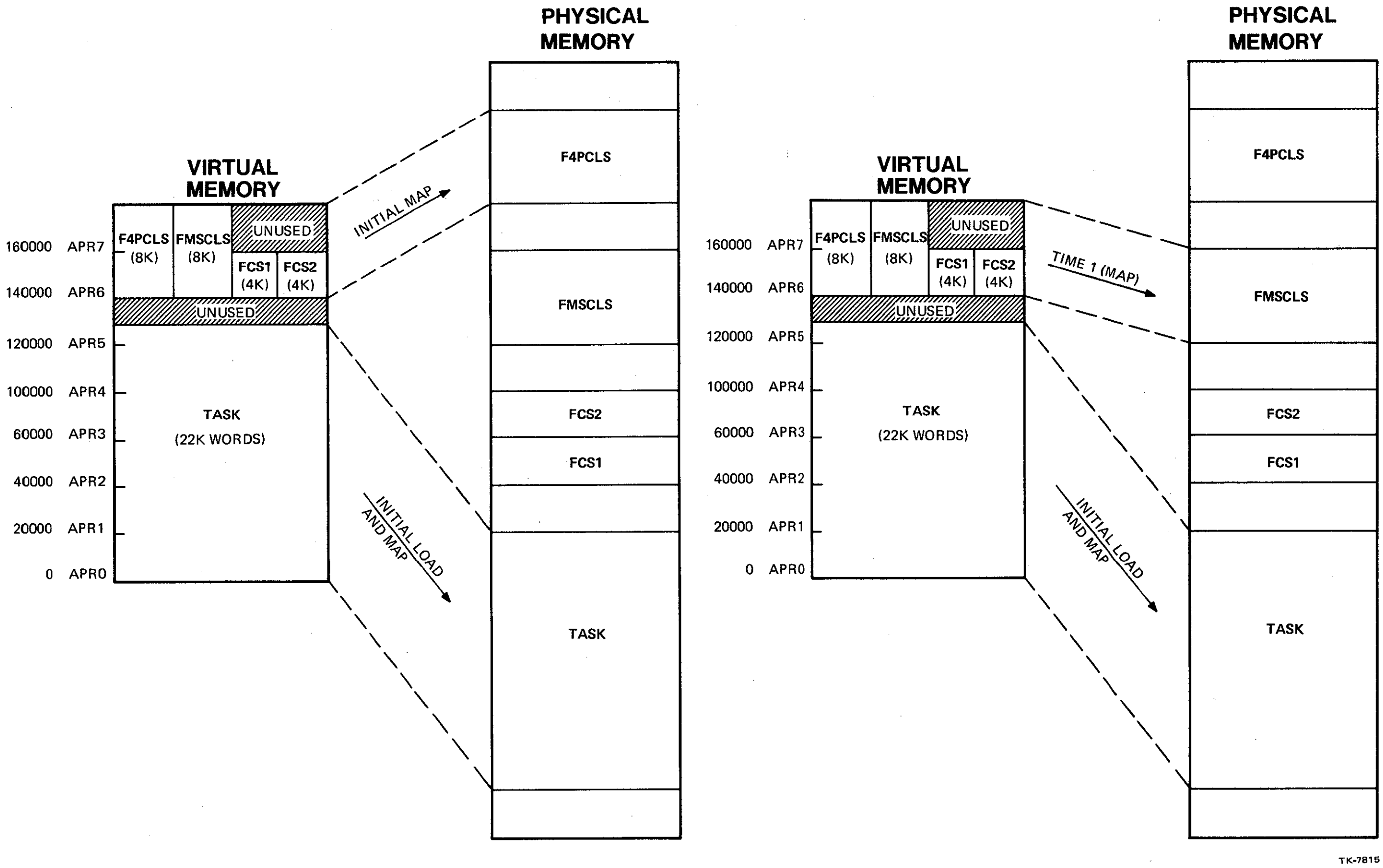


Figure F-7 Cluster Libraries (Sheet 1 of 2)

