EY-0061E-SG-0201

Programming RSX-11M in FORTRAN

Volume II

digital

EY-0061E-SG-0201

Programming RSX-11M in FORTRAN

Student Workbook
Volume II

Prepared by Educational Services
of
Digital Equipment Corporation

Copyright © 1982, Digital Equipment Corporation.
All Rights Reserved.

The reproduction of this material, in part or whole, is strictly prohibited. For copy information, contact the Educational Services Department, Digital Equipment Corporation, Bedford, Massachusetts 01730.

Printed in U.S.A.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may not be used or copied except in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Digital.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECSYSTEM-20	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	RSTS
UNIBUS	VAX	RSX
	VMS	IAS

CONTENTS

Volume I

Volume II

6 OVERLAYING TECHNIQUES

7 STATIC REGIONS

FIGURES

TABLES

EXAMPLES

OVERLAYING TECHNIQUES

INTRODUCTION

Overlays are used to allow a task to be developed and run if the amount of memory available or virtual address space for a task is insufficient. This module explains the various overlay techniques which are available and how to use them.

OBJECTIVES

1. To determine whether to use a disk-resident or memory-resident overlay in a given situation

2. To construct overlay structures using the overlay descriptor language

3. To write tasks using overlays.

RESOURCE

- RSX-11M/M-PLUS Task Builder Manual, Chapters 3 and 4

CONCEPTS

A task may be too large to fit in the available memory. This may happen because it is larger than the total amount of memory on the system. More likely, it is because it is larger than the partition it is to run in, or the available space within the partition. The partition is probably used at the same time by other tasks, hence, the available space may be considerably less than the full partition.

For example, a 2ØK word task may have to fit in 15K words of memory. The task can use overlays and load only portions of the code at a time and just use 15K words of memory. Typically, the pieces which overlay each other contain subroutines.

As an example, consider a task with main code and two subroutines, G and H, which overlay each other. The main code calls subroutine G first, causing G's code to be read into memory. Later, the main code calls subroutine H, causing H's code to be read into the same memory locations, overlaying subroutine G. If the main code later calls G, G's code overlays subroutine H. As the task executes, overlaying is performed whenever necessary. You can choose to have all loading of overlay segments done automatically or you can load them manually with specific calls to a loading routine.

In addition to physical memory limitations, tasks on PDP-11 systems have virtual memory limitations. As we learned in the last module, a task can use a maximum of 32K words of virtual addresses at a time. A task may require, say, 4ØK of virtual memory, thereby exceeding the 32K virtual addressing limit. Overlays loaded from disk would permit this task to run in 32K words or less of physical memory, and allow all of the code loaded at any given time to be addressed. Therefore, 32K words, or less, of code are loaded and addressed at any one time, satisfying the virtual address limit. Or, using a special kind of overlay, all 4ØK words of code can be loaded into memory, but the task maps only 32K words of code at a time. This means that the task stays within the virtual addressing limits even though it uses 4ØK of physical memory.

These special kinds of overlays are called memory-resident overlays. They overlay by remapping rather than by reloading code into memory.

An overlaid task can have several program segments. A program segment consists of part or all of one or more object modules. Each of the object modules consists in turn of one or more program sections (Psects). There is always a single resident root segment. This segment is loaded when the task is first loaded and remains loaded and mapped at all times. In addition, there are overlay segments which either:

- reside on disk unless needed and share virtual address space and physical memory.

- stay in memory once needed and share virtual address space only.

There is one restriction on subroutines in an overlay segment. They cannot call subroutines which are located in a segment which overlays itself. The code for only one segment or the other is available at any one time, and never both. We say that the segments must be logically independent.

There are some drawbacks to using overlays. Additional code is required to handle the overlay structure and the loading and/or mapping of the overlay segments. Also, some execution time is required to load and/or map the overlay segments.

OVERLAY STRUCTURE

Example 6-1 lists the subroutines (corresponding to overlay segments) which each segment calls during the execution of a task. In addition, the sizes of the various modules are listed. If the task is built without overlays, it is 17K words in size.

We can reduce the amount of memory needed to 8K words by using overlays. Figure 6-1 shows a likely overlay structure, using a memory allocation diagram. This picture represents the overlaying or sharing of virtual and/or physical address space in the task. Figure 6-2 shows another method for showing the same overlay structure, an overlay tree. It is easier to draw but doesn't allow you to estimate the size of the task. As the calculation below Figure 6-1 shows, the largest pieces which will ever be needed at any one time are PROG, the root, and overlay segments SUB1 and B. These total 8K words, so this task can run in 8K words of physical memory.

Main Segment:	PROG
PROG calls:	SUB1, SUB2, SUB3
SUB1 calls:	A, B
SUB2 calls:	none
SUB3 calls:	C, D, E

Segment	Size
PROG	4K words
SUB1	2K words
SUB2	3K words
SUB3	1K words
A	1K words
B	2K words
C	1K words
D	2K words
E	1K words
TOTAL	17K words

Example 6-1 Description of An Overlaid Task

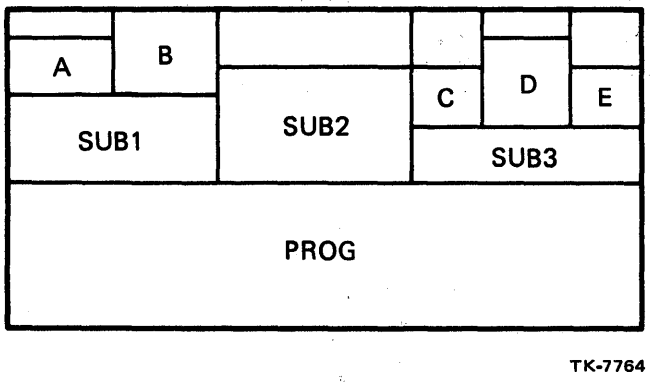


Figure 6-1 A Memory Allocation Diagram

```
Overlaid Task Size = Size of Root + Sum of lengths of segments
                                    using the most overlay
                                    area at any one time
              = Size of PROG + Size of SUB1 + Size of B
              =      4K      +      2K      +     2K    = 8K
```

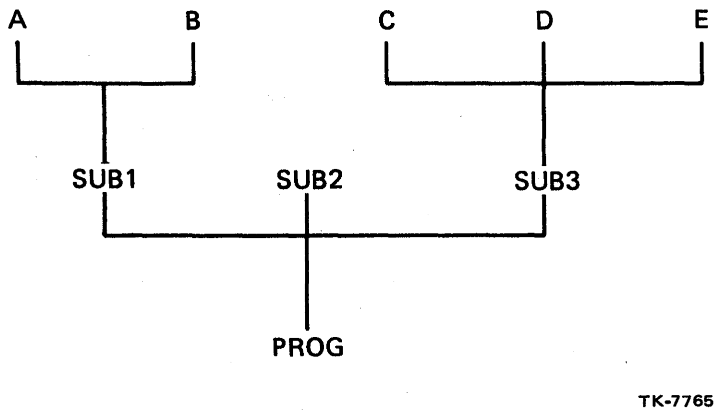


Figure 6-2 An Overlay Tree

STEPS IN PROGRAM DEVELOPMENT USING OVERLAYS

Use the following steps in developing a task which uses overlays:

1. Compile each module, producing a .OBJ file for each.
2. Use the editor to create an overlay descriptor file (defines the overlay structure for the Task Builder).
3. Task-build using the overlay descriptor file as the only input file.

THE OVERLAY DESCRIPTOR LANGUAGE (ODL)

The overlay descriptor language (ODL) is a fairly simple language which is used to define the overlay structure for the Task Builder. Statements are placed in a text file which has a file type 'ODL' (e.g., EXAMPLE.ODL). It is identified to the Task Builder as a special file by using the /OVERLAY_DESCRIPTION input file qualifier (/MP in MCR) in the task-build command line.

ODL Command Line Format

The ODL command lines use the following format:

```
label:   directive   argument-list   ;comment
```

where:

label - a one to six character symbolic, required only on a .FCTR directive.

directive - one of the following

```
.ROOT  - indicates the start of the overlay tree
.END   - indicates the end of input
.FCTR  - allows naming of subtrees
.NAME  - allows naming a segment and assigning
         attributes
.PSECT - allows special placement of a global
         program section (Psect) - typically
         used only in special cases in MACRO-11.
```

argument list - a list of .OBJ files and/or object libraries, separated by hyphens or commas, and grouped together with parentheses.

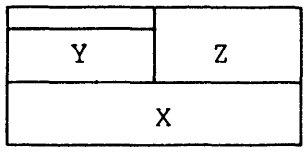
comment - a comment to annotate the line

The separators have the following meaning:

- Parentheses '()'
  - enclose the segments to be overlaid
- The hyphen '-'
  - indicates the concatenation of virtual address space
- The comma ','
  - separates the segments to be overlaid

Examples of ODL:

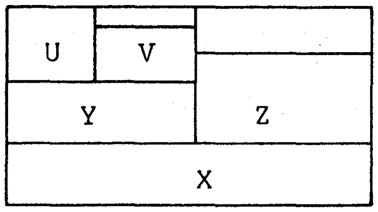
1. X, the root of a task, calls subroutines Y and Z.



```
.ROOT    X-(Y,Z)
.END
```

Explanation: X is the root segment, Y and Z are each overlay segments. Virtual addresses are assigned to X first. Starting after that, Y and Z begin at the same virtual address. Either Y or Z (never both) is loaded and mapped using those virtual addresses.

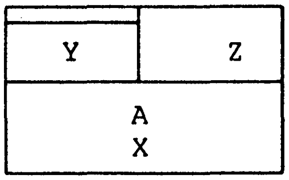
2. Using the information from Example 1, Y calls subroutines U and V.



```
.ROOT    X-(Y-(U,V),Z)
.END
```

Explanation: Add to Example 1. U and V are overlay segments which overlay each other. After the last address for Y, virtual addresses begin for U and V.

3. Using Example 1 again, add subroutine A to the root segment.



```
        .ROOT    X-A-(Y,Z)
        .END
```

Explanation: X and A together make up the root segment. Virtual addresses are assigned first to X and then to A. After that, Y and Z are assigned virtual addresses.

4. Using ODL to describe Example 6-1 (Figures 6-1 and 6-2):

```
        .ROOT    PROG-(SUB1-(A,B),SUB2,SUB3-(C,D,E))
        .END
```

Explanation: PROG is the root segment. SUB1, SUB2, and SUB3 overlay each other, beginning at the same virtual address. A and B overlay each other, beginning after SUB1. C, D, and E overlay each other, beginning after SUB3.

5. Using the .FCTR directive to describe Example 6-1:

```
        .ROOT    PROG-(PART1,SUB2,PART2)
PART1:  .FCTR    SUB1-(A,B)
PART2:  .FCTR    SUB3-(C,D,E)
        .END
```

Explanation: Substitute SUB1-(A,B) for PART1 in the first line. Substitute SUB3-(C,D,E) for PART2 in the first line.

TYPES OF OVERLAYS

There are two types of overlays available, disk-resident overlays and memory-resident overlays. In fact, both are loaded from disk. The distinction is that disk-resident overlays are always loaded from disk every time they are needed, while memory-resident overlays are loaded from disk only the first time they are needed. After that, they remain in memory and remapping is used to overlay segments as needed.

Disk-Resident

Disk-resident overlays are available on all RSX-11M systems. See Figure 6-3 for an example of a task with a root segment and three disk-resident overlays. On initial load, only the root segment MAIN is loaded. Overlay segments are loaded from disk whenever required. This typically occurs when a subroutine in the overlay segment is called. So if the root overlay segment MAIN contains a call for subroutine A, for example, segment A is loaded from disk prior to the transfer of control to A. If, after the subroutine returns control to MAIN, a call is made to subroutine B, segment B is loaded into memory right over segment A. If a call is later made to subroutine C, segment C is loaded right over segment B. This loading of overlay segments is performed whenever necessary. The subroutines may be called in any order and each subroutine may be called any number of times in the course of task execution.

The same starting virtual address is assigned to all three overlay segments, A, B, and C, beginning at the next 32(1Ø) word boundary after the code for MAIN. So A, B, and C use the same virtual addresses and are loaded starting at the same physical address. One virtual address window maps the entire task, just the code in memory is changed when an overlay is loaded.

This technique is useful when the entire task is too large to fit into the space allowed for it. In the example in Figure 6-3, a 22K word task runs in 15K words of physical memory. Disk-resident overlays are the default overlay type. The examples in the previous section all produce disk-resident overlays.

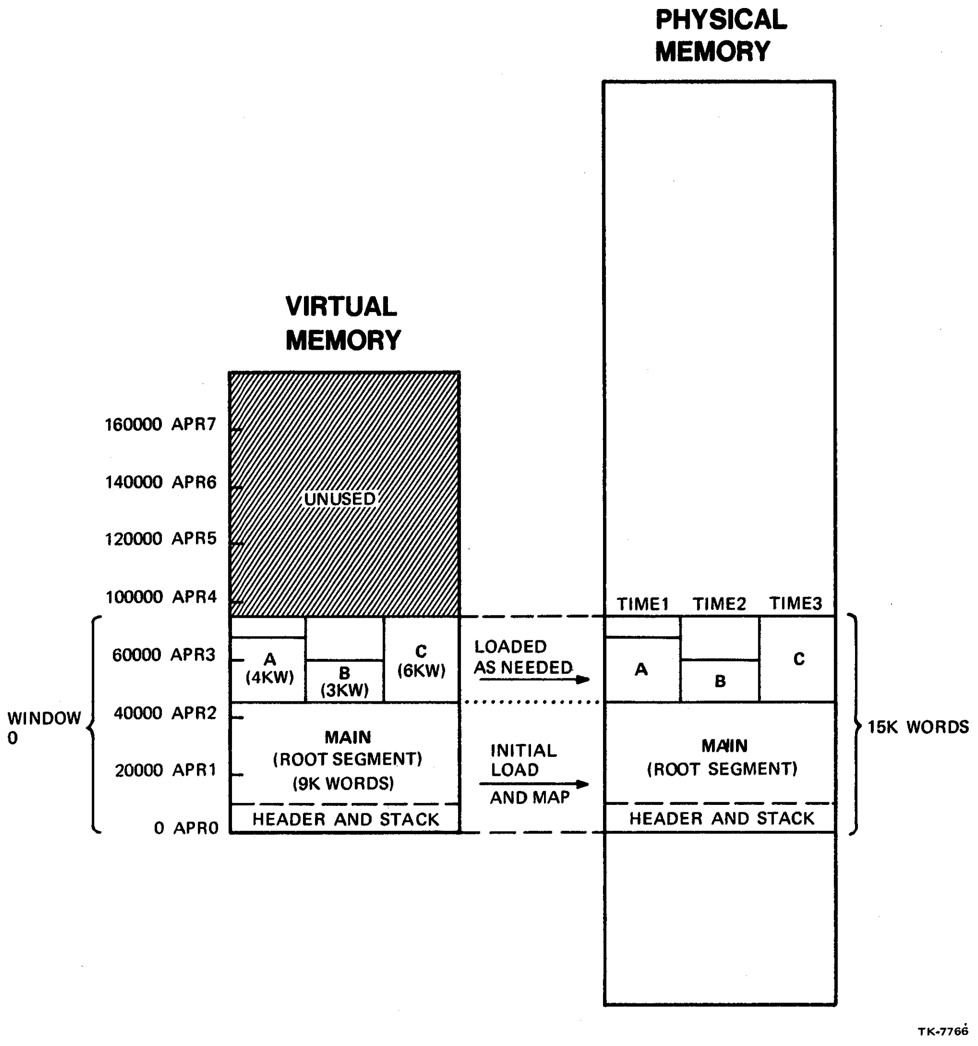


Figure 6-3 An Example of Disk-Resident Overlays

Memory-Resident

Memory-resident overlays are available only on mapped systems which support the memory management directives. See Figure 6-4 for the same task as in Figure 6-3, this time with memory-resident overlays. On initial load, again only the root segment MAIN is loaded. The first time an overlay segment is needed it is loaded from disk. However, once a segment is loaded it remains in memory and is not reloaded from disk.

If subroutine A is called first, overlay segment A is loaded and virtual address window 1 is mapped to A. If, after the subroutine returns control to MAIN, a call is made to subroutine B, then segment B is loaded, but not directly over A. Instead, it is loaded into another area of memory, and then virtual address window 1 is mapped to B. If a call is later made to subroutine C, segment C is loaded into another area of memory, and virtual address window 1 is mapped to C.

The real gain in run-time efficiency is made when an overlay is needed again. If another call is made to A, overlay segment A does not have to be loaded again from disk. It is already memory-resident. Therefore, virtual address window 1 is simply remapped from segment C to segment A. Any additional overlaying is performed by remapping, with no further loading of overlay segments necessary. Again, the subroutines may be called in any order and each subroutine may be called any number of times.

The advantage of this approach is that after the first load, it is much faster than disk-resident overlays. However, there is no savings in the use of physical memory. In fact, a bit more memory is required than with a non-overlaid task. So its main use is for overcoming the 32K word virtual address limit when execution time efficiency is important. A 44K word task can use memory-resident overlays if there is enough memory available and the time necessary for loading disk-resident overlay segments is unacceptable.

The root segment uses one window and each overlay area requires a separate window. This means that virtual addresses for each overlay segment begin at the starting virtual address for the next highest APR, corresponding to a 4K word boundary. Because the root segment is 9K(1Ø), APRs Ø, 1, and 2 must be used to map the root segment. Notice that A, B, and C all begin at virtual address 6ØØØØ, for APR 3.

This means that virtual addresses 44ØØØ-57777 cannot be used by this task. If in fact MAIN were extended, then these virtual addresses would be used. Remember, this doesn't mean that any physical memory is wasted; but it does mean that careful allocation of sizes to the various segments is necessary to avoid wasting virtual address space. Note that the maximum number of overlay areas with memory-resident overlay is seven since the root segment requires one virtual address window and each overlay level requires another virtual address window.

To indicate that you want memory-resident overlays, place an exclamation point (!) before an overlay specification. The '!' applies only to the first level; the next level may have disk-resident overlays or memory-resident overlays again. The only restriction on mixing of types is that once a level has disk-resident overlays, no higher level may have memory-resident overlays.

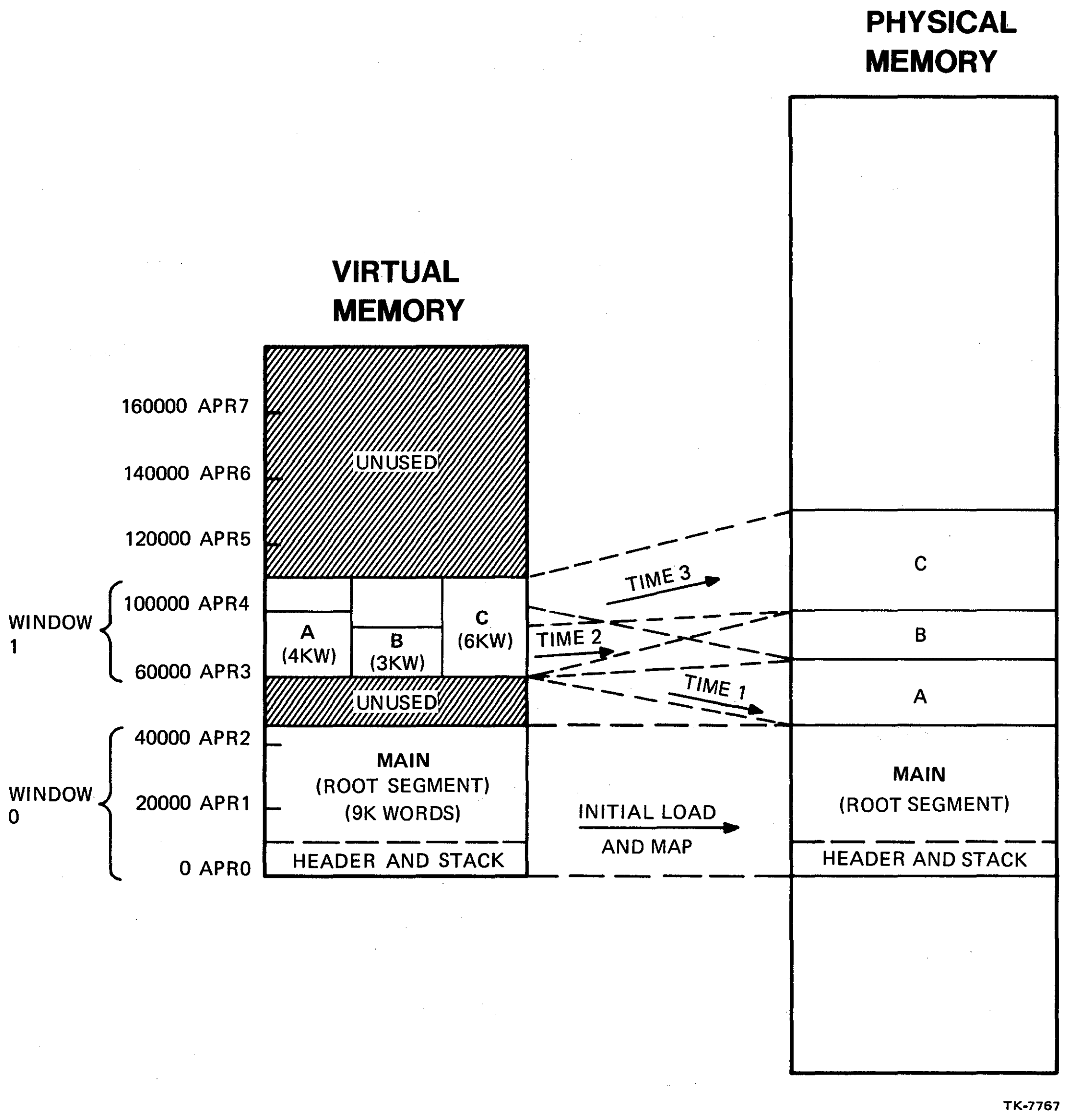
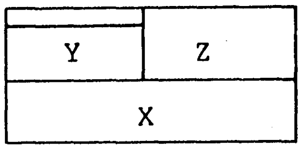


Figure 6-4 An Example of Memory-Resident Overlays

Examples of .ODL files for memory-resident overlays:

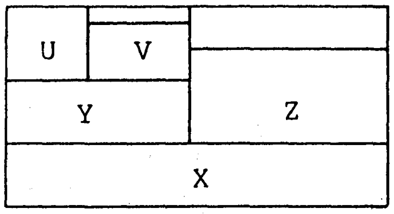
1. X, the root of a task, calls subroutines Y and Z.



```
.ROOT    X-!(Y,Z)
.END
```

The ! makes the overlays memory-resident.

2. Using the information from Example 1, Y calls subroutines U and V.



a. All memory-resident overlays:

```
.ROOT    X-!(Y-!(U,V),Z)
.END
```

b. Some memory-resident overlays, some disk-resident overlays:

```
.ROOT    X-!(Y-(U,V),Z)
.END
```

c. Illegal mixture:

```
.ROOT    X-(Y-!(U,V),Z)
.END
```

Explanation of c.: This mixture is illegal because the first level (Y and Z) is disk-resident. The next higher level cannot have memory-resident overlays. Therefore, U and V cannot be memory-resident.

LOADING METHODS

There are two loading methods, autoload and manual load. With autoload, any necessary loading and/or remapping (in the case of memory-resident overlays) is done automatically and is transparent to the program. With manual load, the overlay segments are loaded by specific user calls to a loading routine. Autoload and manual load cannot be mixed in the same task.

Autoload

When a call is made to a subroutine in an overlay segment, an autoload routine takes control before the transfer to the subroutine is made. It checks to find out whether the required segment is already loaded or loaded and mapped. It performs any necessary loading and/or remapping. Following that, the transfer to the called subroutine is made.

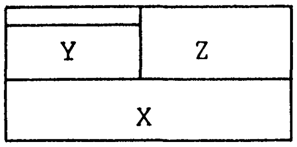
Autoload is path loading, meaning that all segments along the path to the required overlay segment are loaded. For example, in example 2 in the previous section, involving X, Y, U, V, and Z, if a call from segment X is made to subroutine U, both Y and U are loaded. (However, the auto-load routine checks to see if either Y or U is already in memory and if so, the segments are not loaded.)

Autoload is indicated by an asterisk (*) before an overlay specification in an ODL line. An asterisk outside a set of parentheses applies to all levels inside the parentheses.

The advantages of autoload are that it is easy to use and that it does not require changes in the source code. For instance, you could make changes in the ODL commands for the task but you would not have to make any changes in the source code. One disadvantage to autoload is that it increases the size of the segments, since the autoload code plus its data structures must be included in the task. Another disadvantage is that it executes slower than manual load, since the autoload code has to check for whether the required segment is available or not each time an autoloadable segment is called. In addition, autoload must be performed synchronously. See Section 4.1 (on Autoload) in the RSX-11M/M-PLUS Task Builder Manual for more information about autoload.

Examples of autoload:

1. X, the root of a task, calls subroutines Y and Z.



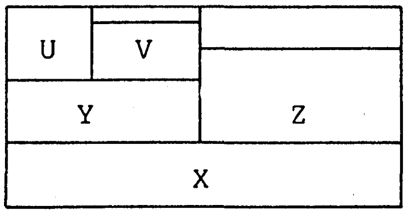
With disk-resident overlays:

```
.ROOT    X-*(Y,Z)
.END
```

With memory-resident overlays:

```
.ROOT    X-*!(Y,Z)
.END
```

2. Using the information from Example 1, Y calls subroutines U and V.



With disk-resident overlays:

```
.ROOT    X-*(Y-(U,V),Z)
.END
```

With memory-resident overlays:

```
.ROOT    X-*!(Y-!(U,V),Z)
.END
```

With some memory resident and some disk resident overlays:

```
.ROOT    X-*!(Y-(U,V),Z)
.END
```

Manual Load

With manual load, you must call the subroutine MNLOAD in the main program or any subroutines to load and/or map any required overlay segment before calling a subroutine in that segment. Additionally, you must keep track of which segments are currently available to avoid a transfer of control to an incorrect segment, and to avoid unnecessary calls to the loading subroutine. Manual load is not path loading. In example 2 of the previous section, if X calls U, it can load just segment U, without loading segment Y, unless that is desired. See Section 4.2 (on Manual Load) in the RSX-11M/M-PLUS Task Builder Manual for more information on manual load.

Manual load is the default loading method. Anytime that a segment is not preceeded by an asterisk (*) in the ODL file, manual load is used.

The advantages of manual load are that smaller overlay segments result, it is usually more run time efficient, and loading of overlay segments can be performed either synchronously or asynchronously. The disadvantages are that the user must keep track of things and that it requires special coding in the source program.

Comparison of a Task With No Overlays, With Disk-Resident Overlays, and With Memory-Resident Overlays

Example 6-1, shown earlier in the module, and repeated below for convenience, shows a main program which calls a subroutine, which in turn calls another subroutine, etc. Note that the sizes shown for the various parts of the task are only approximate.

Main Segment:	PROG
PROG calls:	SUB1, SUB2, SUB3
SUB1 calls:	A, B
SUB2 calls:	none
SUB3 calls:	C, D, E

Segment	Size (in words)
PROG	4K
SUB1	2K
SUB2	3K
SUB3	1K
A	1K
B	2K
C	1K
D	2K
E	1K
Total	17K

Example 6-1 Description of an Overlaid Task

Example 6-2 shows part of the task-build map for the task in Example 6-1 when the task is built with no overlays.

Example 6-3 shows the map when Example 6-1 is built with all disk-resident overlays.

Example 6-4 shows the map when Example 6-1 is built with all memory-resident overlays.

Example 6-2 does not use overlays; therefore no .ODL file is required. Examples 6-3 and 6-4 use overlays; therefore they require a .ODL file. These files are shown along with the map.

Example 6-2 has a root segment but does not have any overlay segments. Note that a single virtual address window maps the entire task. The virtual address limits of the task are ØØØØØØ(8) and 1Ø5357(8), meaning that these virtual addresses are used to reference the task code when it is loaded into memory. The task image is 17792(1Ø) words long; hence 17792(1Ø) words of physical memory are required to load and run the task.

Task-build command:

```
LINK/MAP PROG,SUB1,A,B,SUB2,SUB3,C,D,E,-
LB:[1,1]FOROTS/LIBRARY
```

```
Partition name : GEN
Identification : 01
Task  UIC      : [305,301]
Stack    limits: 000254 001253 001000 00512.
PRG xfr address: 021254
Total address windows: 1.
Task  imase  size  : 17792. words
Task address limits: 000000 105357
R-W disk blk limits: 000002 000107 000106 00070.


*** ROOT SEGMENT: PROG


R/W mem  limits: 000000 105357 105360 35568.
Disk blk limits: 000002 000107 000106 00070.
```

Example 6-2 Map File of Example 6-1 Without Overlays

Example 6-3 with disk-resident overlays, has a root segment, PROG, and eight overlay segments. Note that a single virtual address window maps the entire task when just disk overlays are used; i.e., when no memory resident overlays are used. The overlay description shows the virtual addresses and sizes of the segments. On the right side, the segments are listed, lined up by overlay level. Segments SUB1, SUB2, and SUB3 overlay each other. They all begin at virtual address Ø222ØØ(8), right after the root segment PROG. At various times, virtual addresses starting at Ø222ØØ(8) reference the memory code of the overlay segment which is actually loaded in memory at that time.

Segments A and B overlay each other, beginning with virtual address Ø32234(8), right after SUB1. In a similar way, segments C, D, and E begin at virtual addresses Ø2625Ø(8), right after SUB3. With disk-resident overlays, only virtual addresses ØØØØØØ(8) to Ø42237(8) are used to reference the task in memory, compared to ØØØØØØØ(8) to 1Ø5357(8) without overlays. This task requires only 88ØØ(1Ø) words of memory, compared to 17792(1Ø) words with no overlays.

PROG.ODL file:

```
        .ROOT PROG-L-*(SUB1-L-(A-L,B-L),SUB2-L,SUB3-L-(C-L,D-L,E-L))
    L:  .FACTR LB:[1,1]FOROTS/LIBRARY
        .END
```

Task-build command:

```
         LINK/MAP PROG/OVERLAY_DESCRIPTION
```

Note that LB:[1,1]FOROTS/LIBRARY must be concatenated with each segment in the ODL file. In the remaining examples of ODL files, the concatenation of the library to each segment will not be shown in order to simplify the appearance of the ODL file.

```
Partition name : GEN
Identification : 01
Task  UIC      : [305,301]
Stack     limits: 000260 001257 001000 00512.
PRG xfr address: 021260
Total address windows: 1.
Task  image  size  : 8800. words
Task address limits: 000000 042237
R-W disk blk limits: 000002 000120 000117 00079.


EX63.TSK Overlay description:

Base     Top       Length
----     ---       ------
000000  022177  022200  09344.       PROG
022200  032233  010034  04124.           SUB1
032234  036237  004004  02052.                A
032234  042237  010004  04100.                B
022200  036203  014004  06148.           SUB2
022200  026247  004050  02088.           SUB3
026250  032253  004004  02052.                C
026250  036253  010004  04100.                D
026250  032253  004004  02052.                E
```

Example 6-3 Map File of Example 6-1 With Disk-Resident Overlays

Example 6-4, with memory-resident overlays, also has a root segment, PROG, and eight overlay segments. Notice that three virtual address windows are required for this task, one for the root segment and one for each other overlay level. PROG uses virtual addresses ØØØØØØ(8) to Ø23Ø77(8), slightly more than with Example 6-3. However, segments SUB1, SUB2, and SUB3 begin at virtual address 4ØØØØ(8) corresponding to the next available APR, APR 2, and not right after PROG. This is necessary because the virtual address window must begin with the next APR. Segments A and B begin at 6ØØØØ(8), since the next virtual address window begins with APR3. Segments C, D and E also begin at 6ØØØØ(8) for the same reason. With memory-resident overlays, virtual addresses ØØØØØØ(8) to Ø77777(8) are used and the task requires 18464(1Ø) words in memory. The memory-resident overlay version of the task requires the most virtual memory and also the most physical memory of the three examples.

PROG.ODL file:

```
.ROOT PROG-*!(SUB1-!(A,B),SUB2,SUB3-!(C,D,E))
.END
```

Task-build command:

```
LINK/MAP PROG/OVERLAY_DESCRIPTION
```

```
Partition name : GEN
Identification : 01
Task  UIC      : [305,301]
Stack    limits: 000320 001317 001000 00512.
PRG xfr address: 021320
Total address windows: 3.
Task  image  size  : 18464. words
Task address limits: 000000 077777
R-W disk blk limits: 000003 000122 000120 00080.



EXDOVR.TSK Overlay description:

Base     Top         Length
----     ---         ------
000000  023077  023100  09792.        PROG
040000  050077  010100  04160.             SUB1
060000  064077  004100  02112.                  A
060000  070077  010100  04160.                  B
040000  054077  014100  06208.             SUB2
040000  044077  004100  02112.             SUB3
060000  064077  004100  02112.                  C
060000  070077  010100  04160.                  D
060000  064077  004100  02112.                  E
```

Example 6-4 Map File of Example 6-1 With Memory-Resident Overlays

Table 6-1 refers to Examples 6-2, 6-3, and 6-4.

Table 6-1 Comparison of Overlaying Methods

Method	Task Size	Windows	Advantages and Disadvantages
Non-Overlaid	17792(1Ø) Words of Memory 7Ø(1Ø) Blocks on Disk 1Ø536Ø(8) Virtual Addresses Used	1	Advantages Smallest task size on disk Fastest execution Simplest to develop Disadvantages Maximum task size 32K words Task smaller than 32K words but too large for partition or for available space in partition
Disk-Resident	88ØØ(1Ø) Words of Memory 79(1Ø) Blocks on Disk 42238(8) Virtual Addresses Used	1	Advantages Uses the smallest amount of physical memory Uses the least amount of virtual address space Disadvantages Slowest execution time; overlay segments loaded from disk when needed
Memory-Resident	18464(1Ø) Words of Memory 8Ø(1Ø) Blocks on Disk 1ØØØØØ(8) Virtual Addresses Used	3	Advantages Faster execution than disk-resident over-lays Task resident in memory at one time Disadvantages Uses the most memory and disk space May waste virtual address space Requires space in memory to hold the entire task

Table 6-1 gives a comparison of the three overlaying methods. In addition to the various sizes, it also lists the advantages and disadvantages of each approach. It is also possible to build this task with memory-resident overlays for the first level (SUB1,SUB2 and SUB3) and disk-resident overlays for one or both of the second levels (A and B; or C, D and E).

LIBRARIES

Object libraries, when used, must be specified in the .ODL file. The one exception is the default system library LB:[1,1]SYSLIB.OLB, which is searched automatically for the root and each overlay segment. To allow inclusion of any needed libraries, just specify the library with the /LB qualifier (as in MCR format for TKB). To force the inclusion of a specific module from a library, use the /LB:module form of the /LB qualifier.

Examples:

1.

```
                .ROOT     MAINPG-MYLIB1/LB-LIB-(SUBA,SUBB,CPART)
        CPART:  .FCTR     SUBC1-(SUBC2,SUBC3)
        LIB:    .FCTR     MYLIB2/LB
                .END
```

Explanation: Include all needed modules from MYLIB1.OLB and from MYLIB2.OLB that are referenced in the root segment MAINPG.

2.

```
                .ROOT     MAIN-MYLIB1/LB:MOD4-MYLIB1/LB-(A,B)
                .END
```

Explanation: Include the module MOD4 from MYLIB1.OLB. In addition, the second MYLIB1/LB with no modules listed, causes the inclusion of any other modules from MYLIB1.OLB that are referenced in the root segment MAIN.

Note that if you reference additional library routines from other segments, they will not get resolved properly unless you specify the library again in each referencing overlay segment.

3. Including the FORTRAN OTS Library:

```
                .ROOT     MAIN-LIBRA-(A-LIBRA,B-LIBRA)
       LIBRA:   .FCTR     LB:[1,1]FOROTS/LB   or F4POTS/LB
                .END
```

Include needed modules from FOROTS.OLB (or F4POTS.OLB) in the root segment, in segment A, and in segment B. Notice that you should specify the library in each segment which might need it. Otherwise, if segment A needs a module not already included for the root segment, the library is not searched again for module A unless it is specified again in overlay segment A.

Note that in an installation which makes heavy use of FORTRAN, the appropriate FORTRAN OTS library may have been included in SYSLIB making it unnecessary to include the OTS library in the TKB command. Check with your system manager to see if the OTS library is included in SYSLIB.

Example of Duplicate Code in Overlays

In the above example with a root and two overlay segments, A and B, it is possible that duplicate code will be forced into the two segments. If A and B both need module X from the library, and the root does not need X, then a copy of X would be placed in both segment A and in segment B. This adds to the size of segments A and B but keeps the size of the root smaller. If the size of the root is critical, you may be willing to have the duplicate code appear in A and B. If the size of the root is not critical, force X to be in the root by the following ODL statement:

```
           .ROOT MAIN-LB:[1,1]FOROTS/LB:X-LIB-(A-LIB,B-LIB)
   LIB:    .FCTR LB:[1,1]FOROTS/LB
           .END
```

In general, it is good practice to include a library reference in each segment of the task. If you are concerned with the possibility of duplicate code, you can use the trial and error approach wherein you specify the library only in the root and then note the unresolved symbols that occur. Once you determine from the TKB map which modules are needed in which segments, you can then determine if you want to place certain modules in the root or if you are willing to have duplicate code in various segments.

Duplicate code can also be included from SYSLIB, the default library. If you wish to use the trial and error method on modules from SYSLIB, use the /LONG qualifier in the LINK command (/MA in TKB format). This qualifier causes the Task Builder to list modules included from SYSLIB in the map file.

Note that in the previous example, if X had been required in the root, duplicate code in the overlay segments would not be generated; all references to X would be resolved via the root.

An Overlay Example

Example 6-5 is a simple task with a root segment ROOT and 2 overlay segments, P and Q. During the execution of the task, the following calling sequence is used:

ROOT calls P
ROOT calls Q

Figure 6-5 shows an overlay tree and a memory allocation diagram for this task.

The code for Example 6-5 is separated into three different modules, one for each segment. The source file for the root segment ROOT contains the startup code and controls the overlay loading by calls to the subroutines. The source file for each overlay segment, P and Q, contains the subroutine code.

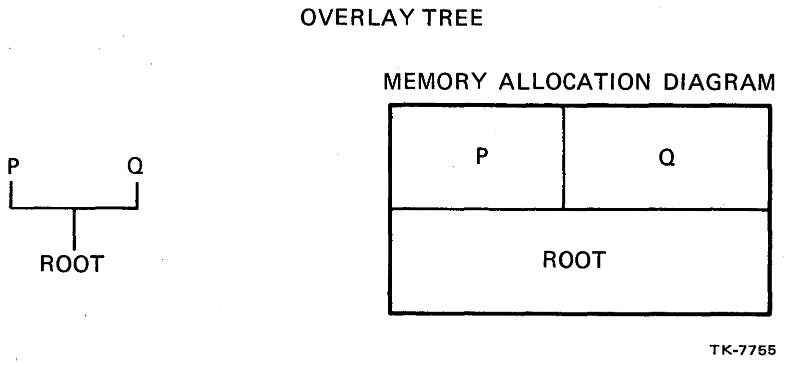


Figure 6-5 Task with Two Overlay Segments

Steps in Program Development for Example 6-5

1. Compile each module.

```
>FORTRAN/LIST ROOT
>FORTRAN/LIST P
>FORTRAN/LIST Q
```

2. Use the editor to create the overlay descriptor file FEXDOVR.ODL for disk-resident overlays.

```
        .ROOT ROOT-LIB-*(P-LIB,Q-LIB)
  LIB:  .FCTR LB:[1,1]FOROTS/LB
        .END
```

3. Task-build using the .ODL file as the input file.

```
>LINK/MAP EXDOVR/OVERLAY_DESCRIPTION
```

LEARNING ACTIVITY

1. To build the above task with memory-resident overlays, how would you modify the .ODL file?

2. To build the above task without overlays, what task-build command would you use?

The following notes are keyed to Example 6-5.

