RSX–11M/M–PLUS

Task Builder Manual

Order No. AA-H266A-TC

RSX-11M Version 3.2
RSX-11M-PLUS Version 1.0

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, June 1979

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 only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL
DEC
PDP
DECUS
UNIBUS
COMPUTER LABS
COMTEX
DDT
DECCOMM
ASSIST-11
VAX
DECnet
DATATRIEVE
DECsystem-10
DECtape
DIBOL
EDUSYSTEM
FLIP CHIP
FOCAL
INDAC
LAB-8
DECSYSTEM-20
RTS-8
VMS
IAS
TRAX
MASSBUS
OMNIBUS
OS/8
PHA
RSTS
RSX
TYPESET-8
TYPESET-11
TMS-11
ITPS-10
SBI
PDT

CONTENTS

CONTENTS

CONTENTS

Page

CONTENTS

CONTENTS

FIGURES

CONTENTS

FIGURES (Cont.)

CONTENTS

FIGURES (Cont.)

CONTENTS

FIGURES (Cont.)

TABLES

PREFACE

MANUAL OBJECTIVES

This manual describes the concepts and capabilities of the RSX-11M/M-PLUS Task Builder.

Working examples are used throughout this manual to introduce and describe features of the Task Builder. Because RSX-11M systems support a large number of programming languages, it is not practical to illustrate the Task Builder features in all of the languages supported. Instead, most of the examples in the main text of this manual are written in MACRO-11.

INTENDED AUDIENCE

Before reading this manual, you should be familiar with the fundamental concepts of your operating system (RSX-11M or RSX-11M-PLUS) and with the operating procedures described in the RSX-11M/M-PLUS MCR Operations Manual. In addition, you should be familiar with the programming concepts described in the RSX-11M/M-PLUS Guide to Program Development.

STRUCTURE OF THIS DOCUMENT

This manual has eight chapters. Their contents are summarized as follows:

- Chapter 1 describes the Task Builder command sequences that you use to interact with the Task Builder.

- Chapter 2 describes the basic Task Builder functions, including the Task Builder's allocation of virtual address space, the resolution of global symbols, and privileged tasks.

- Chapter 3 describes some typical Task Builder features including tasks that access shared regions and device commons, multiuser tasks, tasks that create dynamic regions, virtual program sections and privileged tasks.

- Chapter 4 describes the Task Builder's overlay capability and the language you must use to define an overlay structure.

- Chapter 5 describes the two methods available to you to load overlay segments.

- Chapter 6 lists the Task Builder switches and options in two sections. Both switches and options are listed in alphabetical order in their respective sections, and are printed on colored stock to help you find them quickly.

- Chapter 7 describes the considerations for building a task on one system to run on a system with a different hardware configuration.

- Chapter 8 describes two memory dumps -- Postmortem and Snapshot.

This manual also contains six appendixes. Their contents are summarized as follows:

- Appendix A contains a detailed description of the Task Builder input data structures.

- Appendix B contains a detailed description of the task image file structure.

- Appendix C contains a list of the symbols and program section names reserved for Task Builder use.

- Appendix D contains information on improving Task Builder performance.

- Appendix E describes the fast Task Builder.

- Appendix F contains the Task Builder error messages. These are also printed on colored stock for quick reference.

A Task Builder glossary follows the appendixes.

ASSOCIATED DOCUMENTS

Other manuals closely allied with this document are described in the documentation directory for your operating system. This directory defines the intended audience of each manual in the documentation set and provides a brief synopsis of each manual's contents.

CONVENTIONS USED IN THIS DOCUMENT

In this manual, horizontal ellipses (...) indicate that additional, optional arguments in a statement format have been omitted. For example:

```
input-spec,...
```

This means that one or more input-spec items, separated by commas, can be specified.

Vertical ellipses means that additional lines of code or lines in a Task Builder map file are not pertinent to an example, have been omitted. For example:

```
TKB>input-line
      .
      .
      .
```

This means that one or more of the indicated TKB items have been omitted.

Finally, in the examples of Task Builder command sequences, the portion of the command sequence that you type is printed in red. The Task Builder's responses and prompts are printed in black.

SUMMARY OF TECHNICAL CHANGES

This manual documents RSX-11M-PLUS Task Builder enhancements. The following RSX-11M-PLUS features have been added to the Task Builder.

The MU switch has been added to provide for the building of multiuser tasks. Multiuser tasks allow more than one user to share the read-only portions of a single task.

The following options have also been added:

- ROPAR (read-only partition) specifies the partition in which the read-only portion of a multiuser task is to reside.

- GBLXCL (global exclusion) specifies the symbols that are to be excluded from the symbol definition file of a resident supervisor-mode library.

- RESSUP (user-owned resident supervisor-mode library) specifies that the task expects to access a resident supervisor-mode library.

- SUPLIB (system-owned resident supervisor-mode library) specifies that the task expects to access a system-owned resident supervisor-mode library.

- CMPRT (completion routine) identifies the completion routine in a supervisor-mode library.

These new features are described in Chapter 6. Chapter 3 contains examples that illustrate them.

While you can specify these features in the Task Builder command sequence on both RSX-11M V3.2 and RSX-11M-PLUS systems, the resulting tasks can be installed and run only on RSX-11M-PLUS systems.

CHAPTER 1

INTRODUCTION AND COMMAND SPECIFICATIONS

The basic steps in the development of a program are as follows:

1. You write one or more routines in an RSX-11M/M-PLUS supported source language and enter each routine as an ASCII text file, through an editor.

2. You submit each text file to the appropriate language translator (an assembler or compiler), which converts it to a relocatable object module.

3. You specify the object modules as input to the Task Builder, which combines the object modules into a single task image output file.

4. You install and run the task.

If you find errors in the task when you run it, you make corrections to the text file, using the editor, and then repeat steps 2 through 4.

The Task Builder's main function is to convert relocatable object modules (.OBJ files) into a single task image (.TSK file) that you can install and run on a RSX-11M or RSX-11M-PLUS system. The task is the fundamental executable unit in both systems.

If your program consists of a single object module, the use of the Task Builder is appropriately simple. You specify as input only the name of the file containing the object module produced from the translation of the program, and specify as output the task image file.

Typically, however, programs consist of more than a single object module. In this case, you name each of the object module files as input. The Task Builder links the object modules, resolves references between them, resolves references to the system library, and produces a single task image ready to be installed and executed.

The Task Builder makes a set of assumptions (defaults) about the task image based on typical usage and storage requirements. You can override these assumptions by including switches and options in the task-building terminal sequence. Thus, you can build a task that is tailored to its own input/output and storage requirements.

The Task Builder also produces (upon request) a memory allocation (map) file that contains information describing the allocation of address space, the modules that make up the task image, and the value of all global symbols. In addition, you can request that a list of global symbols, accompanied by the name of each referencing module, be appended to the file (global cross reference).

The following example shows a simple sequence for building a task:

```
>MAC PROG=PROG
>TKB PROG=PROG
>INS PROG
>RUN PROG
```

The first command (MAC) causes the MACRO-11 assembler to translate the source code of the file PROG.MAC into a relocatable object module in the file PROG.OBJ. The second command (TKB) causes the Task Builder to process the file PROG.OBJ to produce the task image file PROG.TSK. The third command (INS) causes the INSTALL processor to add the task to the Executive's directory of executable tasks (System Task Directory). The fourth command (RUN) causes the task to execute.

The example just given includes the command

```
>TKB PROG=PROG
```

This command illustrates the simplest use of the Task Builder. It gives the name of a single file as output and the name of a single file as input.

This chapter describes basic Task Builder command forms and sequences.

1.1 TASK COMMAND LINE

The task command line contains the output file specifications, followed by the input file specifications; they are separated by an equal sign (=). You can specify up to three output files and any number of input files.

You must give the output files in a specific order: the first file you name is the image (.TSK) file, the second is the memory allocation (.MAP) file, and the third is the symbol definition (.STB) file. The map file lists information about the size and location of components within the task. The symbol definition file contains the global symbol definitions in the task and their virtual or relocatable addresses in a format suitable for reprocessing by the Task Builder. You specify this file when you are building a resident library or common. (Resident libraries and commons are described in Chapter 3.) The Task Builder combines the input files to create a single task image that can be installed and executed.

The task command line has the form:

```
task-image-file,map-file,symbol-definition-file=input-file,...
```

You can omit any output file by replacing the file specification with the delimiting comma that would normally follow it. The following commands illustrate the ways the Task Builder interprets the output file names.

Command	Output Files
>TKB IMG1,IMG1,IMG1=IN1	The task image file is IMG1.TSK, the memory allocation (map) file is IMG1.MAP, and the symbol definition file is IMG1.STB.
>TKB IMG1=IN1	The task image file is IMG1.TSK.
>TKB ,IMG1=IN1	The map file is IMG1.MAP.

Command	Output Files
>TKB ,,IMG1=IN1	The symbol definition file is IMG1.STB.
>TKB IMG1,,IMG1=IN1	The task image file is IMG1.TSK and the symbol definition file is IMG1.STB.
>TKB =IN1	This is a diagnostic run with no output files.

1.2 MULTIPLE LINE INPUT

Although you can specify a maximum of three output files, you can specify any number of input files. When you specify several input files, a more flexible format is sometimes necessary -- one that consists of several lines. This multiline format is also necessary when you want to include options in your command sequence (see Section 1.3).

If you type TKB, the Monitor Console Routine (MCR) activates the Task Builder. The Task Builder then prompts for input until it receives a line consisting only of the terminating slash characters (//) For example:

```
>TKB
TKB>IMG1,IMG1=IN1
TKB>IN2,IN3
TKB>//
```

This sequence produces the same result as the single line command:

```
>TKB IMG1,IMG1=IN1,IN2,IN3
```

Both command sequences produce the task image file IMG1.TSK and the map file IMG1.MAP from the input files IN1.OBJ, IN2.OBJ, and IN3.OBJ.

You must specify the output file specifications and the equal sign (=) on the first command line. You can begin or continue input file specifications on subsequent lines.

When you type the terminating slash characters (//), the Task Builder stops accepting input, builds the task, and returns control to MCR.

1.3 OPTIONS

You use options to specify the characteristics of the task you are building. To include options in a task, you must use the multiline format. If you type a single slash (/) following the input file specification, the Task Builder requests option information by displaying ENTER OPTIONS: and prompting for input. For example:

```
>TKB
TKB>IMG1,IMG1=IN1
TKB>IN2,IN3
TKB>/
ENTER OPTIONS:
TKB>PRI=100
TKB>COMMON=JRNAL:RO
TKB>//
```

In this sequence there are two options: PRI=100 and COMMON=JRNAL:RO. The two slashes end option input, initiate the task-build, and return control to MCR upon completion.

NOTE

> When you are building an overlaid task, there are exceptions to the use of the single slash (/). Overlaid tasks are described in Chapter 4.

The RSX-11M/M-PLUS Task Builder provides numerous options. These are described in Chapter 6. The general form of an option is a keyword followed by an equal sign (=) and an argument list. The arguments in the list are separated from one another by colons (:). In the example above, the first option consists of the keyword PRI and a single argument indicating that the task is to be assigned the priority 100. The second option consists of the keyword COMMON and an argument list, JRNAL:RO, indicating that the task accesses a resident common region named JRNAL and that the access is read-only. You can specify more than one option on a line, by using an exclamation point (!) to separate the options. For example:

```
     TKB>PRI=100!COMMON=JRNAL:RO
```

This command is equivalent to the two lines:

```
     TKB>PRI=100
     TKB>COMMON=JRNAL:RO
```

Some options accept more than one set of argument lists. You use a comma (,) to separate the argument lists. For example:

```
     TKB>COMMON=JRNAL:RO,RFIL:RW
```

In this command, the first argument list indicates that the task has requested read-only access to the resident common JRNAL. The second argument list indicates that the task has requested read/write access to the resident common RFIL.

The following three sequences are equivalent:

```
     TKB>COMMON=JRNAL:RO,RFIL:RW

     TKB>COMMON=JRNAL:RO!COMMON=RFIL:RW

     TKB>COMMON=JRNAL:RO
     TKB>COMMON=RFIL:RW
```

1.4 MULTIPLE TASK SPECIFICATIONS

If you intend to build more than one task, you can use the single slash (/) following option input. This directs the Task Builder to stop accepting input, build the task, and request information for the next task-build.

For example:

```
>TKB
TKB>IMG1=IN1
TKB>IN2,IN3
TKB>/
ENTER OPTIONS:
TKB>PRI=100
TKB>COMMON=JRNAL:RO
TKB>/
TKB>IMG2=SUB1
TKB>//
```

The Task Builder accepts the output and input file specifications and the option input; it then stops accepting input upon encountering the single slash (/) during option input. The Task Builder builds IMG1.TSK and then returns to accept more input for building IMG2.TSK.