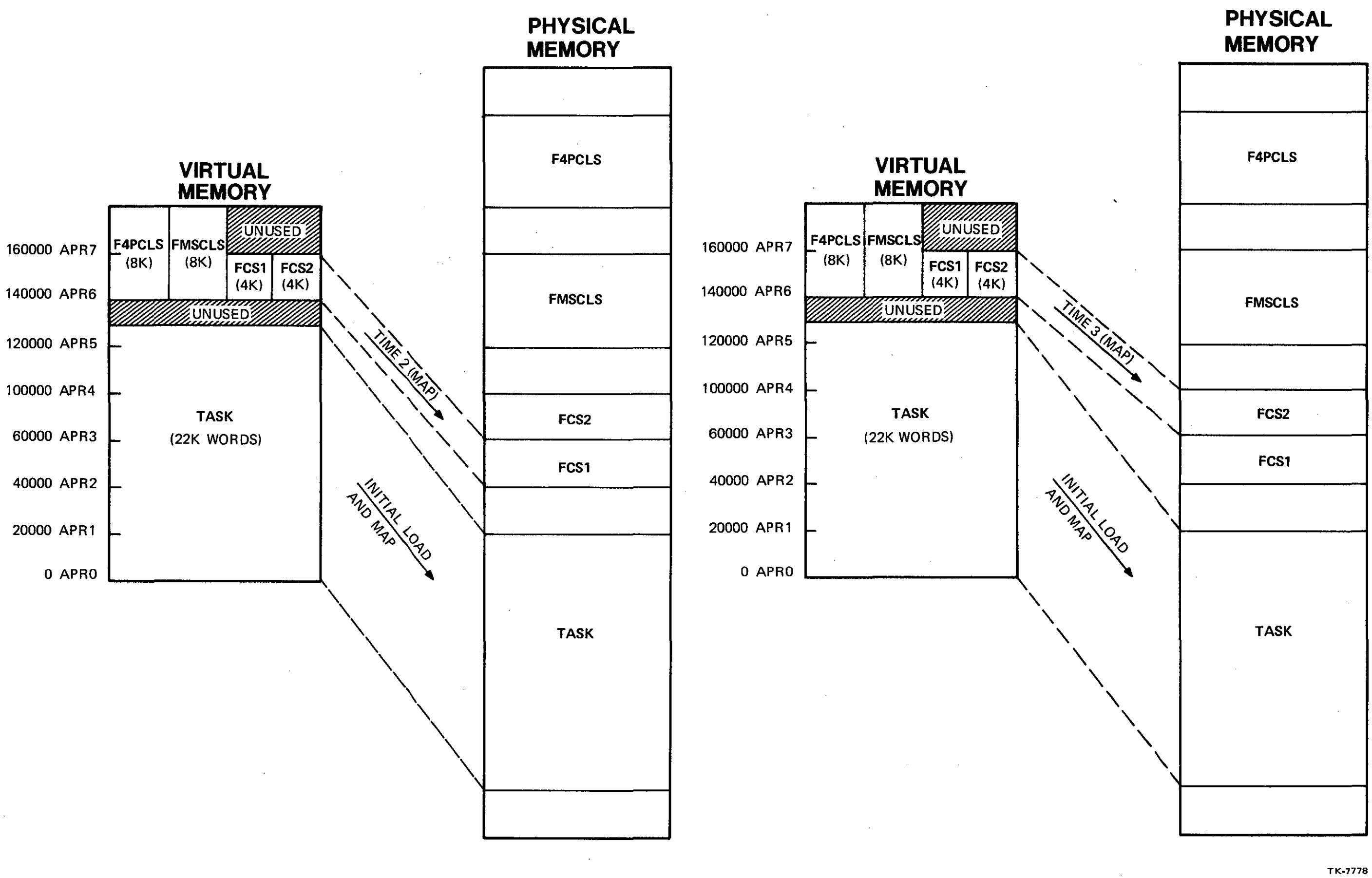


Figure F-7 Cluster Libraries (Sheet 2 of 2)