1. On initial load only the root segment ROOT is loaded.
2. With autoload the call to subroutine P causes the autoload routine to load overlay segment P from disk and then transfer control to the subroutine.
3. Subroutine P displays a message and returns.
4. The call to subroutine Q causes the autoload routine to load overlay segment Q from disk over segment P and then transfer control to the subroutine.
5. Subroutine Q displays a message and returns.

If another call were added to subroutine Q, the autoload routine would check and see that overlay segment Q is already loaded and would then just transfer control to Q. If another call were added to subroutine P, the autoload routine would check and see that overlay segment P is not loaded. Hence, it would load segment P over segment Q and then transfer control.

```
          PROGRAM ROOT
C
C FILE ROOT.FTN
C
C This task calls each of the subroutines P AND Q
C
C Task-build instructions: Use FEXDOVR.ODL as the input
C file.
C
          WRITE (5,50)              ! Display message
❶ 50      FORMAT (' THE ROOT SEGMENT IS NOW RUNNING AND
          1WILL CALL P.')
❷         CALL P
          WRITE (5,150)             ! Display message
150       FORMAT (' THE ROOT SEGMENT WILL NOW CALL Q.')
❹         CALL Q
          WRITE (5,250)             ! Display message
250       FORMAT (' THE ROOT SEGMENT WILL NOW EXIT.')
          CALL EXIT                 ! Exit
          END


          SUBROUTINE P
C
C FILE P.FTN
C
C This subroutine displays a message and then returns
C
❸         WRITE(5,50)               ! Display message
50        FORMAT (' SEGMENT P IS NOW LOADED. SUBROUTINE P
          1IS EXECUTING.')
          RETURN                    ! Return
          END


          SUBROUTINE Q
C
C FILE Q.FTN
C
C This subroutine displays a message and then returns
C
❺         WRITE(5,50)               ! Display message
50        FORMAT (' SEGMENT Q IS NOW LOADED. SUBROUTINE Q
          1IS EXECUTING.')
          RETURN                    ! Return
          END
```

Run Session

```
>RUN EXDOVR
THE MAIN SEGMENT IS RUNNING AND WILL CALL P.
SEGMENT P IS NOW LOADED. SUBROUTINE P IS EXECUTING.
THE MAIN SEGMENT WILL NOW CALL Q.
SEGMENT Q IS NOW LOADED. SUBROUTINE Q IS EXECUTING.
THE MAIN SEGMENT WILL NOW EXIT.
>
```

Example 6-5 A Task with Two Overlay Segments

Changing Example 6-5 to Manual Load

To change the previous example to manual load, the source code in ROOT must be modified to include the calls to subroutine MNLOAD which will cause the loading of the segments. The ODL file must also be modified to remove the autoload indicator (*). The files MLROOT.FTN and MLEXDOVR.ODL on the tape provided with this course are modifications of ROOT.FTN and EXDOVR.ODL. Check UFD [2Ø2,3] for these files. See your course administrator if you have difficulty finding these files.

GLOBAL SYMBOLS IN OVERLAID TASKS

When the Task Builder builds a task, each reference to a subroutine is an unresolved symbol reference which must ultimately be resolved by finding a corresponding subroutine or by finding an entry in the system library. (Each subroutine generates a global symbol definition which can be used to resolve an unresolved global reference symbol.) If no such subroutine or entry in the system library is found, the global symbol is unresolved.

The scope of a global symbol is controlled by the overlay structure. A module can only refer to a global symbol defined on a path which passes through it. Thus, in Figure 6-6, the reference to global symbol R (global symbol and subroutine are used synonymously in this discussion) in segment A1 is undefined because R is not defined in either AØ or CNTRL. AØ and CNTRL form the only path passing through A1. The definition in A2 can't be used because A1 and A2 overlay one another.

In a single segment task with no overlays the same global symbol cannot be defined more than once, or it is multiply defined. With the rules governing global symbols in overlays, however, the same name can be used for two different global symbols as long as they follow these two restrictions:

1. They must be defined on separate paths. Each reference is resolved to the definition on its own path. Only if the same symbol is defined more than once on the same path, is it multiply defined.

2. The two symbols must not be referenced from a segment closer to the root which has paths through both segments. An example is a root segment which references a subroutine N. If the root segment has two overlay segments U and V and each one defines the subroutine N, the Task Builder can't tell which subroutine N to use. Therefore, the reference is ambiguous, since there are several possible ways to resolve the reference.

Figure 6-6 shows an example overlay tree with a number of global symbol definitions. The various references are resolved as follows:

Q is defined in AØ and BØ	Reference in A22 resolved in AØ Reference in A1 resolved in AØ Reference in B1 resolved in BØ
R is defined in A2	Reference in A22 resolved to A2 Reference in A1 undefined Reference in CNTRL resolved to A2 (if autoload, through an autoload vector)
S is defined in AØ and BØ	Reference in A1 resolved to AØ Reference in A21 resolved to AØ Reference in A22 resolved to AØ Reference in B1 resolved to BØ Reference in B2 resolved to BØ Reference in CNTRL ambiguous
T is defined in AØ and A21	Symbol multiply defined

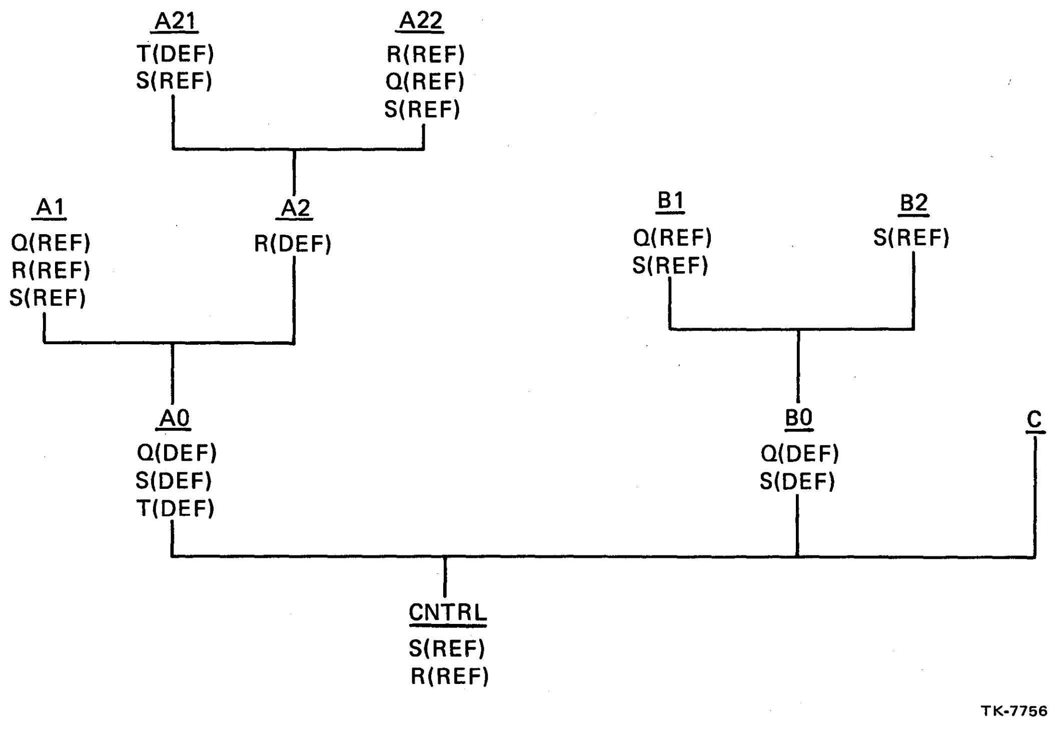


Figure 6-6 Resolution of Global Symbols

Data References in Overlays

Data local to an overlay segment is only available while the segment is loaded. When the segment is overlaid by another segment, any updating of local data that had been made while the segment was loaded will be lost. The next time the same segment is loaded from the disk, the original data values will be brought into memory. For this reason it is strongly recommended that data required by more than one segment be placed in the root.

If you wish to share data between overlay segments, you must use FORTRAN COMMON or pass arguments in the CALL (discussed below). Note that if you want to share data between overlay segments A and B, and if updating of the data can be done by either segment, it is not sufficient to simply place the COMMON in A and B; it must also be placed in the root segment.

By placing the same COMMON in the root, you are assured that A and B will always be referring to the same data in the COMMON since the root segment is always loaded. In FORTRAN-77 another way to place a COMMON in the root is to use the FORTRAN SAVE common-name statement in one of the segments. This will force the task-builder to place the named common in the root. The .PSECT ODL statement can also be used to force the placement of a common in the root segment.

Another way of sharing small amounts of data between two overlays is to have the data passed from the root to each overlay as an argument to the CALL. If the segment changes one of the data values passed as an argument, it will then be changed in the root segment. The changed value can then be passed to the next overlay, etc.

Example 6-6 is a more complex example of the use of overlays. The program calling sequence is as follows:

```
    MAIN calls A
    A calls JOB1 or JOB2 (in module JOBXX)
    MAIN calls B
    Loop through three time
        MAIN calls A
        A calls JOB1 or JOB2
    End of loop
    MAIN calls TOTAL (in the root segment)
```

The following notes are keyed to Example 6-6.

1. Task-build instructions.

2. COMMON OTHER is defined in the root segment MAIN, and is referred to in overlay A and in overlays JOB1 and JOBXX. The entire allocation of space for OTHER is in MAIN; no space is reserved for OTHER in the overlays.

   The use of the COMMON OTHER by the MAIN segemnt and the the overlay segments allows the overlays to access data provided by MAIN and to pass a result back to MAIN via the fourth argument in OTHER. This argument is called variously ANS in MAIN, ARG(4) in overlay A, SUM in overlay JOB1 and ANS in JOBXX.

3. COMMON TOTCOM is also defined in MAIN and is referenced in overlays JOB1 and JOBXX. Allocation for TOTCOM is in MAIN. Subroutine TOTAL displays the grand total, which has been accumulated in TOTCOM in variable TOT, but the subroutine does not refer to COMMON TOTCOM. Since MAIN passes the argument TOT to subroutine TOTAL, the subroutine does not have to use TOTCOM. This illustrates how shared data may be passed between overlay segments via the argument list.

4. Note that subroutine A calls JOB2, which is the name of the subroutine, and that the ODL file uses JOBXX which is the file name. File names are always used in ODL; not subroutine names. In general, file names and subroutine names should be the same simply to avoid confusion.

5. Note that neither COMMON OTHER or COMMON TOTCOM appear in segment B since the segment does not refer to any variables in either COMMON.

6. Argument TOT is is COMMON TOTCOM. Since the argument is passed to subroutine TOTAL, TOTAL does not need a reference to COMMON TOTCOM.

```
      PROGRAM MAIN
C
C FILE MAIN.FTN
C
C This program prints a message and then calls subroutine
C A. Subroutine A asks whether to perform job 1 or job 2.P
C It then calls either subroutine JOB1 or JOB2 which
C performs the operation and displays the results. MAIN
C then calls subroutine B which displays a message. MAIN
C then calls subroutine A 3 more times, keeping a grand
C total of the operations. Finally, it displays the
C grand total and exits.
C
❶ C Task-build instructions: Use FMRMAIN.ODL as the input
C file.
C
      COMPLEX DUMMY(1024)     ! Leave space to make
C                             !  segment larger
    ❷ COMMON /OTHER/OP1,OP,OP2,ANS
      INTEGER OP1,OP,OP2,ANS
      DATA OP1,OP2/5,2/
C
    ❸ COMMON /TOTCOM/TOT
      INTEGER TOT             ! Total
C
      TYPE *,'THE MAIN SEGMENT IS RUNNING AND WILL
     1CALL A'
      CALL A                  ! Call subroutine A
      TYPE *,'THE MAIN SEGMENT WILL NOW CALL B'
    ❺ CALL B                  ! Call subroutine B
      DO 10, I=1,3
      TYPE *,'THE MAIN SEGMENT WILL NOW CALL A'
      ANS = 0                 ! Clear answer in case
C                             !  of no operation
10    CALL A                  ! Call subroutine A
      TYPE *,'THE MAIN SEGMENT WILL CALL TOTAL'
    ❻ CALL TOTAL(TOT)         ! Call routine to
C                             !  display grand total
      TYPE *,'THE MAIN SEGMENT WILL NOW EXIT'
      CALL EXIT               ! EXIT
      END
```

Example 6-6 Complex Example Using Overlays
(Sheet 1 of 4)

```
      SUBROUTINE A
C
C FILE A.FTN
C
C This subroutine displays a message and then asks which
C of two jobs to do. It calls the appropriate subroutine
C to do the job, displays the results, and then returns
C to the main program
C

    ❷ COMMON /OTHER/ARG
      INTEGER ARG(4)
      INTEGER BUFF
C
      TYPE 1
1     FORMAT (T8,'SEGMENT A IS NOW LOADED. SUBROUTINE
      1 A IS EXECUTING.')
      TYPE 2
2     FORMAT ('$',T8,'DO YOU WANT TO DO JOB 1 OR JOB 2
      1? ')
      ACCEPT 3,BUFF
3     FORMAT (I6)
      IF (BUFF.NE.1) GOTO 10   ! Is it job 1?
      CALL JOB1                ! Call subr to do job 1
      GOTO 20                  ! Branch to display code
10    IF (BUFF.NE.2) GOTO 1000! Is it job 2?
    ❹ CALL JOB2                ! Call subr to do job 2
20    TYPE 21,ARG
21    FORMAT (T8,I2,1X,A2,I2,' = ',I3/)
      GOTO 2000
1000  TYPE 1001
1001  FORMAT (T8,'NO SUCH JOB. SORRY.')
2000  RETURN                   ! Return
      END
```

Example 6-6 Complex Example Using Overlays
(Sheet 2 of 4)

```
      SUBROUTINE JOB1
C
C FILE JOB1.FTN
C
C This subroutine performs an addition operation. The
C operands, operator, and sum are held in one common
C block, and the total in another.
C
    ❷ COMMON /OTHER/NUM1,OPRATR,NUM2,SUM
      INTEGER NUM1,OPRATR,NUM2,SUM
    ❸ COMMON /TOTCOM/TOT
      INTEGER TOT
C
      INTEGER DUMMY(1024)       ! Leave space to make
C                               !  module larger
C
      TYPE 1                    ! Display message
1     FORMAT (T16,'SEGMENT JOB1 IS NOW LOADED.',
      1/,T16'SUBROUTINE JOB1 IS EXECUTING.')
      SUM = NUM1 + NUM2         ! Calculate sum
      TOT = TOT + SUM           ! Add to grand total
      OPRATR = '+'              ! Move operand for
C                               !  output display
      RETURN
      END


    ❹ SUBROUTINE JOB2
C
C FILE JOBXX.FTN
C
C This subroutine performs a multiplication operation.
C The operands, operator, and product are held in one
C common block, the running total in another.
C
    ❷ COMMON /OTHER/OP1,OPRATR,OP2,ANS
      INTEGER OP1,OPRATR,OP2,ANS
    ❸ COMMON /TOTCOM/TOT
      INTEGER TOT
      REAL DUMMY(1024)          ! Leave space to make
C                               !  module larger
      TYPE 1                    ! Display message
1     FORMAT (T16,'SEGMENT JOBXX IS NOW LOADED.',
      1/,T16,'SUBROUTINE JOB2 IS EXECUTING.')
      ANS = OP1 * OP2           ! Calculate product
      TOT = TOT + ANS           ! Add this to grand total
      OPRATR = '*'              ! Move operand for
C                               !  output display
      RETURN
      END
```

Example 6-6 Complex Example Using Overlays
(Sheet 3 of 4)

```
        ❺ SUBROUTINE B
C
C FILE B.FTN
C
C This subroutine displays a message and returns
C
          TYPE 1
1         FORMAT (T8,'SEGMENT B IS NOW LOADED. SUBROUTINE
          1B IS EXECUTING.')
          RETURN
          END

        ❻ SUBROUTINE TOTAL (TOT)
C
C FILE TOTAL.FTN
C
C Subroutine to display grand total. The grand total
C location is passed as a subroutine argument
C
          INTEGER TOT
          TYPE 1,TOT
1         FORMAT ('  THE GRAND TOTAL IS ',I3,'.'/)
          RETURN
          END
```

Run Session

```
>RUN MRMAIN
THE MAIN SEGMENT IS RUNNING AND WILL CALL A
          SEGMENT A IS NOW LOADED. SUBROUTINE A IS EXECUTING.
          DO YOU WANT TO DO JOB 1 OR JOB 2? 1
                    SEGMENT JOB1 IS NOW LOADED.
                    SUBROUTINE JOB1 IS EXECUTING.
          5 + 2 =  7

THE MAIN SEGMENT WILL NOW CALL B
          SEGMENT B IS NOW LOADED. SUBROUTINE B IS EXECUTING.
THE MAIN SEGMENT WILL NOW CALL A
          SEGMENT A IS NOW LOADED. SUBROUTINE A IS EXECUTING.
          DO YOU WANT TO DO JOB 1 OR JOB 2? 2
                    SEGMENT JOBXX IS NOW LOADED.
                    SUBROUTINE JOB2 IS EXECUTING.
          5 * 2 = 10

THE MAIN SEGMENT WILL NOW CALL A
          SEGMENT A IS NOW LOADED. SUBROUTINE A IS EXECUTING.
          DO YOU WANT TO DO JOB 1 OR JOB 2? 2
                    SEGMENT JOBXX IS NOW LOADED.
                    SUBROUTINE JOB2 IS EXECUTING.
          5 * 2 = 10

THE MAIN SEGMENT WILL NOW CALL A
        SEGMENT A IS NOW LOADED. SUBROUTINE A IS EXECUTING.
        DO YOU WANT TO DO JOB 1 OR JOB 2? 1
                SEGMENT JOB1 IS NOW LOADED.
                SUBROUTINE JOB1 IS EXECUTING.
         5 + 2 =  7

THE MAIN SEGMENT WILL CALL TOTAL
THE GRAND TOTAL IS  34.

THE MAIN SEGMENT WILL NOW EXIT
>
```

Example 6-6 Complex Example Using Overlays
(Sheet 4 of 4)

LEARNING ACTIVITY (Using Example 6-6)

1. **Draw an overlay tree or a memory allocation diagram.** Since the questions below assume a particular overlay structure, check your answer before doing questions 2 through 4.

2. **What** .ODL file would you use for autoload and all disk-resident overlays?

3. **What** .ODL file would you use for autoload and all memory-resident overlays?

4. **What** .ODL file would you use for autoload and A and B memory-resident and JOB1 and JOBXX disk-resident?

CO-TREES

Sometimes there are subroutines which must be callable from several or all different overlay segments in a task. One solution is to place the subroutines in the root. Since they are always loaded, they are then available from the root and all overlay segments. If this causes the task to become too large and the subroutines are logically independent (don't call each other), another solution is available. You can set up a separate overlay area and place the subroutines in it so that they overlay each other.

For example, Figure 6-7 shows an overlaid task with subroutines X and Y in the root. They are placed there so that the root and every other segment can call them. If this makes the task too large, set up a separate overlay area and place X and Y in it so they overlay each other (Figure 6-8). X and Y are in a separate overlay area, therefore, they can overlay each other and still be called from the root and every other segment in the task.

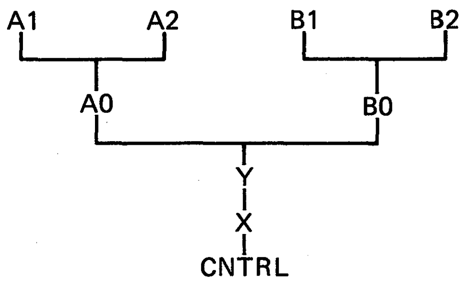
The two overlay areas, the main one and the separate one for the extra subroutines, are defined by a multiple tree structure. The tree for the main code is called the main tree and the other one is called a co-tree. The co-tree root may contain code but it does not have to. In the example in Figure 6-8, the root of the co-tree is null (or is a dummy root) and contains no code. A root is needed to set up the overlay structure. Only the root of the main tree is loaded on initial load. The co-tree roots are loaded when they are first needed and remain loaded after that. Other than that, loading of overlay segments works just like a single-tree overlay structure.

The .ODL files are listed above the files for the task without co-trees and with co-trees. The co-trees are separated in the .ODL file by a comma. With autoload, an asterisk (*) should be specified on the co-tree roots as well as in the normal places. This is necessary because the co-tree roots are loaded like overlay segments the first time they are needed. Also, note that the .NAME directive is used to specify that CNTRL2 is just a name for the null root segment of the co-tree.

For additional information on co-trees and an example, see Section 3.5 (on Multiple-Tree Structures) in the <u>RSX-11M/M-PLUS Task Builder Manual</u>. In particular, note the use of the /NOFU or /FU switch used with TKB.

.ODL File with no co-trees:

```
.ROOT   CNTRL-X-Y-*(AØ,(A1,A2),BØ-(B1,B2))
.END
```



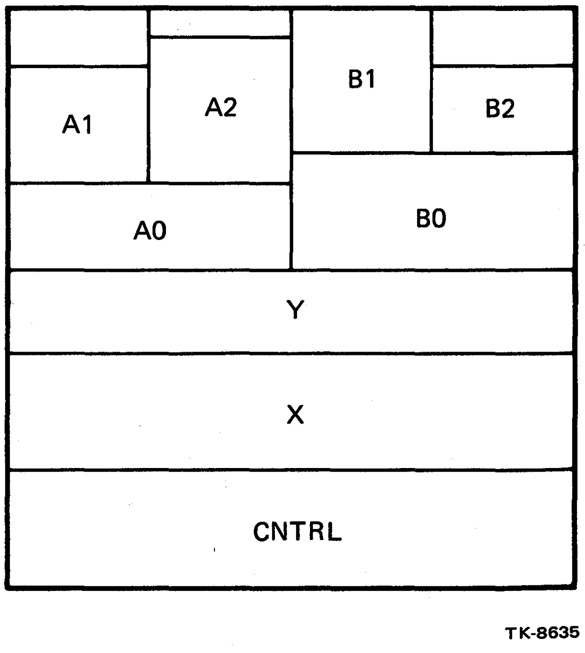


Figure 6-7 Task Without Co-Trees

```
.ODL File with Co-Trees
     .NAME   CNTRL2
     .ROOT   CNTRL-*(AØ-(A1,A2),BØ-(B1,B2)), *CNTRL2- *(X,Y)
     .END
```

The segment CNTRL2 is a dummy root used for loading purposes only.

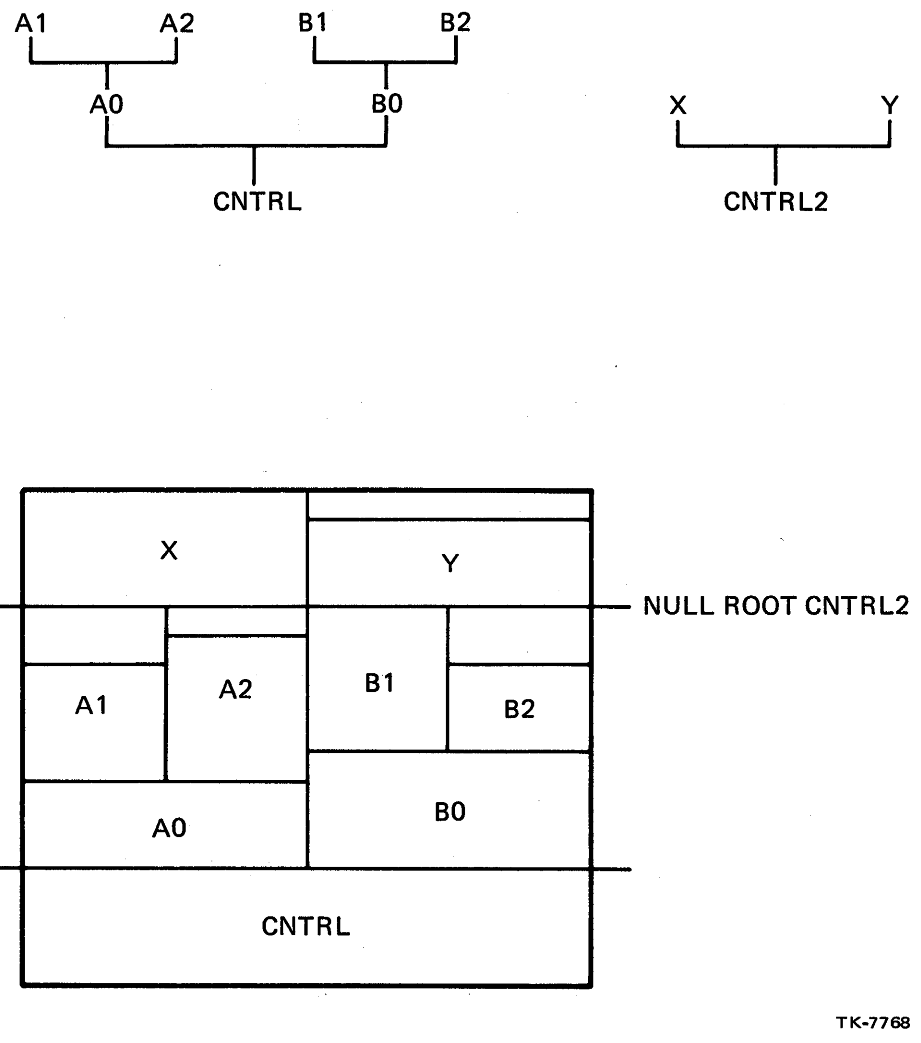


Figure 6-8 Use of Co-Trees

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 in that book or in on-line files.

If you think that you have mastered the material, ask your course administrator to record your progress in 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.

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.

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.

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 for any tasks which use the region. For a task using the region, the virtual addressing limit applies to the total of all regions used plus the task's code.

Static regions also 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 (must be writen in MACRO-11 but can be used in a FORTRAN CALL)	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 by 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, either the user did not specify an APR, or APR 6 was specified. When task B was built, the user specified APR 2.

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) through 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.

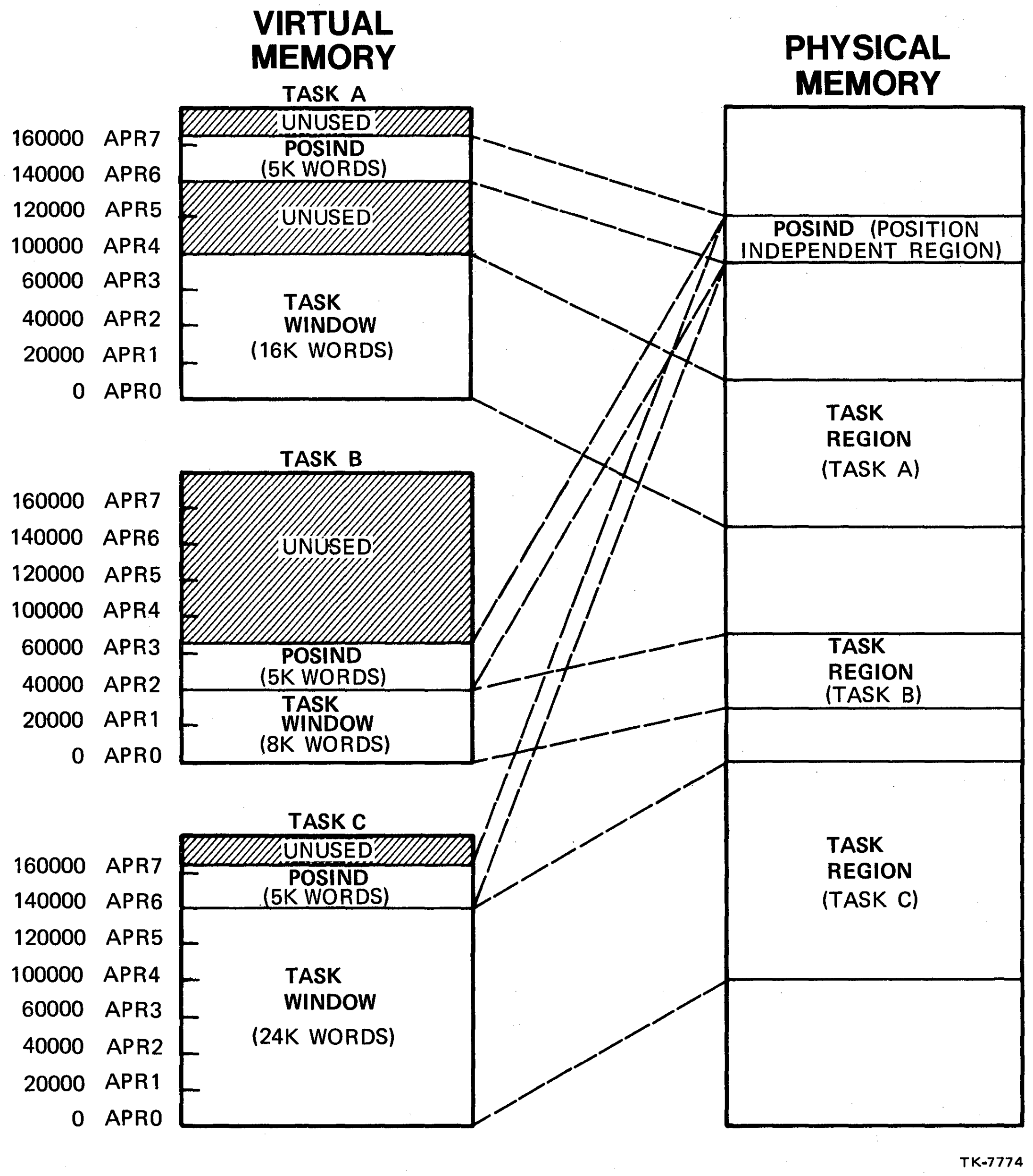


Figure 7-1 Tasks Using a Position Independent Shared Region

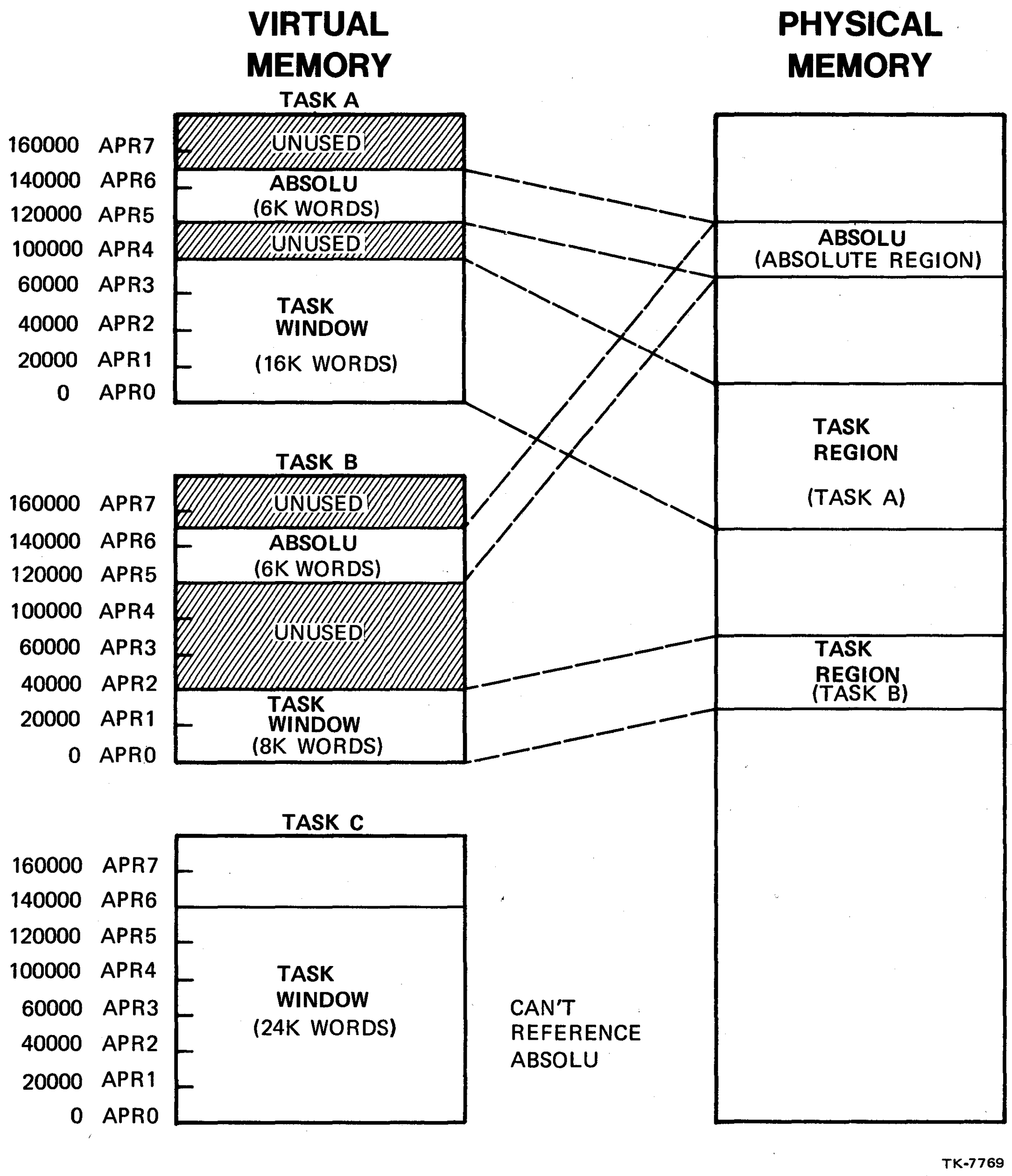


Figure 7-2 Tasks Using an Absolute Shared Region

Because of the added flexibility of a position independent region, i.e., any APR can be used to map the region, it might seem that there is no reason to ever use an absolute region with its attendant APR restrictions. However, there are coding restrictions for position independent regions which require the use of highly specialized coding techniques. Because of these restrictions, the decision to create a position independent or an absolute region is usually based on these coding restrictions rather than on flexibilty alone.

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. Compile 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 the linkage to the region.

After task-building the shared region, task-build the referencing task. It can be written and compiled 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. FORTRAN COMMON is used for this purpose.
- Subroutine call to a subroutine defined in a shared region.

PROCEDURE FOR CREATING SHARED REGIONS AND REFERENCING TASKS

Creating a Resident Common

1. Code the shared region. Typically consists of a COMMON statement and DATA statements which allow you to initialize the COMMON.

2. Choose position independent for a resident common.

3. Compile the shared region.

4. 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.

5. Task-build the shared region.

   - Symbol definition file (.STB) required.

   - Build position independent and /SHAREABLE:COMMON. This causes the Task Builder to include the COMMON names in the .STB file so that references to them in the referencing task are properly resolved. The /SHAREABLE:LIBRARY switch used in task-building resident libraries causes the COMMON (Psect for MACRO) names to be omitted from the .STB file. This avoids task-builder errors in the case of unintentional duplication of Psect names.

   - Use required switches and options (see Table 7-2).

6. 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.

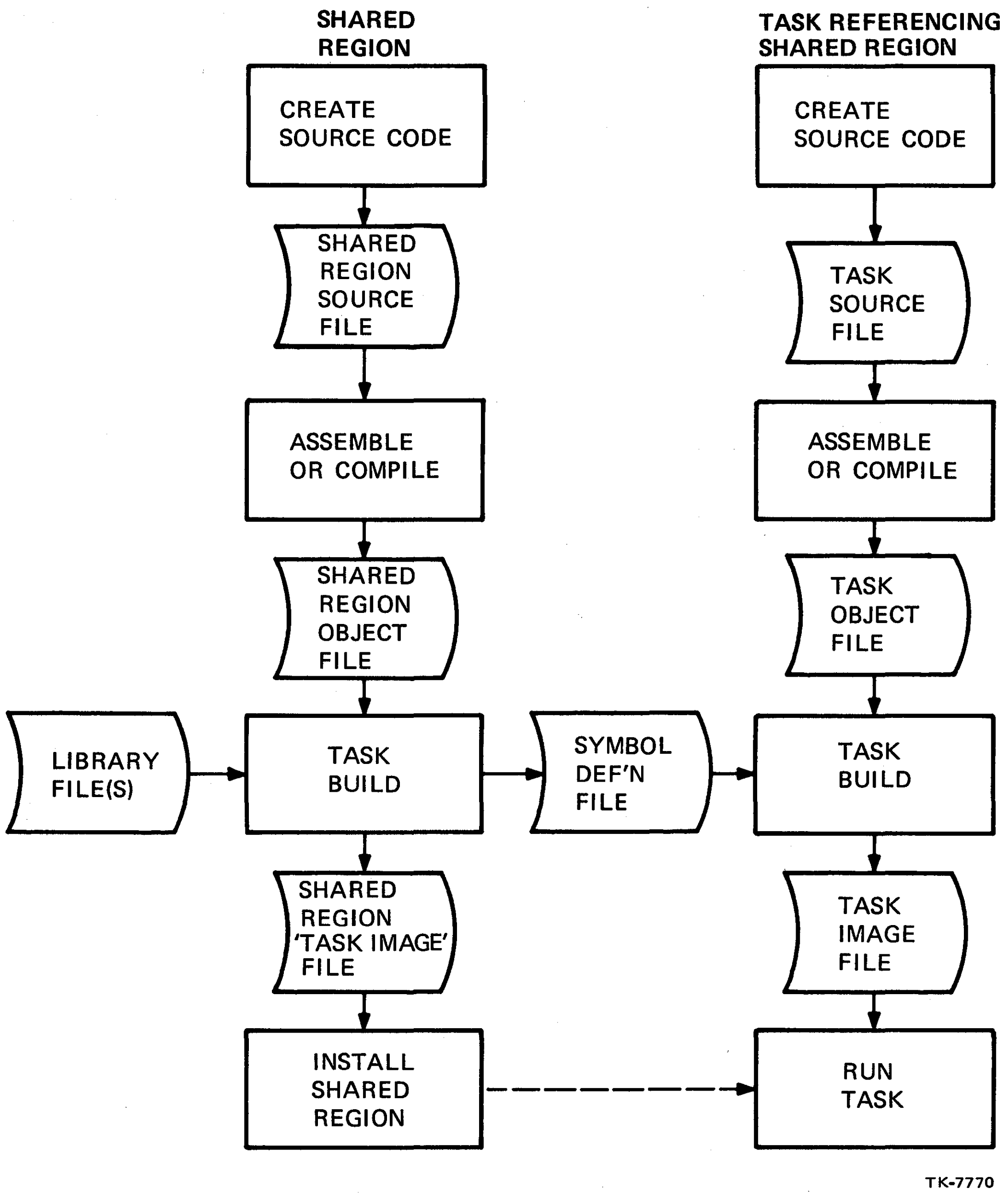


Figure 7-3 Program Development for Shared Regions

The required switches and options in Table 7-2 are needed for various 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, that value is used as a maximum, for length checking. A task-builder error results if the length of the region is longer than the length specified. If you specify a length of Ø, the region is set up with the size needed for the code, so long as it doesn't exceed the normal 32K word virtual addressing limit.

Table 7-2 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, Ø or maximum)	PAR=GEN If base and length not specified, information 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 len, use base=Ø, length=Ø or max

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

1. Code the shared region.

   See COMWP.FTN in Example 7-1. The following note is keyed to the example:

   ❶ Create the FORTRAN named COMMON, MYDATA, and put data into the array I.