1.5 INDIRECT COMMAND FILES

You can enter commands to the Task Builder directly from the keyboard, or indirectly through the indirect command file facility. To use the indirect command file facility, you prepare a file that contains the Task Builder commands you want to be executed. Later, after you invoke the Task Builder, you type an at sign (@) followed by the name of the indirect command file.

For example, suppose you create a file called AFIL.CMD containing the following:

```
IMG1,IMG1=IN1
IN2,IN3
/
PRI=100
COMMON=JRNAL:RO
//
```

Later, you can type:

```
>TKB
TKB>@AFIL
TKB>

     or simply
>TKB @AFIL
```

When the Task Builder encounters the at sign (@), it directs its search for commands to the file named AFIL.CMD. The example above is equivalent to the keyboard sequence:

```
>TKB
TKB>IMG1,IMG1=IN1
TKB>IN2,IN3
TKB>/
ENTER OPTIONS:
TKB>PRI=100
TKB>COMMON=JRNAL:RO
TKB>//
```

When the Task Builder encounters two terminating slash characters (//) in the indirect command file, it terminates indirect command file processing, builds the task, and exits to MCR.

When the Task Builder encounters a single slash (/) in an indirect command file and the slash is the last character in the file, the Task Builder directs its search for commands to the terminal. For example, suppose the file AFIL.CMD in the last example is changed to read:

```
IMG1,IMG1=IN1
IN2,IN3
/
```

Later, you can type:

```
>TKB
TKB>@AFIL
```

In this case, the Task Builder goes to the terminal and prompts:

```
ENTER OPTIONS:
TKB>
```

From this point, you input options to the Task Builder directly from the keyboard. If you then conclude option input from the keyboard with double slashes (//), the Task Builder suspends command processing, as described above, and exits to MCR following the task-build. If you conclude option input with a single slash (/), the Task Builder prompts for new command input following the task-build of IMG1.TSK, as follows:

```
TKB>
```

Using the single slash (/) following option input in indirect command files is a convenient way to return control to your terminal between successive task-builds. For example, suppose you create two indirect command files. The first, AFIL.CMD, contains:

```
IMG1,IMG1=IN1
IN2,IN3
/
PRI=100
COMMON=JRNAL
/
```

The second, AFIL1.CMD, contains:

```
IMG2,IMG2=IN4
IN5,IN6
/
PRI=100
//
```

Then, the terminal sequence to build these two tasks is:

```
>TKB
TKB>@AFIL
TKB>@AFIL1
>
```

NOTE

For interaction with a Task Builder indirect command file as described above, you must use the multiline format when you specify the indirect command file.

The Task Builder permits two levels of indirection in file references. That is, the indirect command file referenced in a terminal sequence can contain a reference to another indirect command file. For example, if the file BFIL.CMD contains all the standard options that are used by a particular group of users at an installation, you can modify AFIL to include an indirect command file reference to BFIL.CMD as a separate line in the option sequence.

The contents of AFIL.CMD would then be:

```
     IMG1,IMG1=IN1
     IN2,IN3
     /
     PRI=100
     COMMON=JRNAL:RO
     @BFIL
     //
```

To build these files, you type:

```
     >TKB
     TKB> @AFIL
```

Suppose the contents of BFIL.CMD are:

```
     STACK=100
     UNITS=5!ASG=DT1:5
```

The terminal equivalent of building these files is:

```
>TKB
TKB>IMG1,IMG1=IN1
TKB>IN2,IN3
TKB>/
ENTER OPTIONS:
TKB>PRI=100
TKB>COMMON=JRNAL:RO
TKB>STACK=100
TKB>UNITS=5!ASG=DT1:5
TKB>//
```

The indirect command file reference must appear on a separate line. For example, if you modify AFIL.CMD by adding the @BFIL reference on the same line as the COMMON=JRNAL:RO option, the substitution would not take place and the Task Builder would report an error.

1.6 COMMENTS IN LINES

You can include comments at any point in the command sequence, except in lines that contain file specifications. You begin a comment with a semicolon (;) and terminate it with a carriage return. All text between these delimiters is a comment.

For example, in the indirect command file, AFIL.CMD, described in Section 1.5, you can add comments to provide more information about the purpose and the status of the task.

```
;
; TASK 33A
;
; DATA FROM GROUP E-46 WEEKLY
;
IMG1,IMG1=
;
; PROCESSING ROUTINES
;
IN1
;
; STATISTICAL TABLES
;
IN2
;
; ADDITIONAL CONTROLS
;
IN3
/
PRI=100
;
COMMON=JRNAL:RO ; RATE TABLES
;
; TASK STILL IN DEVELOPMENT
;
//
```

1.7 FILE SPECIFICATIONS

The Task Builder adheres to the standard RSX-11M/M-PLUS conventions for file specifications. For any file, you can specify the device, the User File Directory (UFD), the file name, the file type, the file version number, and any number of switches.

The file specification has the form:

```
device:[group,member]filename.type;version/sw1/sw2.../swn
```

When you specify files by name only the Task Builder applies the default switch settings for device, group, member, type, version. For example:

```
>TKB
TKB>IMG1,IMG1=IN1
TKB>IN2,IN3
TKB>//
```

If the current User Identification Code (UIC) of the terminal that the Task Builder is running on is [200,200], the task image file specification of the example is assumed to be:

```
SY0:[200,200]IMG1.TSK;1
```

That is, the Task Builder creates the task image file on the system device (SY0:) under UFD [200,200]. The default type for a task image file is .TSK and if the name IMG1.TSK is new, the version number is 1. The default settings for all the task image switches also apply. Switch defaults are described in detail in Chapter 6.

For example:

```
>TKB
TKB>[20,23]IMG1/CP/DA,IMG1/CR=IN1
TKB>IN2;3,IN3
TKB>//
```

This sequence of commands instructs the Task Builder to create a task image file IMG1.TSK;1 and a memory allocation (map) file MP1.MAP;1 (actually, it produces IMG1.TSK and IMG1.MAP with versions one higher than the current versions) under UFD [20,23] on the device SY:. The task image is checkpointable and contains the standard debugging aid (ODT). The Task Builder outputs the map to the line printer with a global cross-reference listing appended to it. The Task Builder builds the task from the latest versions of IN1.OBJ, IN3.OBJ, and the specific version of IN2.OBJ. The input files are all found on the system device.

For some files, a device specification is sufficient. In the example above, the map file is fully specified by the device LP:. The map listing is produced on the line printer, but is not retained as a file.

This example also used switches /CP, /CR, and /DA. The code, syntax, and meaning for each switch are given in Chapter 6.

1.8 SUMMARY OF SYNTAX RULES

The syntax rules for issuing commands to the Task Builder are as follows:

1. A task-build command can take any one of four forms. The first form is a single line:

   ```
   >TKB task-command-line
   ```

   The second form has additional lines for input file names:

   ```
   >TKB
   TKB>task-command-line
   TKB>input-line
           .
           .
           .
   TKB>terminating-symbol
   ```

   The third form allows you to specify options:

   ```
   >TKB
   TKB>task-command-line
   TKB>/
   ENTER OPTIONS:
   TKB>option-line
           .
           .
           .
   TKB>terminating-symbol
   ```

The fourth form has both input lines and option lines:

```
>TKB
TKB>task-command-line
TKB>input-line
        .
        .
        .
TKB>/
ENTER OPTIONS:
TKB>option-line
        .
        .
        .
TKB>terminating-symbol
```

The terminating symbol can be:

/   if you intend to build more than one task

// if you want the Task Builder to return control to MCR

2. A task command line has one of the three forms:

```
output-file-list=input-file,...

=input-file,...

@indirect-command-file
```

The third form is an indirect command file specification as described in Section 1.5.

3. An ouput file list has one of the three forms:

```
task-image-file,map-file,symbol-definition-file

task-image-file,map-file

task-image-file
```

The task-image-file is the file specification for the task image file; map-file is the file specification for the memory allocation (map) file; and symbol-definition-file is the file specification for the symbol definition file. Any of the specfications can be omitted, so that, for example, the following form is permitted:

```
task-image-file,,symbol-definition-file
```

4. An input line has one of two forms:

```
input-file,...

@indirect-command-file
```

Both input-file and indirect-command-file are file specifications.

5. An option line has one of two forms:

   option!...

   @indirect-command-file

   The indirect-command-file is a file specification.

6. An option has the form:

   keyword=argument-list,...

   The argument-list is:

   arg:...

   The syntax for each option is given in Chapter 6.

7. A file specification conforms to standard RSX-1M/M-PLUS comventions. It has the form:

   device:[group,member]filename.type;version/sw1/sw2.../swn

   device: The name of the physical device on which the volume containing the desired file is mounted. The name consists of two ASCII characters followed by an optional 1- or 2-digit octal unit number and a colon; for example, LP: or DT1:.

   group The group number, in the range of 1 through 377(8).

   member The member number, in the range 1 through 377(8).

   filename The name of the desired file. The file name can contain up to 9 alphanumeric characters.

   type The 3-character file type idectification. Files having the same name but a different function are distinguished from one another by the file type; for example, CALC.TSK and CALC.OBJ.

   version The version number, in octal, of the file. Various versions of the same file are distinguished from one another by this number; for example, CALC.OBJ;1 and CALC.OBJ;2.

   All components of a file specification are optional. The combination of the group number and the member number is the User File Directory (UFD) that contains the file name.

CHAPTER 2

TASK BUILDER FUNCTIONS

The process of building a task involves three distinct Task Builder functions. First, the Task Builder is a linker. It collects and links the relocatable object modules that you specify to it into a single task image and resolves references to global symbols across the module boundaries.

Second, the Task Builder assigns addresses to the task image. On mapped systems, the Task Builder assigns addresses for a task beginning at zero. The Executive then relocates the addresses at runtime. On unmapped systems, the Task Builder assigns addresses for a task beginning at the base address of the partition in which the task is to run. The addresses of tasks that run on unmapped systems are not relocated at runtime.

NOTE

Unless otherwise indicated, references to tasks that run on mapped systems assume that the tasks are nonprivileged and residing within system-controlled partitions.

Third, the Task Builder builds data structures into the task image that are required by the INSTALL processor to install the task and by the Executive to run it.

This chapter describes the three Task Builder functions in detail.

2.1 LINKING OBJECT MODULES

When the language translators convert symbolic source code within a module to object code, they assign provisional 16-bit addresses to the code. A single assembly or compilation produces a single object module. In their simplest form, each module begins at 0 and extends upward to the highest address in the module. Three object modules produced at separate times might have the address limits shown in Figure 2-1.

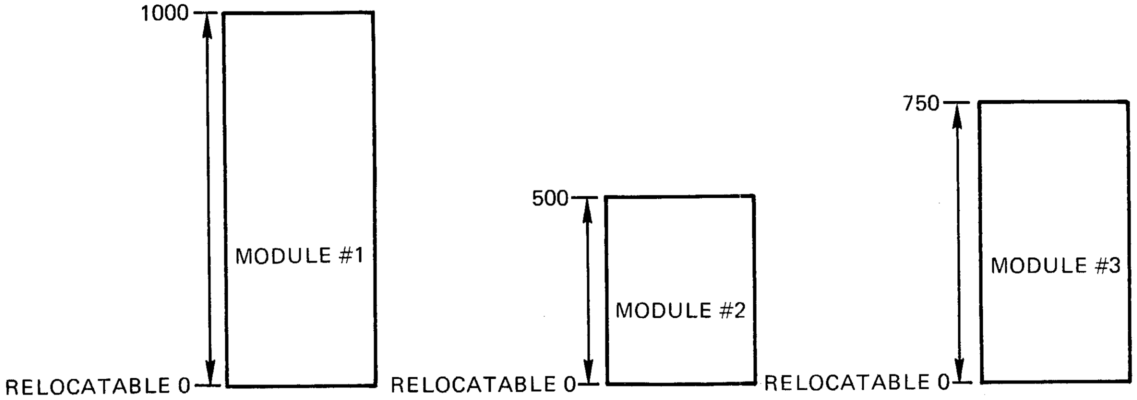


Figure 2-1 Relocatable Object Modules

If these modules represent the separate modules of a single program, the Task Builder links them together and modifies the provisional addresses to one of the following:

- A single sequence of addresses beginning at 0 and extending upward to the sum of all of the addresses of each module (mapped system)
- A single sequence of addresses beginning at a base address assigned at task build time and extending upward to the sum of all the addresses of each module (unmapped system)

For example, Figure 2-2A shows the three modules linked for a mapped system, and Figure 2-2B shows the modules linked for an unmapped system.