APPENDIX G
ADDITIONAL EXAMPLES

The following examples should be available on-line, probably under UFD [2Ø2,1]. They are needed for the Tests/Exercises. Therefore, they are listed here in case they are not available on-line at your site.

```
        .TITLE  READF
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; File READF.MAC
;
; This task starts up, sets event flag 1, reads the
; event flags, moves them into registers R0-R3 and then
; exits. It uses the $ form of the directive calls.
;
; The flags are returned as follows:
;
;                 word 0 = event flags 1-16
;                 word 1 = event flags 17-32
;                 word 2 = event flags 33-48
;                 word 3 = event flags 49-64
;         bit set means flag is set,
;         bit clear means flag is clear
;-

        .MCALL  RDAF$,SETF$,EXIT$S,DIR$ ; System macros

BUFF:   .BLKW   4                       ; Buffer for event flag
                                        ;  values
READ:   RDAF$   BUFF                    ; DPB for Read All Event
                                        ;  Flags directive

SETF:   SETF$   1                       ; DPB for Set Event Flag
                                        ;  directive

START:  CLR     R4                      ; Clear error counter
        DIR$    #SETF                   ; Set event flag 1
        BCS     ERR1                    ; Branch on dir error
        DIR$    #READ                   ; Read the event flags
                                        ;  (1 - 64).
        BCS     ERR2                    ; Branch on dir error
        MOV     BUFF,R0                 ; Move the event flag
        MOV     BUFF+2,R1               ;  values into the
        MOV     BUFF+4,R2               ;  registers
        MOV     BUFF+6,R3
        IOT                             ; Trap and display
                                        ;  registers

; Come here on directive errors
ERR2:   INC     R4                      ; R4=2 for read error
ERR1:   INC     R4                      ; R4=1 for set event
                                        ;  flag error
        MOV     $DSW,R0                 ; Error code into R0
        IOT                             ; Trap and display the
                                        ;  registers
        .END    START
```

Example G-1 Reading the Event Flags (for Exercise 1-1)

```
        .TITLE  CSI
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;
; CSI illustrates the use of the command string
; interpreter. This task accepts a command line from the
; terminal in the form:
;
;       dev:[x,y]filename.filetype;version/switch
;
; where switch can be:
;                               DE - Delete file
;                               DI:N - Display N copies of file

        .MCALL  GCMLB$,GCML$,CSI$,CSI$1,CSI$2
        .MCALL  CSI$SV,CSI$SW,CSI$ND
        .MCALL  FSRSZ$,FDBDF$,FDRC$A,FDOP$A,FINIT$
        .MCALL  QIOW$S,QIOW$,DIR$,EXIT$S
        .MCALL  DELET$,OPEN$R,OPEN$W,GET$,PUT$,CLOSE$

        .NLIST  BEX

;       LOCAL DATA

TYPE1:  QIOW$   IO.WVB,5,1,,,,<ERR1,SIZ1,40>
TYPE2:  QIOW$   IO.WVB,5,1,,,,<ERR2,SIZ2,40>
TYPE3:  QIOW$   IO.WVB,5,1,,,,<ERR3,SIZ3,40>
TYPE4:  QIOW$   IO.WVB,5,1,,,,<ERR4,SIZ4,40>
ERR1:   .ASCII  /GET COMMAND LINE ERROR/
        SIZ1=.-ERR1
ERR2:   .ASCII  /CSI ERROR. ILLEGAL COMMAND/
        SIZ2=.-ERR2
ERR3:   .ASCII  /CSI ERROR. FILE SPEC ERROR/
        SIZ3=.-ERR3
ERR4:   .ASCII  /ERROR PERFORMING TASK/
        SIZ4=.-ERR4
BUFF:   .BLKB   100.                    ; Output text buffer
TBUFF:  .BLKB   132.                    ; Transfer buffer
FMT:    .ASCIZ  /YOU HAVE REQUESTED A %7A JOB/
        .EVEN
DATA:   .WORD 0                         ; Argument block
DELTXT: .ASCII  /DELETE/<0>             ; ASCII text
TYTXT:  .ASCII  /TYPE/<0><0><0>
NOTXT:  .ASCII  /NOTHING/
        .EVEN

        CSI$                            ; Define CSI offsets
CBLK:   .BLKB   C.SIZE                  ; allocate CSI storage
        .EVEN

        DEMSK = 1                       ; Delete mask
        DIMSK = 2                       ; Display mask