2. Compile the shared region.

```
>FORTRAN/LIST COMWP
```

3. If necessary, create the common type partition.

   We will make a partition COMWP, eight blocks = 1000(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 2 zeroes.  (For example, base
!address 123400(8)=1234 blocks,
!length=20000(8) bytes = 200(8) blocks)
!  Eliminate the partition TSTPAR
>SET NOPARTITION:TSTPAR
!  Create the partition COMWP
>SET PARTITION:COMWP/BASE:1234/SIZE:10/COMMON
!  Check to see if this worked correctly
>SHOW PARTITIONS
```

   Later, to eliminate the partition and to replace TSTPAR, use these 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,LB:[1,1]FOROTS/LIB
Option? STACK=Ø
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 with FORTRAN COMMON). /NOHEADER indicates that no task header be included in the task image since this is not an executable task. /SYMBOL_TABLE indicates that a .STB file be created. (COMWP.STB). /CODE:PIC indicates a position independent region. STACK=Ø indicates no stack space is needed since this is not an executable task. PAR=COMWP indicates the partition is COMWP. The Task Builder gets the length (for a maximum check) from the partition on the system.

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.

```
        BLOCK DATA COMWP
C
C File COMWP.FTN
C
C Program to create and initialize a resident common
C
C Task-build instructions: Must include /SHAREABLE:COMMON
C and /NOHEADER switches; STACK=0 and PAR=COMWP options.
C Must create .STB file. May be /CODE:PIC or absolute
C (the default). OTS library NOT required.
C
    ❶  COMMON /MYDATA/ I(256)
        DATA I /128*5,128*10/
        END


        PROGRAM COMGP
C
C File COMGP.FTN
C
C Task to read data from a static region and print it
C out at TI:. It uses a COMMON to reference the data.
C
C Task-build instructions:
C
C       LINK/MAP/OPTION COMGP,LB:[1,1]FOROTS/LIBRARY
C       Option? RESCOM=COMWP/RO
C       Option? <RET>
C
    ❶  COMMON /MYDATA/ L(256) ! Common to reference
C                             !        shared region
C Loop through to display region, 8 numbers on a line
        DO 50 J = 1,249,8
        WRITE (5,10) (L(K),K=J,J+7) ! Write values
10      FORMAT (' ',I2,7I8)
50      CONTINUE
        CALL EXIT
        END
```

Example 7-1 Resident Common Referenced with FORTRAN COMMON (Sheet 1 of 2)

```
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 FORTRAN COMMON (Sheet 2 of 2)

Creating a Referencing Task

1. Code the task, using the FORTRAN COMMON used in creating the region.

2. Compile the task.

3. Task-build the task.

   - Specify shared regions by using one of the following options:

     - RESCOM=common name - for a user resident common. The .STB and .TSK files may be on any device and in any UFD, using normal defaults.

       Append /RO or /RW for read-only or read-write access.

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

       Append :RO or :RW for read-only or read/write access.

       (Note that a colon (:) is used for COMMON and a slash (/) is used for RESCOM when appending the RO or RW switches.)

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.FTN 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.FTN in Example 7-1. The following note is keyed to the example:

   ❶ The same FORTRAN named COMMON, MYDATA, is used here as in COMWP.FTN to set up referencing.

2. Compile 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
   ```

Accessing a Region for Read-Only or Read/Write

Whether read-only or read/write access 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, read-only access is required.

However, when QIOs are issued and the buffer is in the shared region, the situation is more involved. 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 (IO.SPR) for illegal user buffer. This condition does not cause errors in the example because FORTRAN WRITEs create the output string in a buffer within the referencing task area and the QIOs do the writes from the referencing task area. However, if you issue QIOs directly, the above problem can exist.

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.

CREATING AND REFERENCING A SHARED LIBRARY

Example 7-2 contains a shared library, LIB.MAC, and a referencing task USELIB.FTN. The program LIB.MAC and the associated comments are included to illustrate how a MACRO program can be called from a FORTRAN program. Some knowledge of MACRO-11 is required to have a full understanding of the example. The FORTRAN user need only know the order of the arguments in the CALL in order to use these subroutines.

The shared library contains four simple arithmetic routines to add, subtract, multiply, and divide two numbers. They are all written to be reentrant and, in addition, they are written so that they can be called from a FORTRAN program with a standard FORTRAN subroutine call.

```
          INTEGER OP1,OP2,ANS
          CALL AADD(OP1,OP2,ANS)
```

The argument list is set up as follows:

```
         *******************************
         *     R5      *    COUNT=3    *    word, word
         *******************************
         *       address of OP1        *     longword
         *******************************
         *       address of OP2        *     longword
         *******************************
         *       address of ANS        *     longword
         *******************************
```

Note that subroutines written in FORTRAN cannot be included in a resident library because the code generated by FORTRAN is not reentrant. 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(1Ø) and 2(1Ø), and displays just the answers for the four operations. The following notes are keyed to Example 7-2.

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

❷ 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.

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

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

❺ 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.

❻ Task-build instructions needed to tie the task to the library.

Task-Building the Shared Library and the Referencing Task

The instructions for task-building the library and the referencing task are included in Example 7-2; however one point should be emphasized.

When Task Building the library, you must use the /SHAREABLE:LIBRARY switch to avoid task-builder errors when building the referencing task. Whether the library is to be a system resident library or a user resident library is determined strictly by where the .STB and the .TSK file for the library reside. If they are in LB:[1,1], the library is a system resident library. If the .STB and .TSK files exist in other than LB:[1,1], the library is a user resident library.

When task building a referencing task, the option (not switch) RESLIB=library name or LIBR=library name must be used. If the option LIBR is used, the search for the library will be done only in UFD LB:[1,1]. If the option RESLIB is used, the search for the library will be done on the default device and UFD, or on the device and UFD specified with the library name; for example:

```
>LINK/OPTIONS/MAP COMPG
Option?  RESLIB=DB2:[200,5]LIBA1/RO
```

The above comments also apply to the creation and referencing of a common region. The only difference is that when the common is task-built, the /SHAREABLE:COMMON switch is generally used and when the common is referenced, the option COMMON=name is used for a system resident common, and RESCOM=name is used for a user resident library.

```
        .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.
;-
     ❷  .PSECT  AADD,RO,I,GBL,REL,CON
❶ AADD:: CALL   $SAVAL            ; Save all registers
       ┌MOV    @2(R5),R0          ; Move 1st operand
     ❸ │ADD    @4(R5),R0          ; Add 2nd operand
       └MOV    R0,@6(R5)          ; Store result
        RETURN                    ; Restore regs and return

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

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

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

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

```
        PROGRAM USELIB
C
C File USELIB.FTN
C
C FORTRAN task to use resident library LIB
C
C Task-build instructions:
C
C       >LINK/CODE:FPP/MAP/OPTION USELIB,LB:[1,1]FOR-
C     6 ->OTS/LIBRARY
C       Option? RESLIB=LIB/RO
C       Option? <RET>
C
        INTEGER ANS,OP1,OP2
        DATA OP1,OP2 /8,2/
C
        CALL AADD(OP1,OP2,ANS)   ! Add operands
        TYPE 100, ANS            ! Print results
C
        CALL SUBB(OP1,OP2,ANS)   ! Subtract operands
        TYPE 100, ANS            ! Print results
C
        CALL MULL(OP1,OP2,ANS)   ! Multiply operands
        TYPE 100, ANS            ! Print results
C
        CALL DIVV(OP1,OP2,ANS)   ! Divide operands
        TYPE 100, ANS            ! Print results
C
        CALL EXIT
C
100     FORMAT (' THE ANSWER = ',I2,'.')
        END
```

Run Session

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

Figure 7-2 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. The I/O page does not contain physical memory, but peripheral device registers instead. Therefore, a device common does not contain data the way a regular resident common does. It 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. 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.

While a device common region can be created in FORTRAN, by its nature, referencing must be done via MACRO-11. For an example see the RSX-11M/M PLUS Task Builder Manual.

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

- 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 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, FMS (Forms Management Services), and FCS (File Control Services). If you write FORTRAN programs which use these products, you may find it useful to just read the last few pages, which 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 and Exercises Book. They are all lab problems. Check your answers against the solutions provided, either the on-line file (under UFD [2Ø2,2]) or the hard copy in the Tests and Exercises Book.

If you feel that you have mastered the material, have your course administrator record your progress on your progress plotter. You will then be ready to begin a new module.

If you feel that you have not yet mastered the material, return to the 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 access existing static 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 upon 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 automatically is 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	FORTRAN Calls
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	UNMAP

REQUIRED DATA STRUCTURES

Each memory management directive requires that you set up one of two data structures within your task; namely 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 FORTRAN calls are quite straightforward. Their format is either:

```
CALL XXXX(wdb,idsw)

    or

CALL XXX(rdb,idsw)
```

where wdb is the name of an 8 word integer array for the Window Descriptor Block

rdb is the name of an 8 word integer array for the Region Descriptor Block

Examples:

```
INTEGER WDB(8),RDB(8)
    .       .      .
    .       .      .
CALL CRAW(WDB,IDSW)
CALL CRRG(RDB,IDSW)
```

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:

- Attach Region (ATRG)
- Create Region (CRRG)
- Detach Region (DTRG)

Figure 8-1 shows the 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. It 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. Also used to return a size when attaching an existing region.

Name of Region - up to six characters in Radix-5Ø. Assigned when a region is created and used when attaching to a region.

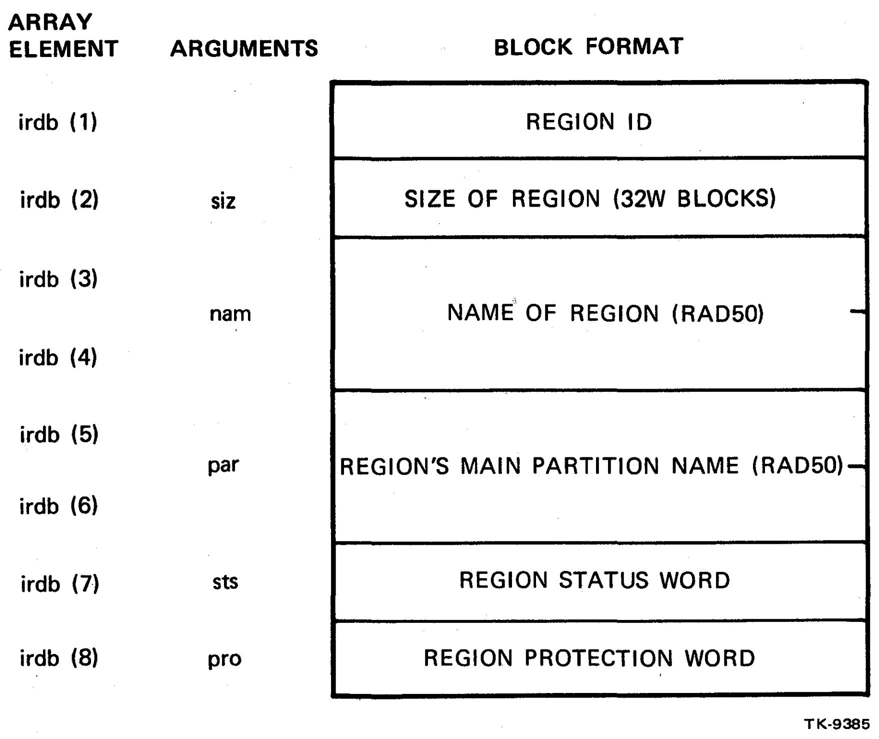
Region's Main Partition Name - up to six characters in Radix-5Ø. The name of the system controlled partition.

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. Table 8-1 lists 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: System, Owner, Group, and World:

```
World     Group      Owner     System
DEWR      DEWR       DEWR      DEWR
111Ø      111Ø       ØØØØ      ØØØØ     = 167ØØØ(8)
```

A 1 means access is denied, a Ø means access is permitted. The example means world and group have read access; owner and system have all access.



ARRAY ELEMENT	ARGUMENTS	BLOCK FORMAT
irdb (1)		REGION ID
irdb (2)	siz	SIZE OF REGION (32W BLOCKS)
irdb (3)	nam	NAME OF REGION (RAD50)
irdb (4)		
irdb (5)	par	REGION'S MAIN PARTITION NAME (RAD50)
irdb (6)		
irdb (7)	sts	REGION STATUS WORD
irdb (8)	pro	REGION PROTECTION WORD

Figure 8-1 The Region Definition Block

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

Just as in other modules, the symbols shown are those used in the documentation and by MACRO programmers. The symbols can be converted to FORTRAN acceptable variable names by dropping the period in the symbol. Values may be assigned by using the DATA statement.

Creating an RDB in FORTRAN

Example:

Create an RDB for a region with the following specifications:

```
    Size in 32(1Ø) 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

    DIMENSION IRDB(8)
        .        .
        .        .
        .        .
    DATA IRDB/Ø,2,3RMYR,3REG ,3RGEN,3R    ,"242,"177Ø17/
```

In the above, the region status word (word 7 = 242(8)), is the sum of 2ØØ(8) + 4Ø(8) +2(8). See table 8-2 for meanings.

The region protection word is 177Ø17(8), which breaks down as follows:

World	Group	Owner	System
DEWR	DEWR	DEWR	DEWR
1111	111Ø	ØØØØ	1111

Example:

Create an RDB for a region with the following specifications:

Size in 32(1Ø) word blocks = 1ØØØ(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

```
    DIMENSION IRDB(8)
        .        .
        .        .
    DATA IRDB/Ø,"1ØØØ,3RXXX,3RX  ,Ø,Ø,"1ØØ,"17ØØØØ/
```

Note that any value the Region Descriptor Block could be changed dynamically at run time by using input values to change various parts of the RDB.

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:

- Create Address Window (CRAW)
- Eliminate Address Window (ELAW)
- Map Address Window (MAP)
- Unmap Address Window (UNMAP)
- Send-by-Reference (SREF)
- Receive-by-Reference (RREF)

Figure 8-2 shows the layout of the WDB.

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

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; 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.

Offset in Region (32 word blocks) - The offset within the region at which mapping is to begin. 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. 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. 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 eight-word buffer for sending or receiving data as part of the Send-by-Reference and Receive-by-Reference directives.

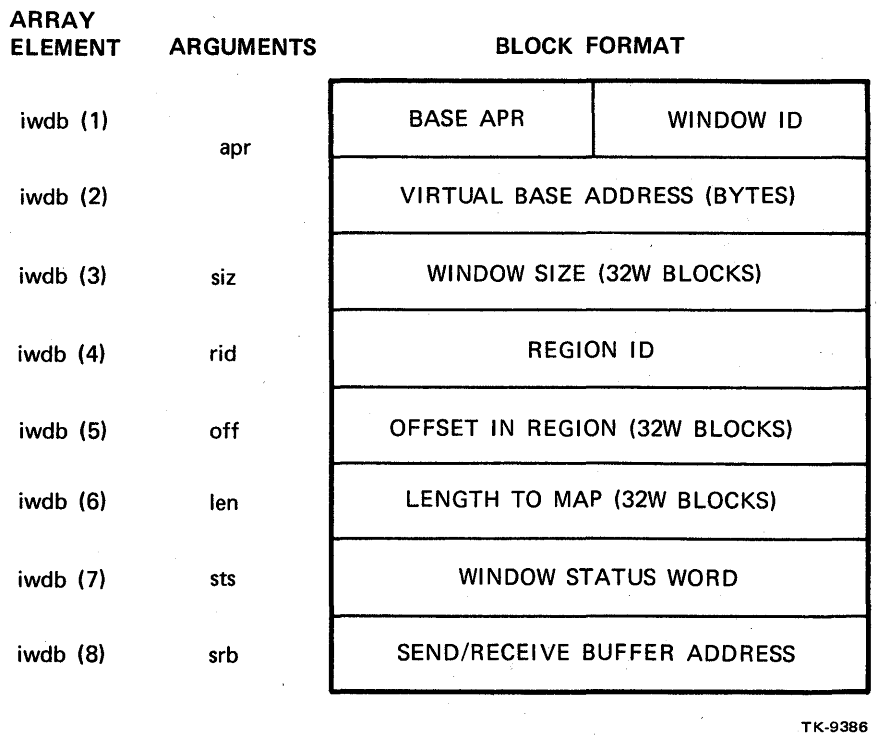


Figure 8-2 The Window Definition Block

Creating a WDB in FORTRAN

Example:

Create a WDB to describe a window with the following:

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

```
      DIMENSION IWDB(8)
          .        .
     DATA IWDB/"34ØØ,Ø,1ØØ,Ø,Ø,1ØØ,"2Ø1,Ø/
```

Note that the APR number (7 in the example) must be placed in the high byte of the first word in the WDB. This can be done by putting 34ØØ(8) into IWDB(1). 34ØØ(8) is ØØØØØ111 ØØØØØØØØ(2) which puts a 7 in the high byte for the base APR. This can also be done by setting IDWB(1)=7*256.

Word 7 (2Ø1(8)) is the window status word. See Table 8-3 for the definitions of the bits in this word.

Create a WDB to describe a window with the following:

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

```
     DIMENSION IWDB(8)
          .        .
          .        .
     DATA IWDB/"24ØØ,Ø,2ØØ,Ø,5,1Ø,"412,Ø/
```

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)

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). It is recommended that a task always issue the Detach Region directive rather than depend on the EXIT processing code to issue the Detach. The reason for this is that if a task is fixed and EXITs, then no detach is done. If you run the fixed task over and over, you could run out of pool.

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 and also what attaching and mapping mean, look over the sections on Windows and Regions in the last few pages of Module 5, the Memory Management module.

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, plus 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:

1. Size of region (in 32(1Ø) word blocks)

2. Name of region (becomes a private region if no name)

3. Name of partition (defaults to partition of task)

4. Region Status Word - mark for delete or do not delete (default is mark for delete)

5. Region protection word - determines permissible access to region

In the following discussion, the MACRO symbols are used for the various Window Status Word bits. See Table 8-3 (Window Status Word) for definitions and values.

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. The following code segment illustrates how to examine RS.CPR to see if the the region was successfully created.

```
          INTEGER RSCPR,RDB(8)
          DATA RSCPR/"1ØØØØØ/
               .       .
               .       .
          I=RDB(7).AND."1ØØØØØ
```

Now test I. If I is Ø, the region was not created; otherwise it was.

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 the last detach. The following notes are keyed to the example.

❶ Set up the RDB. RS.NDL(1ØØ(8)) in the region status word (RDB(7)) 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)

                                                170ØØØ(8)
```

Bit set means access denied

❷ Issue directive to create region, specifying the RDB address and the DSW as the only arguments.

❸ Check for directive error.

❹ Display message and exit.

```
        PROGRAM CRERG
C
C File CRERG.FTN
C
C CRERG creates a named region and exits, leaving the
C region in existence.
C
C RDB = Region Definition Block for region with the
C following properties:
C       Size              = 100 (32. word blocks)
C       Name              = MYREG
C       Partition         = GEN
C       Protection        = WO:None,SY:RWED
C                           OW:RWED,GR:RWED
C       Do not mark for delete on last detach
C
        INTEGER RDB(8)
C Initialize the RDB
     ❶  DATA RDB/0,"100,3RMYR,3REG ,3RGEN,3R    ,
        1"000100,"170000/
C Create region
     ❷  CALL CRRG(RDB,IDS)
C Branch on error
     ❸  IF(IDS.LT.0)GOTO 800
C Write success message
        WRITE (5,15)
15   ❹  FORMAT (' CRERG SUCCESSFULLY CREATED MYREG')
C Go to common exit
        GOTO 1000
C Write create error message
800     WRITE(5,850)IDS
850     FORMAT(' ERROR IN CREATING REGION, DSW = ',I4)
1000    CALL EXIT
        END
```

Run Session

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

Example 8-1 Creating a 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 a task exits without issuing an explicit detach, and the task is not fixed, 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 a detach, no detach is issued 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 when you attempt to create a virtual address window.

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, additional input parameters are needed. See the following section on Mapping to a region.

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 assumes, 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 shorter of length of window and length remaining in region. If defaulted, is set to the shorter of the two.

Window status word - actual access desired (read-only, or read/write). Read access is always requested by default so a request for write access actually requests read/write access, and a request for no access actually requests read access.

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 access desired is used here in addition to that declared when attaching because several 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 access or read/write access.

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

❶ Task-building with the WNDWS=1 option causes the Task Builder to allocate space in the task header for one additional window block. You must also use the VSECT option to create a virtual section starting at 160000(8) for an extent of 20000(8). APR 7 must be used to map the section because the section's beginning address is 160000(8). The name of the virtual section is DATA. This ties the FORTRAN named COMMON DATA to the virtual section.

❷ RDB for region. Note that RDB(7), the region status word, is 152(8). This is the combination of the following:

```
RS.NDL = 100(8)
RS.ATT =  40(8)
RS.DEL =  10(8)
RS.WRT =   2(8)
         ------
         152(8)
```

See Table 8-1 for the above definitions.

❸ 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 (200(8)) means that the Create Address Window directive will both create the window and map it to the region. WS.RED (1(8)) is automatic, even though not specified. WS.WRT (2(8)) indicates to map with write access. The sum of the two needed octal codes is 202(8).

❹ Create region and attach.

❺ Move region ID, returned in RDB(1) after attach, into WDB(4) for mapping.

❻ Create a virtual address window and map it to the region. The virtual address window begins with APR 7, so the base address in the window is 160000(8), corresponding to the base address in the region.

7 Place a byte count, 4ØØ(1Ø), 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 the information about the amount of data.

8 Move 1ØØ(1Ø) words of ASCII "AB" and 1ØØ(1Ø) words of ASCII "12" into the region. This gives us 2ØØ(1Ø) words or 4ØØ(1Ø) bytes of data.

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

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

```
      PROGRAM CREURG
C
C File CREURG.FTN
C
C CREURG creates a named region (attached on creation),
C creates a virtual address window (mapped on creation),
C places ASCII data in the region, detaches from the
C region and exits, leaving the region in existence.
C It places a count word in the first word of the
C region, telling how many bytes of data follow.
C
C Task-build instructions:
C
C     [>LINK/MAP/OPTIONS/CODE:FPP CREURG,LB:[1,1]FOROTS-
C     |->/LIBRARY
C  ❶  |Option? VSECT=DATA:160000:20000
C     |Option? WNDWS=1
C     |Option? <RET>
C     [
C RDB = Region Definition Block for region with the
C following properties:
C     [Size              = 100(8) (32. word blocks)
C     |Name              = MYREG
C     |Partition         = GEN
C  ❷  |Protection        = WO:None,SY:RWED
C     |                    OW:RWED,GR:RWED
C     |Do not mark for delete on last detach
C     [Attach with write and delete access
C
C WDB = Window Definition Block for window with the
C following properties:
C     [APR                = 7
C     |Size               = 100octal (32. word blocks)
C  ❸  |Offset in region = 0 (32. word blocks)
C     |Length in region = 100octal (32. word blocks)
C     |Map on create with write access
C     [
      INTEGER RDB(8),WDB(8)
      COMMON /DATA/ IDATA(201)
C Initialize the RDB
  ❷   DATA RDB/0,"100,3RMYR,3REG ,3RGEN,3R    ,"000152,"170000/
C Initialize the WDB
  ❸   DATA WDB/"3400,0,"100,0,0,"100,"202,0/
C Call routine to create and attach region
  ❹   CALL CRRG(RDB,IDS)
C Check for error
      IF(IDS.LT.0)GOTO 800
C Create address window and map to region
  ❺   WDB(4)=RDB(1)
  ❻   CALL CRAW(WDB,IDS)
```

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

```
C Check for error
        IF(IDS.LT.0)GOTO 810
C Place data in region - 1st word is a byte count
    ❼   IDATA(1)=400
       ┌DO 10 J=2,101
10  ❽  │IDATA(J)='AB'
       │DO 20 K=102,201
20     └IDATA(K)='12'
C Detach from region
    ❿   CALL DTRG(RDB,IDS)
C Check for error
        IF(IDS.LT.0)GOTO 820
C Write message
    ❾   TYPE *,'CREURG HAS CREATED AND INITIALIZED THE
        1REGION'
C Branch to common exit
        GOTO 1000
C Write create error message
800     WRITE(5,805)IDS
805     FORMAT(' ERROR IN CREATING REGION, DSW = ',I4)
C Go to common exit
        GO TO 1000
C Write attach error message
810     WRITE(5,815)IDS
815     FORMAT(' ERROR IN CREATING WINDOW AND MAPPING,
        1DSW = ',I4)
        GOTO 1000
C Write detach error message
820     WRITE(5,825)IDS
825     FORMAT(' ERROR IN DETACHING FROM REGION, DSW = '
        1,I4)
C Common exit
1000    CALL EXIT
        END
```

Run Session

```
>RUN CREURG
 CREURG HAS CREATED AND INITIALIZED THE REGION
>RUN ATTURG
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABAB12121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212
1212121212121212
>
```

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

The following notes are keyed to Example 8-3.

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

2. The RDB for attaching to the region. The only required information is the region name and the region status word. The partition name and the size, although included, are not required. RS.MDL (2ØØ(8)) (set) marks the region for delete when we do an explicit detach. We need delete access to mark the region for delete (RS.DEL=1Ø(8)). In addition, attach with read (RS.RED=1(8)) and write (RS.WRT=2(8)) access so we can map with read/write access. The sum of the region status codes above is 213(8).

3. The WDB for the virtual address window. We map the entire region (length = 1ØØ (8) 32-word blocks) starting from the beginning (offset = Ø). WS.MAP means create the address window and map. Map with read (WS.RED) and write (WS.WRT) access. The sum of the window status codes is 2Ø3(8).

4. Attach to the region.

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

6. First word in the region contains a character or byte count. Convert it to a word count.

7. Print the contents of the region, 64(1Ø) characters per line. This technique is used to demonstrate how to control the width of the output and to make the run session fit on an 8 1/2 by 11 inch page with margins. If the full terminal buffer width (typically 8Ø(1Ø) or 132(1Ø)) is acceptable, the FORMAT could be 39A2 or 65A2.

8. Detach from the region. Explicit detach required to mark the region for delete.

```
      PROGRAM ATTURG
C
C File ATTURG.FTN
C
C FORTRAN program to attach the existing region MYREG,
C create a virtual address window (mapped on creation),
C read ASCII data out of the region, detach from the
C region, and exit. The region is marked for delete
C and will be deleted on last detach.
C The first word in the region contains a count of how
C many bytes of data are in the region.
C
C Task-build with these options:
C                  VSECT=REGION:160000:20000
C    1             WNDWS=1
C
      INTEGER RDB(8),WDB(8)
      INTEGER IDATA(2048)       ! Array for addressing
C                               !  region (Full 4KW)
C This common block will align with the address window
      COMMON /REGION/IDATA
C RDB = Region definition block with the following properties:
C       Size            0 (32.-word blocks) - returned
C                                when attached
C       Name            MYREG
C 2     Partition       GEN
C       Mark for delete on last detach
C       Attach with delete, read and write access
C Initialize the RDB
      DATA RDB /0,0,3RMYR,3REG ,3RGEN,3R    ,"000213
     1,0/
C
C WDB = Window definition block with the following properties:
C       APR                7
C       Size               200 octal (32.-word blocks)
C       Offset in region   0 (32.-word blocks)
C 3     Length of window   0 octal (32.-word blocks) -
C                                defaults to shortest
C                                available length
C       Map on create with read and write access
C Initialize the WDB
      DATA WDB /"3400,0,"200,0,0,0,"203,0/
```

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

```
    C
    C Attach region
❹            CALL ATRG (RDB,IDS)
    C Check for error on attach
             IF (IDS .LT. 0) GOTO 100
    C Move region id to WDB
             WDB(4)=RDB(1)
❺   C Create and map window
             CALL CRAW (WDB,IDS)
    C Check for error
             IF (IDS .LT. 0) GOTO 200
    C Get byte count and convert to word count
❻            NWORD=(IDATA(1)+1)/2
    C Print contents of region
❼   10       WRITE (5,11) (IDATA(I),I=2,NWORD)
    11       FORMAT (' ',32A2)
    C Detach from region and delete it
❽            CALL DTRG (RDB,IDS)
    C Check for error
             IF (IDS .LT. 0) GOTO 300
    C And Jump to exit


             GOTO 500
    C
    C Error messages
    100      WRITE (5,101) IDS
    101      FORMAT (' ERROR ATTACHING TO REGION, DSW =',I4)
             GOTO 500
    200      WRITE (5,201) IDS
    201      FORMAT (' ERROR IN CREATING WINDOW, DSW =',I4)
             GOTO 500
    300      WRITE (5,301) IDS
    301      FORMAT (' ERROR DETACHING FROM REGION, DSW =',I4)
    C
    500      CALL EXIT
             END
```

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 appropriate access privileges 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. 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 the last detach.

If you are using an event flag for synchronization, note that the flag should be used to notify the sender when the receiver receives the region by reference. It is not the same as the 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 (2ØØ(8)) for receive and map
                      WS.RCX (1ØØ(8)) for receive data or exit

Buffer Address - 1Ø(1Ø) word buffer for sender task
                 name (in Radix-5Ø 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 that the task is to EXIT if there are no packets in the Receive-by-Reference queue.

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

2. 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.

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

4. Fill the region with ASCII M's.

5. Send-by-Reference to RCVREF (Example 8-4). Event flag 1 will be set when RCVREF actually does a Receive-by-Reference.

6. Display message saying region created and sent. Then wait for event flag 1 to be set.

7. Display message saying RCVREF received region.

8. Exit. The Executive will detach us from the region. Note that even if SNDREF exits before REVREF received, 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.

```
      PROGRAM SNDREF
C
C File SNDREF.FTN
C
C This program creates a 64-word unnamed region and
C fills it with ASCII characters.  It then sends it by
C reference to task RCVREF, and waits for RCVREF to
C receive the region.(This is signalled by event flag
C #1.)  SNDREF then prints a message and exits.  Since
C the area is unnamed, it is automatically deleted when
C the last attached task exits.
C
C Task-build instructions:
C
C       >LINK/MAP/CODE:FPP/OPTIONS SNDREF,LB:[1,1]FOROTS-
C       ->/LIBRARY
C       Option? WNDWS=1
C       Option? VSECT=DATA:160000:200
C       Option? <RET>
C
C Install and run instructions: RCVREF must be installed.
C Run SNDREF first, then run RCVREF.
C
C       RDB = Region definition block with the following
C       properties:
C               Size            2 32-word blocks
C               Name            none
C               Partition       GEN
C               Protection      WO:none,SY:RWED,OW:RWED,
C                               GR:none
C               Attach on creation
C               Read and write access desired on attach
C
C       WDB = Window definition block with the following
C       properties:
C               APR                 7
C               Size                2 32-word blocks
C               Offset in region    0 32-word blocks
C               Length of region    2 32-word blocks
C               Map on create with write access
C
      INTEGER RDB(8),WDB(8),RCV(2)
C This common block will align with the address window
      COMMON /DATA/IDATA(64)
C Initialize the RDB
❶     DATA RDB/0,"2,0,0,3RGEN,3R   ,"43,"170017/
C Initialize the WDB
❷     DATA WDB/"3400,0,"2,0,0,"2,"202,0/
C Name of receiver task
      DATA RCV/3RRCV,3RREF/
```

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

```
C Code
          CALL CRRG(RDB,IDS)        ! Create region
   3      IF (IDS .LT. 0) GOTO 100 ! Check for error
          WDB(4)=RDB(1)             ! Move region id to WDB
          CALL CRAW(WDB,IDS)        ! Create window
          IF (IDS .LT. 0) GOTO 200 ! Check for error
C Fill region with data
   4      DO 10 I=1,64
10        IDATA(I)='MM'
C Send-by-reference to receiver task, set event flag 1
C when received
   5      CALL SREF(RCV,1,WDB,,IDS)
          IF (IDS .LT. 0) GOTO 400 ! Check for error
          TYPE *,' SNDREF HAS CREATED THE REGION AND HAS
   6      1 SENT IT TO RCVREF.'    ! Display message
          CALL WAITFR(1,IDS)        ! Now wait for reception
          IF (IDS .LT. 0) GOTO 500 ! Check for error
          TYPE *,' RCVREF HAS RECEIVED IT. SNDREF IS NOW
   7      1EXITING.'                ! Write message
          GOTO 600                  ! And so exit
C Error handling code
100       WRITE (5,110)IDS
110       FORMAT (' ERROR CREATING REGION, DSW = ',I4)
          GOTO 600
200       WRITE (5,210)IDS
210       FORMAT (' ERROR CREATING WINDOW, DSW = ',I4)
          GOTO 600
400       WRITE (5,410)IDS
410       FORMAT (' ERROR IN SEND-BY-REFERENCE, DSW = ',I4)
          GOTO 600
500       WRITE (5,510)IDS
510       FORMAT (' ERROR ON WAIT, DSW = ',I4)

600  8    CALL EXIT
          END
```

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

The following notes are keyed to Example 8-5.

1. WDB for virtual address window. The size is 2ØØ(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.

2. Create the virtual address window.

3. WS.MAP (2ØØ(8)) must be set in the Window Status Word (word 7) of the Window Definition Block, so that the task will map as part of the Receive-by-Reference.

4. Receive-by-reference and map.

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

6. Display all characters with one WRITE.

7. 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. So if you receive and map in one call, issue the Create Address Window directive with the WS.MAP bit clear, and 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.

```
      PROGRAM RCVREF
C
C File RCVREF.FTN
C
C Program to receive-by-reference a region from SNDREF,
C map to the region, read ASCII data from the region,
C detach from the region, and exit. The region will be
C deleted on last detach.
C
C Task-build instructions: Include these options
C                 WNDWS=1
C                 VSECT=DATA:160000:20000
C
C Install and run instructions: RCVREF must be installed.
C Run SNDREF first, then run RCVREF.
C
C WDB = Window definition block with:
C       APR                  7
C       Size                 200(8) 32-word blocks
C                                   Allow for full APR use
C   These are filled in on receive, as set by sender:
C       Offset in region  0 32-word blocks
C       Length of region  0 32-word blocks
C                                  reset after mapping
C       Access               0
C NOTE: Must map after receiving (or as part of receive)
      INTEGER WDB(8)
   ❶  DATA WDB/"3400,0,"200,0,0,0,0,0/
C This common block will align with the address window
      COMMON /DATA/IDATA("10000)
C Create address window--do not map at this time
   ❷  CALL CRAW(WDB,IDS)
C Check for error on create
      IF (IDS .LT. 0) GOTO 200
C Now set WDB status for mapping--will be done by receive-by-reference
   ❸  WDB(7)=WDB(7)+"200
C Receive-by-reference and map
   ❹  CALL RREF(WDB,,IDS)
C Check for error
      IF (IDS .LT. 0) GOTO 100
```

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

```
C Calculate number of words of data - length in blocks
C returned at WDB(6)
    5   NCHAR = 32*WDB(6)
    6  [WRITE(5,10) (IDATA(I),I=1,NCHAR)
10     [FORMAT (' ',32A2)
C Go exit
        GOTO 300
C       Error messages
100     WRITE(5,110)IDS
110     FORMAT (' ERROR ON RECEIVE-BY-REFERENCE, DSW =',I4)
        GOTO 300
200     WRITE(5,210)IDS
210     FORMAT (' ERROR CREATING WINDOW, DSW =',I4)
300  7  CALL EXIT
        END
```

Run Session

```
>INS RCVREF
>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
```

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

THE MAPPED ARRAY AREA

A large core resident data area may be set up by using the FORTRAN VIRTUAL statement. The VIRTUAL statement provides the Task Builder with the information required to create a mapped array area. The VIRTUAL statement is very similar to the DIMENSION statement except that all space reserved for a VIRTUAL array is in a separate area within the task region.

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.

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. However, due to the subscript limitation of 32767 in FORTRAN, a single array cannot have more than 32767 elements. Typically, the virtual address window maps only a portion of the region at a time using a single APR. Once you have referenced an element in a virtual array, the APR is set up to map to the nearest 4K boundary in the array. Hence, assuming an integer array IARRAY, if your first reference is to IARRAY(1), then any element in the virtual array between IARRAY(1) and IARRAY(4Ø96) can be maped with the current setting of the APR.

However, if a reference is made to an element with a subscript higher than 4Ø96, the APR used for the 4K window must be remapped. Hence, consecutive references to IARRAY(1), IARRAY(5ØØØ), IARRAY(2), IARRAY(5ØØ1), etc., will cause a remapping on each reference, thereby inducing some additional overhead. Note that all mapping is transparent to the user; however, knowledge of how mapping is performed, and when, can aid you in designing your programs to reduce the overhead required by remapping.

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.

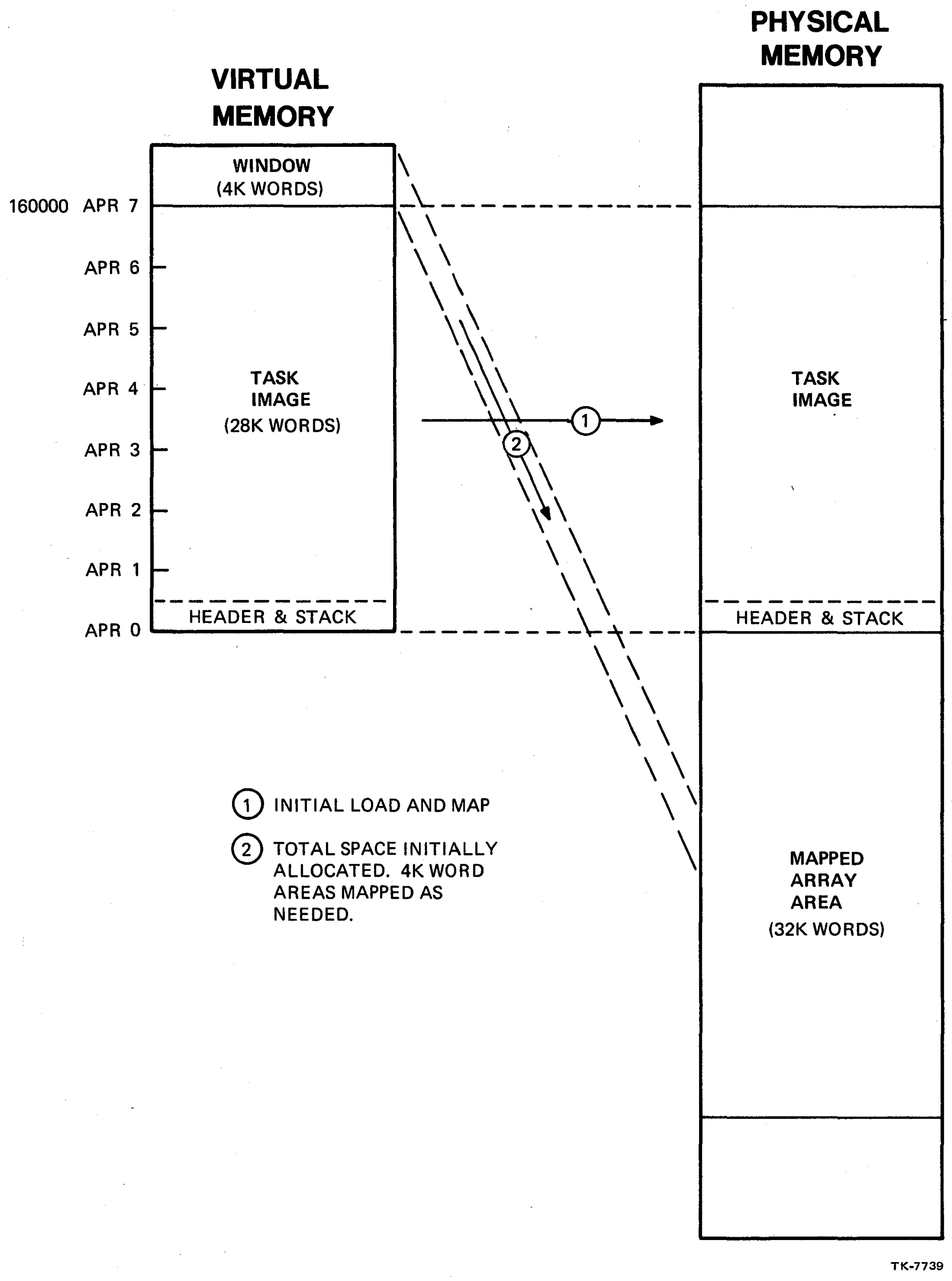


Figure 8-3 The Mapped Array Area

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

❶ Create the virtual array IDATA with 32000 elements.

❷ Data to be placed into various parts of the virtual array.

❸ Put 'A1' into IDATA(1) and 'G7' into IDATA(2). After the first reference, the mapping is set up to allow reference to any element up to IDATA(4096) without remapping.

❹ Put data into elements IDATA(4097) and IDATA(4098). Note that the window had to be remapped to access the second 4K of the mapped array. This is transparent to the user.

❺ Put data into the third 4K block. Remapping needed.

❻ Put data into the fourth 4K block. Remapping needed.

❼ Retrieve data from each of the four 4K blocks. Remapping required for each reference. 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.