2.1.1 Allocating Program Sections

The language translators process source code and the Task Builder links object modules within the context of program sections. A program section is a block of code or data that consists of three elements:

- a name
- a set of attributes
- a length

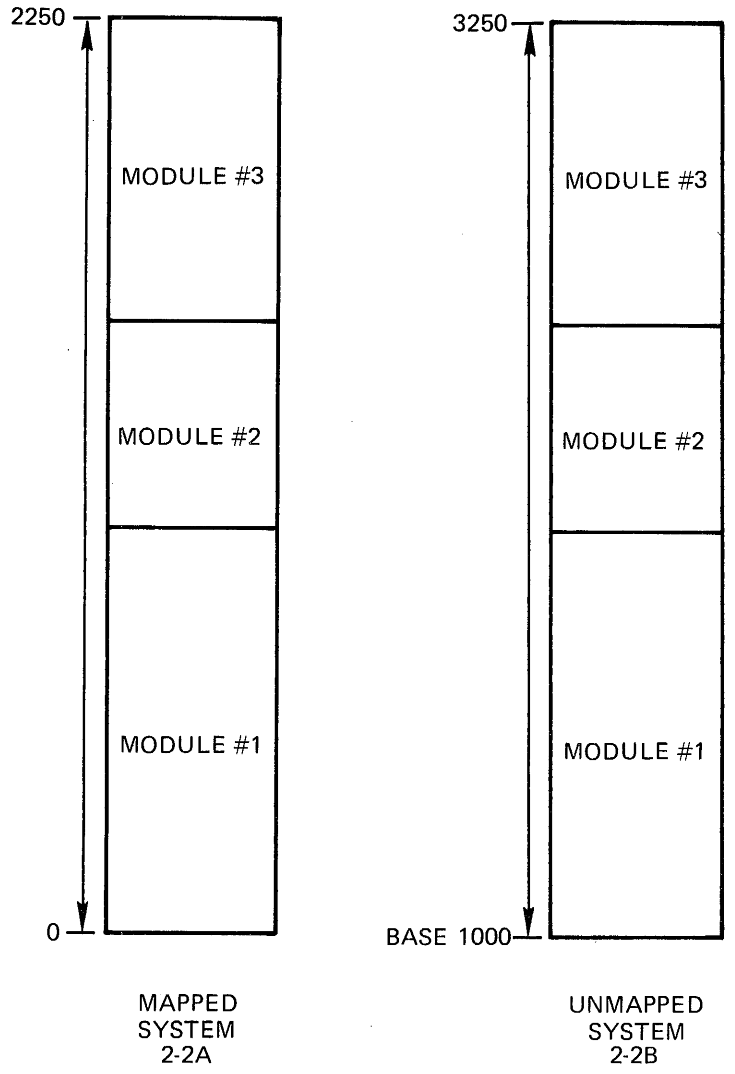


Figure 2-2 Modules Linked for Mapped and Unmapped Systems

Program sections are important because they are the basic unit used by the Task Builder to determine the placement of code and data in a task image. The language translators maintain a separate location counter for each program section in a program. The name of each program section, its attributes, and its length are conveyed to the Task Builder through the object module.

You can create as many program sections within a module as you wish by explicitly declaring them (with the COMMON statement in FORTRAN or the .PSECT directive in MACRO-11, for example) or you can leave the creation of program sections to the language translator. If you do not explicitly create a program section in your source code, the language translator you are working with will create a "blank" program section within each module translated. This program section will appear on your listings and maps as . BLK.. For more information on explicitly declared program sections, see your language reference manual.

A program section's name is the name by which the language translator and Task Builder reference it. When processing files, both the language translator and the Task Builder create internal tables that contain program section names, attributes and lengths.

Program section attributes define a program section's contents, its placement in a task image, and, in some cases, the allowed mode of access (read/write or read-only).

A program section's length determines how much address space the Task Builder must reserve for it.

When a program consists of more than one module, it is not unusual for program sections of the same name to exist in more than one of the modules. Therefore, as the Task Builder scans the object modules, it collects scattered occurrences of program sections of the same name and combines them into a single area of your task image file. The attributes listed in Table 2-1 control the way the Task Builder collects and places each program section in the task image.

Table 2-1
Program Section Attributes

Attribute	Value	Meaning
access-code	RW	Read/write: data can be read from, and written into, the program section
	RO	Read-only: data can be read from, but cannot be written into, the program section
type-code [1]	D	Data: the program section contains data
	I	Instruction: the program section contains either instructions, or data and instructions
scope-code	GBL	Global: the program section name is recognized across overlay segment boundaries, the Task Builder allocates storage for the program section from references outside the defining overlay segment.
	LCL	Local: the program section name is recognized only within the defining overlay segment; the Task Builder allocates storage for the program section from references within the defining overlay segment only

[1] Do not confuse these codes with the I and D space hardware on PDP-11 systems.

(continued on next page)

Table 2-1 (Cont.)
Program Section Attributes

Attribute	Value	Meaning
allocation-code	CON	Concatenate: all references to a given program section name are concatenated; the total allocation is the sum of the individual allocations
	OVR	Overlay: all references to a given program section name overlay each other; the total allocation is the length of the longest individual allocation
relocation-code	REL	Relocatable: the base address of the program section is relocated relative to the base address of the task
	ABS	Absolute: the base address of the program section is not relocated; it is always 0
memory-code [2]	HIGH	High: the program section is to be loaded into high-speed memory
	LOW	Low: the program section is to be loaded into low-speed memory

[2] Not used by the Task Builder.

The type-code and scope-code are meaningful only when you define an overlay structure for a task. These codes are described in later chapters within the context of the descriptions of overlays. The Task Builder does not use the memory-code.

The Task Builder uses a program section's access-code and allocation-code to determine its placement and size in a task image. It divides address space into read/write and read-only areas, and places the program sections in the appropriate area according to access-code.

The Task Builder uses a program section's allocation-code to determine its starting address and length. If a program section's allocation-code indicates that the Task Builder is to overlay it, the Task Builder places each allocation to the program section from each module at the same address within the task image. The Task Builder determines the total size of the program section from the length of the longest allocation to it.

If a program section's allocation-code indicates that the Task Builder is to concatenate it, the Task Builder places the allocation from the modules one after the other in the task image, and determines the total allocation from the sum of the lengths of each allocation.

The Task Builder always allocates address space for a program section beginning on a word boundary. If the program section has the D (data) and CON (concatenate) attributes, the Task Builder appends to the last byte of the previous allocation all storage contributed by subsequent modules. It does this regardless of whether that byte is on a word or nonword boundary. For a program section with the I (instruction) and CON attributes, however, the Task Builder allocates address space contributed by subsequent modules beginning with the nearest following word boundary.

For example, suppose three modules, IN1, IN2, and IN3, are to be task-built. Table 2-2 lists these modules with the program sections each contains and their access codes and allocation codes.

Table 2-2
Program Sections for Modules IN1, IN2, and IN3

File Name	Program Section Name	Access Code	Allocation Code	Size (octal)
IN1	B A C	RW RW RO	CON OVR CON	100 300 150
IN2	A B	RW RW	OVR CON	250 120
IN3	C	RO	CON	50

In this example, the program section named B, with the attribute CON (concatenate), occurs twice. Thus, the total allocation for B is the sum of the lengths of each occurence; that is, 100 + 120 = 220. The program section named A also occurs twice. However, it has the OVR (overlay) attribute so its total allocation is the largest of the two sizes, or 300. Table 2-3 lists the individual program section allocations.

Table 2-3
Individual Program Section Allocations

Program Section Name	Total Allocation
B A C	220 300 220

The Task Builder then groups the program sections according to their access codes, and alphabetizes each group as shown in Figure 2-3.

NOTE

The example shown in Figure 2-3 represents the Task Builder's default allocation of program sections. For information on altering this default, see the description of the SQ switch in Chapter 6.

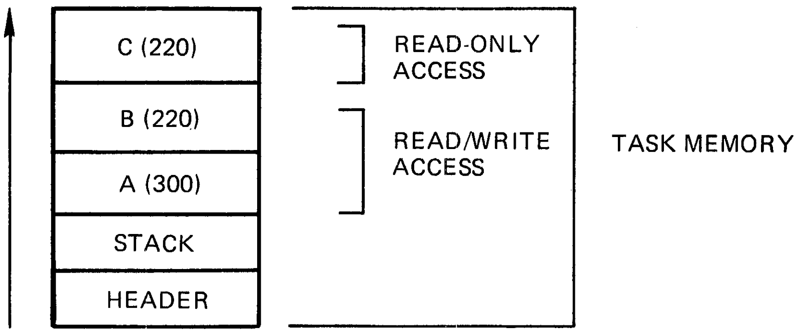


Figure 2-3 Allocation of Task Memory

2.1.2 Resolving Global Symbols

The Task Builder resolves references to global symbols across module boundaries and any references (explicit or implicit) to the system library. When the language translators process a text file, they assume that references to global symbols within the file are defined in other, separately assembled or compiled modules. As the Task Builder links the relocatable object modules, it creates an internal table of the global symbols it encounters within each module. If, after the Task Builder examines and links all the object modules, references remain to symbols that have not been defined, the Task Builder assumes that it will find the definition for the symbols within the default system object module library (LB:[1,1]SYSLIB.OLB). If undefined symbols still remain after SYSLIB is examined, the Task Builder flags the symbols as undefined. If you have not specified an output map in your Task Builder command sequence, the Task Builder reports the names of the undefined symbols to you on your terminal. If you have specified an output map, the Task Builder outputs to your terminal only the fact that the task contains undefined symbols. The names of the symbols appear on your map listing.

When creating the task image file, the Task Builder resolves global references as shown in the following example. Table 2-4 lists the three files IN1, IN2, and IN3, showing the program sections within each file, the global symbol definitions within each program section, and the references to global symbols in each program section.

Table 2-4
Resolution of Global Symbols for IN1, IN2, and IN3

File Name	Program Section Name	Global Definition	Global Reference
IN1	B A C	B1 B2	A L1 C1 XXX
IN2	A B	A B1	B2
IN3	C		B1

In processing the first file, IN1, the Task Builder finds definitions for B1 and B2 and references to A, L1, C1, and XXX. Because no definition exists for these references, the Task Builder defers the resolution of these global symbols. In processing the next file, IN2, the Task Builder finds a definition for A, which resolves the previous reference, and a reference to B2, which can be immediately resolved.

When all the object files have been processed, the Task Builder has three unresolved global references -- C1, L1, and XXX. Assume that a search of the system library LB:[1,1]SYSLIB.OLB resolves L1 and XXX and the Task Builder includes the defining modules in the task's image. Assume also that the Task Builder cannot resolve the global symbol C1. The Task Builder lists it as an undefined global symbol.

The relocatable global symbol B1 is defined twice. The Task Builder lists it as a multiply-defined global symbol. The Task Builder uses the first definition of a multiply-defined symbol.

Finally, an absolute global symbol (for example, symbol=100) can be defined more than once without being listed as multiply defined as long as each occurrence of the symbol has the same value.

2.2 ASSIGNING ADDRESSES

The primary addressing mechanism of the PDP-11 is the 16-bit computer word. The maximum physical address space that the PDP-11 can reference at any one time is a function of the length of this word. The highest number that can be represented in 16 bits is 177777(8) or 65,535(10). Because the PDP-11 is a byte-addressable machine, the 16-bit word length allows it to address up to 65,535 bytes (32K words) of physical address space at any one time. The amount of address space that a machine can reference at any one time is called virtual address space.

2.2.1 Unmapped Systems

In an unmapped system, the machine's virtual address space and its physical address space coincide exactly, as shown in Figure 2-4.

In an unmapped system, the machine's address space is limited to 32K words. All of the machine's physical memory and all of its device registers are accessible to all tasks running on the system. The top 4K words of address space are reserved for the UNIBUS addresses that correspond to the peripheral device registers (the I/O page), and a segment of low memory is occupied by the Executive. Therefore, in an unmapped system, the largest task size is 32K words minus the I/O page and the size of the Executive. Figure 2-5 shows the memory layout for an unmapped system.

Unmapped systems contain only user-controlled partitions. When the Task Builder links the relocatable object modules of a task that is to run on an unmapped system, it requires that you specify the partition in which the task is to run, the partitions base address and length. The Task Builder will set the base address of the task to the base address of the partition. This means that the task's location in physical memory is bound to the partition and does not change. Because all of physical memory in an unmapped system is directly addressable, and the task's location within memory does not change, the addresses that the Task Builder assigns coincide exactly with the physical addresses of the machine and, therefore, do not need to be relocated at runtime.