```

Example G-2 Using the Routines GCML and CSI (for Exercise 10-6)
(Sheet 1 of 3)

```
SWTBL:                                ; Switch descriptor table
        CSI$SW  DE,DEMSK              ; Delete switch = DE
        CSI$SW  DI,DIMSK,,,,NUM       ; Display switch = DI,
                                      ;  also allow DI:N
        CSI$ND                        ; End of switch table

        CSI$SV  OCTAL,COPY,2,NUM;       Value N for /DI:N is
                                      ;  in octal and will
                                      ;  be stored in COPY
        CSI$ND                        ; End of switch value
                                      ;  table

;GET COMMAND LINE BLOCK DEFINITIONS

        FSRSZ$  1                     ; GCML uses record I/O

GBLK:   GCMLB$  ,CSI,,5               ; Prompt with 'CSI' on
                                      ;  LUN 5
FDB:    FDBDF$                        ; FDB for file to delete
                                      ;  or display.
        FDRC$A  ,TBUFF,132.           ; URB AT TBUFF, length
                                      ;  132.
        FDOP$A  1,CBLK+C.DSDS         ; LUN 1, dataset
                                      ; descriptor from CSI

; NOTE: Need a 2nd FDB for display

        .EVEN

JMPTBL: .WORD   NONE,DELETE,DISPLY ; Jump table for
                                      ;  subroutines depending
                                      ;  on switches

COPY:   .WORD   0                     ; Value for N in /DI:N
        .ENABLE LSB

START:  FINIT$                        ; Initialize FCS, this
                                      ;  is normally done with
                                      ;  an OPEN statement.
                                      ;  For delete we do not
                                      ;  need an open statement.
NEXT:   GCML$   #GBLK                 ; Prompt and get command
        BCC     10$                   ; Branch if command OK
; Check for ^Z. If ^Z, exit.
        CMPB    #GE.EOF,GBLK+G.ERR ; Is it ^Z?
        BNE     REALER                ; Branch on other error
        EXIT$S                        ; Exit
REALER: DIR$    #TYPE1                ; Display error text for
                                      ;  get command line error
        EXIT$S                        ; Exit

; Parse input for illegal characters

10$:    CSI$1   #CBLK,GBLK+G.CMLD+2,GBLK+G.CMLD ; Format
                                      ; is CSI addr, addr of
                                      ; command, length of
                                      ; command
```

Example G-2 Using the Routines GCML and CSI (for Exercise 10-6)
(Sheet 2 of 3)

```
        BCC     20$                     ; Branch on OK command
        DIR$    #TYPE2                  ; Display error text for
                                        ;  illegal command
        EXIT$S                          ; Exit

; Create a dataset descriptor from the file specification

20$:    CSI$2   #CBLK,OUTPUT,#SWTBL ; Expect output file
                                        ;  spec
        BCC     30$                     ; Branch on file spec OK
        DIR$    #TYPE3                  ; Display text for file
                                        ;  spec error
        EXIT$S                          ; Exit

; Call the appropriate subroutine

30$:    MOV     #FDB,R0                 ; Address of file
                                        ;  descriptor
        MOV     CBLK+C.MKW1,R1          ; Mask value = 0, 1, or 2
        ASL     R1                      ; Double for word offset
                                        ;  into jump table
        CALL    @JMPTBL(R1)             ; Call the subroutine
        BR      NEXT                    ; Get next command line

; Subroutine NONE, entered if no switches specified

NONE:   MOV     #NOTXT,DATA             ; Set up for output of
                                        ;  message

; Common display message code

OUTM:   MOV     #BUFF,R0                ; Set up for $EDMSG
        MOV     #FMT,R1                 ;
        MOV     #DATA,R2                ;
        CALL    $EDMSG                  ; Edit message
        QIOW$S  #IO.WVB,#5,#1,,,,<#BUFF,R1,#40> ; Display
        RETURN                          ; Return

; Subroutine DELETE - Just display a message

DELETE: MOV     #DELTXT,DATA            ; Set up for output of
                                        ;  message
        BR      OUTM                    ; Branch to common
                                        ;  display code

; Subroutine DISPLY - Just display a message

DISPLY: MOV     #TYTXT,DATA             ; Set up for output of
                                        ;  message
        BR      OUTM                    ; Branch to common
                                        ;  display code
        .END    START
```

Example G-2 Using the Routines GCML and CSI (for Exercise 10-6)
(Sheet 3 of 3)

APPENDIX H
LEARNING ACTIVITY ANSWER SHEET

Learning Activity 2-1 (Directives)

1. Either: a) Do some work, then check the flag by using the CLEF$ 35. directive. Check the DSW. IS.SET (=+2) means the flag was set; IS.CLR (=Ø) means the flag was clear, or b) read flags 4 through 64 using RDAF$ and then test bit 2 of the third word in the buffer to read flag 35. In either case, keep doing more specific work and periodically check the flag.

2. The Executive, would only set event flag 1 for Task A. It would not set Task B's event flag 1; therefore, Task B wouldn't realize that the data had been sent.

3. Local flags are accessible only to the task itself. They are specifically provided for synchronization between the Executive and a task.