```
        PROGRAM VIRTAR
C
C File VIRTAR.FTN
C
C VIRTAR makes use of the mapped array area by using a
C FORTAN virtual array. It places data in 4 different
C 4K word blocks of the area and then displays the
C data at the terminal.
C
        INTEGER DATA,DATB,DATC,DATD,DATG
C Set up the virtual array in the mapped array area
    ❶   VIRTUAL IDATA(32000)
C Define data values to be placed in the array
       [DATA DATA,DATB,DATC //'A1','B2','C3'/
    ❷  [DATA DATD,DATG //'D4','G7'/
C Place data in 1st 4KW block/ IDATA(1) - IDATA(4096)
       [IDATA(1)=DATA
    ❸  [IDATA(2)=DATG
C Place data in 2nd 4KW block/ IDATA(4097) - IDATA(8192)
       [IDATA(4097)=DATB
    ❹  [IDATA(4098)=DATG
C Place data in 3rd 4KW block/ IDATA(8193) - IDATA(12288)
       [IDATA(8193)=DATC
    ❺  [IDATA(8194)=DATG
C Place data in 4th 4KW block/ IDATA(12289) -IDATA(16384)
       [IDATA(12289)=DATD
    ❻  [IDATA(12290)=DATG
C Write data from 1st 4KW block
        WRITE   (5,100) IDATA(1),IDATA(2)
C Write data from 2nd 4KW block
 [      WRITE   (5,100) IDATA(4097),IDATA(4098)
 [C Write data from 4th 4KW block
❼[      WRITE   (5,100) IDATA(12289),IDATA(12290)
 [C Write data from 3rd 4KW block
 [      WRITE   (5,100) IDATA(8193),IDATA(8194)
100     FORMAT (' ',A2,A2)
        CALL EXIT
        END
```

Run Session

```
>RUN VIRTAR
A1G7
B2G7
D4G7
C3G7
>
```

Example 8-6 Use of the Mapped Array Area

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 in that book or in on-line files.

If you think that you have mastered the material, ask your course administrator to record your progress in 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 I/O

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 (RLØ1/Ø2, RPØ6, RMØ2/Ø3), 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 RLØ2s). 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(1Ø) words or 512(1Ø) bytes long. Some devices divide or format their volumes into pieces which are 256(1Ø) 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(1Ø) word area on the disk, starting with logical block Ø. 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 Ø.

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(1Ø) words	Numbered in increments relative to the beginning of the volume, starting with Ø
Virtual	256(1Ø) 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(1Ø) 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.

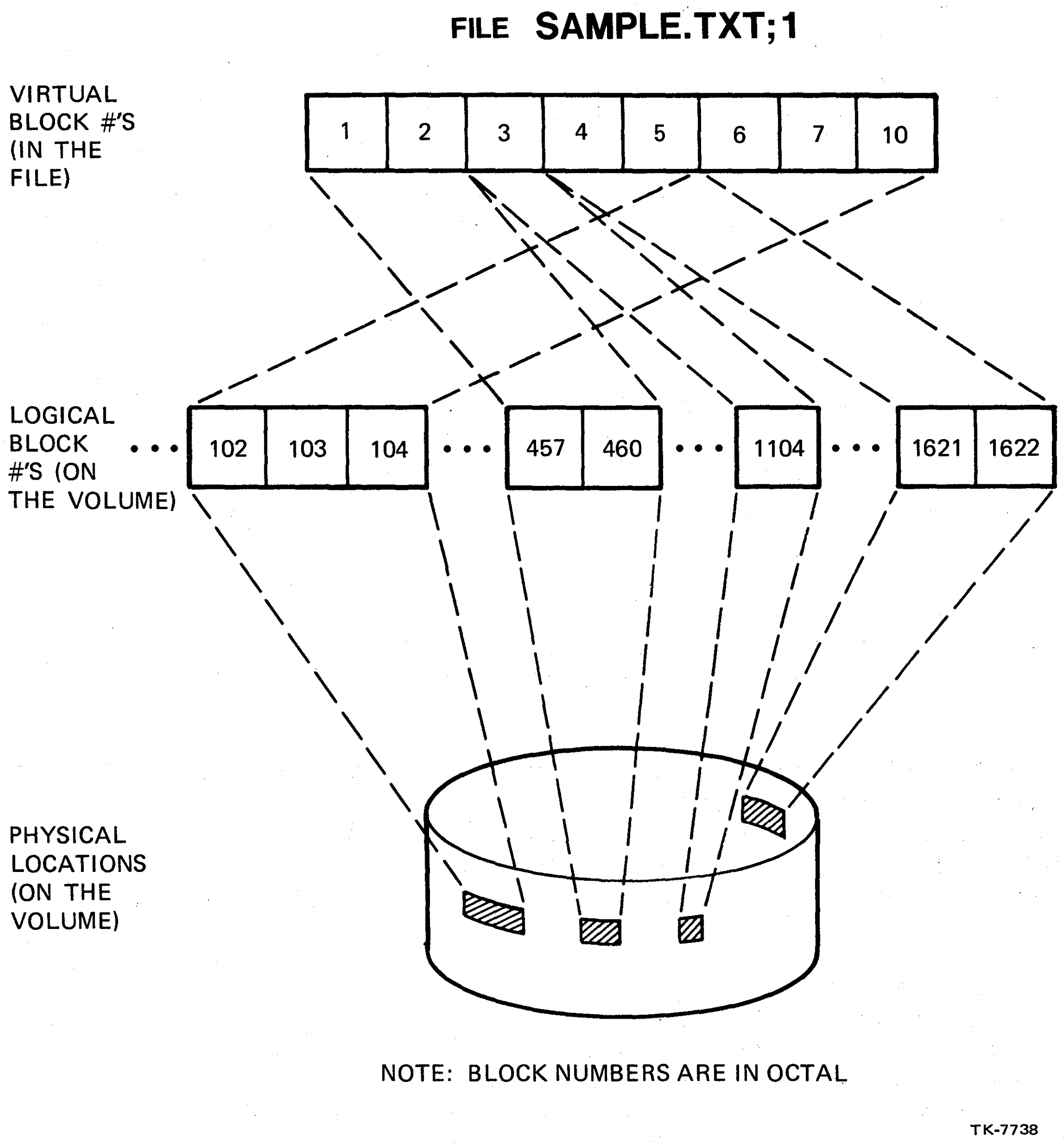

FILE SAMPLE.TXT;1
VIRTUAL BLOCK #'S (IN THE FILE)
1
2
3
4
5
6
7
10
LOGICAL BLOCK #'S (ON THE VOLUME)
102
103
104
457
460
1104
1621
1622
PHYSICAL LOCATIONS (ON THE VOLUME)
NOTE: BLOCK NUMBERS ARE IN OCTAL
TK-7738

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

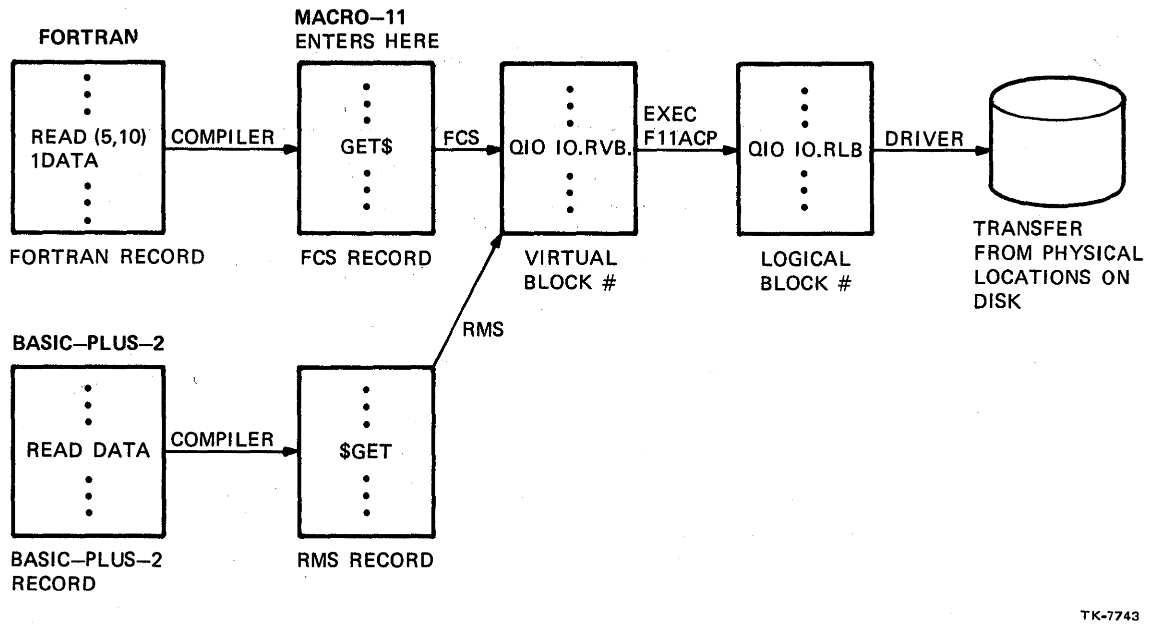


Figure 9-2 How the Operating System Converts Between Virtual, Logical, and Physical Blocks

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(1Ø). Virtual block 1 = logical block 336851(1Ø), vb 2 = lb 336852(1Ø), vb 3 = lb 336853(1Ø), vb4 = lb 336854(1Ø), and vb 5 = lb 336855(1Ø). The next group of blocks, starting with virtual block 6 has 51(1Ø) blocks and begins at logical block 3369ØØ(1Ø) up through logical block 33695Ø(1Ø). The last 17(1Ø) virtual blocks (virtual blocks 57(1Ø) to 73(1Ø)) begin at logical block 337ØØ6(1Ø) up through logical block 337Ø22(1Ø). These file retrieval pointers are updated each time a change in block allocation occurs as a result of a file I/O operation.

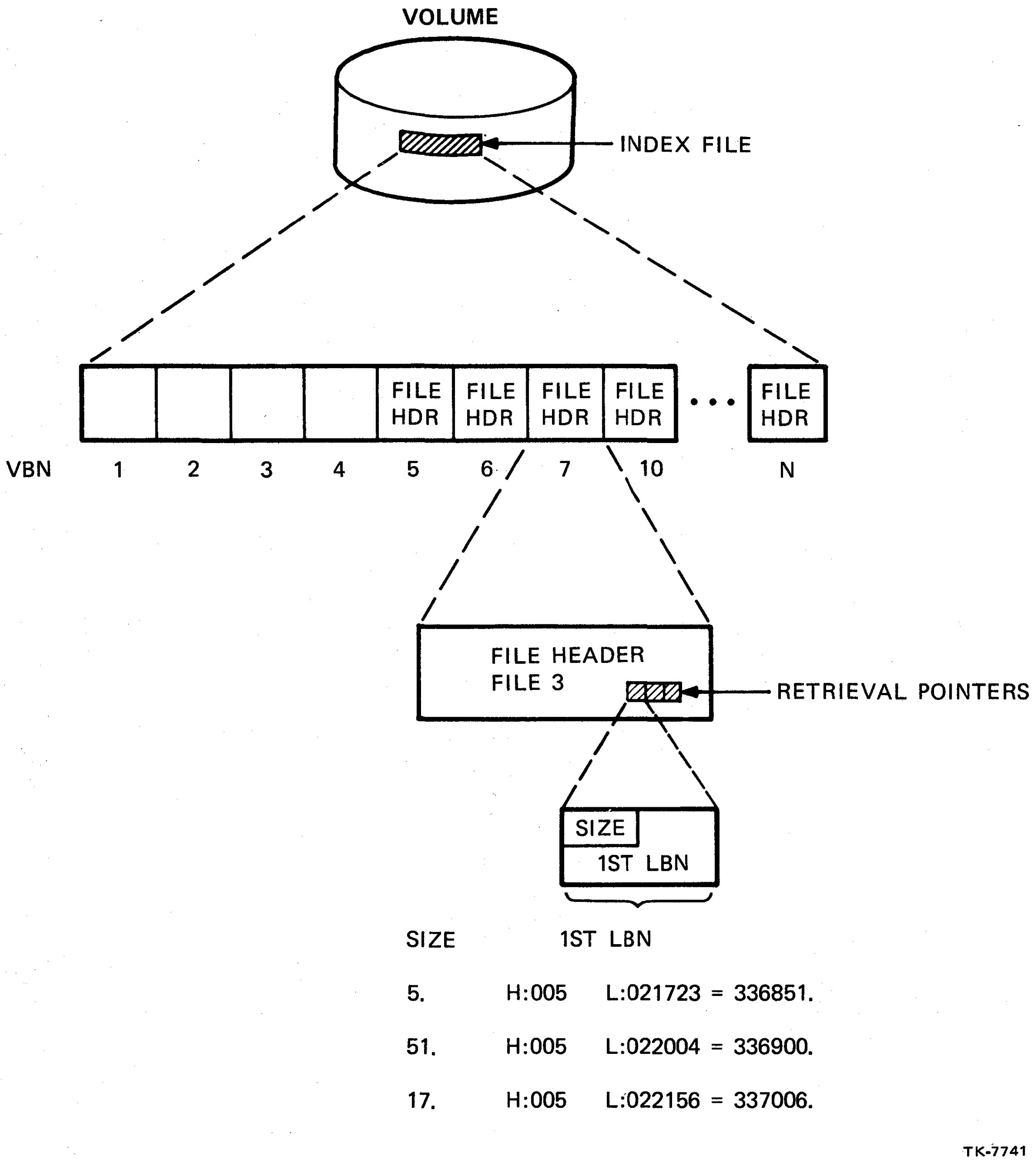


Figure 9-3 FILES-11 Structures Used to Support Virtual-to-Logical Block Mapping

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.

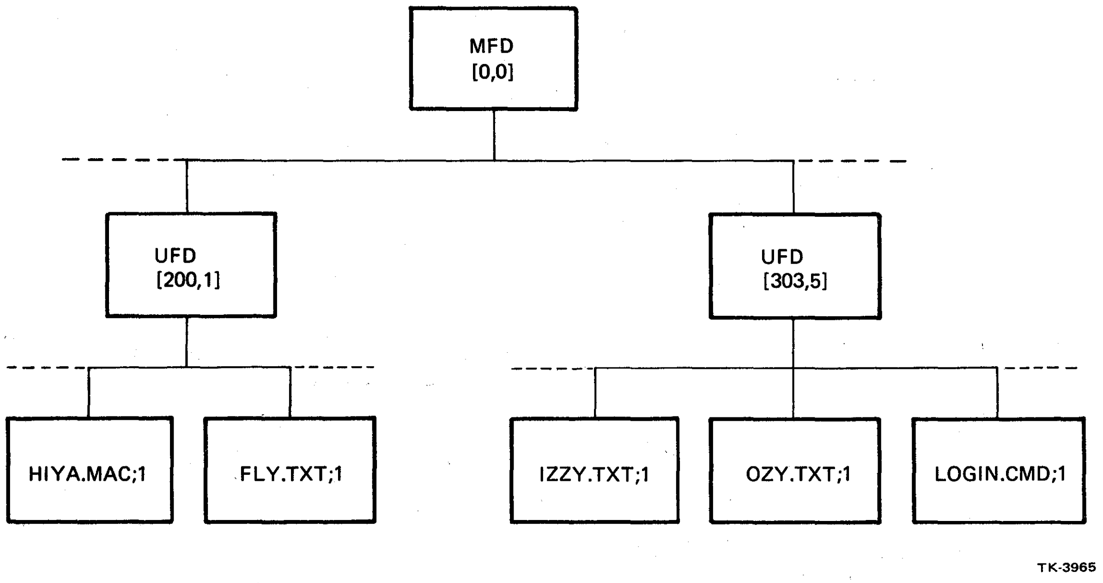


Figure 9-4 Directory and File Organization on a Volume

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 a file is opened a second time during a task's execution.

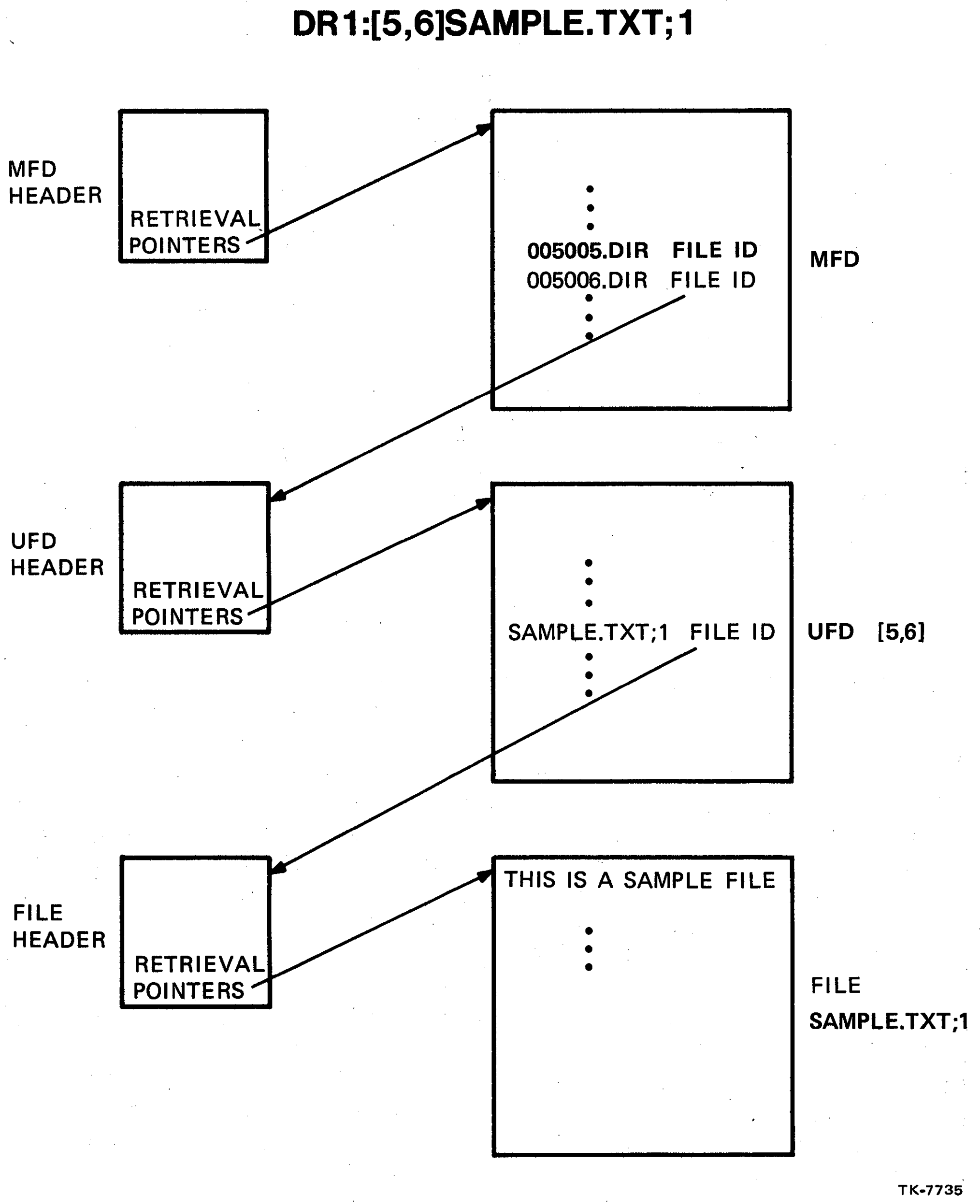


Figure 9-5 Locating a File on a FILES-11 Volume

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.

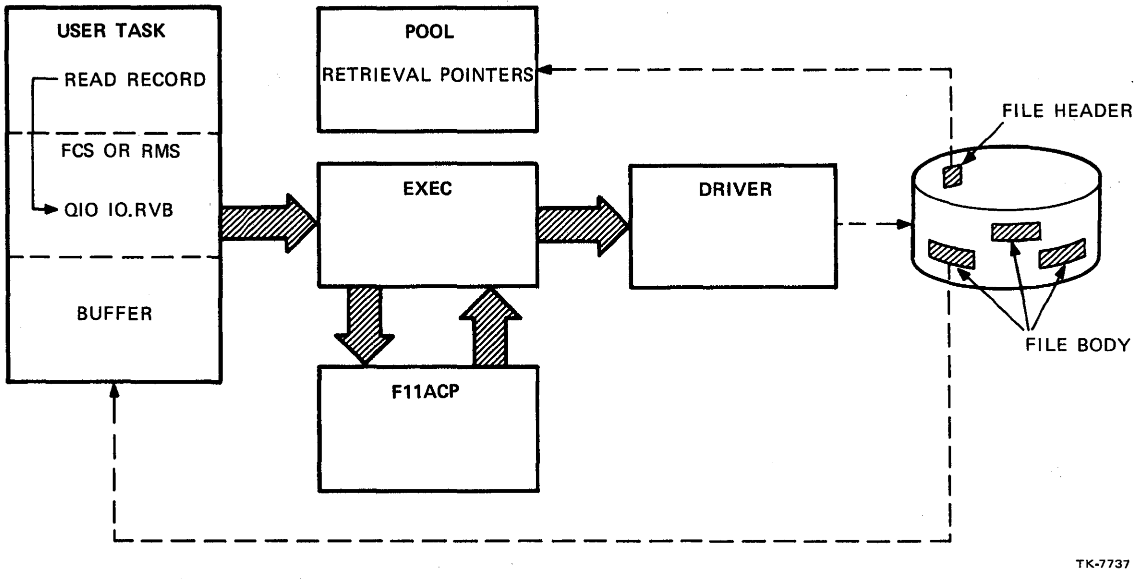


Figure 9-6 Flow of Control During the Processing of an I/O Request

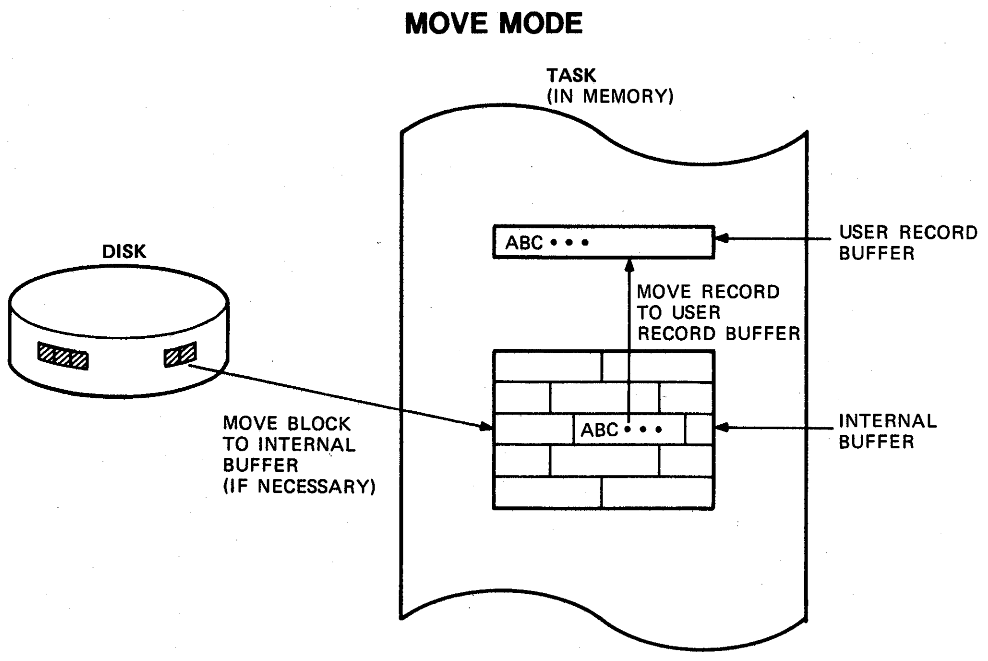
OVERVIEW AND COMPARISON OF FCS AND RMS

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.



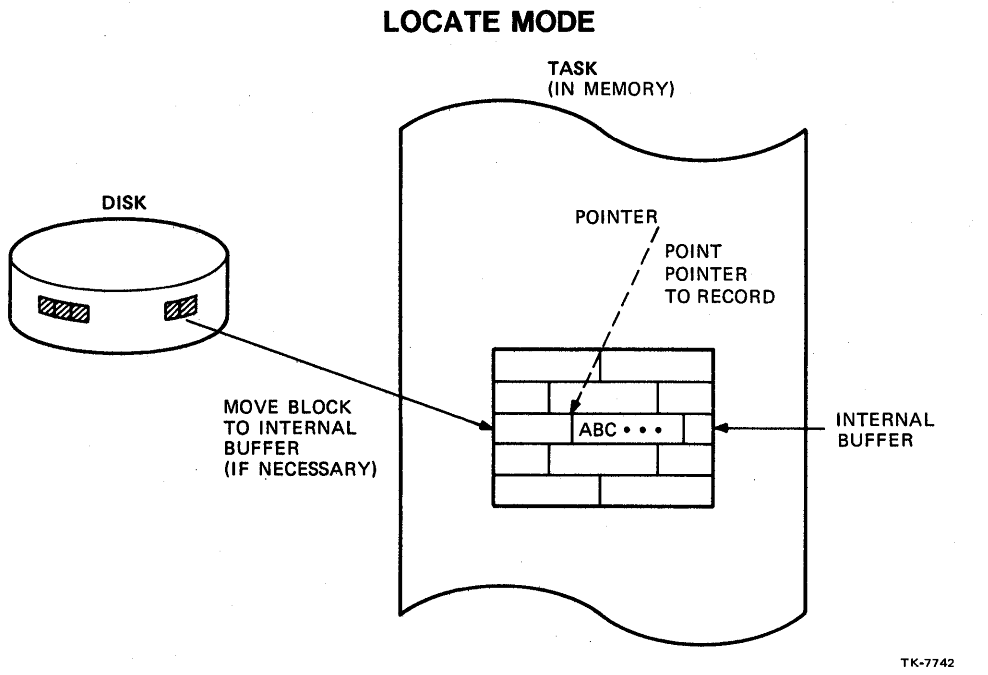


Figure 9-7 Move Mode and Locate Mode

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.

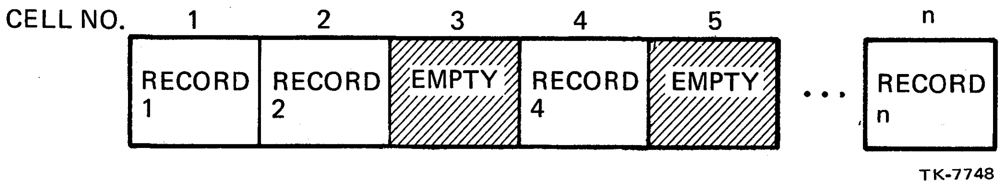


Figure 9-8 Sequential Files

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(1Ø))

```
ØØØ Ø61 Ø62 Ø63 Ø64 Ø65 Ø4Ø Ø4Ø Ø4Ø Ø4Ø Ø4Ø Ø4Ø Ø4Ø Ø4Ø Ø4Ø Ø4Ø Ø4Ø
     1   2   3   4   5   _   _   _   _   _   _   _   _   _   _   _
Ø2Ø Ø4Ø xxx Ø61 Ø62 Ø63 Ø4Ø Ø61 Ø62 Ø63 Ø64 Ø4Ø Ø4Ø Ø4Ø Ø4Ø Ø4Ø Ø4Ø
     _  pad  1   2   3   _   1   2   3   4   _   _   _   _   _   _
Ø4Ø Ø4Ø Ø4Ø Ø4Ø xxx 1Ø1 1Ø1 1Ø1 1Ø1 Ø4Ø 1Ø2 1Ø2 1Ø2 1Ø2 Ø4Ø 1Ø3 1Ø3
     _   _   _  pad  A   A   A   A   _   B   B   B   B   _   C   C
Ø6Ø Ø4Ø 1Ø4 Ø4Ø Ø4Ø Ø4Ø xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx
     _   D   _   _   _  pad
```

Variable-Length Records

```
ØØØ ØØ5 ØØØ Ø61 Ø62 Ø63 Ø64 Ø65 xxx Ø1Ø ØØØ Ø61 Ø62 Ø63 Ø4Ø Ø61 Ø62
             1   2   3   4   5  pad          1   2   3   _   1   2
Ø2Ø Ø63 Ø64 Ø16 ØØØ 1Ø1 1Ø1 1Ø1 1Ø1 Ø4Ø 1Ø2 1Ø2 1Ø2 1Ø2 Ø4Ø 1Ø3 1Ø3
     3   4           A   A   A   A   _   B   B   B   B   _   C   C
Ø4Ø Ø4Ø 1Ø4 xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx
     _   D
```

Sequenced Variable-Length Records

```
ØØØ ØØ7 ØØØ ØØ1 ØØØ Ø61 Ø62 Ø63 Ø64 Ø65 xxx Ø12 ØØØ ØØ2 ØØØ Ø61 Ø62
                     1   2   3   4   5  pad                  1   2
Ø2Ø Ø63 Ø4Ø Ø61 Ø62 Ø63 Ø64 Ø2Ø ØØØ ØØ3 ØØØ 1Ø1 1Ø1 1Ø1 1Ø1 Ø4Ø 1Ø2
     3   _   1   2   3   4                   A   A   A   A   _   B
Ø4Ø 1Ø2 1Ø2 1Ø2 Ø4Ø 1Ø3 1Ø3 Ø4Ø 1Ø4 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, 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.

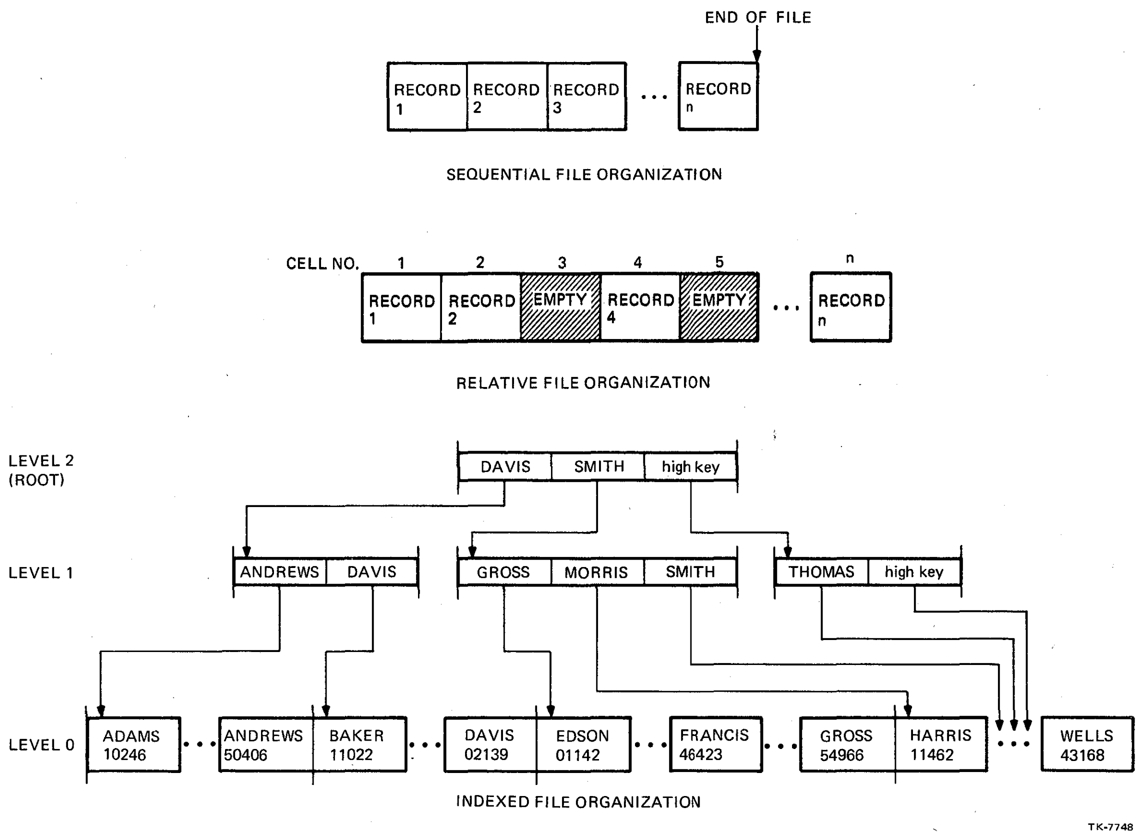


Figure 9-9 RMS File Organizations

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 record's original location, which points to 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(1Ø) 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

* Not available in FORTRAN.

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(1Ø) 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

INTRODUCTION

The File Control Services (FCS) subsystem provides the means through which tasks perform I/O. In FORTRAN, calls to the FCS routines are made indirectly through the FORTRAN Object Time System (OTS).

While the FORTRAN programmer need not know how the the data structures or the various calls to the FCS subroutines are used, this module is presented as a brief introduction to FCS.

The first example, illustrating how a FORTRAN program creates a file, also shows the MACRO code needed to perform the same function.

Further examples illustrate how some of the FCS features can be incorporated by using the FORTRAN OPEN statement and the appropriate forms of the READ and WRITE statements.

The major portion of this module contains a brief summary, with examples, of the FORTRAN READ, WRITE and OPEN statements and the various file and record types used in writing a FORTRAN program.

The FORTRAN programmer should be aware that each of the above, READ, WRITE, and OPEN, are translated into FCS data structures at compile time, and CALLs to FCS routines at execution time.

OBJECTIVES

1. To choose file characteristics for a specific application and create files with those characteristics

2. To write tasks which read or write data using record I/O.

RESOURCES

1. FORTRAN IV User's Guide

2. FORTRAN IV-Plus User's Guide

3. FORTRAN-77 User's Guide

FILE ORGANIZATION VS. RECORD ACCESS

A clear distinction must be made between the organization of a file and the record access to a file.

A file's organization refers to how the file was created via the keyword ORGANIZATION in the OPEN.

The two possibilities are:

```
    ORGANIZATION='SEQUENTIAL'
    ORGANIZATION='RELATIVE'
```

'INDEXED' is a third ORGANIZATION, but will not be discussed here.

Once established, the file ORGANIZATION cannot be changed. The default is 'SEQUENTIAL'.

The record access to a file determines how a particular program wants to access a file, again via the OPEN. The choices are:

```
    ACCESS='SEQUENTIAL'
    ACCESS='DIRECT'
    ACCESS='APPEND'
    (ACCESS='INDEXED' will not be discussed.)
```

Figure 1Ø-1 shows the possible combinations of ORGANIZATION and ACCESS.

ORGANIZATION	ACCESS
SEQUENTIAL	SEQUENTIAL
"	APPEND
"	DIRECT (if fixed length records)
RELATIVE	DIRECT
"	SEQUENTIAL

Figure 1Ø-1 Possible Combinations of ORGANIZATION and ACCESS

READ AND WRITE ACCESS TO A FILE

When a file is opened via the OPEN statement, the default is that the file is opened for read and write access. The OPEN keyword READONLY is used to restrict a program from write access. If you are the 'WORLD' to a file (i.e., not SYSTEM, OWNER, or GROUP) which has 'WORLD' protection set to R (read), and you attempt to open that file without using the READONLY keyword in the OPEN, the open will fail.

TYPES OF RECORDS IN A FILE

(Sometimes Referred to as 'Record Format')

There are three types of records (record formats) possible in a file via the OPEN keyword RECORDTYPE:

```
RECORDTYPE='VARIABLE'
RECORDTYPE='FIXED'
RECORDTYPE='SEGMENTED'
```

Type VARIABLE consists of variable length records where the record length is kept in the first two bytes of each record.

Type FIXED consists of records all of the same length as specified in the RECORDSIZE keyword in the OPEN. Since the size is fixed, it is not kept as an extra two bytes in the record; it is kept in the header of the file and is available when the file is opened.

Type SEGMENTED consists of records which contain a single logical record having one or more variable length records (segments). The length of a segmented record is arbitrary; however, the length of each segment is determined by the value of the RECORDSIZE keyword. The default size is 133. The segmented record is unique to FORTRAN and can be used only with unformatted sequential files under sequential access.

Because there is no set limit on the size of a segmented record, each segment contains control information to indicate that the segment is one of the following:

- The first segment in the segmented record (control word=1)
- The last segment in the segmented record (control word=2)
- The only segment in the segmented record (contol word=3)
- None of the above: i.e., a continuation record (control word=Ø)

The control word is kept as the first two bytes in the segment if the record is FIXED and in the third and fourth bytes if the record is VARIABLE.

When you wish to access an unformatted sequential file that contains fixed length or variable length records, which was not created by FORTRAN, you must specify RECORDTYPE='FIXED' or RECORDTYPE='VARIABLE' when you open the file. If you do not specify a RECORDTYPE, the default OPEN of the file will be RECORDTYPE='SEGMENTED' and the first word (if FIXED) or the second word (if VARIABLE) will be treated as a control word causing almost certain errors in the data.

FORMATTED AND UNFORMATTED RECORDS

A READ or WRITE statement can be formatted or unformatted. The main difference in the two is that a formatted READ or WRITE uses ASCII data while an unformatted READ or WRITE uses untranslated binary data.

The FORM='FORMATTED' or FORM='UNFORMATTED' is used as appropriate. The default is FORMATTED for ORGANIZATION='SEQUENTIAL' and UNFORMATTED for ORGANIZATION='RELATIVE'.

DECLARING THE SIZE OF A RECORD

The keyword RECORDSIZE is used to declare a specific size for a record. The defaults are as follows:

FORMATTED	133 bytes
UNFORMATTED (fixed)	128 bytes
UNFORMATTED (variable)	126 bytes

Note that you must specify the TKB option MAXBUF=n if you exceed a record size of 133, where n is the size in bytes of the record.

SUMMARY OF KEYWORDS IN THE OPEN STATEMENT

ORGANIZATION	=	'SEQUENTIAL'
	=	'RELATIVE'
ACCESS	=	'SEQUENTIAL'
	=	'DIRECT'
	=	'APPEND' (sequential only)
READONLY		to disallow WRITEs
RECORDTYPE	=	'FIXED'
	=	'VARIABLE'
	=	'SEGMENTED'
FORM	=	'FORMATTED'
	=	'UNFORMATTED'
RECORDSIZE	=	n

The remainder of this module is a series of examples illustrating the various types of files and how they are OPENed and ACCESSed.

Example 1Ø-1, CRESEQ, creates a file, VARI.ASC, of variable length records. Since the records are variable in length, the byte count for each record is kept in the first two bytes of the record itself.

As can be seen from the run session, the first record contains a single character, 1. Therefore bytes Ø and 1 are ØØ1 and ØØØ. The next byte is 61, which is ASCII for 1 followed by a byte of ØØØ. Since the record has an odd number of bytes, the record is padded with a ØØØ byte.