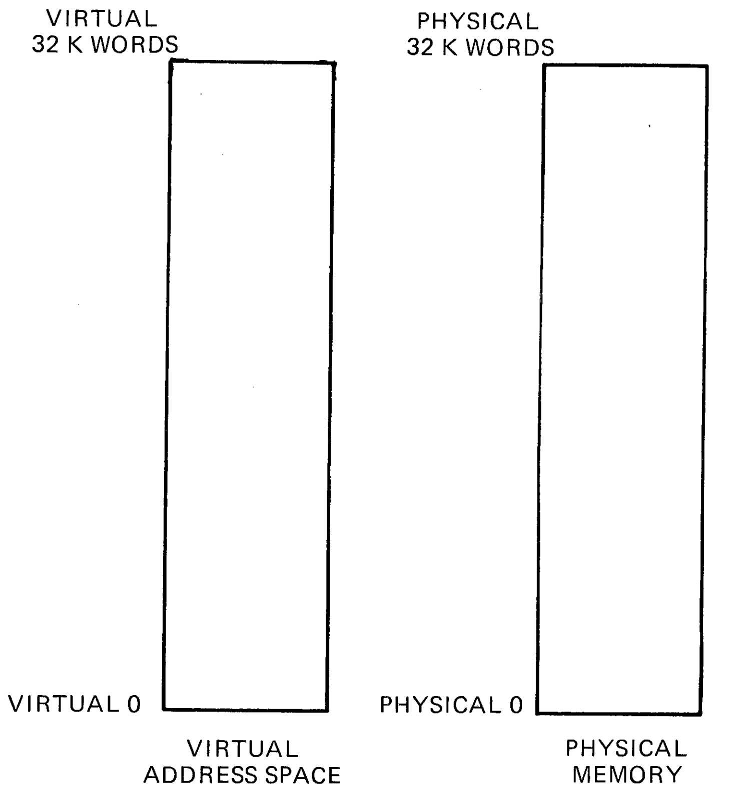


Figure 2-4 Unmapped Memory

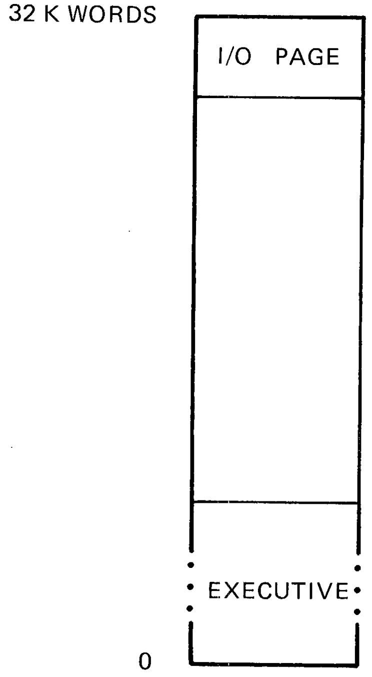


Figure 2-5 Layout for Unmapped System

2.2.2 Mapped Systems

In a mapped system, the relationship between virtual address space and physical address space is different from that of an unmapped system. The primary addressing mechanism for a mapped system is still the 16-bit word, and virtual address space is still 32K words. However, a mapped system has a much greater physical memory capacity and, therefore, physical memory and virtual address space do not coincide.

To address all of physical memory in a mapped system, a machine must have an effective word length of 18 or 22 bits, depending on the model of the machine. When the Task Builder links the relocatable object modules of a task that is to run on a mapped system, it assigns 16-bit address to the task image. The memory management unit's function (under control of the Executive) is to convert the task's 16-bit addresses to effective 18- or 22-bit physical addresses. The mechanical job of task relocation is performed by the Executive and the memory management unit at task runtime. Figure 2-6 illustrates the relationship between physical memory and virtual address space in a mapped system.

The memory management unit divides a machine's 32K words of virtual address space into eight 4K word segments or pages. Each page has two registers associated with it:

- A 16-bit Page Description Register (PDR) which contains control and access information about the page with which it is associated
- A 16-bit Page Address Register (PAR) which is an address relocation register

The PDRs and PARs are always used as a pair. Each pair is called an Active Page Register (APR). Figure 2-7 shows how the memory management unit divides the 32K words of virtual address space.

NOTES

1. A detailed description of the Memory Management unit is beyond the scope of this manual. For a complete description of your machine's memory management unit, refer to the Processor Handbook for your machine.

2. Mapped machines have up to three modes of operation: Kernel, Supervisor, and User (11/34 machines do not have supervisor mode). The information in this chapter is relevant to User mode only.

The Executive allocates only as many APRs as are necessary to map a given task into physical memory. Therefore, a 4K word task requires one APR; a 6K word task requires two. Figure 2-8 illustrates this mapping.

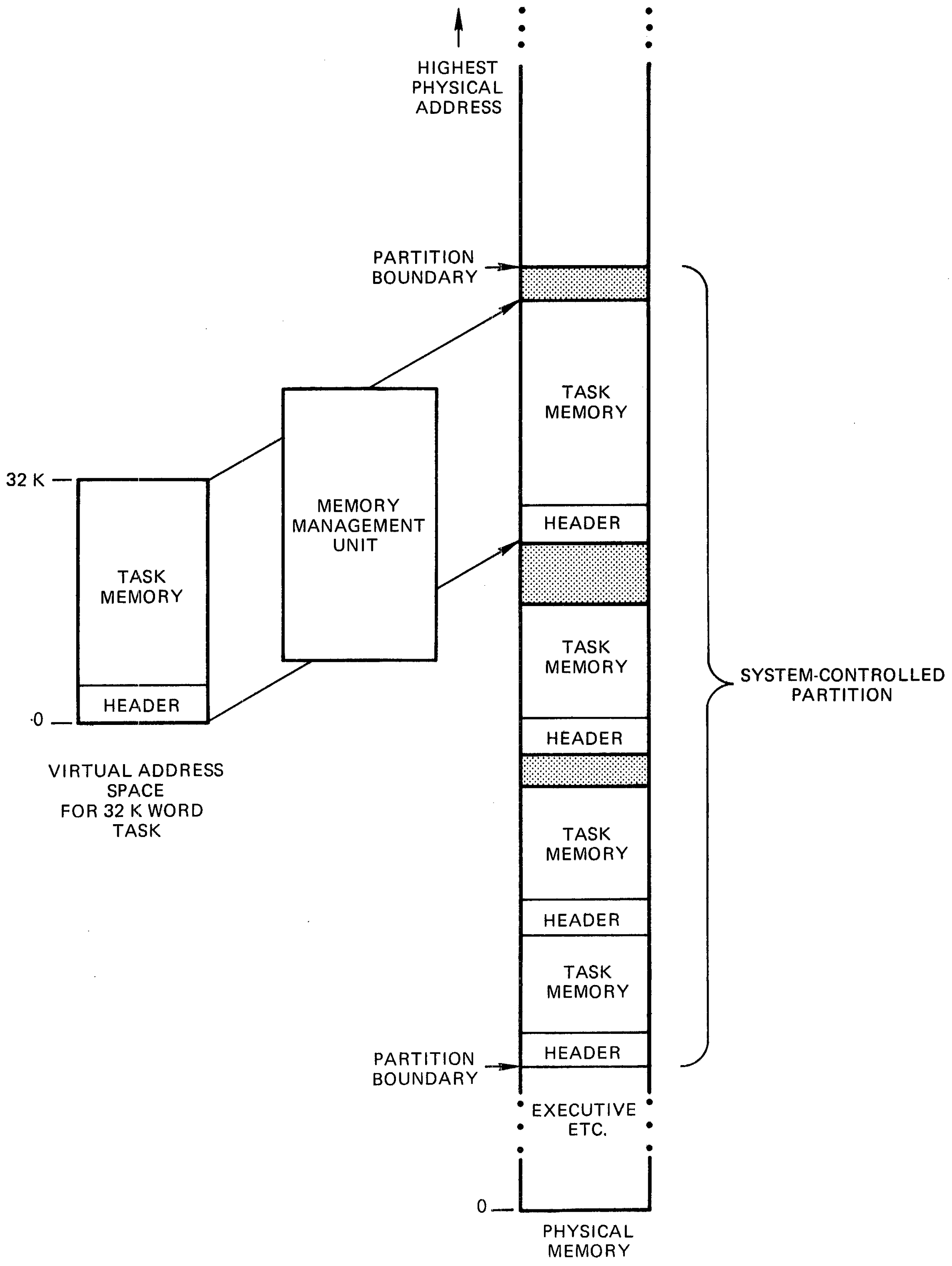


Figure 2-6 Task Relocation in a Mapped System

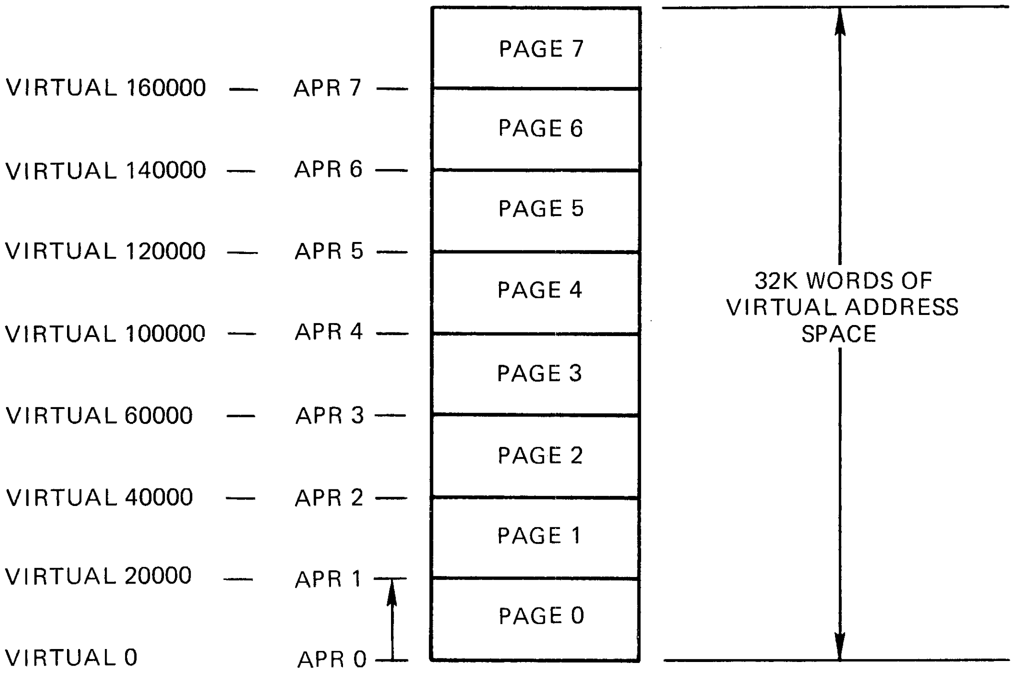


Figure 2-7 Memory Management Unit's Division of Virtual Address Space

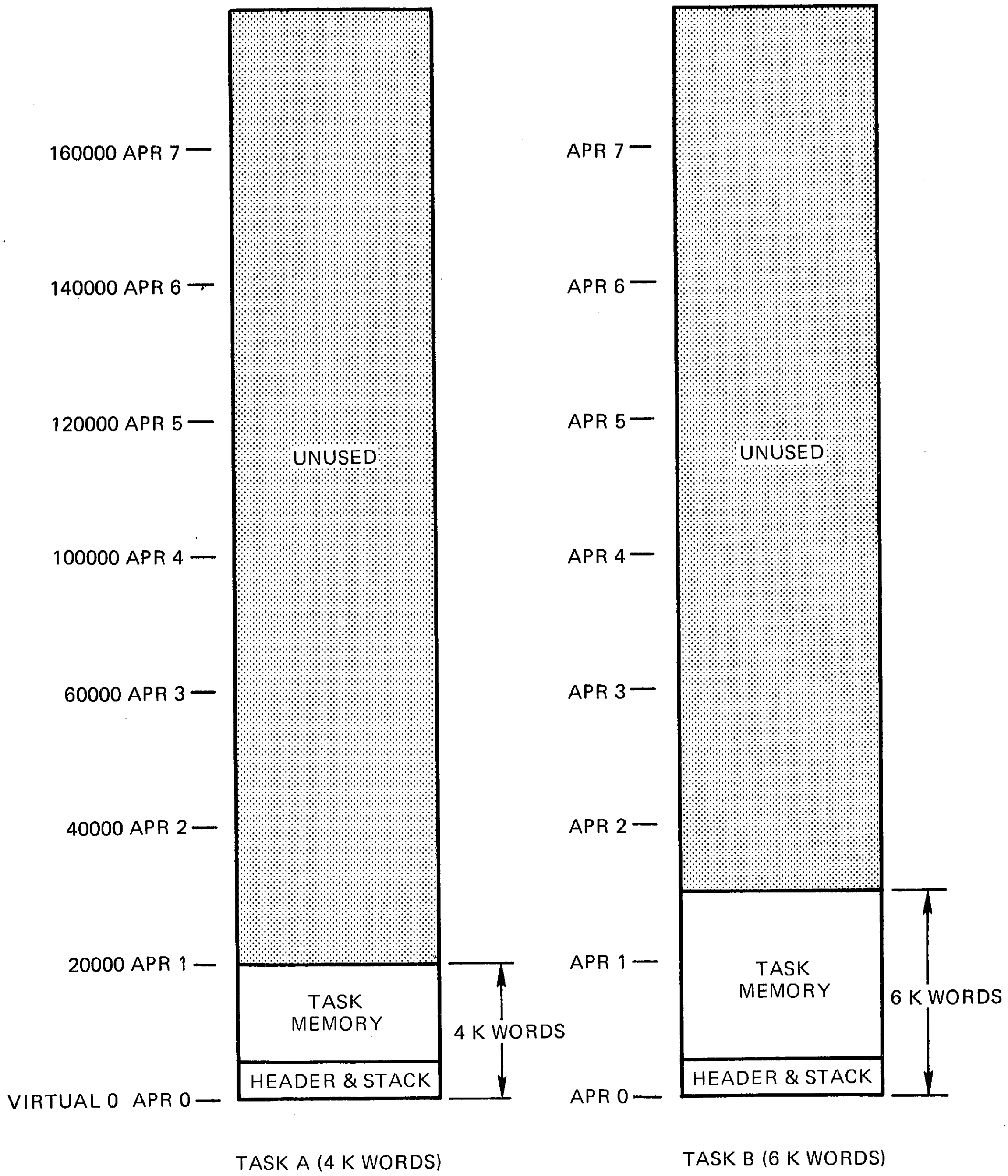


Figure 2-8 Mapping for 4K Word and 6K Word Tasks

Finally, the layout of the virtual address space for a task that is to run in a mapped system is different in most cases from that of a task that is run in an unmapped system. Unless a task is privileged, the I/O page and the Executive are not normally part of a task's virtual address space and, unlike an unmapped system, a task is inhibited by the system from accessing any portion of physical memory that it does not specifically own. Because the I/O page and the Executive are not part of a task's virtual address space, a task can be a full 32K words long on a mapped system.