Learning Activity 6-1 (Overlays)

1.

```
.ROOT-*!(P,Q)
.END
```

2. LINK/MAP ROOT,P,Q

Learning Activity 6-2

1. Diagram

```
JOB1     JOBXX
  |        |
  ----------
       |
       A              B
       |              |
       ----------------
               |
             TOTAL
               |
              MAIN
```

2.
```
.ROOT MAIN-TOTAL-*(A-(JOB1,JOBXX),B)
.END
```

3.
```
.ROOT MAIN-TOTAL-*!(A-!(JOB1,JOBXX),B)
.END
```

4.
```
.ROOT MAIN-TOTAL-*!(A-(JOB1,JOBXX),B)
.END
```

Learning Activity 10-1 (File Control Services)

Without a User Record buffer (no spanning of blocks):

```
      FDBDF$
      FDRC$A   FD.PLC        ; Use locate mode
      FDOP$A   1,DFNB        ; Use LUN 1, default name block
DFNB: NMBLK$   YOURS,MAC     ; File Spec
```

With a User Record Buffer

```
      FDBDF$
      FDRC$A  FD.PLC,URB,80.; 80.= maximum record size,
                            ; Record size can be checked after
                            ; the file is opened as well.

      FDOP$A  1,DFNB
DFNB: NMBLK$  YOURS.MAC
```

You can use a dataset descriptor as well.

If you use a default name block to specify TI:, use:

```
     NMBLK$ ,,,TI,0
```

GLOSSARY

GL

ASYNCHRONOUS SYSTEM TRAP (AST) - A system condition which occurs as a result of a specified event such as completion of an I/O request.

On occurrence of the event, control passes to an AST service routine, and the AST is added to an Executive first-in first-out queue for the task in which the service routine appears.

ATTACH - Device: Dedicate a physical device unit for exclusive use by the task that requested attachment.

A task attaches a given device by issuing a QIO directive, or QIO and WAIT directive, specifying the I/O function IO.ATT.

Region: Include a region in a task's logical address space.

A task attaches a region by issuing an Attach Region directive or by being the target of another task's Send-By-Reference directive.

CLUSTER LIBRARIES - A special setup with shared resident libraries which permits a task to use the same virtual address window to map several difficult libraries. For example, the resident FORTRAN Object Time System and the resident FCS library could use the same virtual addresses. The run-time routines map and remap the regions as they are needed, somewhat similar to what happens with regular memory-resident overlays.

DATASET DESCRIPTOR - A six-word area in the user task containing sizes and addresses of ASCII data strings, which FCS consults in order to obtain a run-time file specification.

A dataset descriptor for a given file is a user-created data structure which contains a file specification for that file.

When the filename block associated with a given file does not contain sufficient information to enable FCS to do run-time file processing on that file, FCS tries to get the needed information from the file's dataset descriptor, if specified. Otherwise, FCS consults the file's default filename block, if specified, in order to get the desired information.

DEFAULT FILENAME BLOCK - An area in the user task that supplies FCS with those default values that are needed to build a routine file specification.

When the filename block associated with a given file does not contain sufficient information to allow FCS to process the file, and when a dataset descriptor does not contain the needed information, then FCS consults the default filename block associated with the file to obtain the missing information.

A default filename block may be used to supply a default name, extension, and/or version for the file. The MACRO programmer uses the NMBLK$ macro to create this block at assembly time.

DETACH - Device: Free an attached physical device unit for use by tasks other than the one that attached it.

A physical device unit can only be detached by means of an IO.DET I/O function issued by the task that attached it, or by the Executive, if the task is terminated with the device still attached.

Region: Remove a region from a task's logical address space.

A task detaches a region by issuing a Detach Region directive or by exiting.

DIRECTIVE STATUS WORD - A word in the user task header into which the Executive returns status information about the most recently called directive.

After processing a directive, the Executive passes the status of that directive to the issuing task by putting a success or error code into the task's Directive Status Word, which is assigned the global label $DSW. If $DSW is negative, the Executive rejected the directive; if $DSW is +1, the directive was successful.

EVENT FLAG - A software flag which can be specified in a program request to indicate to the issuing task which of several specified events has occurred.

There are 96(1Ø) event flags.

Event flags	1	-	32(1Ø)	are local
	33(1Ø)	-	64(1Ø)	are system global flags
	65(1Ø)	-	96(1Ø)	are group global flags

Local flags are used for intra-task synchronization, while group global and system global flags are used for inter-task synchronization and communication.

EXECUTIVE DIRECTIVE - A program request for Executive services.

An Executive directive is issued from a FORTRAN program by calling a subroutine in the system object library. It is issued from a MACRO-11 program by invoking a macro in the system macro library.

FILE DESCRIPTOR BLOCK (FDB) - The tabular data structure which provides FCS with information needed to perform I/O operations on a file.