The next record contains an even number of bytes (2), so the record need not be padded.

Although the examples use several defaults, in order to illustrate the various defaults, it is recommended that no defaults be used when creating a file with an OPEN statement. Hence, in Example 1Ø-2, the complete OPEN is as follows:

```
      OPEN(UNIT=1,NAME='VARI.ASC',CARRIAGECONTROL='LIST',
      1    ORGANIZATION='SEQUENTIAL',ACCESS='SEQUENTIAL',
      2    TYPE='NEW',FORM='FORMATTED')
```

While it may seem a bit tedious to include all options in the OPEN, it aids greatly in the readability of the program and relieves any question as to what was meant in the OPEN.

```
        PROGRAM CRESEQ  !CREATE FILE SEQUENTIALLY
C
C FILE CRESEQ.FTN
C
C This task creates a file VARI.ASC of variable-length
C records using sequential record access.  The records
C are input from the terminal and written to the file.
C The process stops when the operator types CTRL/Z at
C the terminal.
C
        BYTE BUFF(80)
        INTEGER LEN
C OPEN FILE - Default access is sequential, default form
C is formatted I/O with sequential access
        OPEN    (UNIT=1,NAME='VARI.ASC',TYPE='NEW',
        1       CARRIAGECONTROL='LIST')
C Loop
10      READ (5,11,END=100) LEN,BUFF     ! Read record
11      FORMAT (Q,80A1)
        WRITE (1,12) (BUFF(I),I=1,LEN)   ! Write record
12      FORMAT (80A1)                    !  to file
        GO TO 10
C Close file and exit
100     CLOSE   (UNIT=1)
        CALL EXIT
        END
```

Run Session

```
>RUN CRESEQ
1
22
333
4444
Now is the time for all good.
```

```
Dump of DR2:[305,301]VARI.ASC;6 - File ID 40554,5,0
                 Virtual block 0,000001 - Size 512. bytes


000000  001 000 061 000 002 000 062 062 003 000 063 063 063 000 004 000
000020  064 064 064 064 035 000 116 157 167 040 151 163 040 164 150 145
000040  040 164 151 155 145 040 146 157 162 040 141 154 154 040 147 157
000060  157 144 056 000 000 000 000 000 000 000 000 000 000 000 000 000
```

Example 10-1 Creating a Sequential File with Variable Length Records

Example 10-2 shows the equivalent MACRO code to produce the same file as Example 10-1.

Example 10-3, SEQFOR, reads the first five records from the file VARI.ASC and displays them on the terminal.

```
        .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
```

Example 10-2 MACRO Equivalent of Example 10-1 (Sheet 1 of 2)

```
; 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



        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
1
22
333
4444
Now is the time for all good.
```

```
Dump of DR2:[305,301]VARI.ASC;6 - File ID 40554,5,0
                 Virtual block 0,000001 - Size 512. bytes


000000  001 000 061 000 002 000 062 062 003 000 063 063 063 000 004 000
000020  064 064 064 064 035 000 116 157 167 040 151 163 040 164 150 145
000040  040 164 151 155 145 040 146 157 162 040 141 154 154 040 147 157
000060  157 144 056 000 000 000 000 000 000 000 000 000 000 000 000 000
```

Example 1Ø-2 MACRO Equivalent of Example 1Ø-1 (Sheet 2 of 2)

```
        PROGRAM SEQFOR
C
C File SEQFOR.FTN
C
C This task reads the first 5 records from the file
C VARI.ASC using sequential access and formatted reads.
C It displays the records at TI:.
C
        INTEGER REC(40)
C Open file
        OPEN (UNIT=1,NAME='VARI.ASC',TYPE='OLD')
C                                   ! Defaults to
C                                   ! sequential access,
C                                   ! formatted reads
        DO 100 I=1,5
C Read record from file
        READ (1,10) N,REC
10      FORMAT (Q,40A2)
C Write record at terminal
        WRITE (5,20) (REC(K),K=1,(N+1)/2)
20      FORMAT (' ',40A2)
100     CONTINUE
C Close file and exit
        CLOSE (UNIT=1)
        CALL EXIT
        END
```

Run Session

```
>RUN SEQFOR
1
22
333
4444
Now is the time for all good.
```

Example 1Ø-3 Program to Read File Created in 1Ø-1

Example 1Ø-4, CRESEQFIX, creates a file, FIXED.ASC, containing fixed length records of 16 bytes each. In a file of fixed length records, the size of each record is kept in the header of the file rather than in the first two bytes of the record itself. In the file dump you will see that the first input record, containing a 1, creates a record consisting of 61 (ASCII) and 15 blanks (4Ø(8)). The next record is 62, 62, and 14 blanks, etc.

One advantage of a file of fixed length records is that the file may be accessed in DIRECT (or random) mode for both READ and WRITE. The disadvantage of a fixed length record is that, assuming a 16-byte record, a record containing one byte and a record containing 16 bytes occupies the same space on the disk. (Direct access is not available on a tape or cassette.) If you have a wide disparity in record sizes, say 1Ø and 8Ø, it may not be practical to use fixed length records. However, where disk space is not a problem, using direct access to a sequential file might be very useful.

```
        PROGRAM CRESEQFIX    !CREATE FILE SEQUENTIALLY
C
C FILE CRESEQFIX.FTN
C
C This task creates a file  FIXED.ASC of  fixed-length
C records using sequential record access.  The records
C are input from the terminal and written to the file.
C The process stops when the operator types  CTRL/Z at
C the terminal.
C
        BYTE BUFF(80)
        INTEGER LEN
C OPEN FILE - Default access is sequential, default form
C       is formatted I/O with sequential access.
C
        OPEN    (UNIT=1,NAME='FIXED.ASC',TYPE='NEW',
        1 RECORDTYPE='FIXED',RECORDSIZE=16)
C Loop
10      READ (5,11,END=100) LEN,BUFF       ! Read record
11      FORMAT (Q,80A1)
        WRITE (1,12) (BUFF(I),I=1,LEN)     ! Write record
12      FORMAT (80A1)                      !  to file
        GO TO 10
C Close file and exit
100     CLOSE   (UNIT=1)
        CALL EXIT
        END
```

```
Run Session

1
22
333
4444
Now is the time for all good.
```

```
Dump of DR2:[305,301]FIXED.ASC;3 - File ID 40573,6,0
                 Virtual block 0,000001 - Size 512. bytes


000000  061 040 040 040 040 040 040 040 040 040 040 040 040 040 040 040
000020  062 062 040 040 040 040 040 040 040 040 040 040 040 040 040 040
000040  063 063 063 040 040 040 040 040 040 040 040 040 040 040 040 040
000060  064 064 064 064 040 040 040 040 040 040 040 040 040 040 040 040
000100  122 157 163 145 163 040 141 162 145 040 162 145 144 056 040 040
```

Example 1Ø-4 Creating a File With Sequential,
Fixed Length Records

Example 1Ø-5, READFIXED, prompts you for the record number of the record you want from the file FIXED.ASC, displays the record and then allows you to replace the record if you wish. Note that the file was created as a sequential file with fixed length records and is being accessed as DIRECT. Since the record size is in the header of the file, it is not necessary to describe the record size in the OPEN. Note that both the READ and the WRITE to unit 1 use the formatted, direct form, i.e.:

```
     READ (1'NO,1Ø)
                              and
     WRITE (1'NO,1Ø)
```

One precaution here is that if you attempt to replace a record with a longer record (in this case 16 bytes) than the original, the new record will be truncated on the right.

As you can see from the run session in CRESEQFIX, the third record originally contained 333. This was replaced with "Now is the Time", as is shown by running READFIXED a second time and displaying record 3 again.

```
        PROGRAM READFIXED
C
C File READFIXED.FTN
C
C This task asks you which record you want from  FIXED.ASC,
C and displays the record on the terminal. It then asks  if
C you wish to replace the record and if so asks for the new
C record.
C
        CHARACTER*16 REC,NEW
C Open file
        OPEN (UNIT=1,NAME='FIXED.ASC',TYPE='OLD',ACCESS='DIRECT',
        1 FORM='FORMATTED')
C Read record from file
        TYPE *, 'Enter record number you want.'
        READ *,NO
        READ (1'NO,10)REC  !Get record number NO
10      FORMAT (A16)
C Write record at terminal
        WRITE (5,20) REC
        TYPE *, 'Do you want to replace the record? Y or N '
        READ(5,10)ANS
        IF (ANS.EQ.'N'.OR.ANS.EQ.'n') GO TO 100
        TYPE *, 'Enter new record.'
        READ(5,10)NEW
20      FORMAT (' ',A16)
        WRITE(1'NO,10)NEW
100     CONTINUE
C Close file and exit
        CLOSE (UNIT=1)
        CALL EXIT
        END
```

Run Session

```
>RUN READFIXED
Enter record number you want.
3
333
Do you want to replace the record? Y or N
Y
Enter new record.
Now is the time.
>RUN READFIXED
Enter record number you want.
3
Now is the time.
>
```

Example 10-5 Reading a Fixed Length Record

Example 1Ø-6, DIRFOR, illustrates the creation of a file via direct access. The example creates record 1 through record 5, in order. It is not necessary to create the records in order, nor must there be a record n-1 if record n exists. Hence you may have a sparse file, containing only those records whose record numbers are specifically used in a WRITE.

Note that the RECORDSIZE = 1Ø is used in the OPEN. Since this is a formatted record, the recordsize of 1Ø means that each record will be 1Ø bytes. Hence the first record, containing 1,1,1,1,1, is filled with five blanks (4Ø,4Ø,4Ø,4Ø,4Ø). The fifth record, which contains just a 5, is filled with nine blanks. The rest of the file is filled with zeros.

```
        PROGRAM DIRFOR
C
C File DIRFOR.FTN
C
C This task creates a file DIRFOR.DAT using direct
C access formatted writes.
C
C Direct access formatted writes are available in
C FORTRAN IV-PLUS and FORTRAN-77 only
C
        INTEGER REC(10)
C
C Open file
        OPEN (UNIT=2,NAME='DIRFOR.DAT',ACCESS='DIRECT',
        1 TYPE='NEW',FORM='FORMATTED',RECORDSIZE=10)
        DO 100 I=1,5
C Prompt for input
        WRITE (5,25)
25      FORMAT ('$ INPUT UP TO 10 DIGITS: ')
C Read record from terminal
        READ (5,50)N,REC
50      FORMAT (Q,10I1)
C Write record to disk
        WRITE (2'I,80) (REC(K),K=1,N)
80      FORMAT (10I1)
100     CONTINUE
        CLOSE (UNIT=2)
        CALL EXIT
        END
```

Run Session

```
>RUN DIRFOR
INPUT UP TO 10 DIGITS: 11111
INPUT UP TO 10 DIGITS: 2222
INPUT UP TO 10 DIGITS: 3333333333
INPUT UP TO 10 DIGITS: 444
INPUT UP TO 10 DIGITS: 5
>
```

```
Dump of DR2:[305,301]DIRFOR.DAT;17 - File ID 40653,10,0
                 Virtual block 0,000001 - Size 512. bytes


000000  061 061 061 061 061 040 040 040 040 040 062 062 062 062 040 040
000020  040 040 040 040 063 063 063 063 063 063 063 063 063 063 064 064
000040  064 040 040 040 040 040 040 040 065 040 040 040 040 040 040 040
000060  040 040 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

Example 10-6 Creating a Direct Access File

Example 1Ø-7, DIRUNF, creates a file with unformatted, direct access records. Since the file is unformatted, the record size of 5 does not refer to five bytes but rather to five storage units where a storage unit is defined as four bytes. Hence each record is 2Ø bytes long. Note that the file dump shows words rather than bytes. This is because the data type is INTEGER which has two bytes for each value. The first record contains five words of ØØØØ1 padded with five words of ØØØØØ to pad out the 2Ø-byte record.

```
        PROGRAM DIRUNF
C
C File DIRUNF.FTN
C
C This task creates a file DIRUNF.DAT using direct
C access unformatted writes.
C
        INTEGER REC(10)
C Open file
        OPEN (UNIT=4,NAME='DIRUNF.DAT',ACCESS='DIRECT',
        1 TYPE='NEW',RECORDSIZE=5)    ! Defaults to
C                                     ! unformatted
        DO 100, I=1,5
C Prompt fpr input
        WRITE (5,25)
25      FORMAT (' INPUT UP TO 10 DIGITS:')
C Read record from terminal
        READ (5,10) N,REC
10      FORMAT (Q,10I1)
C Write record to disk
        WRITE (4'I) (REC(K),K=1,N)
100     CONTINUE
        CALL EXIT
        END
```

```
Run Session
>RUN DIRUNF
INPUT UP TO 10 DIGITS: 11111
INPUT UP TO 10 DIGITS: 2222
INOUT UP TO 10 DIGITS: 3333333333
INPUT UP TO 10 DIGITS: 444
INPUT UP TO 10 DIGITS: 5
>
```

```
Dump of DR2:[305,301]DIRUNF.DAT;13 - File ID 40661,5,0
                  Virtual block 0,000001 - Size 512. bytes


000000    000001 000001 000001 000001 000001 000000 000000 000000
000020    000000 000000 000002 000002 000002 000002 000000 000000
000040    000000 000000 000000 000000 000003 000003 000003 000003
000060    000003 000003 000003 000003 000003 000003 000004 000004
000100    000004 000000 000000 000000 000000 000000 000000 000000
000120    000005 000000 000000 000000 000000 000000 000000 000000
000140    000000 000000 000000 000000 000000 000000 000000 000000
```

Example 10-7 Creating an Unformatted, Direct Access File

Example 1Ø-8, SEQUNF, illustrates the SEGMENTED record type, even though the OPEN does not contain RECORDTYPE = 'SEGMENTED'. This is because SEGMENTED is the default record type for an UNFORMATTED, SEQUENTIAL file. This is the default file type created by an unformatted WRITE in FORTRAN. Hence, if there had been no OPEN statement, and the write statement was as shown:

```
        WRITE(1)(REC(K),K=1,N)
```

the file created would default to FORØØ1.DAT (ØØ1 because 1 was used in the WRITE) and the record type would be SEGMENTED. The advantage of a file with segmented records is that there is no limit to its size, i.e., a single record could be many physical blocks on a disk. The disadvantage of a file with segmented records is that it cannot be read by any other high level languages.

```
        PROGRAM SEQUNF
C
C This task creates a file SEQUNF.DAT using sequential
C unformatted writes
C
        BYTE REC(10)
C
C Open file
        OPEN (UNIT=1,NAME='SEQUNF.DAT',TYPE='NEW',
        1 FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
C Loop for 5 records
        DO 100 I=1,5
C Prompt for input
        WRITE (5,25)
25      FORMAT ('$ INPUT UP TO 10 DIGITS: ')
C Read record from terminal
        READ (5,50) N,REC
50      FORMAT (Q,10I1)
C Write record to disk
        WRITE (1) (REC(K),K=1,N)
100     CONTINUE
        CLOSE (UNIT=1)
        CALL EXIT
C Error routine
900     WRITE (5,950)
950     FORMAT (' THERE WAS A FILE OPEN ERROR')
        CALL EXIT
        END
```

Run Session

```
>RUN SEQUNF
INPUT UP TO 10 DIGITS: 11111
INPUT UP TO 10 DIGITS: 2222
INPUT UP TO 10 DIGITS: 3333333333
INPUT UP TO 10 DIGITS: 444
INPUT UP TO 10 DIGITS: 5
>
```

```
Dump of DR2:[305,301]SEQUNF.DAT;16 - File ID 40675,3,0
                Virtual block 0,000001 - Size 512. bytes

000000    000014 000003 000001 000001 000001 000001 000001 000012
000020    000003 000002 000002 000002 000002 000026 000003 000003
000040    000003 000003 000003 000003 000003 000003 000003 000003
000060    000003 000010 000003 000004 000004 000004 000004 000003
000100    000005 000000 000000 000000 000000 000000 000000 000000
```

Example 10-8 Creating a Segmented File

Example 10-9, FWRITE, illustrates how a Block I/O routine written in MACRO can be called by a FORTRAN program. Block I/O is not directly available in FORTRAN.

```
        .TITLE  FWRITE
        .IDENT  /01/
        .ENABL  LC              ; Enable lower case
;+
; FWRITE is a FORTRAN-callable block I/O subroutine.
;
; Subroutine call:
;
; CALL FWRITE (ilun,ibuf,isiz,ivb,iefn,iosb,ierr)
;
;       where   ilun    is logical unit number
;               ibuf    is block buffer address
;               isiz    is block buffer size (in bytes)
;               ivb     is address of 2-word v.b. number
;               iefn    is event flag
;               iosb    is I/O status block
;               ierr    is a status code
;                       +1  =   Success
;                       -1  =   $FCHNL ERROR
;                       -2  =   CANNOT CHANGE RECORD ACCESS
;                       -4  =   WRITE$ REJECTED
;-
        .MCALL  WRITE$,FDRC$R ; System FCS macros
IOSB:   .BLKW   2
;
FWRITE::
        MOV     @2(R5),R2       ; Lun
        MOV     @#$OTSV,R3      ; Address of FORTRAN
                                ;  work area
        CALL    $FCHNL          ; Get FORTRAN FDB
        BCS     ERROR1          ; Branch on error
        ADD     #14,R0          ; Point to FCS FDB
        FDRC$R  ,#FD.RWM        ; Change record access
                                ; to block I/O
        BCS     ERROR2          ; Branch on error
        WRITE$  ,4(R5),@6(R5),10(R5),@12(R5),14(R5),#0
                                ; Issue write
        BCS     ERROR3          ; Branch on error
        MOV     #1,@16(R5)      ; Return success code
        RETURN                  ;
ERROR1: MOV     #-1,@16(R5)     ; Return FCHNL failure
                                ;  code
        RETURN
ERROR2: MOV     #-2,@16(R5)     ; Return couldn't change
                                ;  access code
        RETURN
ERROR3: MOV     #-4,@16(R5)     ; Return write rejected
                                ;  code
        RETURN
        .END
```

Example 10-9 Creating a File Using Block I/O (Sheet 1 of 3)

```
      PROGRAM BLOCK1
C
C File BLOCK1.FTN
C
C BLOCK1 creates a file BLOCK.ASC using FWRITE, a
C FORTRAN callable subroutine written in MACRO-11.
C
C Subroutine call:
C
C     CALL FWRITE(ilun,ibuff,isize,ivbn,iefn,iosb,ierr)
C
C     where   ilun    is the logical unit number
C             ibuff   is the array to be written
C             isize   is the size of the buffer (bytes)
C             ivbn    is a 2-integer vbn (high,low)
C             iefn    is an event flag number
C             iosb    is a 2-integer I/O status block
C             ierr    is an status code,
C                     +1  =   SUCCESS
C                     -1  =   $FCHNL ERROR
C                     -2  =   CANNOT CHANGE RECORD ACCESS
C                     -4  =   WRITE$ REJECTED
C
C Task-build instructions:
C
C     >LINK/MAP/CODE:FPP BLOCK1,FWRITE,LB:[1,1]F4POTS-
C     ->/LIBRARY
C
      INTEGER WDBUFF(256),IVBN(2)
      INTEGER ISIZE,IEFN,IOSB(2),IERR
      BYTE IOST(2),CHAR,CHBUFF(512)
C
      EQUIVALENCE (IOSB,IOST) ! For accessing I/O status
      EQUIVALENCE (WDBUFF,CHBUFF) ! For accessing data
      DATA ILUN,ISIZE,IEFN /1,512,2/
C Get virtual block #
      TYPE 5
5     FORMAT ('$VIRTUAL BLOCK NUMBER (LOW ONLY): ')
      ACCEPT 6,IVBN(2)
6     FORMAT (I6)
      IVBN(1) = 0             ! High VBN = 0
C Get character to insert
      TYPE 7
7     FORMAT ('$CHARACTER: ')
      ACCEPT 8,CHAR
8     FORMAT (1A1)
C Fill buffer with character
      DO 9,I=1,ISIZE
9     CHBUFF(I) = CHAR
```

Example 10-9 Creating a File Using Block I/O (Sheet 2 of 3)

```
C Open file
        OPEN (UNIT=ILUN,NAME='BLOCK.ASC',TYPE='NEW')
C Call subroutine to write block of data
        CALL FWRITE (ILUN,WDBUFF,ISIZE,IVBN,IEFN,IOSB,
        1IERR)
        IF (IERR .LT. 0) GOTO 200
        TYPE 20
20      FORMAT (' 1 BLOCK BEING WRITTEN TO FILE')
C Wait for write to complete
        CALL WAITFR(IEFN,IDSW)
        IF (IDSW .LT. 0) GOTO 40  ! Check for dir error
        IF (IOST(1) .LT. 0) GOTO 100 ! Chcek for I/O
C                                   !  error on write
        WRITE (5,30)IOSB(2)
30      FORMAT (' WRITE COMPLETED,',I6,' BYTES WRITTEN
        1TO FILE')
        GOTO 300
C
40      TYPE 45,IDSW
45      FORMAT (' DIRECTIVE ERROR. IDSW = ',I6)
        GOTO 300
C
100     WRITE (5,110) IOST(1)
110     FORMAT (' I/O ERROR. I/O STATUS = ',I6)
        GOTO 300
C
200     TYPE    210,IERR
210     FORMAT (' FCS ERROR, CODE = ',I6)
C
300     CLOSE (UNIT=ILUN)
        CALL EXIT
        END
```

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;14 - File ID 40701,2,0
                 Virtual block 0,000001 - Size 512. bytes

      Contains whatever was previously in that block on the disk

Dump of DR2:[305,301]BLOCK.ASC;14 - File ID 40701,2,0
                 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
                                        .
                                        .
                                        .
000760  145 145 145 145 145 145 145 145 145 145 145 145 145 145 145 145
```

Example 10-9 Creating a File Using Block I/O (Sheet 3 of 3)

APPENDICES

APPENDIX A GLOSSARY

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(1Ø) byte (256(1Ø) word) block of data on a block addressable volume.

To achieve device independence, each block addressable volume is organized into logical blocks, numbered Ø 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 Ø 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.

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

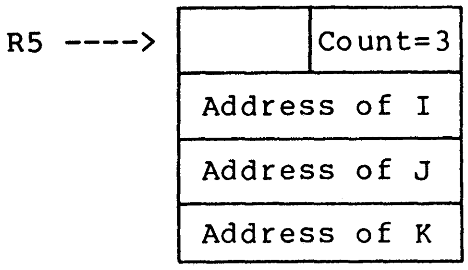
CALLING A MACRO-11 SUBROUTINE FROM A FORTRAN PROGRAM

FORTRAN Program Call:

```
                CALL SUBNAM (I,J,K)
```

MACRO translation:

1. Set up table of arguments.



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 SUBROUTINE 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:0 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 20K 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:

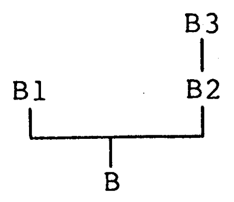
   LINK A,B

   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 100(10) words

   B3.OBJ has Psect Q (LCL or GBL) - length 50(10) words

   When task-built:

   If CON, yields 150(10) words in Psect Q in overlay segment B2 (allocation collected, since it is all in the same overlay segment).

   If OVR instead, 100(10) words in Psect Q in overlay segment B2. B3's 50(10) words are the same as B2's first 50(10) 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(100)
```

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=Ø
Option? PAR=OVRLIB:14ØØØØ:4ØØØØ
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.

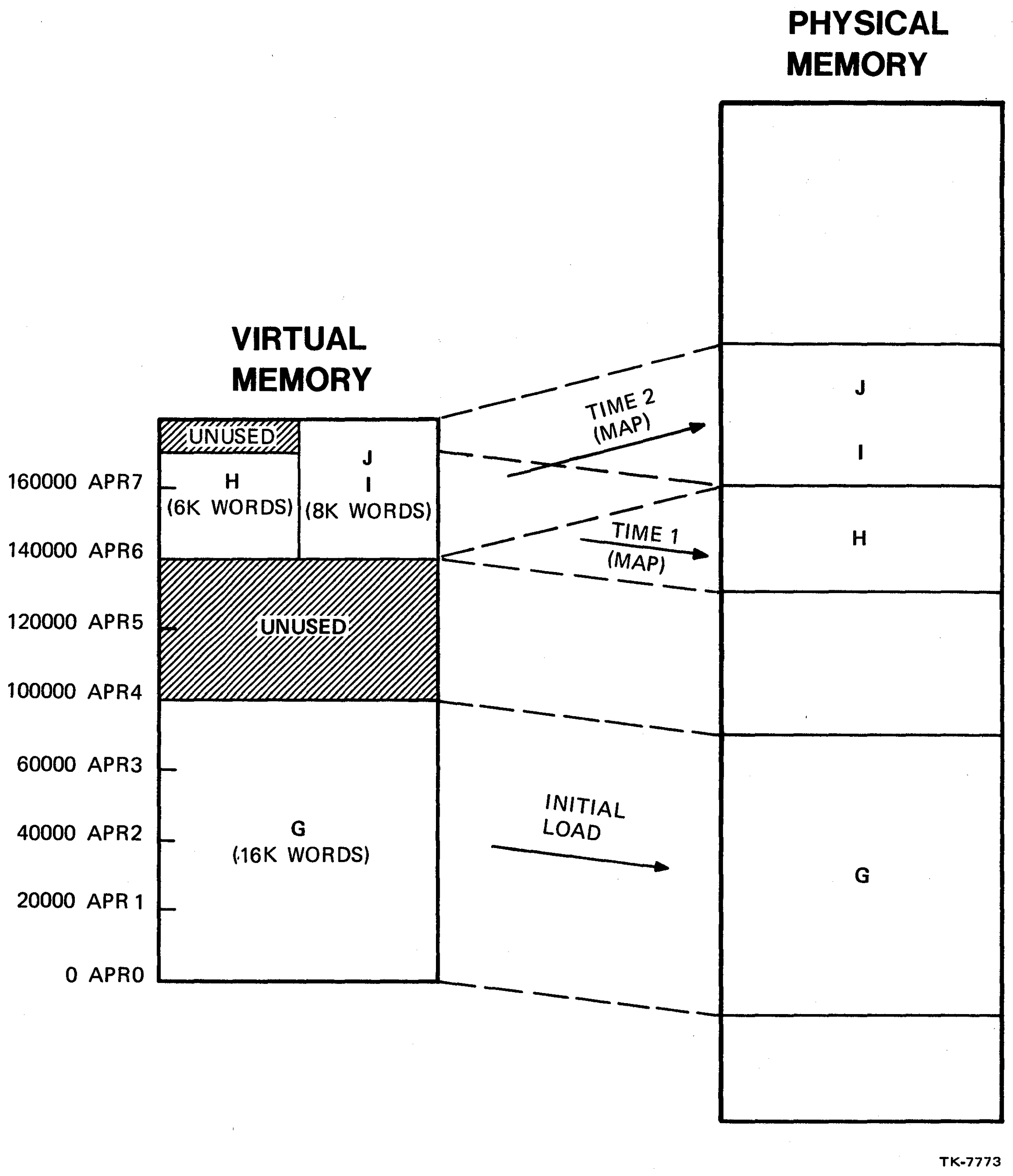


Figure F-1 A Shared Region With Memory-Resident Overlays

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>
```

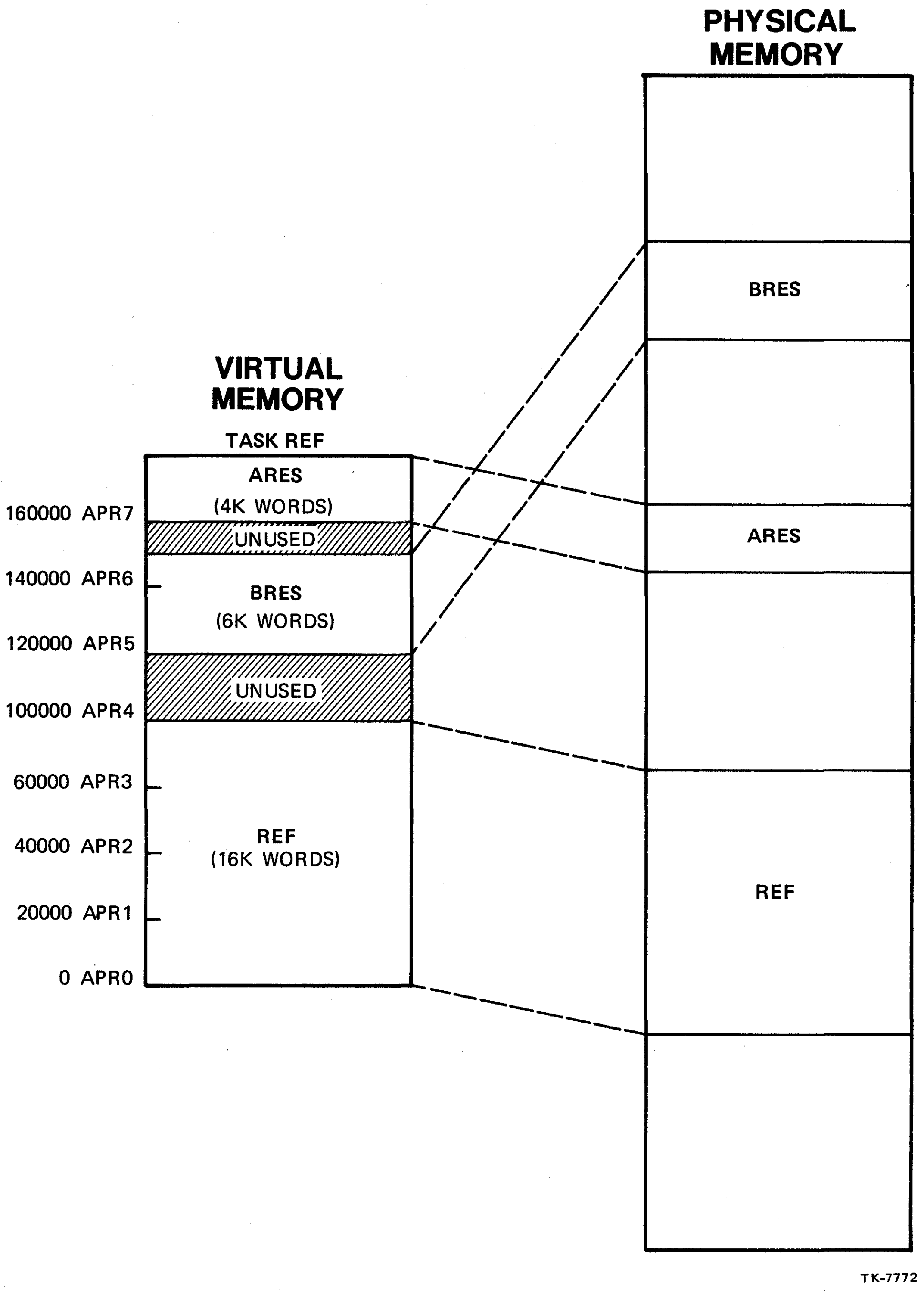


Figure F-2 Referencing Two Resident Libraries

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=0
Option? PAR=F4PRES:120000:60000
Option? <RET>
```

Referencing task is created using normal procedure to reference the library F4PRES.

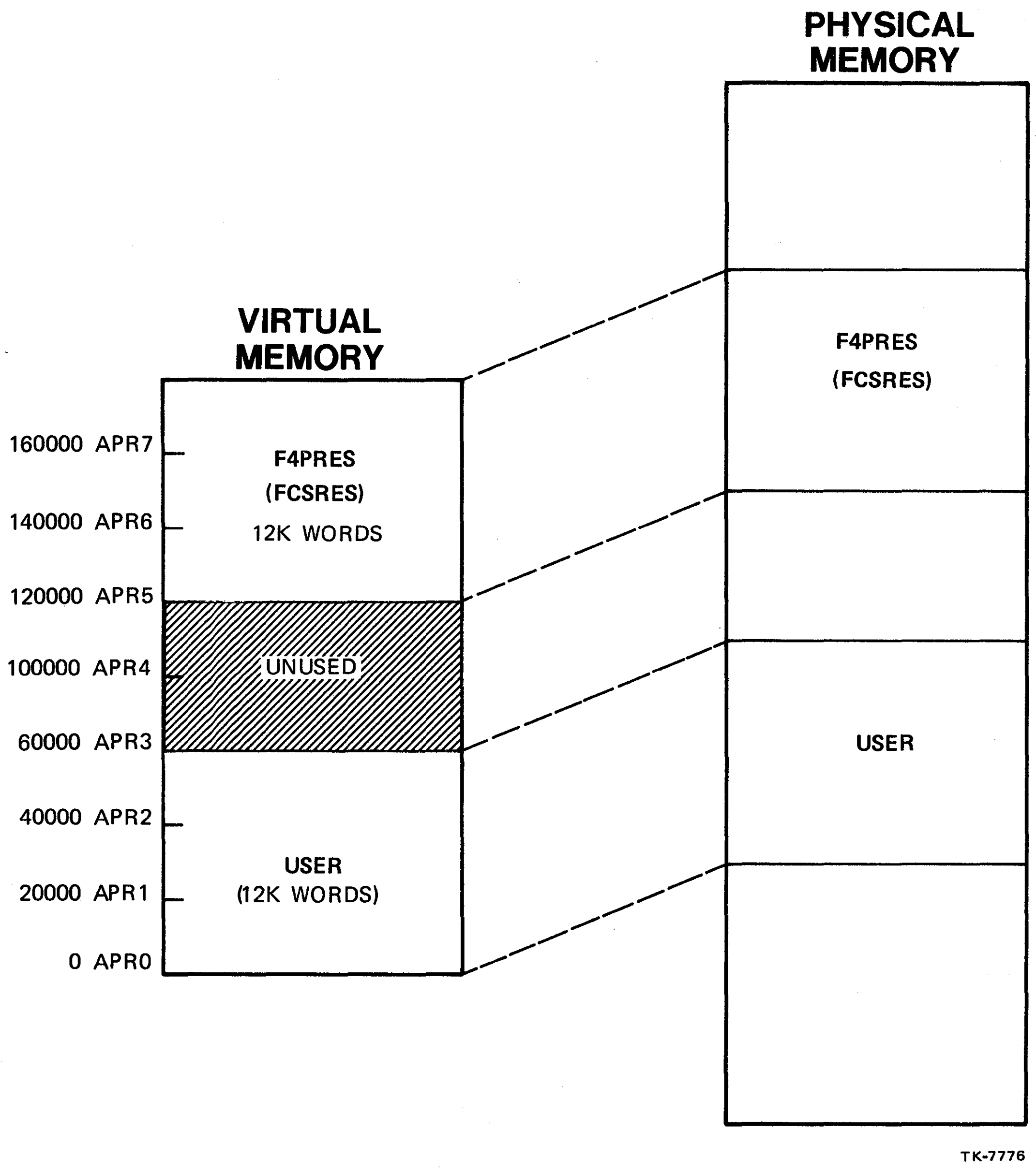


Figure F-3 Referencing Combined Libraries

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.

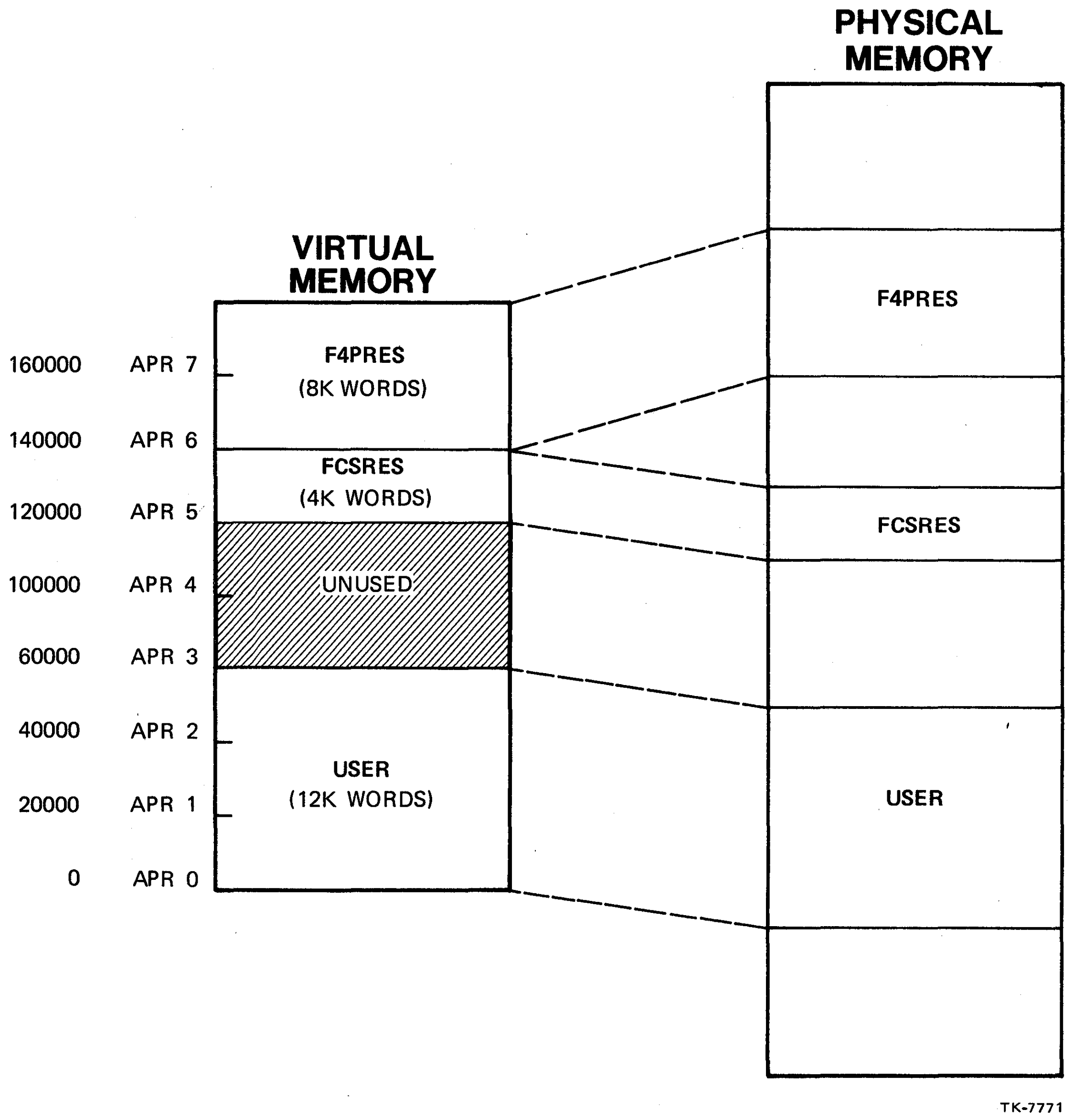


Figure F-4 Building One Library, Then Building a Referencing Library

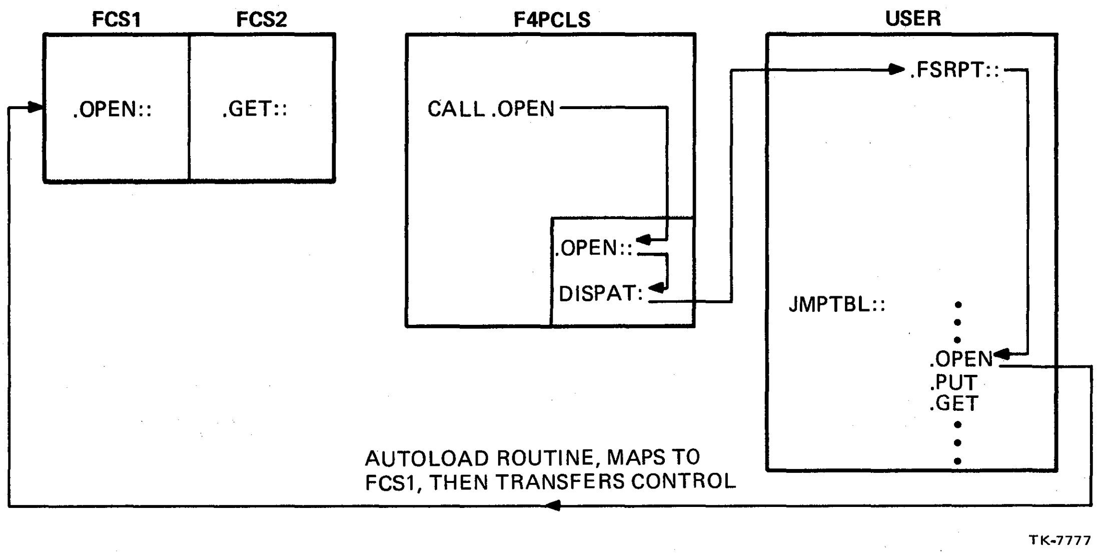


Figure F-5 Revectoring

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 is 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.

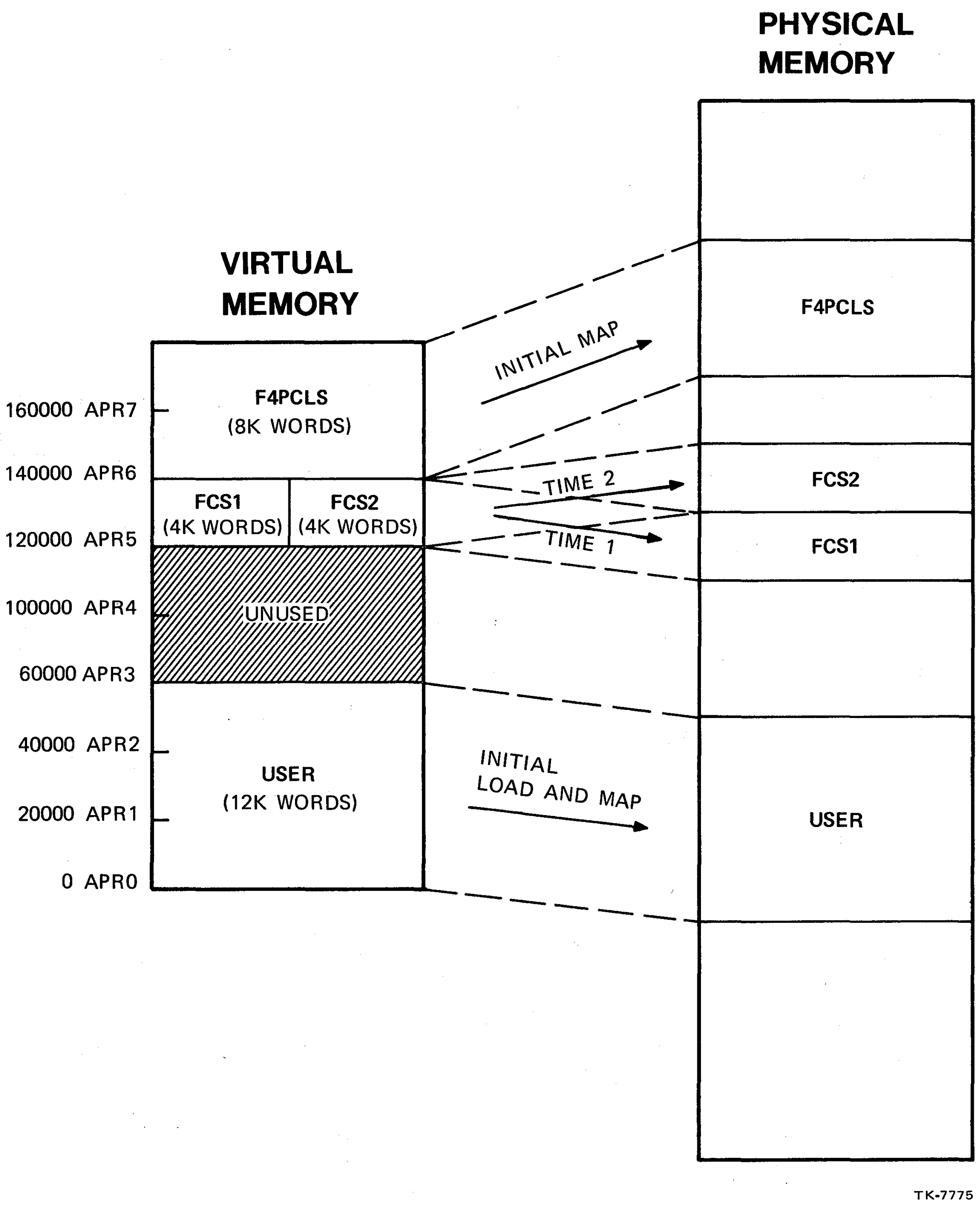


Figure F-6 Using Revectoring When Referenced Library Has Overlays

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 <u>RSX-11M/M-PLUS Task Builder Manual</u> 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>
```