2.3 BUILDING SYSTEM DATA STRUCTURES

It is the Task Builder's responsibility to build the data structures required by other system programs, and to incorporate them into the task image. The Executive (which is the system task responsible for the allocation of system resources) must have access to the data for all tasks on the system. It must know, for example, a task's size and priority, and it must have information about the way each task expects to use the system. It is the Task Builder's responsibility to allocate space in the task image for the data structures required by the Executive. The Task Builder initializes some of these structures, while the INSTALL processor initializes others when you install the task.

The disk image file created by the Task Builder contains the linked task and all of the information required by the system programs to install and run it. In its simplest form, the disk image file consists of three physically contiguous parts:

- The label block group
- The task header
- The task memory image

Figure 2-9 illustrates the basic structure of this file.

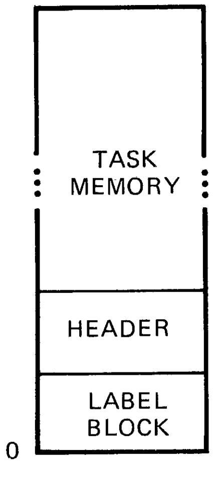


Figure 2-9 Disk Image

NOTE

> Non-runnable images such as resident shared regions do not have a header. Resident shared regions are described in Chapter 3.

The label block group contains data produced by the Task Builder and used by the INSTALL processor. It contains information about the task such as the task's name, the partition in which it runs, its size, priority, and the logical units assigned to it. When you install the task, the INSTALL processor uses this information to create a Task Control Block entry for the task in the System Task Directory (STD file) and to initialize the task's header information.

The task's header contains information that the Executive uses when it runs the task. The header also provides a storage area for saving the task's essential data when the task is checkpointed. The Task Builder creates and partially initializes the header; the INSTALL processor initializes the rest of the header.

The task memory contains the linked modules of the program and therefore the code and data. It also contains the task's stack. The stack is an area of task memory that a task can use for temporary storage and subroutine linkage. It can be referenced through general register 6, the stack pointer (SP). The label block group, the task's header and the task memory are described in detail in Appendix C.

The task's memory image is the part of your task that the system reads into physical memory at runtime. The label block group is not required in physical memory. Therefore, in its simplest form, the task's memory image consists of only two parts: the task header and task memory. Figure 2-10 shows the memory image.

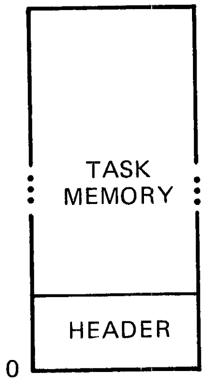


Figure 2-10 Memory Image

2.4 TASK RELOCATION ON MAPPED SYSTEMS

As mentioned earlier, tasks that run on mapped systems must be relocated at runtime. When you build a task that is to run on a mapped system, the Task Builder creates and places in the header of the task one or more 9-word data structures called window blocks. When you install a task the INSTALL processor initializes the window block(s). Once initialized, a window block describes a range of continuous virtual addresses called a window.

A window can be as small as 32 words or as large as 32K words. When a task consists of one continuous range of addresses (a single region task) only one window block is required to describe the entire task from the beginning of its header to the highest virtual address in the task. When a task consists of two or more regions (such as a task that references a shared region as described in Chapter 3), each region must have at least one window block associated with it that describes all or a portion of the region.

When the Executive maps a task into physical memory, it extracts the information it requires to set up the APRs of the memory management unit from the task's window block.

Regardless of the number of regions associated with a task, the region that contains the task's header is always described by window 0. Furthermore, this region is referred to as the task region and is identified as region 0. Figure 2-11 illustrates window block 0.

When you run your task, the Executive determines where in physical memory the task is to reside. The Executive then loads the Page Address Register portion of the APRs with a relocation constant that, when combined with the addresses of the task, yields the 18- or 22-bit physical address range of the task.

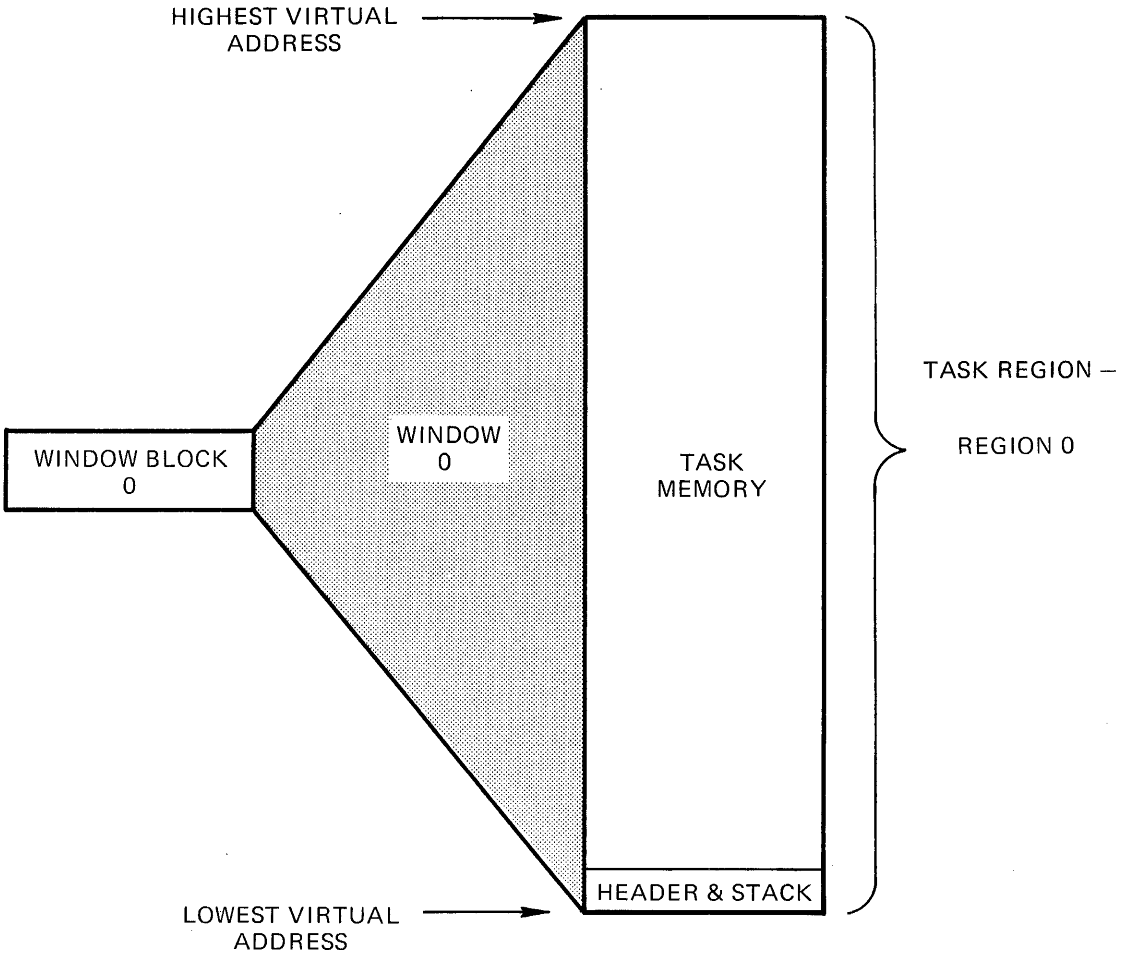


Figure 2-11 Window Block 0

CHAPTER 3

TYPICAL TASK BUILDER FACILITIES

The Task Builder provides you with many facilities for tailoring your tasks to meet your specific requirements. This chapter describes some of these facilities and their applications.

This chapter contains eight working examples. The discussion of the examples assume that you are familiar with the programming concepts described in the RSX-11M/M-PLUS Guide to Program Development and with the first two chapters of this manual.

3.1 SHARED REGIONS

A shared region is a block of data or code that resides in memory and can be used by any number of tasks. Shared regions are useful because they make more efficient use of physical memory:

- By providing a way in which two or more tasks can share their data. This is called a resident common.

- By providing a way in which a single copy of commonly used subroutines can be shared by several tasks. This is called a resident library.

The term "resident" is used to denote a shared region that is built and installed into the system separately from the task that links to it.

Figure 3-1 shows a typical resident common. Task A stores some results in resident common S, and Task B retrieves the data from the common at a later time.

Figure 3-2 shows a typical resident library. In this case, common reentrant subroutines are not included in each task image; instead, a single copy is shared by all tasks.