A task must allocate, through calls to the FDBDF$ macro, or dynamically through the use of run-time macros.

FILE STORAGE REGION (FSR) - The area in user task which FCS uses to buffer all virtual blocks read or written during record processing.

FCS requires one FSR block buffer for each file to be opened at the same time for record I/O. When the task requests a record that is not in the FSR buffer, FCS reads a virtual block from the file into the task's file storage region. On the other hand, FCS writes virtual blocks in the file storage region to the file when a record must be put to the file.

The user task allocates this area by issuing an FSRSZ$ macro.

FILENAME BLOCK - The part of a file's File Descriptor Block which FCS uses for building, and later using, a file specification.

The filename block contains the file's UFD, name, extension, version number, device name, and unit. When a file is initially opened, FCS fills in the filename block from user-supplied information in the dataset descriptor and/or default filename block.

I/O STATUS BLOCK - A two-integer array which receives success or error codes on completion of an I/O request. If an I/O status block has been specified in an I/O request, the Executive clears both words when the I/O operation is queued. On completion, the low byte of the first word contains +1 if the I/O was successful, and a negative error code otherwise.

If the I/O function involved a transfer, the second word contains, on completion, the number of bytes transferred.

LOGICAL ADDRESS SPACE - The set of all physical addresses to which a task has access rights.

If a task is running on a mapped system that includes support for the memory management directives, it may issue directives in order to manipulate its logical address space at run time.

LOGICAL BLOCK - A 512(10) byte (256(10) word) block of data on a block addressable volume.

To achieve device independence, each block addressable volume is organized into logical blocks, numbered 0 to n-1, where n is the number of logical blocks on the volume.

The mapping of logical blocks to physical blocks is handled by the driver.

LOGICAL UNIT NUMBER (LUN) - A number associated with a physical device unit during a task's I/O operations.

The association of a LUN in a task with a given physical device may be done by the Task Builder, by the operator using the REASSIGN command, or at run time by the task, by issuing an Assign LUN directive.

RANDOM ACCESS - A method of I/O to disk files in which records (or virtual blocks) are specified by record (or virtual block) number.

Under FCS, a file must be organized into fixed length records in order for a task to do random access to the file.

FCS supports the use of block I/O, in which virtual blocks are read from, or written to, the file without regard for the structure of those blocks. The FORTRAN language does not support block I/O.

READ/WRITE MODE - An FCS file access method in which the user task uses the READ$ and WRITE$ macros to do block-structured I/O to a file.

REGION - An area consisting of one or more contiguous 32.-word blocks of physical memory.

A region may be named or unnamed, but is always assigned a unique region ID. A region has an associated protection word which specifies the access rights a task may have with respect to that region. Any task that satisfies the region protection word may attach a named region, but no task can attach an unnamed region unless the task has the region ID.

RESIDENT COMMON - A shared region which contains data.

RESIDENT LIBRARY - A shared region containing subroutines and/or functions.

SEQUENTIAL ACCESS - A mode of record access in which the n+1th record in the file is processed after the nth record in the file.

Each record is assigned a record number, and each successive GET or PUT causes the record number to be incremented.

SYNCHRONOUS SYSTEM TRAP (SST) - A "software interrupt" which typically occurs as a result of an error or fault within the executing task.

On recognition of an SST, the Executive aborts the task, unless there is an SST vector table to an SST routine in the task.

VIRTUAL ADDRESS - A 16-bit address which may be directly specified using one of the general purpose registers.

A task specifies a virtual address whenever it uses one of the addressing modes in executing an instruction. Up to 32K virtual word addresses may be specified by a task.

On a mapped system, the memory management hardware dynamically maps virtual addresses to real physical addresses.

VIRTUAL ADDRESS WINDOW - A contiguous chunk of a task's virtual address space.

Each virtual address window in a task begins on a 4K word boundary and consists of one or more 32(10) word blocks of virtual address space. Each window has a unique number assigned to it by the Executive. Window 0 always maps the task's header, stack, and code. A task may divide its virtual address space into eight windows.

VIRTUAL BLOCK - One of the logical blocks belonging to a file.

Each file consists of one or more logical blocks. The logical blocks belonging to a file are called virtual blocks 1, 2, 3, etc. The mapping of virtual blocks in a file to logical blocks on disk is performed by the file system.

WINDOW DESCRIPTOR BLOCK (WDB) - A data structure used in a task in order to represent a dynamically created window.