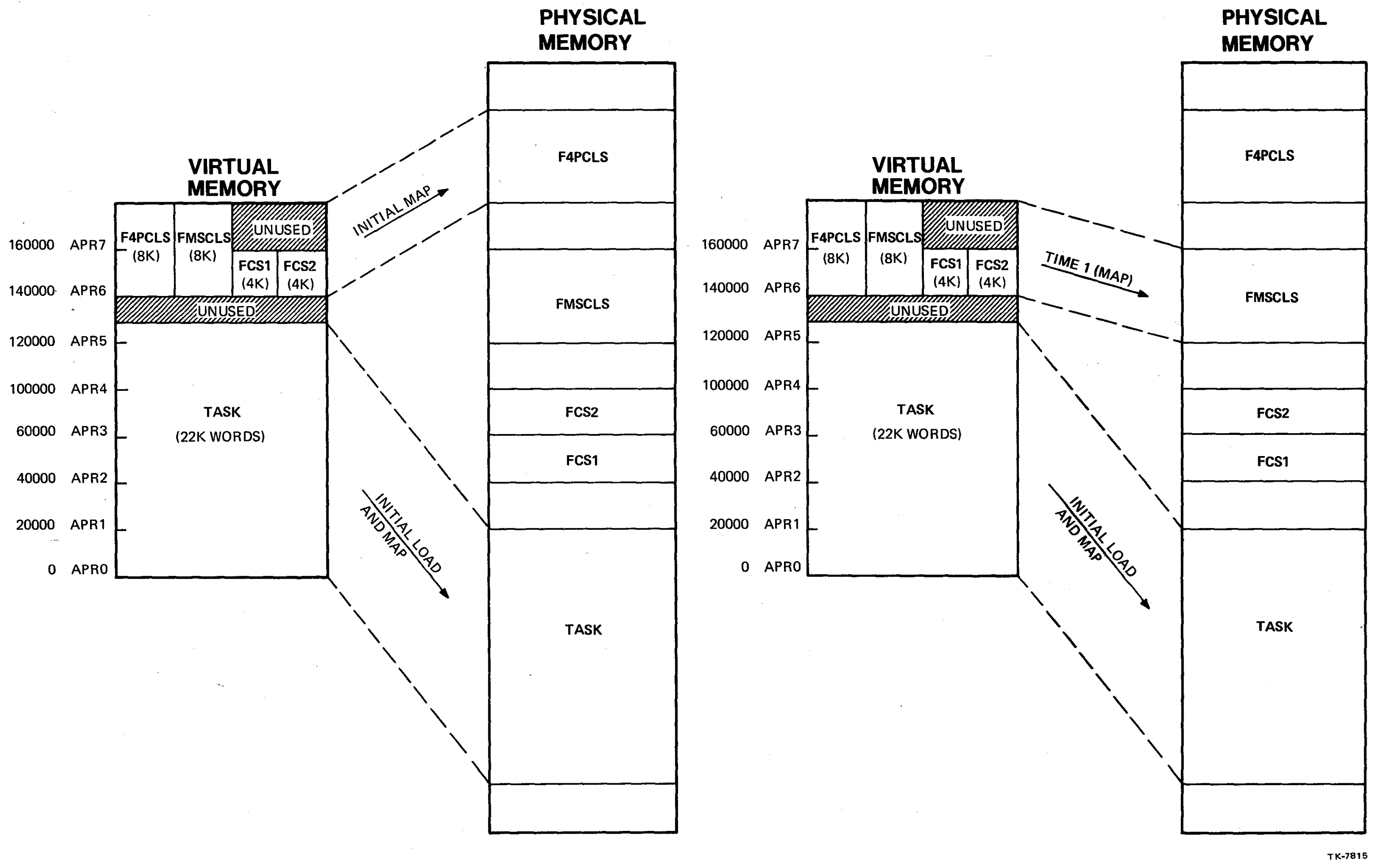


Figure F-7 Cluster Libraries (Sheet 1 of 2)

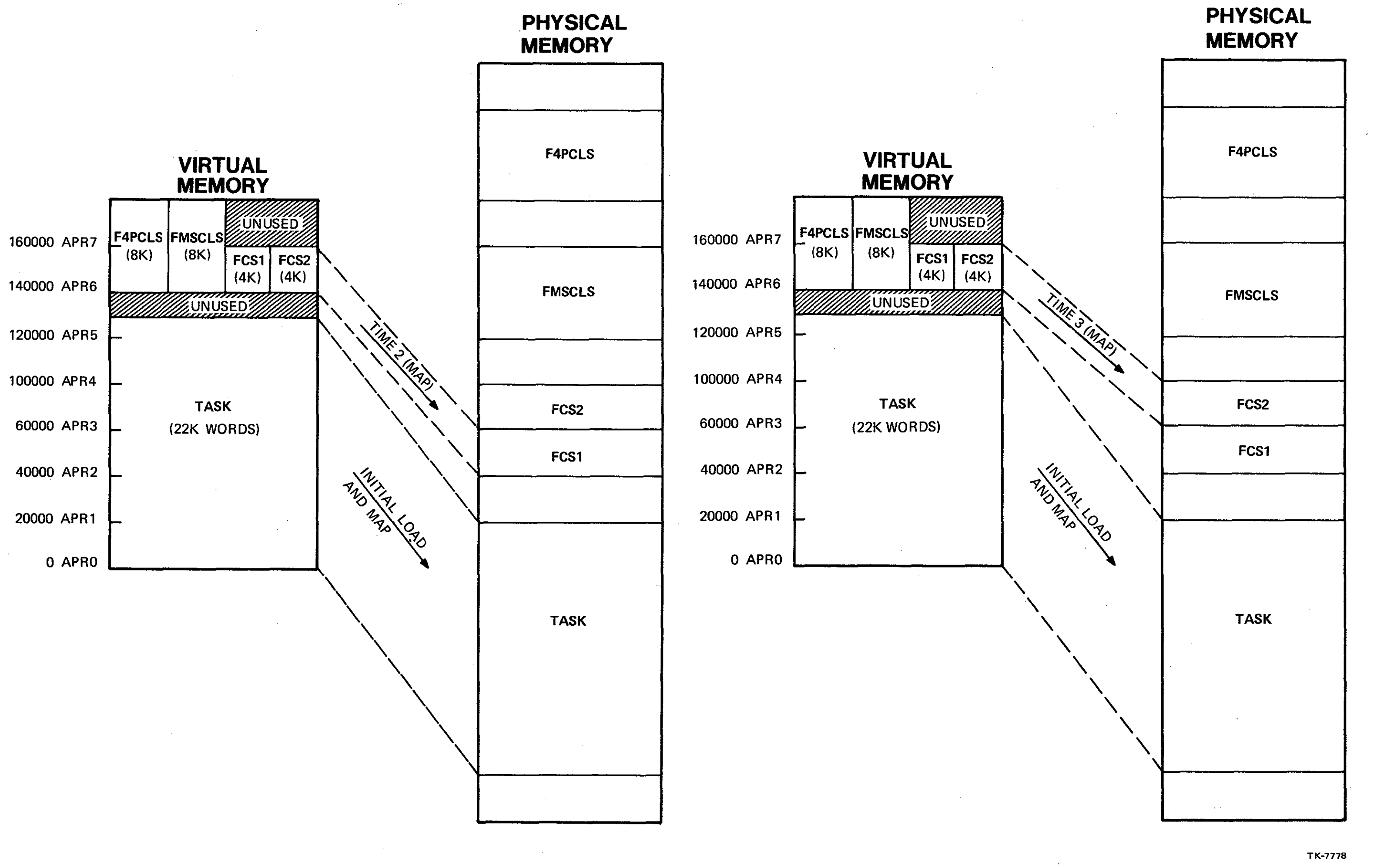


Figure F-7 Cluster Libraries (Sheet 2 of 2)

APPENDIX G ADDITIONAL EXAMPLE

The following example READF.FTN, should be available on-line, probably under UFD [2Ø2,1]. It is needed for the Tests/Exercises. Therefore, it is listed here in case it is not available on-line at your site.

```
      PROGRAM READF
C
C File READF.FTN
C
C This task sets event flag 1 and then reads
C flags 1 to 16 and displays them. The display is
C a series of 16 digits, corresponding to flag
C 16 on the left through flag 1 on the right.
C A 1 indicates that the flag is set, a 0
C indicates that the flag is clear.
C
      INTEGER*2   IEVF(16),IDSW
C Set event flag 1.
      CALL SETEF (1,IDSW)
C Branch on directive error
      IF (IDSW .LT. 0) GOTO 1000
C Read the event flags into the array ievf. Note
C that in FORTAN, we can only read 1 flag at a time
      DO 20 I=1,16
      CALL READEF (I,IDSW)
C Branch on directive error
      IF (IDSW .LT. 0) GOTO 1100
C Check IDSW value, 2 means set, 0 means clear
C Set the ievf value accordingly (1 means set, 0
C means clear)
      IF (IDSW .EQ. 2) GOTO 10
      IEVF(I)=IDSW
      GOTO 20
10    IEVF(I)=1
20    CONTINUE
C Write out flag settings, starting with flag 16.
      WRITE (5,30)
30    FORMAT (' EVENT FLAGS 16. TO 1. ARE:')
      WRITE (5,40) (IEVF(J), J=16,1,-1)
40    FORMAT (' ',16I2)
      CALL EXIT
C Come here on directive errors
1000  WRITE (5,1010) IDSW
1010  FORMAT (' ERROR SETTING FLAG. ERROR CODE = ',I5)
      CALL EXIT
1100  WRITE (5,1110) IDSW
1110  FORMAT (' ERROR READING FLAG. ERROR CODE = ',I5)
      CALL EXIT
      END
```

Example G-1 Reading the Event Flags (For Exercise 1-1)

APPENDIX H
LEARNING ACTIVITY ANSWER SHEET

Learning Activity 2-1 (Directives)

1. Either: a) Do some work, then check the flag by using the CALL CLREF (35,IDSW) directive. Check the DSW. IS.SET (=+2) means the flag was set; IS.CLR (=0) 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)

(Using Example 6-5)

1.

```
        .ROOT-LIB-*!(P-LIB,Q-LIB)
  LIB:  .FACTR LB:[1,1]FOROTS/LB
        .END
```

2.

```
        LINK/MAP ROOT,P,Q,LB:[1,1]FOROTS/LB
```

Learning Activity 6-2

(Using Example 6-6)

1. Overlay tree.

```
            JOB1      JOBXX
              |         |
              -----------
                   |
                   A              B
                   |              |
                   ----------------
                          |
                        TOTAL
                          |
                        MAIN
```

2.

```
            .ROOT MAIN-TOTAL-LIB-*(A-LIB-(JOB1-LIB,JOBXX-LIB),B-LIB)
    LIB:    .FACTR LB:[1,1]FOROTS/LIB
            .END
```

3.

```
            .ROOT MAIN-TOTAL-LIB-*!(A-LIB-!(JOB1-LIB,JOBXX-LIB,B-LIB)
    LIB:    .FACTR LB:[1,1]FOROTS/LIB
            .END
```

4.

```
            .ROOT MAIN-TOTAL-LIB-*(A-LIB-(JOB1-LIB,JOBXX-LIB),B-LIB)
    LIB:    .FACTR LB:[1,1]FOROTS/LB
            .END
```

EY-0061E-TP-0001

Programming RSX-11M in FORTRAN

Tests/Exercises

Prepared by Educational Services
of
Digital Equipment Corporation

CONTENTS

INTRODUCTION

This book contains tests/exercises for two different courses, Programming RSX-11M in MACRO and Programming RSX-11M in FORTRAN. Most of the questions apply to both courses. If a question begins with "In MACRO" or "In FORTRAN", that question applies only to the specified course. Solutions are provided for all tests/exercises. Where it is appropriate, separate solutions are provided for MACRO and FORTRAN. Solutions which involve programs should also be available on-line.

Check the Student Guide in the Student Workbook for your course for information on how to use the tests/exercises.

Using System Services

TEST/EXERCISE

1. Match the function with the type of system service used to perform it.

Function

___ a. The tasks send data back and forth to each other

___ b. The tasks read data from a file on disk

___ c. The tasks get input from an operator at a terminal

Type of System Service

1. System and task information
2. Task control
3. Task communication/coordin-ation
4. I/O to peripheral devices
5. File and record access
6. Memory use

2. Draw a figure to illustrate a method of providing a system service through the Executive.

Using System Services

SOLUTION

1. Match the function with the type of system service used to perform it.

	Function	Type of System Service
3	a. The tasks send data back and forth to each other	1. System and task information
		2. Task control
5	b. The tasks read data from a file on disk	3. Task communication/coordination
4	c. The tasks get input from an operator at a terminal	4. I/O to peripheral devices
		5. File and record access
		6. Memory use

2. Draw a figure to illustrate a method of providing a system service through the Executive.

See Figure 1-1 or 1-2

Directives

TEST/EXERCISE

1. In MACRO-11

   a. Modify the task READF to use the $C form of the Read Event Flags directive.

   b. Modify the task READF to use the $S form of the Read Event Flags directive.

2. In FORTRAN, modify the task READF to set all of the odd numbered flags from 1 to 15(1Ø).

3. Modify WFLAG and SFLAG to use a global event flag instead of a group global event flag. Omit any unnecessary code in the tasks. Check with your instructor to find out which event flag to use.

4. Write a task which does some work and periodically checks a group global event flag. Have it display a message and exit when the flag has been set. Write another task, or modify SFLAG to set the flag.

5. Add a requested exit AST routine to WFLAG.

6. In MACRO-11, add an odd address trap SST routine to the task SST. Include an instruction which causes the trap to occur.

Directives

SOLUTION

1.a

```
        .TITLE  READF
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; File LEX21A.MAC
;
; Modified to use the $C form of the Read All Event ;;EX
; Flags directive
;
; 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
;-

        .MCALL  RDAF$C,SETF$,EXIT$S,DIR$ ; System macros
                                                  ;;EX
BUFF:   .BLKW   4                       ; Buffer for event flag
                                        ;  values

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
        RDAF$C  BUFF                    ; Read the event flags;;EX
                                        ;  (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
```

Directives

SOLUTION

2.
```
C       READF.FTN
C
C File LEX22.FTN
C
C Modified for exercises. Set odd numbered flags. !!EX
C
C This task sets event flag 1 and then reads
C flags 1 to 16 and displays them
C
        INTEGER*2   IEVF(16),IDSW
C Set odd event flags.                            !!EX
        DO 5 K=1,15,2                             !!EX
        CALL SETEF (K,IDSW)                       !!EX
C Branch on directive error
        IF (IDSW .LT. 0) GOTO 1000
5       CONTINUE                                  !!EX
C Read the event flags into the array ievf. Note
C that in FORTAN, we can only read 1 flag at a time
        DO 20 I=1,16
        CALL READEF (I,IDSW)
C Branch on directive error
        IF (IDSW .LT. 0) GOTO 1100
C Check IDSW value, 2 means set, 0 means clear
C Set the ievf value accordingly (1 means set, 0
C means clear)
        IF (IDSW .EQ. 2) GOTO 10
        IEVF(I)=IDSW
        GOTO 20
10      IEVF(I)=1
20      CONTINUE
C Write out flag settings, starting with flag 16.
        WRITE (5,30)
30      FORMAT (' EVENT FLAGS 16. TO 1. ARE:')
        WRITE (5,40) (IEVF(J), J=16,1,-1)
40      FORMAT (' ',16I2)
        CALL EXIT
C Come here on directive errors
1000    WRITE (5,1010) IDSW
1010    FORMAT (' ERROR SETTING FLAG. ERROR CODE = ',I5)
        CALL EXIT
1100    WRITE (5,1110) IDSW
1110    FORMAT (' ERROR READING FLAG. ERROR CODE = ',I5)
        CALL EXIT
        END
```

SOLUTION

```
        PROGRAM WFLAG
C
C FILE LEX23A.FTN
C
C Modified to use event flag 35(10)             !!EX
C
C This task creates the group global event flags, and
C then clears event flag 65. and waits for it to be set.
C When the flag is set, it writes a message and exits
C
C Install and run instructions:
C
C       Run WFLAG, then run SFLAG. At least one of the
C       tasks must be installed, or else the RUN command
C       will try to install both tasks under the same
C       name (TTnn)
C
        WRITE (5,20)
20      FORMAT (' CLEAR AND WAIT FOR EF 35. TO BE SET')!!EX
        CALL CLREF (35,IDSW)                     !!EX
        IF (IDSW .LT. 0) GOTO 1100
        CALL WAITFR (35,IDSW)                    !!EX
        IF (IDSW .LT. 0) GOTO 1200
        WRITE (5,30)
30      FORMAT (' EF 35. HAS BEEN SET. FWAIT WILL NOW EXIT')
C                                                !!EX
        CALL EXIT
C Error processing
C
1100    WRITE (5,1110) IDSW
1110    FORMAT (' DIRECTIVE ERROR CLEARING EVENT FLAG 35.
        1 DSW = ',I5)                            !!EX
        CALL EXIT
1200    WRITE (5,1210) IDSW
1210    FORMAT (' DIRECTIVE ERROR WAITING FOR EVENT FLAG
        1 35. DSW = ',I5)
        CALL EXIT
        END
```

```
        .TITLE  SFLAG
        .IDENT  /01/
        .ENABL  LC              ; Enable lower case
;+
; FILE LEX23B.MAC
;
; Modified to use event flag 35.                ;;EX
;
; This task sets event flag 65.  It assumes that the
; group global event flags have already been created.
;
; Assemble and task-build instructions:
;
;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]SFLAG
;       LINK/MAP SFLAG,LB:[1,1]PROGSUBS/LIBRARY
```

Directives

SOLUTION

4.
```
        .TITLE  LEX24
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; FILE LEX24.MAC
;
; This program creates the group global event flags,
; clears event flag 65., does some work and periodically
; cehcks event flag 65. When the flag is set it writes a
; message and exits.
;
; Assemble and task-build instructions:
;
;       MACRO/LIST/OBJECT:WFLAG LB:[1,1]PROGMACS/LIB-;;EX
;       RARY,dev:[ufd]LEX24                          ;;EX
;       LINK/MAP WFLAG,LB:[1,1]PROGSUBS/LIBRARY
;
; Install and Run instructions:
;
;       Run WFLAG, then run SFLAG. At least one of the
;       tasks must be installed, or else the RUN command
;       will try to install both tasks under the same
;       name, TTnn.
;-
        .MCALL  EXIT$S,WTSE$C,CLEF$C,CRGF$C ; System
                                            ;  macros
        .MCALL  TYPE                    ; Supplied macro

START:  CLR     R0                      ; R0 used to identify
                                        ;  the error
        TYPE    <LEX24 IS CREATING THE GROUP GLOBAL EVENT FLAGS>
        CRGF$C                          ; Create group global
                                        ;  event flags
        BCC     OK                      ; Branch on directive ok
; If group global event flags already exist,
; just display message and continue
        CMP     $DSW,#IE.RSU            ; Check for efs already
                                        ;  in existence
        BNE     ERR1                    ; Branch on any other
                                        ;  dir error
        TYPE    <GROUP GLOBAL EVENT FLAGS ALREADY EXIST>
OK:     TYPE    <CLEAR EF 65. WORK UNTIL IT IS SET>
        CLEF$C  65.                     ; Clear event flag 65.
        BCS     ERR2                    ; Branch on directive
                                        ;  error
AGAIN:  CLR     R1                      ; Clear counter      ;;EX
; Loop 2**16 times, then check flag                          ;;EX
LOOP:   INC     R1                      ; Increment counter ;;EX
        BNE     LOOP                    ; Not yet cycled, loop;;EX
                                        ;  again             ;;EX
```

SOLUTION

```
                WRITE (5,10)
        10      FORMAT (' LEX24 IS CREATING THE GROUP GLOBAL EVENT FLAGS')
        C                                                         !!EX
                CALL CRGF (,IDSW)
                IF (IDSW .LT. 0) GOTO 900
        15      WRITE (5,20)
        20      FORMAT (' CLEAR EF 65. WORK UNTIL IT IS SET')
                CALL CLREF (65,IDSW)
                IF (IDSW .LT. 0) GOTO 1100
        22      DO 25 K=1,65535                                  !!EX
        25      CONTINUE                                         !!EX
                WRITE (5,28)                                     !!EX
        28      FORMAT (' COUNTER HAS CYCLED')                   !!EX
                CALL READEF (65,IDSW)                            !!EX
                IF (IDSW .LT. 0) GOTO 1200                       !!EX
                IF (IDSW .NE. 2) GOTO 22                         !!EX
                WRITE (5,30)
        30      FORMAT (' EF 65. HAS BEEN SET. LEX24 WILL NOW EXIT')
                CALL EXIT
        C Error processing
        C
        C Check for code of -17, meaning flags already exist
        900     IF (IDSW .NE. -17) GOTO 1000
        C In that case, just dislay a message and continue.
                WRITE (5,910)
        910     FORMAT (' GROUP GLOBAL EVENT FLAGS ALREADY EXIST')
                GOTO 15
        C Here for fatal errors, display message and exit
        1000    WRITE (5,1010) IDSW
        1010    FORMAT (' DIRECTIVE ERROR CREATING GROUP GLOBAL
                1EF''S. DSW = ',I5)
                CALL EXIT
        1100    WRITE (5,1110) IDSW
        1110    FORMAT (' DIRECTIVE ERROR CLEARING EVENT FLAG 65.
                1 DSW = ',I5)
                CALL EXIT
        1200    WRITE (5,1210) IDSW
        1210    FORMAT (' DIRECTIVE ERROR READING EVENT FLAG
                1 65. DSW = ',I5)                                !!EX
                CALL EXIT
                END
```

SOLUTION

```
        WTSE$C  65.             ; Wait for event flag 65
                                ;  to be set
        BCS     ERR3            ; Branch on directive
                                ;  error
        TYPE    <EF 65. HAS BEEN SET. WFLAG WILL NOW EXIT>
        EXIT$S
; AST Service routine                                 ;;EX
REXAST: TYPE    <WHY ME? NOT THIS TIME!!> ; Type message
                                                      ;;EX
        ASTX$S                  ; AST exit to return ;;EX
ERR3:   INC     R0              ; R0 = 3 if error on
                                ; wait for dir
ERR2:   INC     R0              ; R0 = 2 if error on
                                ;  clear flag dir
ERR1:   INC     R0              ; R0 = 1 if error on
                                ;  create group flags dir
ERR0:   MOV     $DSW,R1         ; Place DSW in R1, leave
                                ;  R0=0 for specify ;;EX
                                ;  requested exit AST err
        IOT                     ; Trap and dump registers
        .END    START
```

```
        PROGRAM WFLAG
C
C FILE LEX25.FTN
C
C Modified to include a Requested Exit AST      !!EX
C
C This task creates the group global event flags, and
C then clears event flag 65. and waits for it to be set.
C When the flag is set, it writes a message and exits
C
C Install and run instructions:
C
C       Run WFLAG, then run SFLAG. At least one of the
C       tasks must be installed, or else the RUN command
C       will try to install both tasks under the same
C       name (TTnn)
C
        EXTERNAL REXAST                 !!EX
C Set up Requested Exit AST             !!EX
        CALL SREA (REXAST,IDSW)         !!EX
        IF (IDSW .LT. 0) GOTO 950       !!EX
        WRITE (5,10)
10      FORMAT (' WFLAG IS CREATING THE GROUP GLOBAL EVENT FLAGS')
        CALL CRGF (,IDSW)
        IF (IDSW .LT. 0) GOTO 900
15      WRITE (5,20)
20      FORMAT (' CLEAR AND WAIT FOR EF 65. TO BE SET')
        CALL CLREF (65,IDSW)
        IF (IDSW .LT. 0) GOTO 1100
        CALL WAITFR (65,IDSW)
        IF (IDSW .LT. 0) GOTO 1200
        WRITE (5,30)
```

SOLUTION

6.

```
        .TITLE  SST
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;
; FILE LEX26.MAC
;
; Modified to include an odd address trap      ;;EX
;
; This task sets up an SST vector table to handle SST's
; for BPT, IOT, and odd address traps. It then executes
; instructions to cause these traps to occur. In each
; SST routine, a message is displayed and then the task
; continues. Finally, a TRAP instruction is executed.
; Since no user SST routine is specified for TRAP, the
; Executive aborts the task.
;
; Assemble and task-build instructions:
;
;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]LEX26
;       LINK/MAP LEX26,LB:[1,1]PROGSUBS/LIBRARY
;
        .MCALL  SVTK$C,EXIT$S       ; External system macros
        .MCALL  TYPE                ; External supplied macro
;
VTABLE: .WORD   ODDTRP,MPTVIO,BPT,IOT ; SST vector table
;                                                  ;;EX
START:  SVTK$C  VTABLE,4            ; Have Executive set up
                                    ; SST table
        BPT                         ; BPT instruction
        TST     1                   ; Test location 1, ;;EX
                                    ;  causing an odd  ;;EX
                                    ;  addr trap       ;;EX
        CLR     120000              ; Clear location 120000,
                                    ;  causing a memory
                                    ;  protect violation
        IOT                         ; IOT instruction
        EXIT$S                      ; Exit
NEW:    TRAP                        ; TRAP instruction
;
```

Using the QIO Directive

TEST/EXERCISE

1. Modify SYNCHQ or ASYNCQ to write prompting text (e.g., "TYPE SOME TEXT: ") before issuing the read.

2. In MACRO-11, modify NUMER, replacing the error handling code with code which writes out an error message plus the appropriate status code. Refer to SYNQER for sample error messages.

3. Modify NOECHO to use one QIO directive to both write the prompt and read the input. Also, have the read timeout if no key is struck for 2Ø(1Ø) seconds, in which case, display a timeout message and exit.

4. Write a task which prints a message on every terminal in the system. The task should break through any pending I/O at the terminal. (Note: This task must be task-built as a privileged task, using the /PRIVILEGED:Ø qualifier in the task-build command; /PR:Ø in MCR)

Using the QIO Directive

SOLUTION

1.
```
        .TITLE  SYNCHQ
        .IDENT  /01/
        .ENABL  LC                 ; Enable lower case
;+
; FILE LEX31.MAC
;
; Modified to display prompting text             ;;EX
;
; This task reads a line of text from the terminal,
; converts all upper case characters to lower case, and
; prints the converted message back at the terminal. It
; uses synchronous QIO directives.
;-
        .MCALL  QIOW$C,QIOW$S,EXIT$S ; External system
                                     ;  macros

IOSB:   .BLKW   2                  ; I/O Status Block
BUFF:   .BLKB   80.                ; Text buffer
PRMPT:  .ASCII  /TYPE SOME TEXT: / ; Prompt          ;;EX
LPRMPT  =.-PRMPT                   ; Length of prompt  ;;EX
        .EVEN                                        ;;EX

START:  CLR     R5                 ; Error Count
        CLR     R4                 ; Error indicator - 0
                                   ;  means directive error
                                   ;  (DSW in R3), neg
                                   ;  means I/O error
                                   ;  (I/O status in R3)
        QIOW$C  IO.WVB,5,1,,IOSB,,<PRMPT,LPRMPT,40>
                                   ; Display prompt    ;;EX
        BCS     ERR3               ; Branch on dir error;;EX
        TSTB    IOSB               ; Check for I/O error;;EX
        BLT     ERR3A              ; Branch on I/O error;;EX
        QIOW$C  IO.RVB,5,1,,IOSB,,<BUFF,80.> ; Issue
                                             ;  read
        BCS     ERR1               ; Branch on dir error
        TSTB    IOSB               ; Check for I/O error
        BLT     ERR1A              ; Branch on I/O error
        MOV     IOSB+2,R0          ; Get count of characters
                                   ;  typed in
        CLR     R1                 ; Offset into buffer to
                                   ;  character
LOOP:   CMPB    BUFF(R1),#'A       ; Check for upper case
                                   ;  ASCII character
        BLT     NEXT               ; Branch if below range
        CMPB    BUFF(R1),#'Z
        BGT     NEXT               ; Branch if above range
; Here if upper case, move to register R2 and convert
        MOVB    BUFF(R1),R2        ; Move to register
        ADD     #32.,R2            ; Convert to lower case
        MOVB    R2,BUFF(R1)        ; Replace in message
```

Using the QIO Directive

SOLUTION

```
        PROGRAM ASYNCQ
C
C FILE  LEX31.FTN
C
C Modified to display prompting text          !!EX
C
C This program reads a line of text from the terminal,
C converts any upper case characters to lower case and
C prints the converted message back at the terminal.
C It uses asynchronous QIOs and an event flag for
C synchronization.
C
        BYTE IOSB(4),IBUF(80)
        DIMENSION IPAR(6),K(10)
        EQUIVALENCE (NUM,IOSB(3))
        REAL PRMPT(4)                          !!EX
        DATA PRMPT /'TYPE',' SOM','E TE','XT: '/!!EX
        DATA IOWVB/"11000/
        DATA IORVB/"10400/
        DATA IVFC/"40/
C Set up values for the QIO
        IUNIT=5
C Set up for QIO to issue prompt               !!EX
        CALL GETADR(IPAR(1),PRMPT(1))          !!EX
        IPAR(2)=16                             !!EX
        IPAR(3)="40                            !!EX
C Issue asynchronous write                     !!EX
        CALL QIO(IOWVB,IUNIT,5,,IOSB,IPAR,IDS) !!EX
        IF (IDS .LT. 0) GOTO 780               !!EX
        CALL WAITFR(5,IDS)                     !!EX
        IF (IDS .LT. 0) GOTO 785               !!EX
        IF (IOSB(1) .LT. 0) GOTO 790           !!EX
C Set up for read                              !!EX
        IPAR(3)=0                              !!EX
        IPAR(2)=80
C Get the address of the I/O buffer
        CALL GETADR(IPAR(1),IBUF(1))
C Issue the QIO
        CALL QIO(IORVB,IUNIT,5,,IOSB,IPAR,IDS)
C Check the directive status
        IF (IDS .LT. 0) GO TO 800
C Do some work while I/O operation is being performed
        DO 50 I=1,10
        K(I)=64*I
50      CONTINUE
C       Wait for I/O to complete
        CALL WAITFR(5,IDS)
C       Check directive status
        IF (IDS .LT. 0) GO TO 805
C Check the I/O status
        IF (IOSB(1) .LT. 0) GO TO 810
```

Using the QIO Directive

SOLUTION

2.
```
        .TITLE  NUMER
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; FILE LEX32.MAC
;
; Modified to include error message code        ;;EX
;
; This task does a simple addition and outputs the
; results. It demonstrates the use of $EDMSG for
; formatting messages with numeric data
;-
        .MCALL  QIOW$,EXIT$S,DIR$ ; System macros
        .MCALL  QIOW$S            ; System macros ;;EX
        .NLIST  BEX                     ; Do not list binary
                                        ;  extensions
; Data
A:      .WORD   10                      ; 1st addend and start
                                        ;  of argument block
B:      .WORD   22                      ; 2nd addend
C:      .BLKW   1                       ; Location for sum
;
OUT:    QIOW$   IO.WVB,5,1,,IOSB,,<BUF,,40> ;QIO for
                                        ;  output message
IOSB:   .BLKW   2                       ; I/O status block
;
; Set up for $EDMSG
;
BUF:    .BLKB   80.                     ; Output buffer
FMES:   .ASCIZ  /%D. WAS ADDED TO %D., GIVING %D./
                                        ; Format string
; Set up for error messages using $EDMSG        ;;EX
        .EVEN                                   ;;EX
ARG:    .BLKW   1                       ; Argument block;;EX
FMT1D:  .ASCIZ  /DIRECTIVE ERROR ON WRITE, DSW = %D/ ;;EX
FMT1I:  .ASCIZ  'I/O ERROR ON WRITE, I/O STATUS = %D';;EX
        .EVEN                                   ;;EX

        .LIST   BEX                     ; List binary extensions
        .EVEN                           ; Move to word boundary
START:  MOV     A,C                     ; Move 1st addend to sum
                                        ;  word
        ADD     B,C                     ; Add 2nd addend to form
                                        ;  sum
; Set up for call to $EDMSG
        MOV     #BUF,R0                 ; Addr of output buffer
        MOV     #FMES,R1                ; Addr of format string
        MOV     #A,R2                   ; Addr of argument block
        CALL    $EDMSG                  ; Make call, character
                                        ; count returned in R1
```

Using the QIO Directive

SOLUTION

3.

```
        .TITLE  NOECHO
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; FILE LEX33.MAC
;
; Modified to combine QIOs and include timeout  ;;EX
;
; This task writes a prompt and then issues a QIO to read
; from the terminal without echo.  It then displays the
; word which was entered.
;
; Assemble and task-build instructions:
;
;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[uic]LEX33
;       LINK/MAP LEX33,PROGSUBS/LIBRARY
;-
        .MCALL  EXIT$S,QIOW$C,QIOW$S ; System macros
        .MCALL  DIRERR,IOERR    ; Supplied macros
;
; Data
;
        .NLIST  BEX                     ; Don't list of binary
                                        ;  extensions
MES:    .ASCII  /SECRET WORD: / ; Prompt messege
LEN     =       .-MES                   ; Length of prompt
BUFF:   .ASCII  <15>/NO LONGER A SECRET WORD: /
                                        ; Preceding remark
BLEN    =       .-BUFF                  ; Length of Remark
BUF:    .BLKB   80.                     ; Input buffer
TMOMS:  .ASCII  /READ TIMED OUT/ ; Timeout message ;;EX
LTMOMS  =.-TMOMS                                   ;;EX
        .EVEN                           ; Word align for IOSB
IOSB:   .WORD   0                       ; IOSB is broken into
LENT:   .WORD   0                       ;  two parts for
                                        ;  convenience.
; Define functions locally to allow us of an assignment
; statement to shorten directive statement
IO.RPR  =004400                         ; Define functions
TF.RNE  =20                             ;
TF.TMO  =200                            ;
IO.FNC  =<IO.RPR!TF.RNE!TF.TMO> ; QIO function code
        .LIST BEX                       ; List binary extensions
;
; Code
;
START:  QIOW$C  IO.FNC,5,1,,IOSB,,<BUF,80.,2,MES,LEN,44>
                                        ; Issue read after ;;EX
                                        ; prompt         ;;EX
        BCS     DERR1                   ; Branch on dir error
```

Using the QIO Directive

SOLUTION

```
      PROGRAM NOECHO
C
C File LEX33.FTN
C
C Modified to use read after prompt and to timeout !!EX
C
C This task prompts for input, reads it without echo and
C then skips to the next line and displays the input
C text and exits.
C
      BYTE    BUFF(80),IOSB(4),CR(1)
      INTEGER PARM(6)
      REAL    PROMPT(4)                  ! Prompt    !!EX
C
      DATA    IOFNC   /"4620/            ! QIO       !!EX
C                                        !  function!!EX
C                                        !  code     !!EX
      DATA    ISTMO /2/                  ! Timeout   !!EX
C                                        !  status   !!EX
      DATA    CR /"15/                   ! Carriage return character
      DATA    PROMPT /'SECR','ET W','ORD:','    '/
C                                        ! Text      !!EX
C Set up the I/O parameter list
      CALL GETADR (PARM(1),BUFF(1))      ! buffer address
      PARM(2) = 80                       ! Buffer length
      PARM(3) = 2                        ! Timeout = 2 !!EX
C                                        !  * 10 sec   !!EX
      CALL GETADR (PARM(4),PROMPT(1))    ! Prompt addr !!EX
      PARM(5) = 13                       ! Prompt length!!EX
      PARM(6) = "44                      ! Vertical    !!EX
C                                        !  format contr!!EX
C Issue read no echo, read after prompt, with timeout !!EX
      CALL WTQIO (IOFNC,5,1,,IOSB,PARM,IDS)
      IF (IDS .LT. 0) GO TO 100          ! Dir error?
      IF (IOSB(1) .LT. 0) GO TO 110      ! I/O error?
C Check for timeout
      IF (IOSB(1) .NE. ISTMO) GOTO 1     ! Branch if no!!EX
C                                        !  timeout    !!EX
      TYPE *,'READ TIMED OUT'            ! Display     !!EX
C                                        !  message    !!EX
      CALL EXIT                          !  and exit   !!EX
1     WRITE (5,2) CR,(BUFF(I),I=1,IOSB(3)) ! Echo input
2     FORMAT (' ',A1,'NO LONGER A SECRET WORD: ',80A1)
      CALL EXIT
C
C Error conditions
C
100   TYPE *, 'DIRECTIVE ERROR ON READ. STATUS = ',IDS
      CALL EXIT
110   TYPE *, 'I/O ERROR ON READ. CODE = ',IOSB(1)
      CALL EXIT
      END
```

Using the QIO Directive

SOLUTION

```
        PROGRAM LEX34
C+
C FILE LEX34.FTN
C
C Solution to Module 3, Lab Exercise 4
C
C Task does a write breakthrough to all terminals.
C
C Task-build with /PRIVILEGED:0 qualifier
C-
        INTEGER TTUNIT,DSW
        DATA TTUNIT/0/          ! First output to TT0:
        INTEGER PARAM(6),IOSB(2)
        BYTE SUCCOD(2)          ! I/O success codes
        EQUIVALENCE (SUCCOD,IOSB) ! First bytes of IOSB
        INTEGER IEIDU           ! Mnemonic for "Illegal
        DATA IEIDU/-99/         ! Device or Unit" DSW code
        INTEGER IOFCOD          ! I/O function code
C                               !  mnemonic
        DATA IOFCOD/"501/       ! Write logical block,
C                               !  write breakthrough,
C                               !  and restore cursor
C
C Load parameter list
        CALL GETADR(PARAM(1),'HELLO THERE')
        PARAM(2) = 11           ! Length of string
        PARAM(3) = "40          ! Blank for carr. ctrl.
10      CALL ASNLUN(4,'TT',TTUNIT,DSW) ! Assign LUN 4 to
C                                      !  TTn:
        IF (DSW.LT.0) GOTO 900
        CALL WTQIO(IOFCOD,4,1,,IOSB,PARAM,DSW)
        IF (DSW.LT.0) GOTO 910  ! Directive error
        IF (SUCCOD(1).NE.1) GOTO 920    ! I/O error
        TTUNIT = TTUNIT+1
        GOTO 10
C
C Error from ASNLUN.  If ASNLUN failed because of illegal
C unit number, must have passed the last terminal.  Exit.
900     IF (DSW.EQ.IEIDU) CALL EXIT
        TYPE 905,DSW            ! Other error
905     FORMAT (' ERROR ON ASNLUN. DSW = ',I6)
        CALL EXIT
910     TYPE 915,TTUNIT,DSW
915     FORMAT (' DIRECTIVE ERROR ON QIO TO TT',O2,':'/
        1 ' DSW = ',I6)
        CALL EXIT
920     TYPE 925,TTUNIT,SUCCOD(2),SUCCOD(1),IOSB(2)
925     FORMAT (' I/O ERROR ON QIO TO TT',O2,':'/
        1 ' I/O STATUS BLOCK = ',I4,' ,',I4,' /',I6)
        CALL EXIT
        END
```

Using Directives for Intertask Communication

TEST/EXERCISE

1. Modify RECV1 and SEND1 to synchronize using Suspend and Resume directives instead of event flags.

2. Modify RECV2 so that the display includes the name of the sending task in addition to the data.

3. Write another sender task to send data to RECV2. Modify the receiver so that it receives data from your task only, not from SEND2.