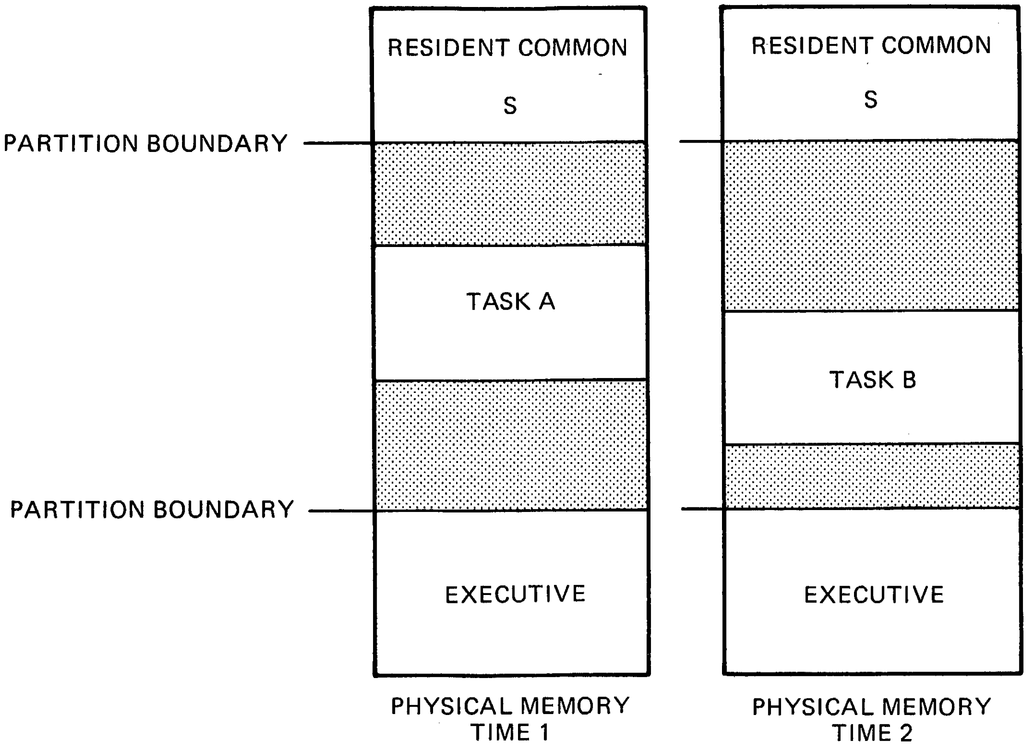


Figure 3-1 Typical Resident Common

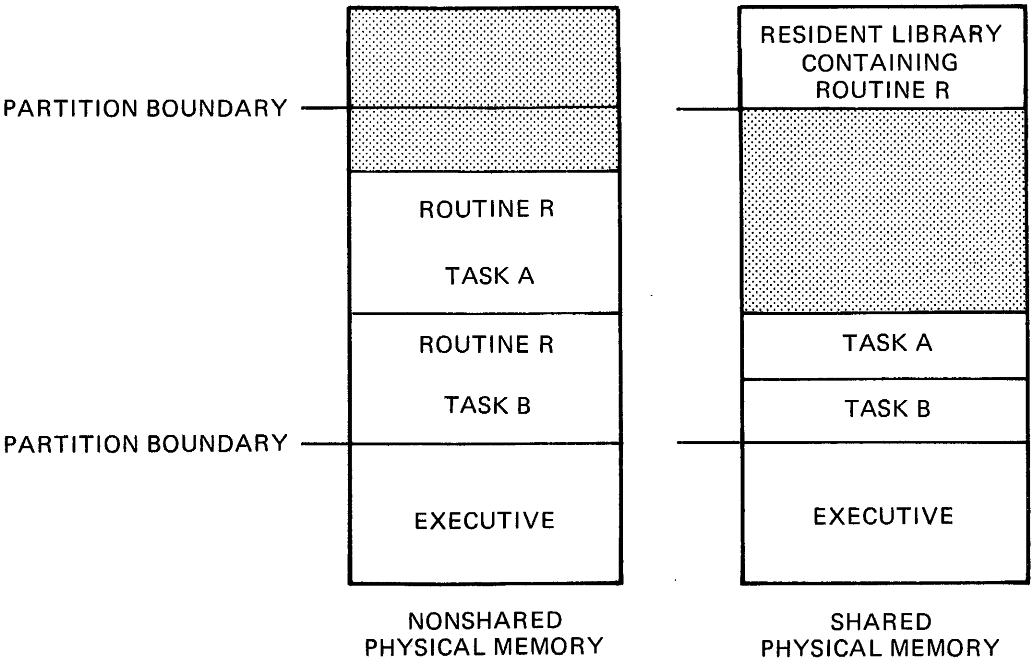


Figure 3-2 Typical Resident Library

When you build a shared region, you must specify in the Task Builder command sequence an output image file name for it. But, because a shared region is not an executable unit, it is not a task. It does not require a header or a stack area. Therefore, when you build a shared region, you always attach the negated header switch (/-HD) to the image file specification. This switch tells the Task Builder to suppress the header within the image. To suppress the stack area, in the Task Builder command sequence during option input, you specify STACK=0. (Refer to Chapter 6 for a complete description of the HD switch and the STACK option.)

In an RSX-11M system, a shared region must reside in its own partition. Therefore, when you generate your system, you must consider the physical memory requirements of any shared regions that you expect to reside within your system. If you do not consider these requirements at system generation time, later, when you build a shared region, you will have to go back and create a common partition for the region.

In an RSX-11M-PLUS system, shared regions do not have to reside within partitions of their own; you can install a shared region in any partition large enough to hold it. In fact, the partition for which the shared region was built does not have to exist in the system at the time the shared region is installed. If you attempt to install a shared region in a partition that does not exist, the INSTALL processor will install it in partition GEN and print the following message on your terminal:

```
INS--PARTITION parname NOT IN SYSTEM DEFAULTING TO GEN
```

When you build a shared region, you must specify the partition in which the region is to reside. You specify the partition name in the Task Builder command sequence during option input. (Refer to Chapter 6 for a description of the PAR option.)

3.1.1 The Symbol Definition File

When you build a shared region, you must specify a symbol definition (.STB) file in the Task Builder command sequence. This file contains linkage information about the region. Later, when you build a task that links to the region, the Task Builder uses this .STB file to resolve calls from within the referencing task to locations within the region.

The contents of an .STB file for a shared region depend on whether the shared region is position independent or absolute. The effects of declaring a shared region position independent or absolute and the resulting contents of the .STB file are described in the following sections.

3.1.2 Position-Independent Shared Regions

A position-independent shared region can be placed anywhere in a referencing task's virtual address space when the system on which the task runs has memory management hardware.

For example, in Figure 3-3, two tasks refer to the shared region S -- task A and task B. The shared region S is 4K words long and therefore requires that much space in the virtual address space of both tasks. Task A is 6K words long and requires two APRs (APR 0 and APR 1) to map its task region. The first APR available to map the shared region is APR 2. Therefore, APR 2 can be specified when task A is built.

Task B is 16.5K words long. It requires five APRs to map its task region. The first APR available to map the shared region S in task B is APR 5. Therefore, APR 5 can be specified when task B is built.

If you do not specify which APR the Task Builder is to use to map a position-independent shared region, the Task Builder will automatically select the highest set of APRs available in the referencing task's virtual address space. In Figure 3-3, for example, if APR 2 in task A and APR 5 in task B had not been selected at task-build time, the Task Builder would have automatically selected APR 7 in both cases.

Declaring a region to be position independent causes the Task Builder to include in the .STB file for the region an entry for each program section in the region. Each entry declares the program section's name, attributes, and length. In addition, the Task Builder includes in the .STB file every symbol in the shared region and its value relative to the beginning of the region.

You specify that a shared region is position independent when you build it by attaching the PI switch to the image file specification for the region. (Refer to Chapter 6 for a description of the PI switch.) You should declare a region position independent if:

- The region contains code that will execute correctly regardless of its location in the address space of the referencing task.
- The region contains data that is not address dependent.
- The region contains data that will be referenced by a FORTRAN program (such data must reside in a named common).

Because the program section name is preserved in a position-independent region, you should observe the following precautions when building and referring to the region:

- No code or data in the region should be included in the blank (. BLK.) program section.
- No code or data in a referencing task should appear in a program section of the same name as a program section in the shared region.
- The order in which memory is allocated to program sections (alphabetic or sequential) must be the same for the shared region and its referencing tasks. (Chapter 2 describes alphabetic ordering of program sections. Refer to the description of the SQ switch in Chapter 6 for an explanation of sequential ordering of program sections.)

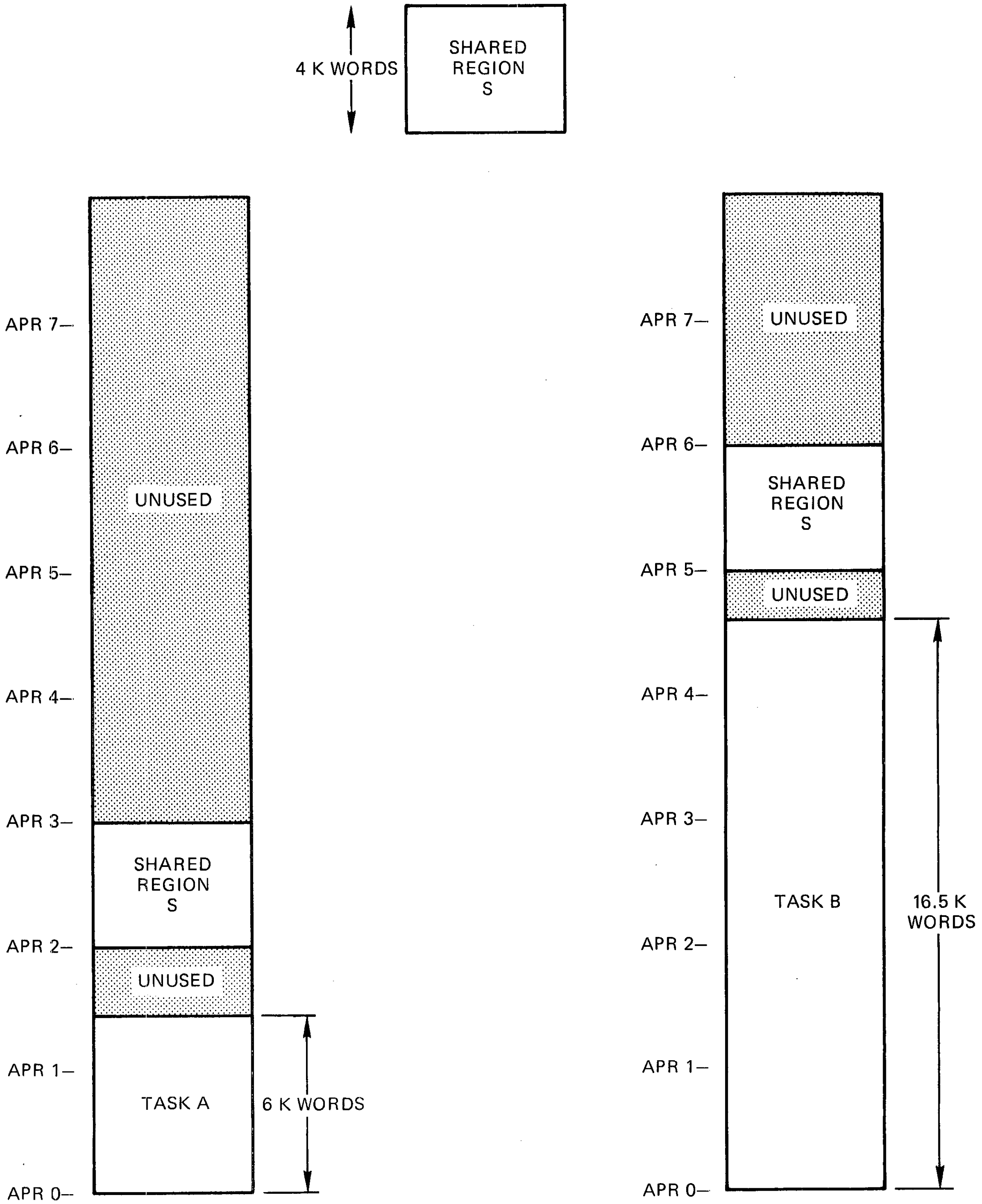


Figure 3-3 Specifying APRs for a Position-Independent Shared Region

3.1.3 Absolute Shared Regions

When a shared region is absolute, the only program section name that will appear in the .STB file for the region will be the absolute program section name (. ABS.). The Task Builder includes in the .STB file for the region each symbol in the region and its value. But, because the Task Builder does not include the program section names of an absolute shared region in its .STB file, all code or data in the region must be referred to by global symbol name.

When a shared region is absolute, you select the virtual addresses for it when you build it. Thus, an absolute shared region is fixed in the virtual address space of all tasks that refer to it.

Figure 3-4 shows three tasks (task C, task D, and task E) and a single absolute shared region, L. The absolute shared region L is 6K words long and is built to occupy virtual addresses 120000(8) to 150000(8). These addresses correspond to APR 5 and APR 6, respectively. Tasks C and D can be linked to region L because at the time they are built APR 5 and APR 6 are unused in both tasks. However, task E is 23K words long and even though it has 8K words of virtual address space available to map the shared region, APR 5 (which corresponds to virtual address 120000, the base address of the shared region) has been allocated to the task region. If shared region L were position independent, task E could be linked to it.

You specify that a shared region is absolute when you build it by simply omitting the PI switch from the task image file. You establish the virtual address for the region by specifying the base address of the region as a parameter of the PAR option.

You should build a shared region absolute if:

- The region contains code that must appear in a specific location in the address space of a referencing task.
- The region contains data that is address dependent.
- The region contains program sections of the same name as program sections in referencing tasks.

Because the Task Builder does not place program section names in the .STB file of an absolute shared region, the Task Builder places no restrictions on the way the program sections are ordered in either the absolute shared region or the tasks that reference it.

3.1.4 Linking to a Shared Region

When you build a task that links to a shared region, you must indicate to the Task Builder the name of the shared region and the type of access the task requires to it (read/write or read-only). In addition, if the shared region is position independent, you can specify which APR the Task Builder is to allocate for mapping the region into the task's virtual address space. Four options are available for this action:

- RESLIB (Resident Library)
- RESCOM (Resident Common)
- LIBR (System-Owned Resident Library)
- COMMON (System-Owned Resident Common)