4. Modify SPAWN so that it spawns CLI..., MCR..., or ...DCL several different times and sends a different MCR or DCL command line each time. Display the exit status after each command executes.

5. Write a parent task and an offspring task. Have the parent spawn the offspring. Have the offspring emit status to the parent every five seconds for 3Ø seconds and then exit. Have the parent display each status value. Optional: Use an AST routine in the parent for synchronization.

Using Directives for Intertask Communication

SOLUTION

1.
```
        .TITLE  SEND1
        .IDENT  /01/
        .ENABL  LC              ; Enable lower case
;+
; FILE LEX41A.MAC
;
; Modified to use Suspend and Resume directives for;;EX
; synchronization                                  ;;EX
;
; This task prompts at TI: for a line of text and sends
; the data to RECV1 for processing.  Synchronization is
; handled through a common event flag.
;
; Assemble and task-build instructions:
;
;       >MACRO/LIST/OBJECT:SEND1 LB:[1,1]PROGMACS/LI-;;EX
;       ->BRARY,dev:[ufd]LEX41A
;       >LINK/MAP SEND1,LB:[1,1]PROGSUBS/LIBRARY
;
; Install and run instructions: RECV1 must be installed
; and run prior to running SEND1. RECV1 continues to run
; until it receives 3 data packets.
;-
        .MCALL  SDAT$C,EXIT$S,RSUM$C ; System macros;;EX
        .MCALL  TYPE,INPUT,DIRERR ; Supplied macros
;
;
BUFFER: .BLKB   26.             ; Data buffer to be sent
;
        .ENABL  LSB             ; Enable local symbol
                                ;  blocks
;
START:: TYPE    <TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS>
                                ; Type prompt
        INPUT   #BUFFER,#26.    ; Get text to send
        SDAT$C  RECV1,BUFFER    ; Send data to RECV1 ;;EX
        BCC     1$              ; Branch on directvie ok
        DIRERR  <UNABLE TO QUEUE DATA TO RECV1> ; Display
                                ; error message and exit
1$:     RSUM$C  RECV1           ; Resume RECV1      ;;EX
        BCC     5$              ; Branch on directive ok;;EX
        DIRERR  <UNABLE TO RESUME RECV1> ;          ;;EX
5$:     EXIT$S                  ; Exit              ;;EX
        .END    START
```

Using Directives for Intertask Communication

SOLUTION

```
        .TITLE  RECV1
        .IDENT  /01/
        .ENABL  LC                ; Enable lower case
;+
; FILE LEX41B.MAC
;
; Modified to use Suspend and Resume for synchronization;;EX
;
; This task and receives data from any sender task
; (e.g., SEND1).It prints the data on TI:.  Then it
; waits for another data packet. It does this until it
; has received 3 messages and then exits.
;
; This task synchronizes with its sender through an
; event flag.
;
; Assemble and task-build instructions:
;
;       >MACRO/LIST/OBJECT:RECV1 LB:[1,1]PROGMACS/LIB-;;EX
;       ->RARY,dev:[ufd]RECV1                        ;;EX
;       LINK/MAP RECV1,LB:[1,1]PROGSUBS/LIBRARY
;
; Install and run instructions: RECV1 must be installed
; and run before running SEND1.
;-
        .MCALL  RCVD$C,EXIT$S,SPND$S; System macros ;;EX
        .MCALL  TYPE,DIRERR       ; Supplied macros
;
;
RBUFF:  .BLKW   15.               ; Receive buffer
;
        .ENABL  LSB               ; Enable local symbol
                                  ;  blocks
;
START:  MOV     #3,R5             ; Initialize message
                                  ;  counter
AGAIN:  SPND$S                    ; Suspend self until;;EX
                                  ;  message arrives
        BCC     3$                ; Branch on directive ok
        DIRERR  <SUSPEND DIRECTIVE FAILED> ; Display ;;EX
                                  ; error message and exit
; We get here when resumed by SEND1                  ;;EX
3$:     RCVD$C  ,RBUFF            ; Receive from anyone
        BCC     5$                ; Branch on directive ok
        DIRERR  <RECEIVE DIRECTIVE FAILED IN "RECV1">
                                  ; Display error message
                                  ;  and exit
; Successful receipt
5$:     TYPE    <DATA RECEIVED BY "RECV1":> ; Display
                                            ;  data
```

SOLUTION

2.

```
        .TITLE  RECV2
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;
; FILE LEX42.MAC                                ;;EX
;
; Modified to display the sender task name in addition ;;EX
; to the data                                          ;;EX
;
; This task receives data from another task. It prints
; the data, along with a header, on TI:.  Then it waits
; for another data packet, continuing this until it has
; received 3 messages.
;
; This task synchronizes with its sender using RCST$.
; Because of this synchronization, the tasks can be run
; in any order, with any relative priorities.
;
; Assemble and task build instructions:
;
;       >MACRO/LIST/OBJECT:RECV2 LB:[1,1]PROGMACS/LIB-;;EX
;       ->RARY,dev:[ufd]LEX42A                       ;;EX
;       >LINK/MAP RECV2,LB:[1,1]PROGSUBS/LIBRARY
;
; Install and run instructions: RECV2 must be installed.
;
        .MCALL  RCST$C,RCVD$C,EXIT$S ; System macros
        .MCALL  TYPE,DIRERR             ; Supplied macros
;
RBUFF:  .BLKW   15.                     ; Receive buffer
TASKNM: .BLKW   3                       ; Buffer for task name;;EX
;
        .ENABL  LSB                     ; Enable local symbol
                                        ;  blocks
;
START:  MOV     #3,R5                   ; Set up message counter
RECEIV: RCST$C  ,RBUFF                  ; Receive from anyone
        BCC     5$                      ; Branch on directive ok
        DIRERR  <RECEIVE DIRECTIVE FAILED IN "RECV2">
                                        ; Display error message
                                        ;  and exit
; Successful receipt or unstopped by another task. First
; check for unstopped after being stopped, in which case
; we have to receive the data
5$:     CMP     $DSW,#IS.SET            ; Were we stopped due to
                                        ;  no data
        BNE     6$                      ; If not, we have a data
                                        ;  packet
        RCVD$C  ,RBUFF                  ; Now get the packet
        BCC     6$                      ; Branch on directive ok
```

SOLUTION

```
C
        INTEGER RBUFF(15)       ! Receive buffer
        INTEGER DSW,ISSET
        INTEGER TASKNM(3)       ! Buffer for ASCII form!!EX
C                               !  of task name       !!EX
        DATA ISSET/2/           ! DSW code mnemonic
C
C
        DO 100, I=1,3
        CALL RCST(,RBUFF,DSW)   ! Receive from anyone
        IF (DSW.GE.0) GOTO 50
        Type *,'RECEIVE DIRECTIVE FAILED IN "RECV2".
        1 DSW = ',DSW           ! Display error message
        GOTO    1000            !  and exit
C
C Successful receipt or unstopped by another task. First
C check for unstopped after being stopped, in which case
C we have to receive the data
50      IF (DSW.NE.ISSET) GOTO 60 ! Were we stopped due
C                                 !  to no data? If not
C                                 !  (NE), we have a
C                                 !  data packet
C Stopped due to no data:
        CALL RECEIV(,RBUFF,,DSW)  ! Now get the packet
        IF (DSW.EQ.1) GOTO 60
        TYPE *,'RECEIVE DIRECTIVE FAILED AFTER "RECV2"
        1UNSTOPPED. DSW = ',DSW   ! Display error
        GOTO 1000                 !  message and exit
C Display data
60      CALL R50ASC (6,RBUFF,TASKNM)          !!EX
        TYPE 75,TASKNM,(RBUFF(J),J=3,15)      !!EX
75      FORMAT (' DATA RECEIVED BY "RECV2":'/1X,3 !!EX
        1A2,1X,13A2)                          !!EX
100     CONTINUE
C Have received 3 messages
        TYPE *,'"RECV2" HAS RECEIVED 3 MESSAGES AND WILL
        1 NOW EXIT'
1000    CALL EXIT               ! Exit
        END
```

Using Directives for Intertask Communication

SOLUTION

```
        PROGRAM LEX43A
C
C FILE LEX43A.FTN                      !!EX
C
C A second sender task to send data to RECV2    !!EX
C
C This task prompts at TI: for a line of text and sends
C the data to RECV2 for processing.  The receiver will
C continue to run until it receives 3 messages.
C Synchronization is handled through RECV2's stop bit.
C RECV2 and LEX43A may be run in any order.
C
C Install and run instructions: LEX43B must be  !!EX
C installed under the name RECV2.               !!EX
C
        BYTE BUFFER(26)         ! Send buffer
        INTEGER DSW
        REAL RECV2
        DATA RECV2/5RRECV2/     ! Receiving task name
        INTEGER IEITS,IEACT     ! Error mnemonics
        DATA IEITS,IEACT/-8,-7/
C
        TYPE *,'TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS'
        READ (5,5) BUFFER
5       FORMAT (26A1)
        CALL SEND(RECV2,BUFFER,,DSW) ! Send data to RECV2
        IF (DSW.EQ.1) GOTO 10
        TYPE *,'UNABLE TO QUEUE DATA TO "RECV2". DSW = '
        1,DSW
10      CALL USTP(RECV2,DSW)    ! Unstop RECV2
        IF (DSW.EQ.1) GOTO 20   ! Branch on directive ok
        IF (DSW.EQ.IEITS) GOTO 20 ! Isn't he stopped?
C                               !  That's ok, he'll pick
C                               !  up data when he
C                               !  executes RCDS$
        IF (DSW.EQ.IEACT) GOTO 20 ! Is he not active? If
C,                              !  not, he'll pick up
C                               !  data when activated
        TYPE *,'UNABLE TO UNSTOP "RECV2". DSW = ',DSW
                                ! Any other error is bad
20      CALL EXIT               ! Exit
        END
```

SOLUTION

```
; Successful receipt or unstopped by another task. First
; check for unstopped after being stopped, in which case
; we have to receive the data
5$:     CMP     $DSW,#IS.SET      ; Were we stopped due to
                                  ;  no data
        BNE     6$                ; If not, we have a data
                                  ;   packet
        RCVD$C  LEX43A,RBUFF      ; Now get the packet
        BCC     6$                ; Branch on directive ok
        DIRERR  <RECEIVE DIR FAILED AFTER "RECV2" UNSTOPPED>
                                  ; Display error message
                                  ;  and exit
6$:     TYPE    <DATA RECEIVED BY "RECV2":> ; Display
                                            ;  text and
        TYPE    #RBUFF+4,#26.               ;  data sent
;       SOB     R5,RECEIV         ; Decrement message
                                  ;  counter. Receive again
                                  ;  if haven't received 3
                                  ;  yet
        DEC     R5                                  ;;EX
        BEQ     DONE                                ;;EX
        JMP     RECEIV                              ;;EX
DONE:   TYPE    <"RECV2" HAS RECEIVED 3 MESSAGES AND WILL NOW EXIT>
                                                    ; Type exit
                                                    ;  message
        EXIT$S                    ; Exit
        .END    START
```

```
        PROGRAM RECV2
C
C FILE LEX43B.FTN                 !!EX
C
C Modified to receive only from LEX43A          !!EX
C NOTE: TASK WILL EXIT WITH A NO DATA QUEUED ERROR IF!!EX
C SEND2 SENDS DATA. MORE COMPLICATED CODE IS NEEDED  !!EX
C TO CHECK FOR SEND2 SENDING DATA AND UNSTOPPING RECV2!!EX
C
C This task receives data from another task (e.g. SEND2).
C It prints the data, along with a header, on TI:.  Then
C it waits for another data packet, continuing this
C until it has received 3 messages.
C
C This task synchronizes with its sender using RCST.
C Because of this synchronization, the tasks can be run
C in any order, with any relative priorities.
C
C Install and run instructions: LEX43B must be        !!EX
C installed under the name RECV2.                     !!EX
C
```

SOLUTION

4.

```
        .TITLE  SPAWN
        .IDENT  /02/
        .ENABL  LC              ; Enable lower case
;
; File LEX44.MAC                                        ;;EX
;
; This program spawns MCR..., passes it a series of     ;;EX
; command lines, waits for each to exit, and            ;;EX
; displays each command's exit status.                  ;;EX
;
; Assemble and task-build instructions:
;
;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]LEX44
;       LINK/MAP LEX44,LB:[1,1]PROGSUBS/LIBRARY
;
        .MCALL  SPWN$,EXIT$S,WTSE$C,QIOW$S,QIOW$C
                                ; System macros
        .MCALL  DIRERR,IOERR    ; Supplied macros
        .NLIST  BEX             ; Inhibit listing of
                                ;  binary extensions

CMD1:   .ASCII "PIP *.MAC/LI"   ; Command line          ;;EX
LEN1    =.-CMD1                 ; Length of command     ;;EX
CMD2:   .ASCII  /ACT/                                   ;;EX
LEN2 = .-CMD2                                           ;;EX
CMD3:   .ASCII  /TIM/                                   ;;EX
LEN3 = .-CMD3                                           ;;EX

SMES:   .ASCII  /SPAWN IS STARTING AND WILL SPAWN/      ;;EX
        .ASCII  / MCR COMMANDS/ ; Startup message       ;;EX
LSMES   =.-SMES                 ; Length of message
        .EVEN
IOSB:   .BLKW   2               ; I/O status block
EXSTAT: .BLKW   8.              ; Exit status block

CMDTBL: .WORD   CMD1,LEN1       ; Table indexing        ;;EX
        .WORD   CMD2,LEN2       ; MCR commands          ;;EX
        .WORD   CMD3,LEN3                               ;;EX
        .WORD   0               ; End of table          ;;EX

SPAWN:  SPWN$   MCR...,,,,,,1,,EXSTAT                   ;;EX

BUFF:   .BLKB   80.             ; Output message buffer
; Format string:
FMT:    .ASCII  /%NSPAWN REPORTING: COMMAND/            ;;EX
        .ASCIZ  / COMPLETED. EXIT STATUS WAS %D.%N/     ;;EX
        .EVEN
START:  QIOW$C  IO.WVB,5,1,,IOSB,,<SMES,LSMES,40>
        BCS     ERR1D           ; Branch on dir error
        TSTB    IOSB            ; Check for I/O error
        BLT     ERR1I           ; Branch on I/O error
```

Using Directives for Intertask Communication

SOLUTION

```
          PROGRAM SPWN
C
C File LEX44.FTN
C
C This program spawns ...DCL, passes it a series of !!EX
C command lines, waits for each to exit, and        !!EX
C displays each command's exit status.              !!EX
C
C Data
          INTEGER EXSTAT(8),PLIST(6),DSW
          BYTE BUFF(80)
C Commands to be spawned:                           !!EX
C
C         DIR *.MAC                                 !!EX
C         SHOW TASKS/ACTIVE                         !!EX
C         SHOW TIME                                 !!EX
C
          REAL CMD(5,3)                             !!EX
          DATA CMD/'DIR ','*.MA','C'    , 0    , 0 ,
          1        'SHOW',' TAS','KS/A','CTIV','E',
          2        'SHOW',' TIM','E'    , 0    , 0/     !!EX
          INTEGER LEN(3)
          DATA LEN/9,17,9/
C
          REAL DCL
          DATA DCL/6R...DCL/
C
C Code
          WRITE (5,15)              ! Write message
15        FORMAT (' SPAWN IS STARTING AND WILL SPAWN ',
          1 'DCL COMMANDS')                         !!EX
          DO 30,I=1,3
          CALL SPAWN(DCL,,,1,,EXSTAT,,CMD(1,I),LEN(I)
          1 ,,,DSW)                                 !!EX
                                    ! Spawn DCL
          IF (DSW.LT.0) GOTO 900    ! Branch on dir error
          CALL WAITFR(1,DSW)        ! Wait for task to exit
          IF (DSW.LT.0) GOTO 910    ! Branch on dir error
          WRITE (5,25) EXSTAT(1).AND."377 ! Display low
                                    ! byte of exit status
25        FORMAT (' SPAWN REPORTING: COMMAND COMPLETED.',
          1 ' EXIT STATUS WAS ',I1,'.')
30        CONTINUE
          CALL EXIT                 ! Exit
C Error handling code
900       TYPE *,'ERROR SPAWNING DCL. DSW = ',DSW
          GOTO 1000
910       TYPE *,'ERROR WAITING FOR EVENT FLAG. DSW = ',DSW
1000      CALL EXIT
          END
```

SOLUTION

```
        DIR$    R4              ; QIOW$ to TI:
        BCS     ERR4
        TST     R5              ; Did offspring exit?
        BGE     3$              ; Yes
        DIR$    #CLEF           ; No. Clear EF 1 again
        BCS     ERR5
        BR      1$              ; Wait
3$:     EXIT$S                  ; Once offspring exits,
                                ;  so should parent
;
ERR1:   DIRERR  <ERROR ON INITIAL CLEF$>
ERR2:   DIRERR  <ERROR SPAWNING LEX45B>
ERR3:   DIRERR  <ERROR ON WTSE$C>
ERR4:   DIRERR  <ERROR ON QIOW$>
ERR5:   DIRERR  <ERROR ON CLEF$>
;
; AST routine, entered when offspring emits status
; (negative status value) or exits (positive status
; value)
;
ASTRTN: SETF$C  1               ; Awaken main code
        BCS     ERR6
        CMP     $DSW,#IS.SET    ; If set, main code is
                                ;  not ready yet
        BEQ     OVRRUN          ; We've been overrun
        TST     STATUS          ; Has offspring exited?
        BGE     4$              ; If so, don't try to
                                ;  reconnect
        CNCT$C  LEX45B,,ASTRTN,STATUS
        BCS     ERR7
4$:     TST     (SP)+           ; Clean up stack from AST
        ASTX$S                  ; Let main code run
;
; If a new status comes in before we're done with the old
; one, something is wrong.  Stop everything.
;
OVRNMS: .ASCII  /STATUS RECEIVED BEFORE READY. /
        .ASCII  /  ABORTING BOTH TASKS./
OVRNML = .-OVRNMS
        .EVEN
;
OVRRUN: QIOW$C  IO.WVB,5,3,,,,<OVRNMS,OVRNML,40>
        ABRT$C  LEX45B          ; Abort offspring
        BCS     ERR8
        EXIT$S                  ; Exit this task
;
ERR6:   DIRERR  <ERROR FROM SETF$ IN AST ROUTINE>
ERR7:   DIRERR  <ERROR CONNECTING TO OFFSPRING>
ERR8:   DIRERR  <ERROR ABORTING OFFSPRING>
        .END    START
```

SOLUTION

```
        .TITLE  LEX45B
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;+
; File LEX45B.MAC
;
; Solution to Module 4, Lab Exercise 5 - Part B,
; offspring task
;
; This task is spawned by LEX45A.  It emits a negative
; status every 5 seconds, then exits after 30 seconds
; (6 emits, then an exit).
;
; If an emit status fails because this task was not
; connected to the parent, another emit status will be
; tried 5 seconds later.  Two consecutive failures cause
; this task to exit with an error message.
;
; This task must be installed under task name LEX45B.
;-
        .MCALL  EMST$S,QIOW$C,WTSE$C,MRKT$C,EXIT$S
        .MCALL  DIRERR
;
NCNCT:  .ASCII  /LEX45B NOT CONNECTED TO ANY PARENT/
        .BYTE   15,12
        .ASCII  /WILL TRY AGAIN IN 5 SECONDS/
NCNCTL = .-NCNCT
        .EVEN
;
START:  CLR     R0                  ; R0 = exit status
        CLR     R1                  ; R1 =  0 means last
                                    ;  attempt to emit status
                                    ;  suceeded. R0 < 0 means
                                    ;  it failed because we
                                    ;  were not connected
        MOV     #6,R3               ; R3 = number of emits
                                    ;  yet to be issued
EMST:   DEC     R3                  ; Set timer (again)?
        BMI     EXIT                ; No, Just exit
        MRKT$C  1,5,2               ; Set timer for 5 seconds
        BCS     ERR1
        DEC     R0                  ; Use status < 0 when
                                    ;  emitting
        EMST$S  ,R0                 ; Emit to parent
        BCS     1$                  ; Failed.  Why?
        CLR     R1                  ; Note success
        BR      WAIT                ; Wait for 5 secs to pass
1$:     CMP     $DSW,#IE.ITS        ; Failed because not
                                    ;  connected?
        BNE     ERR2                ; Any other reason, quit
```

SOLUTION

```
C
        INTEGER DSW,IEITS
        DATA IEITS/-8/              ! Error mnemonic
        LOGICAL*1 ERLAST            ! Flag if last EMST
C                                   !  failed because we were
C                                   !  not connected
        DATA ERLAST/.FALSE./
C
        DO 50,I=1,6                 ! Issue 6 EMSTs
        CALL MARK (1,5,2,DSW)       ! Set timer for 5 seconds
        IF (DSW.LT.0) GOTO 900
        CALL EMST(,(-I),DSW)        ! Emit to parent
        IF (DSW.LT.0) GOTO 20       ! Failed.  Why?
        ERLAST = .FALSE.            ! Note success
        GOTO 30                     ! Wait for 5 secs to pass
20      IF (DSW.NE.IEITS) GOTO 910 ! Failed for reason
C                                   !  other than not
C                                   !  connected
        IF (ERLAST) GOTO 910        ! Failed last time too?
C                                   !  Then give up.
        ERLAST = .TRUE.             ! Else note we failed
                                    !  this time
C                                   ! And announce the
C                                   !  problem:
        TYPE 25
25      FORMAT ('LEX45B NOT CONNECTED TO ANY PARENT'/
        1 'WILL TRY AGAIN IN 5 SECONDS')
C                                   ! And try again in 5 secs
30      CALL WAITFR(1,DSW)          ! Wait for 5 secs to pass
        IF (DSW.LT.0) GOTO 920
50      CONTINUE
        CALL EXIT                   ! Exit (with success)
C
C Directive errors
C
900     TYPE *,'ERROR ON MRKT. DSW = ',DSW
        GOTO 1000
910     TYPE *,'ERROR EMITTING TO PARENT. DSW = ',DSW
        GOTO 1000
920     TYPE *,'ERROR ON WAITFR. DSW = ',DSW
1000    CALL EXIT
        END
```

Memory Management Concepts

TEST/EXERCISE

1. Write 'M' if the statement applies to mapped systems, 'U' if it applies to unmapped systems, or 'M,U' if it applies to both.

   ___ a. Physical addresses up to 32K words accessible with 16-bit addressing.

   ___ b. Physical addresses up to 128K words accessible with 18-bit addressing.

   ___ c. Program relocation possible without having to program or task-build again.

   ___ d. Detection of memory protection violations.

   ___ e. Program executes only at physical addresses that match the virtual addresses created by the task builder.

   ___ f. Virtual address limit of 32K words.

2. Fill in the headings and the missing values in Figure 1.

Memory Management Concepts

SOLUTION

1. Write 'M' if the statement applies to mapped systems, 'U' if it applies to unmapped systems, or 'M,U' if it applies to both.

   U a. Physical addresses up to 32K words accessible with 16-bit addressing. (M is also acceptable since 32K words is the limit of 16-bit addressing even on a mapped system.)

   M b. Physical addresses up to 128K words accessible with 18-bit addressing.

   M c. Program relocation possible without having to program or task-build again.

   M d. Detection of memory protection violations.

   U e. Program executes only at physical addresses that match the virtual addresses created by the task builder.

   M,U f. Virtual address limit of 32K words.

2. Fill in the headings and the missing values in Figure 1.

Overlaying Techniques

TEST/EXERCISE

The following is an output display from a task.

```
MAIN CALLING SUBROUTINE G
G CALLING SUBROUTINE G1
G1 RUNNING
MAIN CALLING SUBROUTINE H1
H1 RUNNING
MAIN CALLING SUBROUTINE H
H CALLING SUBROUTINE H1
H1 RUNNING
H CALLING SUBROUTINE H2
H2 RUNNING
MAIN EXITING
```

The calling sequence parallels the output display.

1. Draw an overlay tree diagram or a memory allocation diagram for a possible overlay structure for the task.

2. Write the modules MAIN, G, G1, H, H1, and H2. Assemble or compile each one.

3. Task-build and run the task without overlays. Obtain a map.

4. Task-build and run the task with all disk-resident overlays. Obtain a map.

5. Task-build and run the task with all memory-resident overlays. Obtain a map.

Overlaying Techniques

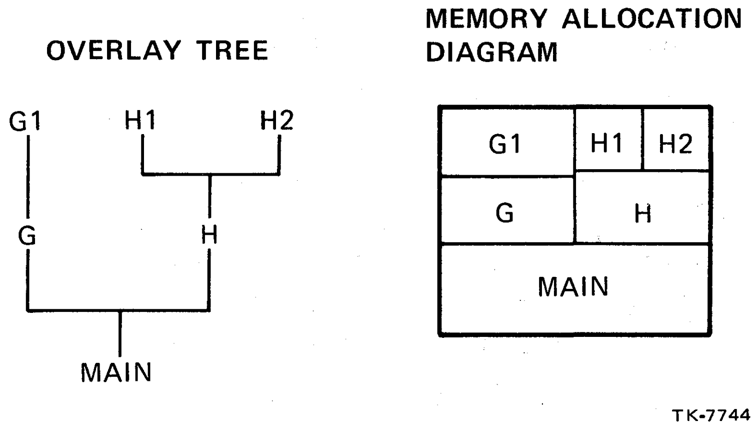
SOLUTION

The following is an output display from a task.

```
MAIN CALLING SUBROUTINE G
G CALLING SUBROUTINE G1
G1 RUNNING
MAIN CALLING SUBROUTINE H1
H1 RUNNING
MAIN CALLING SUBROUTINE H
H CALLING SUBROUTINE H1
H1 RUNNING
H CALLING SUBROUTINE H2
H2 RUNNING
MAIN EXITING
```

The calling sequence parallels the output display.

1. Draw an overlay tree diagram or a memory allocation diagram for a possible overlay structure for the task.



Overlaying Techniques

SOLUTION

```
      PROGRAM MAIN
C
C File LEX6A.FTN
C
C Mainline routine for Module 6, Lab Exercises 1-6.
C Illustrate different overlays and their effects.
C
C For each routine, type message then call routine
C
      TYPE *,'MAIN CALLING SUBROUTINE G'
      CALL    G
      TYPE *,'MAIN CALLING SUBROUTINE H1'
      CALL    H1
      TYPE *,'MAIN CALLING SUBROUTINE H'
      CALL    H
      TYPE *,'MAIN EXITING'
      CALL EXIT
      END
```

```
        .TITLE  G
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;
; File LEX6B.MAC
;
; Subroutine for Module 6, Lab Exercises 1-6.
; Illustrate different overlays and their effects.
;
        .GLOBL  G1                  ; Subroutine called
        .GLOBL  IOFAIL              ; Error routine
        .MCALL  QIOW$C
;
; Messages
;
CG1MS:  .ASCII  /G CALLING SUBROUTINE G1/
CG1ML = .-CG1MS
        .EVEN
;
; Type message then call routine
;
G::     QIOW$C  IO.WVB,5,1,,,,<CG1MS,CG1ML,40>
        BCS     ERROR
        CALL    G1
        RETURN
ERROR:  JMP     IOFAIL
        .END
```

Overlaying Techniques

SOLUTION

```
          SUBROUTINE G1
C
C File LEX6C.FTN
C
C Subroutine for Module 6, Lab Exercises 1-6.
C Illustrate different overlays and their effects.
C
C Type message then return
C
          TYPE *,'G1 RUNNING'
          RETURN
          END
```

```
          .TITLE  H
          .IDENT  /01/
          .ENABL  LC                ; Enable lower case
;
; File LEX6D.MAC
;
; Subroutine for Module 6, Lab Exercises 1-6.
; Illustrate different overlays and their effects.
;
          .GLOBL  H1,H2             ; Subroutines called
          .GLOBL  IOFAIL            ; Error routine
          .MCALL  QIOW$C
;
; Messages
;
CH1MS:    .ASCII  /H CALLING SUBROUTINE H1/
CH1ML =   .-CH1MS
CH2MS:    .ASCII  /H CALLING SUBROUTINE H2/
CH2ML =   .-CH2MS
          .EVEN
;
; Type message then call routine
;
H::       QIOW$C  IO.WVB,5,1,,,,<CH1MS,CH1ML,40>
          BCS     ERROR
          CALL    H1
          QIOW$C  IO.WVB,5,1,,,,<CH2MS,CH2ML,40>
          BCS     ERROR
          CALL    H2
          RETURN
ERROR:    JMP     IOFAIL
          .END
```

Overlaying Techniques

SOLUTION

```
        SUBROUTINE H1
C
C File LEX6E.FTN
C
C Subroutine for Module 6, Lab Exercises 1-6.
C Illustrate different overlays and their effects.
C
C Type message then return
C
        TYPE *,'H1 RUNNING'
        RETURN
        END
```

```
        .TITLE  H2
        .IDENT  /01/
        .ENABL  LC                ; Enable lower case
;
; File LEX6F.MAC
;
; Subroutine for Module 6, Lab Exercises 1-6.
; Illustrate different overlays and their effects.
;
        .GLOBL  IOFAIL            ; Error routine
        .MCALL  QIOW$C
;
; Messages
;
H2RUN:  .ASCII  /H2 RUNNING/
H2RUNL = .-H2RUN
        .EVEN
;
; Type message then return
;
H2::    QIOW$C  IO.WVB,5,1,,,,<H2RUN,H2RUNL,40>
        BCS     ERROR
        RETURN
ERROR:  JMP     IOFAIL
        .END
```

```
        SUBROUTINE H2
C
C File LEX6F.FTN
C
C Subroutine for Module 6, Lab Exercises 1-6.
C Illustrate different overlays and their effects.
C
C Type message then return
C
        TYPE *,'H2 RUNNING'
        RETURN
        END
```

Overlaying Techniques

SOLUTION

4.
```
; Module 6, Lab Exercise 4
;
; .ODL file for building MACRO-11 with all disk resident
; overlays
        .ROOT   LEX6A-PROGSUBS/LB-*(LEX6B-LEX6C,OVRH)
OVRH:   .FCTR   LEX6D-(LEX6E,LEX6F)
;
; LEX6A = MAIN
; LEX6B = G
; LEX6C = G1
; LEX6D = H
; LEX6E = H1
; LEX6F = H2
;
        .END
```

```
; Module 6, Lab Exercise 4
;
; .ODL file for building FORTRAN with all disk-resident
; overlays
        .ROOT   LEX6A-FLIB-*(LEX6B-LEX6C-FLIB,HSEGS)
HSEGS:  .FCTR   LEX6D-FLIB-(LEX6E-FLIB,LEX6F-FLIB)
FLIB:   .FCTR   LB:[1,1]F4POTS/LB
;
; LEX6A = MAIN
; LEX6B = G
; LEX6C = G1
; LEX6D = H
; LEX6E = H1
; LEX6F = H2
        .END
```

5.
```
; Module 6, Lab Exercise 5
;
; .ODL file for MACRO-11 with all memory-resident
; overlays
;
        .ROOT   LEX6A-PROGSUBS/LB-*!(LEX6B-LEX6C,OVRH)
OVRH:   .FCTR   LEX6D-!(LEX6E,LEX6F)
;
; LEX6A = MAIN
; LEX6B = G
; LEX6C = G1
; LEX6D = H
; LEX6E = H1
; LEX6F = H2
;
        .END
```

Overlaying Techniques

SOLUTION

7. Use the map to fill in the following table:

Type of Overlay	Starting Virtual Address of G	Starting Virtual Address of H1
No Overlays		
All Disk-Resident Overlays	Answers will vary depending on students' particular solution.	
All Memory-Resident Overlays		
Disk-Resident and Memory-Resident Overlays		

8.
```
; Module 6, Lab Exercise 8
;
; .ODL file in MACRO-11 to place TOTAL in an overlay
; segment.
; All overlays are disk-resident
        .ROOT   MAIN-*(A-(JOB1,JOBXX),B,TOTAL)
        .END
```

```
; Module 6, Lab Exercise 8
;
; .ODL file in FORTRAN to place TOTAL in an overlay
; segment.
; All overlays are disk-resident
        .ROOT   MAIN-FLIB-*(OVRA,B-FLIB,TOTAL-FLIB)
OVRA:   .FCTR   A-FLIB-(JOB1-FLIB,JOBXX-FLIB)
FLIB:   .FCTR   LB:[1,1]F4POTS/LB
        .END
```

SOLUTION

```
START:  QIOW$C  IO.WVB,5,1,,,,<MES1,LMES1,40> ;Write MES1
        CALL    A                   ; Call subroutine A
        CALL    RTOTAL              ; Call routine to    ;;EX
                                    ;  display running   ;;EX
                                    ;  total             ;;EX
        QIOW$C  IO.WVB,5,1,,,,<MES2,LMES2,40> ;Write MES2
        CALL    B                   ; Call subroutine B
; Set up for loop
        MOV     #3,R4               ; Counter
LOOP:   QIOW$C  IO.WVB,5,1,,,,<MES3,LMES3,40> ; Write MES3
        CLR     ANS                 ; Clear answer in case
                                    ;  of no operation
        CALL    A                   ; Call subroutine A
        CALL    RTOTAL              ; Call routine to    ;;EX
                                    ;  display running   ;;EX
                                    ;  total             ;;EX
        SOB     R4,LOOP             ; Decrement counter and
                                    ;  loop back until done
        QIOW$C  IO.WVB,5,1,,,,<MES4,LMES4,40> ; Write MES4
        CALL    TOTAL               ; Call routine to
                                    ;  display grand total
        QIOW$C  IO.WVB,5,1,,,,<MES5,LMES5,40> ; Write MES5
        EXIT$S                      ; Exit
        .END START
```

```
        PROGRAM MAIN
C
C FILE LEX69A.FTN                  !!EX
C
C Modified to call RTOTAL to display the running    !!EX
C after each call to A                              !!EX
C
C This program prints a message and then calls subroutine
C A. Subroutine A asks whether to perform Job 1 or Job 2.P
C It then calls either subroutine JOB1 or JOB2 which
C performs the operation and displays the results. MAIN
C then calls subroutine B which displays a message. MAIN
C then calls subroutine A 3 more times, keeping a grand
C total of the operations. Finally, it displays the
C grand total and exits.
C
C Task-build instructions: Use LEX69A.ODL as the input!!EX
C file for RTOTAL in the root. Use LEX69B.ODL as the  !!EX
C input file for RTOTAL in the best overlay segment   !!EX
C
```

SOLUTION

```
        .TITLE  RTOTAL
        .IDENT  /01/
        .ENABL  LC                ; Enable lower case
;
; FILE LEX69B.MAC
;
; Subroutine to print the running total
;
        .MCALL  QIOW$S            ; External system macros
        .NLIST  BEX               ; Do not list binary
                                  ;  extensions
RTOFMT: .ASCIZ  /THE TOTAL SO FAR IS %D./ ;Format string
RTOTBF: .BLKB   100.              ; Output buffer
        .EVEN
        .NLIST  BEX               ; List binary extensions

RTOTAL::MOV     #RTOTBF,R0        ; Set up for $EDMSG
        MOV     #RTOFMT,R1        ;
        MOV     #TOT,R2           ;
        CALL    $EDMSG            ; Edit message
        QIOW$S  #IO.WVB,#5,#1,,,,<#RTOTBF,R1,#40>
                                  ; Print it
        RETURN
        .END
```

```
        SUBROUTINE RTOTAL
C
C FILE LEX69B.FTN
C
C Subroutine to print the running total
C
        COMMON /TOTCOM/TOT
        INTEGER TOT
        TYPE 5,TOT
5       FORMAT(' THE TOTAL SO FAR IS', I4,'.')
        RETURN
        END
```

```
; Module 6, Lab Exercise 9
;
; .ODL file in MACRO-11, placing RTOTAL in the root
; segment for testing
; All overlays are memory-resident
        .ROOT   LEX69A-LEX69B-*!(A-!(JOB1,JOBXX),B,TOTAL)
; LEX69A = MAIN modified to call RTOTAL
; LEX69B = RTOTAL
        .END
```

Static Regions

TEST/EXERCISE

1. Create an initialized resident common (size: 32(1Ø) blocks = 1Ø24(1Ø) words, contents: 25(1Ø) in each word). Check with your course administrator to find out where to place the common type partition. Write two tasks, one that modifies all values in the common, and one that reads the values and displays them.

2. Create a resident library using the supplied FORTRAN callable subroutines AADD, SUBB, MULL and DIVV (all in LIB.MAC). Write a task that calls one or more of the routines. For example, write a task that asks for four numbers (A, B, C, and D) and then computes and displays (A * B) + (C * D) = answer.

Static Regions

SOLUTION

1.
```
        .TITLE LEX71A
        .IDENT  /01/
        .ENABL  LC              ; Enable lower case
;+
; File LEX71A.MAC
;
; Program which creates and initializes a common region
; which will be referenced using overlaid Psects.
;
; Size 1024. words, contents all 25's
;
; 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   1024.           ; Repeat count
        .WORD   25.             ; Word of 25(10)
        .ENDR                   ; End repeat range
        .END
```

```
        BLOCK DATA LEX71A
C
C File LEX71A.FTN
C
C Program to create and initialize a resident common
C
C Size is 1024 words, initialized with all 25's
C
C Task-build instructions: Must include /SHAREABLE:COMMON
C and /NOHEADER switches; STACK=0 and PAR=COMWP options.
C Must create .STB file. May be /CODE:PIC or absolute
C (the default). OTS library NOT required.
C
        COMMON /MYDATA/ I(1024)
        DATA I /1024*25/
        END
```

Static Regions

SOLUTION

```
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
```

```
        PROGRAM LEX71B
C
C File LEX71B.FTN
C
C Task to decrement each word in the static common
C region LEX71A. It uses a COMMON to reference
C the data.
C
C Task-build instructions:
C
C       LINK/MAP/OPTION LEX71B,LB:[1,1]FOROTS/LIBRARY
C       Option? RESCOM=LEX71A/RW
C       Option? <RET>
C
        COMMON /MYDATA/ L(1024)! Common to reference
C                              !        shared region
C Decrement values
        DO 5 K=1,1024
        L(K)=L(K)-1
5       CONTINUE
        WRITE (5,10)           ! Display done message
10      FORMAT (' LEX71B HAS MODIFIED THE VALUES IN THE
        1 COMMON LEX71A')
        CALL EXIT
        END
```

Static Regions

SOLUTION

```
; 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
```

```
        PROGRAM LEX71C
C
C File LEX71C.FTN
C
C Task to read data from the static common region LEX71A
C and print it out at TI:. It uses a COMMON to reference
C the data.
C
C Task-build instructions:
C
C       LINK/MAP/OPTION LEX71C,LB:[1,1]FOROTS/LIBRARY
C       Option? RESCOM=LEX71A/RO
C       Option? <RET>
C
        COMMON /MYDATA/ L(1024)! Common to reference
C                              !        shared region
C Loop through to display region, 8 numbers on a line
        DO 50 J = 1,1024,8
        WRITE (5,10) (L(K),K=J,J+7) ! Write values
10      FORMAT (' ',I2,7I8)
50      CONTINUE
        CALL EXIT
        END
```

Static Regions

SOLUTION

```
ADDARG: .WORD   3                   ; For ADD
        .WORD   MURES1              ; First MUL result
        .WORD   MULRES              ; Second result
        .WORD   GRTOT               ; Grand total

; ASCII buffer table.  Initially each entry in this table
; consists of the address of a prompt string followed by
; the address of the buffer to store the input.  After a
; string is input, however, the prompt string address is
; replaced by the length of the input string.  This
; table, with the addition of the final value GRTOT, then
; serves as the $EDMSG argument block.
EDMARG:
ABTBL:  .WORD   APRMT,ASCA
        .WORD   BPRMT,ASCB
CDTBL:  .WORD   CPRMT,ASCC
        .WORD   DPRMT,ASCD
GRTOT:  .WORD                       ; Grand total (numeric
                                    ;  value is inserted
                                    ;  directly into $EDMSG
                                    ;  block)
;
; Other numeric values
M1:     .WORD                       ; First MUL argument
M2:     .WORD                       ; Second MUL argument
MURES1: .WORD                       ; First MUL result
MULRES: .WORD                       ; MUL result

RDPRMT: QIOW$   IO.RPR,5,1,,IOSB,,<,7,,,PLEN,'$>
IOSB:   .BLKW   2

;
; Code
;
START:  QIOW$C  IO.WVB,5,1,,,,<HDRMS,HDRML,40> ; Identify
        MOV     #M1,R5              ; R5 => location to store
                                    ;  binary input values
        MOV     #RDPRMT,R4          ; R4 => "read with
                                    ;  prompt" DPB
        MOV     #ABTBL,R3           ; R3 => ASCII buffer table
        CALL    GETINP              ; Get A
        CALL    GETINP              ; Get B
        MOV     #MULARG,R5          ; R5 => MUL arg block
        CALL    MULL                ; Do first multiply
        MOV     MULRES,MURES1       ; Save result
        MOV     #M1,R5              ; Reset registers
        MOV     #RDPRMT,R4          ;  (FORTRAN calling
        MOV     #CDTBL,R3           ;  convention does not
                                    ;  guarantee they are
                                    ;  preserved.)
```

Static Regions

SOLUTION

```
        PROGRAM LEX72
C+
C File LEX72.FTN
C
C Solution to Module 7, Lab Exercise 2
C
C Task computes sum of products using resident library
C routines.
C
C Task build instructions:
C
C       LINK/MAP/OPTIONS LEX72,LB:[1,1]F4POTS/LIB
C       Option? RESLIB=LIB/RO
C-
        INTEGER A,B,C,D,MURES1,MURES2,GRTOT
C ASCII bytes to make prompting code cleaner
        BYTE ASCA,ASCB,ASCC,ASCD
        DATA ASCA,ASCB,ASCC,ASCD/'A','B','C','D'/
C
        TYPE 5
5       FORMAT (' TASK WILL COMPUTE (A*B)+(C*D)'/
        1 ' ENTER NUMBERS IN DECIMAL.')
C FORMAT statements used repeatedly below:
15      FORMAT ('$ENTER ',A1,': ')
25      FORMAT (I6)
        TYPE 15,ASCA            ! Prompt for
        ACCEPT 25,A             !  and input A
        TYPE 15,ASCB            ! Prompt for
        ACCEPT 25,B             !  and input B
        CALL MULL(A,B,MURES1)   ! MURES1 = A*B
        TYPE 15,ASCC            ! Prompt for
        ACCEPT 25,C             !  and input C
        TYPE 15,ASCD            ! Prompt for
        ACCEPT 25,D             !  and input D
        CALL MULL(C,D,MURES2)   ! MURES2 = C*D
        CALL AADD(MURES1,MURES2,GRTOT)  ! GRTOT = sum
        TYPE 35, A,B,C,D,GRTOT
35      FORMAT (' (',I6,' * ',I6,') + (',I6,' * ',I6,') = ',I6)
        CALL EXIT
        END
```

Dynamic Regions

TEST/EXERCISE

1. Referring to Exercise 1 of Module 7 (Static Regions), modify the tasks that reference the common so that they both map to the common dynamically using the memory management directives.

2. Write a task that creates a dynamic region two blocks long, fills it with a character typed in at the terminal, and leaves it in existence on exit. Write a second task that modifies one value in the region, then displays all the values in the region at the terminal, and finally deletes the region.

3. Modify SNDREF so that it sends the region by reference to a second receiver task, in addition to RCVREF. Write the second receiver task, which should modify values in the region and then display the values in the region at the terminal.

Dynamic Regions

SOLUTION

1.

```
        .TITLE  LEX81B
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; File LEX81B.MAC
;
; LEX71B modified to use memory management directives
;
; Program to attach to the existing region LEX71A, create
; a virtual address window (mapped on creation), decrement
; all values in the region by 1, detach from the region
; and exit.
;
; Assemble and task-build instructions:
;
;       >MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]LEX81B
;       >LINK/MAP/OPTION LEX81B,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$  32.,,LEX71A,LEX71A,<RS.WRT!RS.RED>
;       Define region with:
;               Size            = 32. (32. word blocks)
;               Name            = LEX71A
;               Partition       = LEX71A
;               Attach with read and write access
;
WIN:    CRAW$   WDB     ;DPB for create address window
WDB:    WDBBK$  7,32.,0,0,32.,<WS.MAP!WS.RED!WS.WRT>
;       Define window with:
;               APR               = 7
;               Size              = 32. (32. word blocks)
;               Offset in region = 0 (32. word blocks)
;               Length in region = 32. (32. word blocks)
;               Map on create with read and write access
;
IOSB:   .BLKW   2               ; I/O status block
W       =1024.                  ; # of words in region
DONE:   .ASCII  /LEX81B HAS MODIFIED THE VALUES/ ; Done
        .ASCII  / IN LEX71A/                    ;  message
LDONE   =.-DONE
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
        BCS     ERR2            ; Check for error
```

Dynamic Regions

SOLUTION

```
C WDB = Window definition block with the following properties:
C       APR               7
C       Size              32 (10) (32.-word blocks)
C       Offset in region  0 (32.-word blocks)
C       Length of window  32 (10) (32.-word blocks)
C       Map on create with read and write access
C Initialize the WDB
        DATA WDB /"3400,0,32,0,0,32,"203,0/
C
C Attach region
        CALL ATRG (RDB,IDS)
C Check for error on attach
        IF (IDS .LT. 0) GOTO 100
C Move region id to WDB
        WDB(4)=RDB(1)
C Create and map window
        CALL CRAW (WDB,IDS)
C Check for error
        IF (IDS .LT. 0) GOTO 200
C Decrement values
        DO 50 K=1,1024
        IDATA(K)=IDATA(K)-1
50      CONTINUE
C Detach from region and delete it
        CALL DTRG (RDB,IDS)
C Check for error
        IF (IDS .LT. 0) GOTO 300
C And Jump to exit
        WRITE (5,60)
60      FORMAT (' LEX81B HAS MODIFIED THE VALUES IN
        1 THE COMMON LEX71A')
        GOTO 500
C
C       Error messages
100     WRITE (5,101) IDS
101     FORMAT (' ERROR ATTACHING TO REGION, DSW =',I4)
        GOTO 500
200     WRITE (5,201) IDS
201     FORMAT (' ERROR IN CREATING WINDOW, DSW =',I4)
        GOTO 500
300     WRITE (5,301) IDS
301     FORMAT (' ERROR DETACHING FROM REGION, DSW =',I4)
C
500     CALL EXIT
        END
```

Dynamic Regions

SOLUTION

```
         BCS     ERR2              ; Check for error
         MOV     #160000,R2        ; Set base addr in region
         MOV     #N,R5             ; Loop count
LOOP:    MOV     #BUFF,R0          ; Set up for $EDMSG
         MOV     #FMT,R1
         CALL    $EDMSG            ; Edit data
         QIOW$S  #IO.WVB,#5,#1,,#IOSB,,<#BUFF,R1,#40>
                                   ; Write data
         BCS     ERR3D             ; Check for dir error
         TSTB    IOSB              ; Check for I/O error
         BLT     ERR3I             ; Branch on error
         SOB     R5,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
```

```
         PROGRAM LEX81C
C
C File LEX81C.FTN
C
C LEX71C modified to use memory management directives
C
C Program to attach region LEX71A in partition LEX71A
C create a window and map it to the region upon creation,
C read data out of the region, and detach from it
C
C Task-build with these options:
C                 VSECT=DATA:160000:20000
C                 WNDWS=1
C
         INTEGER RDB(8),WDB(8)
C This common block will align with the address window
         COMMON /DATA/IDATA(1024)
C RDB = Region definition block with the following
C properties:
C        Size              32 (10) (32.-word blocks)
C        Name              LEX71A
C        Partition         LEX71A
C        Protection        WO:none,SY:RWED,OW:RWED,GR:RWED
C        Attach with read access
C Initialize the RDB
         DATA RDB /0,32,3RLEX,3R71A,3RLEX,3R71A,"000001,
         1"170000/
```

Dynamic Regions

SOLUTION

2.

```
        .TITLE  LEX82A
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;
; File LEX82A.MAC
;
; Program to create an named region (attached on
; creation), create a virtual address window (mapped on
; creation), place ASCII data in to region, detach from
; the region and exit, leaving the region in existence.
;
; Task-build instructions:
;
;       Include WNDWS=1 option
;
        .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            = 2 (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 write and delete access
RDB:    RDBBK$  2,MYREG,GEN,<RS.NDL!RS.DEL!RS.WRT!RS.ATT>,170000
;
WIN:    CRAW$   WDB     ; DPB for create address window
;       Define window with:
;               APR               = 7
;               Size              = 2 (32. word blocks)
;               Offset in region = 0 (32. word blocks)
;               Length in region = 2 (32. word blocks)
;               Map on create with write access
WDB:    WDBBK$  7,2,0,0,2,<WS.MAP!WS.WRT>
;
DET:    DTRG$   RDB     ; DPB for detaching region
IOSB:   .BLKW   2       ; I/O status block
BUFF:   .BLKB   80.     ; Input/Output buffer
MES:    .ASCII  /ENTER ASCII CHARACTER: /
LEN     =       .-MES
DNMES:  .ASCII  <15>/LEX82A HAS CREATED AND INITIALIZED/
        .ASCII  / THE REGION/
LDNMES  =.-DNMES
; Error format strings
FCRRER: .ASCIZ  /ERROR CREATING REGION. DSW = %D./
FCRWER: .ASCIZ  /ERROR CREATING WINDOW. DSW = %D./
FDETER: .ASCIZ  /ERROR DETACHING FROM REGION. DSW = %D./
```

SOLUTION

```
        MOV     #FQI2IE,R1        ; QIO write err message
        BR      SHOERR            ; Branch to common code
ERR5:   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
```

```
        PROGRAM LEX82A
C
C File LEX82A.FTN
C
C LEX82A creates a named region (attached on creation),
C creates a virtual address window (mapped on creation),
C places an ASCII character input at TI: at all locations
C in the region, detaches from the region and exits,
C leaving the region in existence.
C
C Task-build instructions:
C
C       >LINK/MAP/OPTIONS/CODE:FPP LEX82A,LB:[1,1]FOROTS-
C       ->/LIBRARY
C       Option? VSECT=DATA:160000:20000
C       Option? WNDWS=1
C       Option? <RET>
C
C RDB = Region Definition Block for region with the
C following properties:
C       Size              = 2 (32. word blocks)
C       Name              = MYREG
C       Partition         = GEN
C       Protection        = WO:None,SY:RWED
C                           OW:RWED,GR:RWED
C       Do not mark for delete on last detach
C       Attach with write and delete access
C
C WDB = Window Definition Block for window with the
C following properties:
C       APR               = 7
C       Size              = 2 (32. word blocks)
C       Offset in region = 0 (32. word blocks)
C       Length in region = 2 (32. word blocks)
C       Map on create with write access
C
```

SOLUTION

```
        .TITLE  LEX82B
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;+
; File LEX82B.MAC
;
; Program to attach to an existing region, create a
; virtual address window (mapped on creation), modify
; the first byte of the region, read ASCII data from the
; region, detach from the region and mark it for delete,
; and finally exit.  The region will be deleted on last
; detach.
;
; Assemble and task-build instructions:
;
;       >MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]-
;       ->LEX82B
;       >LINK/MAP/OPTION LEX82B,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$  0,MYREG,GEN,<RS.WRT!RS.RED!RS.MDL!RS.DEL>
;       Define region with:
                Size              = 0 (32. word blocks)
;                                   returned after attach
;               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$  7,200,0,0,0,<WS.MAP!WS.RED!WS.WRT>
;       Define window with:
;               APR                = 7
;               Size               = 200 (32. word blocks)
;               Offset in region = 0 (32. word blocks)
;               Length in region = 0 (32. word blocks)
;                                       returned when mapped
;               Map on create with read and write access
;
IOSB:   .BLKW   2                   ; I/O status block
RSIZ    =128.                       ; Region size in bytes
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
```

Dynamic Regions

SOLUTION

```
C
        INTEGER RDB(8),WDB(8)
        BYTE IDATA(128)
C This common block will align with the address window
        COMMON /DATA/IDATA
C RDB = Region definition block with the following
C properties:
C       Size            0 (32.-word blocks)
C                         filled in when attached
C       Name            MYREG
C       Partition       GEN
C       Protection      WO:none,SY:RWED,OW:RWED,GR:RWED
C       Mark for delete on last detach
C       Attach with delete, write and read access
C Initialize the RDB
        DATA RDB /0,0,3RMYR,3REG ,3RGEN,3R   ,"000213,
        1"170000/
C
C WDB = Window definition block with the following
C properties:
C       APR               7
C       Size              200(8) (32.-word blocks)
C       Offset in region  0 (32.-word blocks)
C       Length of window  0 (32.-word blocks)
C                           filled in when mapped
C       Map on create with read access
C Initialize the WDB
        DATA WDB /"3400,0,"200,0,0,0,"203,0/
C
C Attach region
        CALL ATRG (RDB,IDS)
C Check for error on attach
        IF (IDS .LT. 0) GOTO 100
C Move region id to WDB
        WDB(4)=RDB(1)
C Create and map window
        CALL CRAW (WDB,IDS)
C Check for error
        IF (IDS .LT. 0) GOTO 200
C Place ASCII Z in first byte
        IDATA(1)='Z'
C Print contents of region
10      WRITE (5,11) IDATA
11      FORMAT (' ',64A1)
C Detach from region and delete it
        CALL DTRG (RDB,IDS)
C Check for error
        IF (IDS .LT. 0) GOTO 300
C And Jump to exit
        GOTO 500
```

Dynamic Regions

SOLUTION

3.

```
        .TITLE  SNDREF
        .IDENT  /01/
        .ENABL  LC                      ; Enable lower case
;+
; File LEX83A.MAC                       ;;EX
;
; Modified to send to a 2nd receiver RCVRF2 in  ;;EX
; addition to RCVREF                            ;;EX
;
; LEX83A 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]-;;EX
;       ->LEX83A
;       >LINK/MAP/OPTION LEX83A,LB:[1,1]PROGSUBS/LIBRARY
;       Option? WNDWS=1
;
; Install and run instructions: RCVREF must be installed.
; LEX83B must be installed as RCVRF2. Run LEX83A first,
; then run RCVREF and RCVRF2 (either one first)
;-
        .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

```

SOLUTION

```
                QIOW$C  IO.WVB,5,2,,,,<MES2,LMES2,40> ; Display
                                                      ;  message
                BCS     6$                  ; Branch on dir error
                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$:     DIRERR  <ERROR ON 2ND WRITE>
        7$:     DIRERR  <ERROR ON 2ND SEND BY REFERENCE>     ;;EX
        8$:     DIRERR  <ERROR ON 2ND WAIT FOR>              ;;EX
                .END    START
```

```
              PROGRAM SNDREF
        C
        C File LEX83A.FTN
        C
        C Modified to send the region by reference to RCVRF2 !!EX
        C in addition to RCVREF                               !!EX
        C
        C This program creates a 64-word unnamed region and
        C fills it with ASCII characters.  It then sends it by
        C reference to task RCVREF, and waits for RCVREF to
        C receive the region.(This is signalled by event flag
        C #1.)  SNDREF then prints a message and exits.  Since
        C the area is unnamed, it is automatically deleted when
        C the last attached task exits.
        C
        C Task-build instructions:
        C
        C       >LINK/MAP/CODE:FPP/OPTIONS LEX83A,LB:[1,1]FO-!!EX
        C       ->ROTS/LIBRARY                                !!EX
        C       Option? WNDWS=1
        C       Option? VSECT=DATA:160000:200
        C       Option? <RET>
        C
        C Install and run instructions: RCVREF must be installed.
        C LEX83B must be installed under the name RCVRF2.    !!EX
        C Run LEX83A first, then run RCVREF and RCVRF2 (in   !!EX
        C either order)
        C
        C       RDB = Region definition block with the following
        C       properties:
        C               Size            2 32-word blocks
        C               Name            none
        C               Partition       GEN
        C               Protection      WO:none,SY:RWED,OW:RWED,
        C                               GR:none
        C               Attach on creation
        C               Read and write access desired on attach
        C
```

SOLUTION

```
C Error handling code
100     WRITE (5,110)IDS
110     FORMAT (' ERROR CREATING REGION, DSW = ',I4)
        GOTO 600
200     WRITE (5,210)IDS
210     FORMAT (' ERROR CREATING WINDOW, DSW = ',I4)
        GOTO 600
400     WRITE (5,410)IDS
410     FORMAT (' ERROR IN SEND-BY-REFERENCE, DSW = ',I4)
        GOTO 600
450     WRITE (5,460)IDS
460     FORMAT (' ERROR IN 2ND SEND-BY-REFERENCE, DSW
        1 = ',I4)                                    !!EX
        GOTO 600                                     !!EX
500     WRITE (5,510)IDS
510     FORMAT (' ERROR ON WAIT, DSW = ',I4)
        GOTO 600                                     !!EX
550     WRITE (5,560)IDS                             !!EX
560     FORMAT (' ERROR ON 2ND WAIT, DSW = ',I4)     !!EX
C
600     CALL EXIT
        END
```

```
        .TITLE  LEX83B
        .IDENT  /01/
        .ENABL  LC                  ; Enable lower case
;
; File LEX83B.MAC
;
; Second reciever for SNDREF (modifed to LEX83A).
; Program to receive-by-reference (mapped on creation),
; modify the first data byte in the region,
; 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 the count of the
; number of bytes of data in the region.
;
; Assemble and task build instructions:
;
;       >MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]
;       ->LEX83B
;       LINK/MAP/OPTIONS LEX83B,LB:[1,1]PROGSUBS/LIBRARY
;       option? WNDWS=1
```

Dynamic Regions

SOLUTION

```
      PROGRAM LEX83B
C
C File LEX83B.FTN
C
C LEX83B receives by reference a region from the task
C LEX83A.  It maps to the region, modifies the first
C byte, prints out the contents, and exits. The region
C is deleted on last detach.
C
C Task-build instructions: Include these options
C                  WNDWS=1
C                  VSECT=DATA:160000:20000
C
C Install and run instructions: LEX83B must be installed.
C as RCVRF2. RCVREF must be installed. Run LEX83A first,
C then run LEX83B and RCVREF (in either order).
C
C WDB = Window definition block with:
C       APR                 7
C       Size                200(8) 32-word blocks
C                                  Allow for full APR
C       Offset in region    0 32-word blocks
C       Length of region    0 32-word blocks (to be filled
C                             in on receive)
C       Read and write access
      INTEGER WDB(8)
      DATA WDB/"3400,0,"2,0,0,"0,"3,0/
      BYTE DATA(128)
C This common block will align with the address window
      COMMON /DATA/DATA
C
C Create address window--do not map at this time
      CALL CRAW(WDB,IDS)
C Check for error on create
      IF (IDS .LT. 0) GOTO 200
C Now set WDB status for mapping--will be done by
C receive-by-reference
      WDB(7)=WDB(7)+"200
C Receive data and map
      CALL RREF(WDB,,IDS)
C Check for error
      IF (IDS .LT. 0) GOTO 100
C Modify first value
      DATA(1)='9'
C Calculate number of bytes of data - length in blocks
C returned at WDB(6)
      NCHAR = 64*WDB(6)
      WRITE(5,10) (DATA(I),I=1,NCHAR)
10    FORMAT (' ',64A1)
C Go exit
      GOTO 300
```

File I/O

TEST/EXERCISE

1. Next to each activity, write O for open, I for I/O operation, or C for close, to identify which step of file I/O is involved.

   ___ a. Records are read from the file.

   ___ b. Access rights to the file are checked.

   ___ c. Existing file is located on disk.

   ___ d. Internal buffers are placed in a pool for re-use.

   ___ e. Records are written to a file.

2. Describe three functions performed by the Files-11 ancillary control processor (F11ACP) when a task creates a new file containing seven blocks.

TEST/EXERCISE

b. A company has a file of customer records. Each record contains the company name, the address, the contact person, and the equipment bought. At different times, the records are accessed using company name, city, or contact person.

c. A company uses COBOL for its applications. It has a payroll file which is processed in order every two weeks.

File I/O

SOLUTION

1. Next to each activity, write O for open, I for I/O operation, or C for close, to identify which step of file I/O is involved.

   <u>I</u> a. Records are read from the file.

   <u>O</u> b. Access rights to the file are checked.

   <u>O</u> c. Existing file is located on disk.

   <u>C</u> d. Internal buffers are placed in a pool for re-use.

   <u>I</u> e. Records are written to a file.

2. Describe three functions performed by the Files-11 ancillary control processor (F11ACP) when a task creates a new file containing seven blocks.

   Any three of the following:

   Allocate a file header

   Initialize the file header

   Set up file retrieval pointers

   Create a directory entry

   Allocate blocks to the file

   Connect a task's LUN to the file

SOLUTION

b. A company has a file of customer records. Each record contains the company name, the address, the contact person, and the equipment bought. At different times, the records are accessed using company name, city, or contact person.

Best answer is RMS only since an indexed file with multiple keys is needed for fastest access. FCS can be used, but access by key value is impossible. You would have to step through the file, checking all records, to locate the one you want.

c. A company uses COBOL for its applications. It has a payroll file which is processed in order every two weeks.

RMS only; COBOL is supported under RMS, but not under FCS.

File Control Services

TEST/EXERCISE

1. Modify CRESEQ so that each record in the file contains the text input from the terminal preceded by "AAAA".

2. Write a task that appends records to a file you have created (using one of the FCS example programs or the editor).

3. In MACRO-11, modify the task CREFXA so that input from the terminal uses FCS routines instead of QIO directives.

4. Write a task that requests input from a terminal of the form:

   n, text

   Use the input to update the nth record of FIXED.ASC, which has fixed length records. Use random access and do not truncate the file.

5. In MACRO-11, modify the task BLOCK1 or BLOCK2 so that it writes or displays two virtual blocks at a time.

6. (Optional) In MACRO-11, modify the task CSI so that the subroutines DISPLY and DELETE actually display and delete the file. Caution: DELET$ delete the highest version of a file if no version number is specified. (See Chapter 6 of the _IAS/RSX I/O Operations Reference Manual_ for information about the routines GCML and CSI.)

SOLUTION

1.
```
        .TITLE  CRESEQ
        .IDENT  /01/
        .ENABL  LC
;+
; File LEX101.MAC
;
; Modified to preced each record with AAAA
;
; CRESEQ creates a file VARI.ASC. 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$A ;
        .MCALL  NMBLK$,OPEN$W,PUT$,CLOSE$ ;
        .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$A  R.VAR,FD.CR     ; Variable length records,
                                ;  Listings - implied
                                ;  carriage return, line
                                ;  feed
        FDRC$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"

; Local Data
BUFF:   .ASCII  /AAAA/          ; USER RECORD BUFFER   ;;EX
INBUF:  .BLKB   80.             ;                      ;;EX
IOST:   .BLKW   2               ; I/O STATUS BLOCK

        .LIST   BEX
        .EVEN

        .ENABL  LSB
```

SOLUTION

```
        PROGRAM CRESEQ  !CREATE FILE SEQUENTIALLY
C
C FILE LEX101.FTN
C
C Modified to precede each record with AAAA     !!EX
C
C This task creates a file of VARI.ASC of
C variable-length records using sequential record access.
C The records are input from the terminal and copied to
C the file.  The process stops when the operator types
C CTRL/Z at the terminal.
C
        BYTE BUFF(84),INBUF(80)                 !!EX
        EQUIVALENCE (BUFF(5),INBUF(1))          !!EX
        INTEGER LEN
        DATA BUFF(1),BUFF(2),BUFF(3),BUFF(4)
        1 /'A','A','A','A'/
C
C
C OPEN FILE
C
C Default access is sequential
C Default is formatted I/O for sequential files
C
        OPEN    (UNIT=1,NAME='VARI.ASC',TYPE='NEW',
        1       CARRIAGECONTROL='LIST')
C
        TYPE    *,'TYPE IN TEXT, TERMINATE EACH RECORD
        1 WITH A CARRIAGE RETURN'
        TYPE    *,'TERMINATE INPUT WITH A CTRL/Z'
C Loop
10      READ (5,11,END=100) LEN,INBUF   ! Read record!!EX
11      FORMAT (Q,80A1)
C
        LEN = LEN+4                     ! Add 4 for A's
C                                                    !!EX
        WRITE (1,12) (BUFF(I),I=1,LEN)  ! Write record
12      FORMAT (80A1)                   !  to file
        GO TO 10
C Close file and exit
100     CLOSE   (UNIT=1)
        CALL EXIT
        END
```

SOLUTION

```
          QIOW$C  IO.RVB,5,1,,IOST,,<BUFF,80.>; Read a
                                      ;  line from TI:
          BCC     DIROK               ; Branch on Directive ok
          MOV     #EFIQIO,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 $EDMSG call
          MOV     #EFDQIO,R1          ;
          BR      SHOERR              ; Branch to show error
                                      ;  and exit
OKIO:     MOV     IOST+2,R1           ; Length of record to R1
          PUT$    #FDB,,R1,ERR2       ; Write next record
          BR      10$                 ; 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
                                      ; $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
```

SOLUTION

3.
```
        .TITLE  CREFXA
        .IDENT  /01/
        .ENABL  LC              ; Enable lower case
;+
; File LEX103.MAC
;
; Modified to use FCS instead of QIO's to get ;;EX
; input from TI:                              ;;EX
;
; 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$,DIR$

        .MCALL  FSRSZ$,FDBDF$,NMBLK$
        .MCALL  FDRC$A,FDAT$A,FDOP$A
        .MCALL  OPEN$W,GET$,PUT$,CLOSE$
        .MCALL  OPEN$R

        .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 CODE. ERROR CODE = %D.?

        .EVEN
        .LIST   BEX             ; Show offsets

        FSRSZ$  2               ; 2 files for record I/O
;                                             ;;EX

FDB:    FDBDF$                  ; File descriptor block
        FDRC$A  ,BUFF,RSIZ      ; User buffer and size
        FDAT$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
```

SOLUTION

```
; Error  Processing
ERR1:
ERR2:
ERR3:
ERR4:    TSTB     F.ERR+1(R0)       ; Directive error or I/O
                                    ;  error
         BEQ      IO                ; Branch on I/O error
DIRERR:  MOV      #EFCDIR,R1        ; Set up for $EDMSG,   ;;EX
                                    ;  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
                                    ; $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
         CLOSE$   #FDBI             ; Close "file" at TI: ;;EX
         EXST$C   EX$ERR            ; Exit with status of 2
         .END     START
```

File Control Services

SOLUTION

```

          .ENABL  LSB                 ; Allow local symbols
                                      ;  to cross Psect
                                      ;  boundaries

START:    OPEN$U  #FDB,,,,,,ERR1      ; Open file for update
                                      ;  (includes extend)
; Clear buffer to all blanks each time
10$:      MOV     #RSIZ,R1            ; Record size
          MOV     #BUFF,R2            ; R2 => buffer
20$:      MOVB    #'  ,(R2)+          ; Move in a blank
          SOB     R1,20$              ; Continue until done

          QIOW$C  IO.RPR,5,1,,IOST,,<BUFF,RSIZ,,INPT,LINPT,'$>
                                      ; Prompt and get input
          CMPB    #IE.EOF,IOST        ; Check for ^Z
          BEQ     EXIT                ; If ^Z, exit
          MOV     #BUFF,R0            ; Set up to convert
          CALL    $CDTB               ;  record # to binary
; Check for good conversion, character after # is
; returned in R2 (it should be a ",")
          CMPB    #',,R2              ; Is it a comma
          BEQ     GOOD                ; Branch on good
                                      ;  conversion
          QIOW$C  IO.WVB,5,1,,,,<CNVER,LCNVER,40>
                                      ; Display error message
          BCS     ERR4                ; Branch on directive
                                      ;  error
          BR      10$                 ; Get next input
GOOD:     PUT$R   #FDB,,,R1,,ERR2     ; Write record to output
                                      ;  file
          BR      10$                 ; Get next input
; Close file, display message, and exit
EXIT:     CLOSE$  #FDB,ERR3           ; Close file
          QIOW$C  IO.WVB,5,1,,,,<BUFF1,LEN1,40> ;Write
                                      ; message to operator
          BCS     ERR4                ; Branch on error
          EXIT$S


ERR1:
ERR2:     CLOSE$  #FDB,ERR3           ; Close file
ERR3:     MOVB    F.ERR(R0),R0        ; Move FCS error code
          MOV     R0,IOST             ;  to argument block
                                      ;  for $EDMSG
          MOV     #IOST,R2            ; Set up for $EDMSG
          TSTB    F.ERR+1(R0)         ; I/O or directive error
          BEQ     IOERR               ; Branch on I/O error
          MOV     #EMESD,R1           ; Set up for dir error
                                      ;  message
          BR      COMME               ; Branch to common code
IOERR:    MOV     #EMESI,R1           ; Set up for I/O error
                                      ;  message
```

SOLUTION

5.

```
        .TITLE  BLOCK2
        .IDENT  /01/
        .ENABL  LC                ; Enable lower case
;+
; File LEX105.MAC                         ;;EX
;
; Modified to work on 2 virtual blocks at a time ;;EX
;
; **-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
        .MCALL  FDBDF$,FDRC$A,FDBK$A,FDOP$A,NMBLK$
        .MCALL  FSRSZ$,OPEN$R,READ$,WAIT$,CLOSE$

        .SBTTL  MESSAGES
        .NLIST  BEX
CR      = 15
LF      = 12
MES1:   .ASCII  /FIRST VIRTUAL BLOCK: /             ;;EX
LEN1    = . - MES1
MES2:   .ASCII  <CR><LF>/HERE ARE THE BLOCKS : /<CR><LF>
;                                                   ;;EX
LEN2    = . - MES2
MES3I:  .ASCIZ  'I/O ERROR FROM OPEN$R, CODE = %D.'
MES3D:  .ASCIZ  /DIRECTIVE ERROR FROM OPEN$R, CODE = %D./
MES4I:  .ASCIZ  'I/O ERROR FROM READ$, CODE = %D.'
MES4D:  .ASCIZ  /DIRECTIVE ERROR FROM READ$, CODE - %D./
MES5I:  .ASCIZ  'I/O ERROR FROM WAIT$, CODE = %D.'
MES5D:  .ASCIZ  /DIRECTIVE ERROR FROM WAIT$, CODE = %D./
BUFF:   .BLKB   80.               ; STORE RESPONSE HERE

        .LIST   BEX
        .EVEN
        .SBTTL  LOCAL STORAGE

        FSRSZ$  0                 ; NO FSR BUFFER NEEDED
                                  ; FOR BLOCK I/O

FDB:    FDBDF$                    ; FDB FOR INPUT FILE
        FDRC$A  FD.RWM            ; READ/WRITE MODE
        FDBK$A  BLOCK,1024.,,,1,IOSB ; EF 1, BUFFER ADR,;;EX
                                  ; SIZE
        FDOP$A  1,,FILE           ; LUN 1, DFNB
FILE:   NMBLK$  BLOCK,ASC         ; NAME IS BLOCK.ASC

VBN:    .WORD   0,1               ; DEFAULT VBN
BLOCK:  .BLKW   512.              ; BLOCK BUFFER        ;;EX
IOSB:   .BLKW   2
```

SOLUTION

```
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
                                    ; FALL INTO COMMON CODE
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
EXIT:
        CLOSE$  #FDB                ; Close the file
        EXST$S  R5                  ; Exit with status
        .END    START
```

File Control Services

SOLUTION

```
        CSI$                    ; Define CSI offsets
CBLK:   .BLKB   C.SIZE          ; allocate CSI storage
        .EVEN

        DEMSK = 1               ; Delete mask
        DIMSK = 2               ; Display mask
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$  3               ; GCML uses record I/O;;EX

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

FDBO:   FDBDF$                  ; FDB for output to TI:;;EX
        FDAT$A  R.VAR,FD.CR     ; Var length records, ;;EX
                                ;  list format        ;;EX
        FDRC$A  ,TBUFF,132.     ; URB at TBUFF, length;;EX
                                ;  132.               ;;EX
        FDOP$A  2,DSPTO         ; LUN 2, dataset      ;;EX
                                ; descriptor at DSPTO ;;EX
DSPTO:  .WORD   LDEV,DEV        ; Dataset descriptor  ;;EX
        .WORD   0,0             ;  for TI:. No UIC or ;;EX
        .WORD   0,0             ;  name needed.       ;;EX
DEV:    .ASCII  /TI:/           ;                     ;;EX
        LDEV=.-DEV              ;                     ;;EX

        .EVEN
JMPTBL: .WORD   NONE,DELETE,DISPLY ; Jump table for
                                ;  subroutines depending
                                ;  on switches
COPY:   .WORD   1               ; Value for N in /DI:N
```

SOLUTION

```
        CALL    OUTMS              ; Call OUTMS, as a     ;;EX
                                   ; subroutine           ;;EX
        RETURN                     ; Return               ;;EX

; Common display message code - a subroutine since it ;;EX
; is not a common return point                        ;;EX

OUTMS:  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
;
; ***WARNING - THE HIGHEST VERSION NUMBER OF THE FILE  ***
; ***WILL BE DELETED IF NO VERSION NUMBER IS SPECIFIED ***

DELETE: MOV     #DELTXT,DATA       ; Set up for output of
                                   ;  message
        CALL    OUTMS              ; Call display         ;;EX
                                   ;  subroutine          ;;EX
        DELET$  #FDB,ERRD          ; Delete file          ;;EX
        RETURN                     ; Return
; Delete error code
ERRD:   MOVB    F.ERR(R0),R5       ; Extend sign on error;;EX
        MOV     R5,DATA+2          ;  and move to arg block;;EX
        MOV     #DELTXT,DATA       ; Move pointer to delete;;EX
                                   ;  text                ;;EX
COMME:  TSTB    F.ERR+1(R0)        ; Check for directive ;;EX
                                   ;  error or I/O error ;;EX
        BEQ     IOERR              ; Branch on I/O error ;;EX
        MOV     #FMTERD,R1         ; Get format string   ;;EX
        BR      DISPER             ; Branch to common    ;;EX
                                   ;  error display code ;;EX
IOERR:  MOV     #FMTERI,R1         ; Get format string   ;;EX
DISPER: MOV     #BUFF,R0           ; Set up for $EDMSG   ;;EX
        MOV     #DATA,R2           ;                     ;;EX
        CALL    $EDMSG             ; Edit message        ;;EX
        MOV     R1,TYPE4+Q.IOPL+2  ; Size of message     ;;EX
        DIR$    #TYPE4             ; Display message     ;;EX
        EXIT$S                     ; Exit                ;;EX

; Subroutine DISPLY - Just display a message

DISPLY: CALL    $SAVAL             ; Save all registers
        MOV     #DITXT,DATA        ; Set up for output of
                                   ;  message
        CALL    OUTMS              ; Branch to common
                                   ;  display code
```