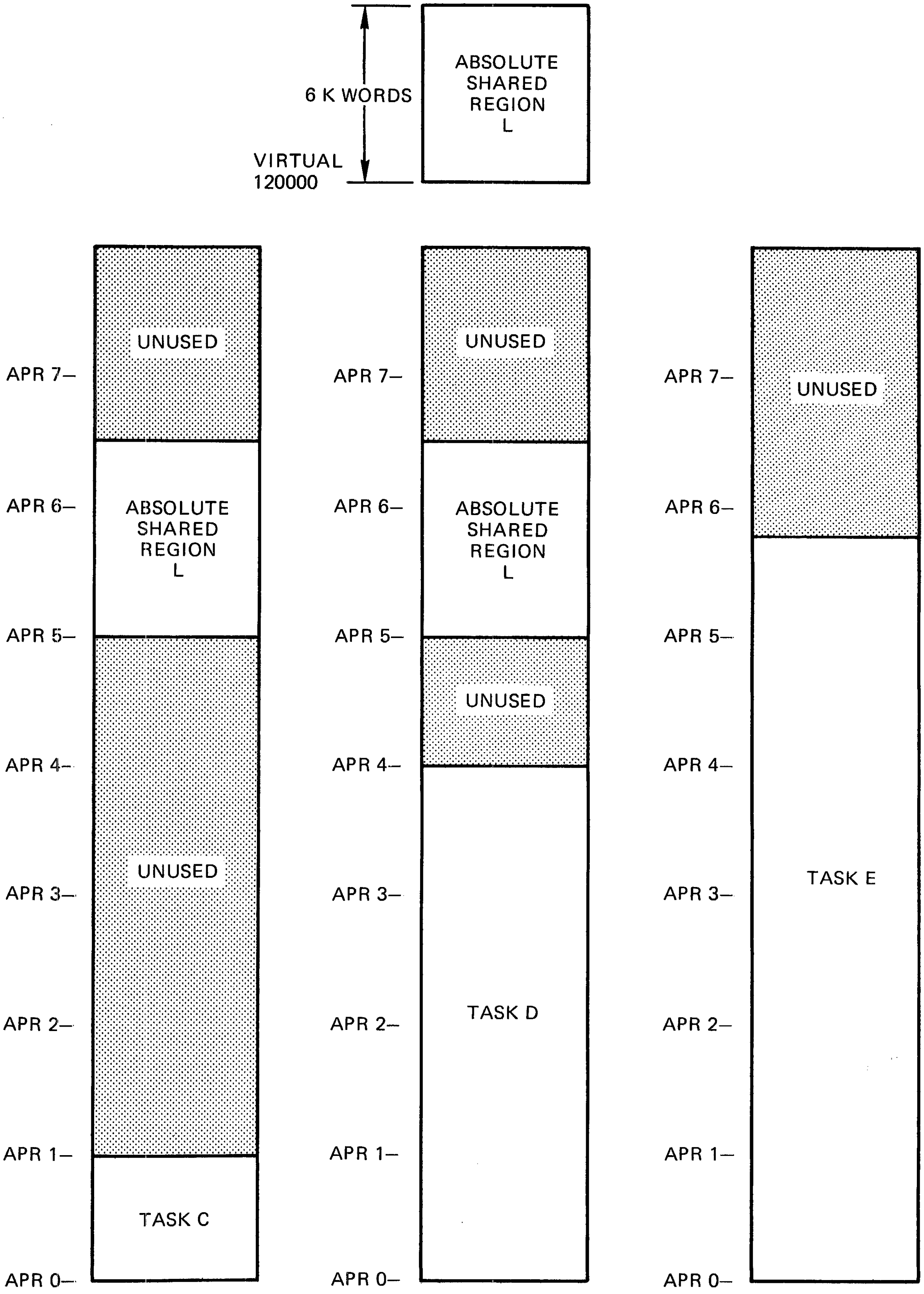


Figure 3-4 Mapping for an Absolute Shared Region

RESLIB and RESCOM accept a complete file specification as one of their arguments. Thus, you can specify a device and UFD indicating to the Task Builder the location of the region's image file and, by implication, its symbol definition file. (Refer to Chapter 1 for more information on file specifications and defaults.)

LIBR and COMMON accept a 1- to 6-character name. When you specify either of these options, the shared region's image file and symbol definition file must reside under UFD [1,1] on device LB0:.

The RESLIB and RESCOM options require that all users of the shared region know the UFD under which the shared region's image file and .STB file reside. The LIBR and COMMON options require only that the users of· the shared region know the name of the shared region. When you specify either LIBR or COMMON, by default, the Task Builder expects to find the shared region's image and .STB files on device LB: under UFD [1,1].

All four options accept two additional arguments:

- The type of access the task requires (RO or RW)
- The first APR that the Task Builder is to allocate for mapping the region into the task's virtual address space. As stated earlier, this argument is valid only when the shared region is position independent.

When you specify any of these options, the Task Builder expects to find a symbol definition file of the same name as the shared region, but with an extension of .STB, on the same device and under the same UFD as the shared region's image file.

The syntax of these options is given in Chapter 6.

When the Task Builder builds a task, it processes first any options that appear in the Task Builder command sequence. When the Task Builder processes one of the four options above, it locates the disk image of the shared region named in the option. The disk image of a shared region does not have a header, but it does have a label block that contains the allocation information about the shared region (for example, its base address, load size, the name of the partition for which it was built). The Task Builder extracts this data from the shared region's label block and places it in the LIBRARY REQUEST section of the label block for the referencing task.

The .STB file associated with the shared region is an object module file. The Task Builder processes it as an input file. If the shared region is position independent, its .STB file contains program section names, attributes and lengths. However, the program section names are flagged within the file as "library" program sections and the Task Builder does not add their allocations to the task image it is building.

If the task links to only one shared region, and if neither the shared region nor the task that links to it contain memory-resident overlays, the Task Builder will allocate two window blocks in the header of the task. (Overlays are described in Chapter 4.) When the task is installed, the INSTALL processor will initialize these window blocks as follows:

- Window block 0 will describe the range of virtual addresses (the window) for the task region.
- Window block 1 will describe the window for the shared region.

Figure 3-5 shows the window-to-region relationship of such a task.

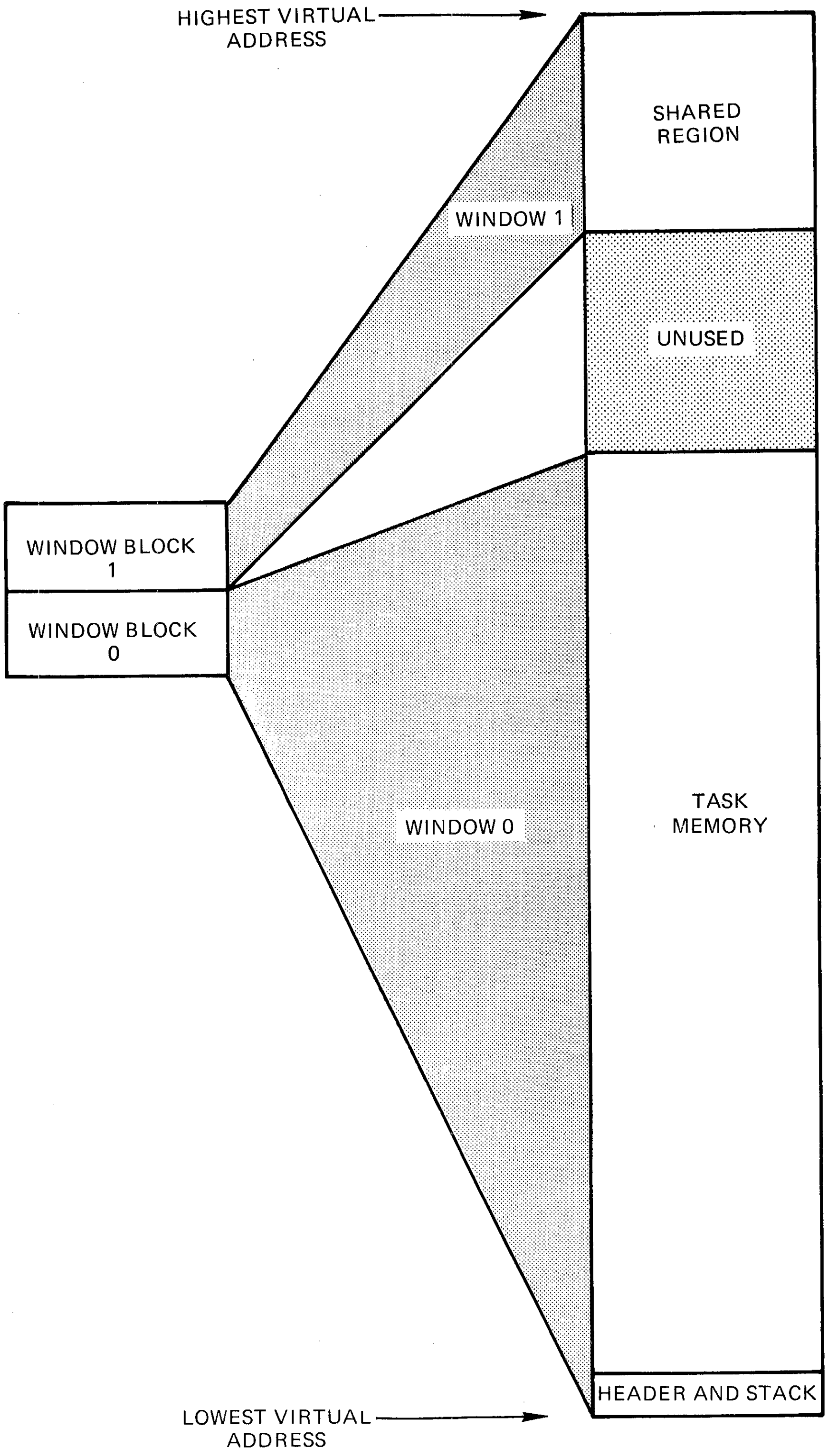


Figure 3-5 Windows for Shared Region and Referencing Task

A shared region need not be installed before a task that links to it is built. The .STB file that you specify when you build the shared region contains all the information required by the Task Builder to resolve references from within a task to locations within the shared region. The only requirement is that you install a shared region before you install a task that links to it.

The number of shared regions to which a task can link is a function of the number of window blocks required to map the task and the regions. In an RSX-11M operating system, if a task is 4K words or less, and each shared region to which the task links is 4K words or less, then a nonprivileged task can access as many as seven shared regions.

In an RSX-11M-PLUS operating system, if a task is 4K words or less, and each shared region to which the task links is 4K words or less, a nonprivileged task can refer to as many as 15 shared regions: seven in user-mode and eight in supervisor-mode. (Supervisor-mode libraries are described in later sections of this chapter.)

Finally, the way the Task Builder processes tasks that link to shared regions leads to an important Task Builder restriction on tasks that link to position-independent shared regions. The Task Builder places all program section names into its internal control section table. This includes program section names from the .STB file of the shared region as well as the program section names from the other input modules. When the Task Builder builds a task that links to a shared region, if the task contains program sections of the same name as program sections in the shared region, the Task Builder will attempt to add the program section allocation in the task to the already existing allocation for the program section of the same name in the shared region. This is not possible because the region's image has already been built, is outside the address space of the task currently being built and cannot be modified. Therefore, the program section names within a task that links to a position-independent shared region must normally be unique with respect to program section names within the shared region.

Should this conflict occur and the program section within the referencing task contain data, when the Task Builder attempts to initialize the program section, it will recognize that it is attempting to store data in an image outside the address limits of the task it is building. The Task Builder will then print the following message on the terminal:

```
     TKB--*DIAG*-LOAD ADDR OUT OF RANGE IN MODULE module-name
```

One exception to the above restriction develops when all of the following conditions exist:

- Both program sections (in the shared region and in the referencing task) have the (D) data and the OVR (overlay) attributes

- The program section in the task is equal to or shorter than the program section in the shared region

- The program section in the task does not contain data.

When all of these conditions exist, there is nothing to be initialized within the shared region. The Task Builder binds the base address of the program section in the task to the base address of the program section in the shared region. If the program section in the task contains global symbols, the Task Builder will assign addresses to them that reflect their location relative to the beginning of the program section. You can use this technique to establish symbolic offsets into resident commons. Examples 1 and 2 in the following sections illustrate how to establish these offsets.

3.1.5 Example 1: Building and Linking to a Common in MACRO-11

The text in this section and the figures associated with it illustrate the development of a MACRO-11 position-independent resident common and the development of two MACRO-11 tasks that share the common. The steps in building a position-independent common can be summarized as follows:

1. You create a source file that allocates the amount of space required for the common. In MACRO-11, either of the assembler directives, .BLKB or .BLKW, provide the means of allocating this space.

2. You assemble the source file.

3. You build the assembled module specifying both a task image file and a symbol definition file.

   You specify the -HD (no header) switch and declare the common to be position independent with the PI switch.

   Under options you specify:

   ```
   STACK=0
   PAR=parname
   ```

   The parname in this PAR option is the name of the partition in which the common is to reside. (The HD and PI switches and the STACK and PAR options are described in Chapter 6.)

   If your system is an RSX-11M system, the common must reside within a common partition of the same name as the common.

   If your system is an RSX-11M-PLUS system, the common can reside within any partition large enough to hold it.

4. You install the common.

Figure 3-6 below shows a MACRO-11 source file that, when assembled and built, will create a position-independent resident common area named MACCOM. The common area consists of two program sections named COM1 and COM2, respectively. Each program section is 512(10) words long.

```
        .TITLE MACCOM
;
;                       COM1 - 512 WORDS
;                       COM2 - 512 WORDS
;
        .PSECT COM1,RW,D,GBL,REL,OVR
        .BLKW 512.
        .PSECT COM2,RW,D,GBL,REL,OVR
        .BLKW 512.

        .END
```

Figure 3-6 Common Area Source File in MACRO-11

Once this common has been assembled, the Task Builder command sequence shown below can be used to build it.

```
TKB>MACCOM/PI-HD,MACCOM/-SP,MACCOM=MACCOM
TKB>/
ENTER OPTIONS:
TKB>STACK=0
TKB>PAR=MACCOM:0:4000
TKB>//
```

This command sequence directs the Task Builder to build a position-independent (/PI), headerless (/-HD), common image file named MACCOM.TSK. It also specifies that the Task Builder is to create a map file, MACCOM.MAP, and a symbol definition file, MACCOM.STB. The Task Builder will create all three files, MACCOM.TSK, MACCOM.MAP, and MACCOM.STB on device SY: under the UFD that corresponds to the terminal UIC. Because /-SP is attached to the map file, the Task Builder will not spool a map listing to the line printer.

Under options, STACK=0 suppresses the stack area in the common's image.

The PAR option specifies that the common area will reside within a common partition of the same name as the common, MACCOM. As stated above, on an RSX-11M system this is a requirement; on an RSX-11M-PLUS system it is not. In addition, the parameters in the PAR option specify a base of zero and a length of octal bytes for the common (Refer to Chapter 6 for descriptions of the switches and options used in this example.)

Figure 3-7 shows the map resulting from this command sequence.

The task attributes section of this map reflects the switches and options of the command string. It indicates that the common resides in a partition named MACCOM, that it was built under terminal UIC [303,3], that it is headerless and position independent, and that it requires one window block to map. The total length of the common is 1024(10) words and its address limits range from 0 to 3777(8). The common image (that portion of the disk image file that eventually will be read into memory) begins at file-relative disk block 3 ❶. The last block in the file is file-relative disk block 6 ❷ and the common image is four blocks long ❸.

The memory allocation synopsis details the Task Builder's allocation for and the attributes of the program sections within the common. For example, reading from left to right, the map indicates that the program section COM1 permits read/write access, that it contains data, and that its scope is global. It also indicates that COM1 is relocatable and that all contributions to COM1 are to be overlaid. Because COM1 has the overlay attribute, the total allocation for it will be equal to the largest allocation request from the modules that contribute to it. (For more information on program section attributes, see Chapter 2.)

Continuing to the right, the first 6-digit number is COM1's base address which is 0 ❹. The next two digits are its length (bytes) in octal and decimal, respectively ❺.

The next line down lists the first object module that contributes to COM1. In this case there is only one: the module MACCOM from the file MACCOM.OBJ;2. The numbers on this line indicate the relative base address of the contribution and the length of the contribution in octal and decimal ❺ . If there had been more than one module input to the Task Builder that contained a program section named COM1, the Task Builder would have listed each module and its contribution in this section.

Notice that there is a program section named . BLK. shown on the map just above the field for COM1. This is the "blank" program section that is automatically created by the language translators. The attributes shown are the default attributes. The allocation for . BLK. is zero because the program sections in MACCOM were explicitly declared. If the program sections had not been explicitly declared, all of the allocation for the common would have been within this program section.

```
MACCOM.TSK;2   MEMORY ALLOCATION MAP  TKB M36         PAGE 1
                     7-FEB-79   13:51


PARTITION NAME : MACCOM                                  ┐
IDENTIFICATION :                                         │
TASK  UIC      : [303,3]                                 │ Task
TASK ATTRIBUTES: -HD,PI                                  │ Attributes
TOTAL ADDRESS WINDOWS: 1.                                │ Section
TASK  IMAGE  SIZE  : 1024. WORDS                         │
TASK ADDRESS LIMITS: 000000 003777                       │
R-W DISK BLK LIMITS: 000003 000006 000004 00004.         ┘

*** ROOT SEGMENT: MACCOM
                        ❶      ❷         ❸

R/W MEM  LIMITS: 000000 003777 004000 02048.
DISK BLK LIMITS: 000002 000005 000004 00004.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                          TITLE    IDENT  FILE
-------                                          -----    -----  ----
. BLK.:(RW,I,LCL,REL,CON)  000000 000000 00000.
COM1  :(RW,D,GBL,REL,OVR)  000000 002000 01024.
                           000000 002000 01024.  MACCOM   01     MACCOM.OBJ;2
COM2  :(RW,D,GBL,REL,OVR)  002000 002000 01024.
                           002000 002000 01024.  MACCOM   01     MACCOM.OBJ;2

                              ❻ ❹         ❺

*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 178.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 8198. WORDS (32. PAGES)
    SIZE OF WORK FILE: 768. WORDS (3. PAGES)

    ELAPSED TIME:00:00:06
```

Figure 3-7 Task Builder Map for MACCOM.TSK

Figure 3-8 is a diagram that represents the disk image file for MACCOM. The circled numbers in Figure 3-8 correspond to the circled numbers in Figure 3-7.

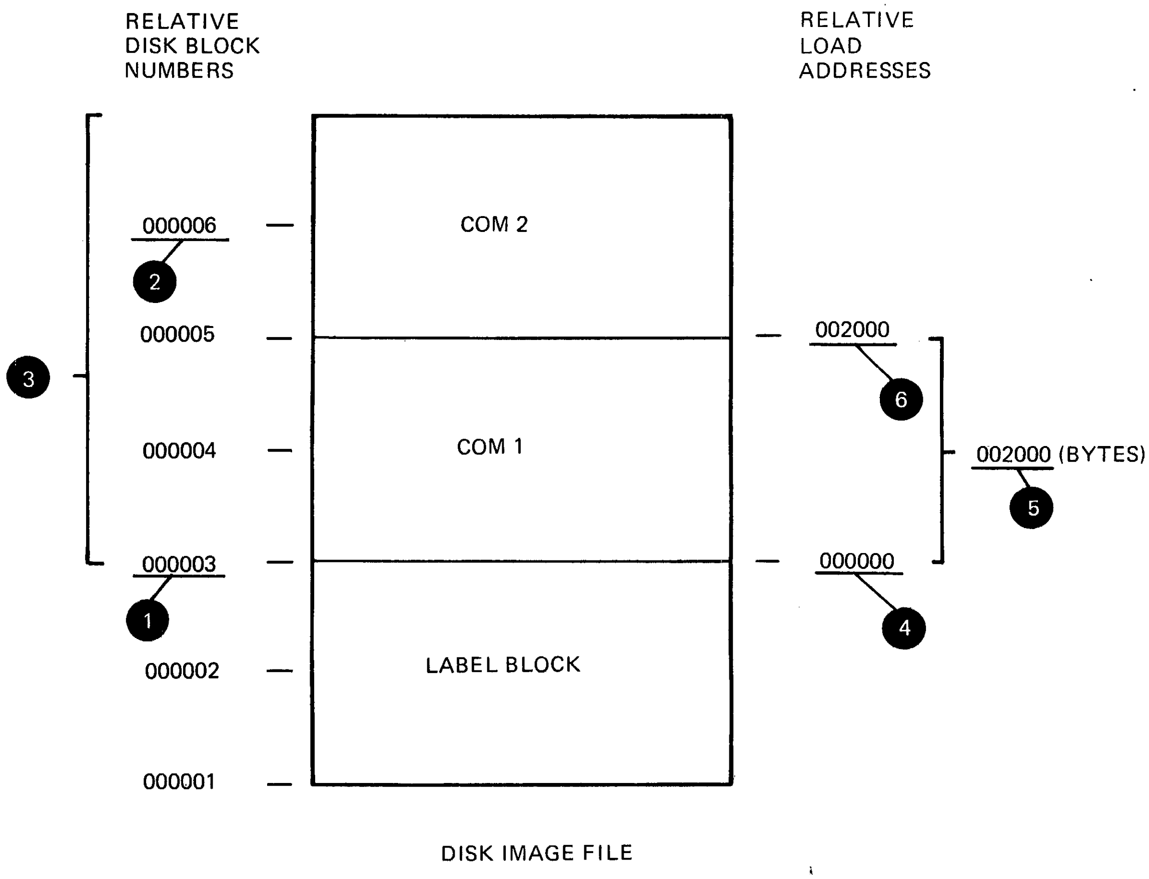


Figure 3-8 Allocation Diagram for MACCOM.TSK

Once you have built MACCOM, you can install it. If your system is an RSX-11M system, the common will be loaded into memory when you install it. It will remain there until you explicitly remove it with the MCR command, REMOVE.

If your system is an RSX-11M-PLUS system, the common will not be loaded until either one of the following occurs:

- A task that is linked to it is run.
- You explicitly fix the common in memory with the MCR command, FIX.

Figures 3-9 and 3-10 show two programs -- MCOM1 and MCOM2, respectively. Both of these programs reference the common area MACCOM created above. MCOM1 in Figure 3-9 accesses the COM1 portion of MACCOM. It inserts into the first ten words of COM1 the numbers 1 through 10 in ascending order. It then issues an Executive directive request for the task MCOM2 and suspends itself.

When MCOM2 runs, it sums the integers left in COM1 by MCOM1 and leaves the result in the first word of COM2. It then issues a resume directive for MCOM1 and exits.

When MCOM1 resumes, it retrieves the answer left in COM2 and calls the system library routine $EDMSG (edit message) to format the answer for output to device TI:.

All of the Executive directives for both programs (RQST$C, SPND$S, QIOW$S, RSUM$C, and EXIT$S) are documented in the RSX-11M-PLUS Executive Reference Manual. The system library routine $EDMSG is documented in the IAS/RSX-11 System Library Routines Reference Manual.

```
        .TITLE  MCOM1
        .IDENT  /01/

        .MCALL  EXIT$S,SPND$S,RQST$C,QIOW$S

OUT:    .BLKW   100.                    ; SCRATCH AREA
FORMAT: .ASCIZ  /THE RESULT IS %D./
MES:    .ASCII  /ERROR FROM REQUEST/
        LEN = . - MES
        .EVEN

;       PSECT - COM1 IS USED TO ACCESS THE FIRST 512. WORDS OF THE
;       COMMON.

        .PSECT  COM1,GBL,OVR,D
INT:    .BLKW   10.

;       PSECT - COM2 IS USED TO ACCESS THE SECOND 512. WORDS OF THE
;       COMMON. IT WILL CONTAIN THE RESULT

        .PSECT  COM2,GBL,OVR,D
ANS:    .BLKW   1

        .PSECT
START:
        MOV     #10.,R0                 ; NUMBER OF INTEGERS TO SUM
        MOV     #1,R1                   ; START WITH A 1
        MOV     #INT,R3                 ; PLACE VALUES IN 1ST 10 WORDS
                                        ; OF COMMON
10$:    MOV     R1,(R3)+                ; INITIALIZE COMMON
        INC     R1                      ; NEXT INTEGER
        DEC     R0                      ; ONE LESS TIME
        BNE     10$                     ; TO INITIALIZE
        RQST$C  MCOM2                   ; REQUEST THE SECOND TASK
        BCS     ERR1                    ; REQUEST FAILED
        SPND$S                          ; WAIT FOR MCOM2 TO SUM THE INTEGERS
        MOV     #OUT,R0                 ; ADDRESS OF SCRATCH AREA
        MOV     #FORMAT,R1              ; FORMAT SPECIFICATION
        MOV     #ANS,R2                 ; ARGUMENT TO CONVERT
        CALL    $EDMSG                  ; DO CONVERSION
        QIOW$S  #IO.WVB,#5,#1,,,,<#OUT,R1,#40>
        EXIT$S
ERR1:
        QIOW$S  #IO.WVB,#5,#1,,,,<#MES,#LEN,#40>
        EXIT$S
        .END    START
```

Figure 3-9 MACRO-11 Source Listing for MCOM1

```
        .TITLE  MCOM2
        .IDENT  /01/



        .MCALL  EXIT$S,QIOW$S,RSUM$C

MES:    .ASCII  /ERROR FROM RESUME/
        LEN = . - MES
        .EVEN

;       PSECT - COM1 IS USED TO ACCESS THE FIRST 10. WORDS OF THE
;       COMMON.

        .PSECT  COM1,GBL,OVR,D
INT:    .BLKW   10.

;       PSECT - COM2 IS USED TO ACCESS THE SECOND 10. WORDS OF THE
;       COMMON. IT WILL CONTAIN THE RESULT

        .PSECT  COM2,GBL,OVR,D
ANS:    .BLKW   1

        .PSECT

START:
        MOV     #10.,R0                         ; NUMBER OF INTEGERS TO SUM
        MOV     #INT,R3                         ; PLACE VALUES IN 1ST 10 WORDS
                                                ; OF COMMMON
        CLR     ANS                             ; INITIALIZE ANSWER
10$:    ADD     (R3)+,ANS                       ; ADD IN VALUES
        DEC     R0                              ; ONE LESS VALUE
        BNE     10$                             ; TO SUM

        RSUM$C  MCOM1                           ; RESUME MCOM1
        BCS     ERR                             ; RESUME FAILED
        EXIT$S
ERR:
        QIOW$S  #IO.WVB,#5,#1,,,,<#MES,#LEN,#40>
        EXIT$S
        .END    START
```

Figure 3-10 MACRO-11 Source Listing for MCOM2

Note that both tasks MCOM1 and MCOM2 contain .PSECT declarations that establish program section names that are the same as program section names within the position-independent common to which the task is linked (MACCOM). As stated earlier, in most circumstances this would be illegal. In this application, however, the .PSECT directives have been placed into the tasks to establish symbolic offsets in the resident common. When either task is built, the Task Builder will assign to the symbol INT: the base address of program section COM1, and to the symbol ANS: the base address of program section COM2. Figure 3-11 illustrates this assignment.

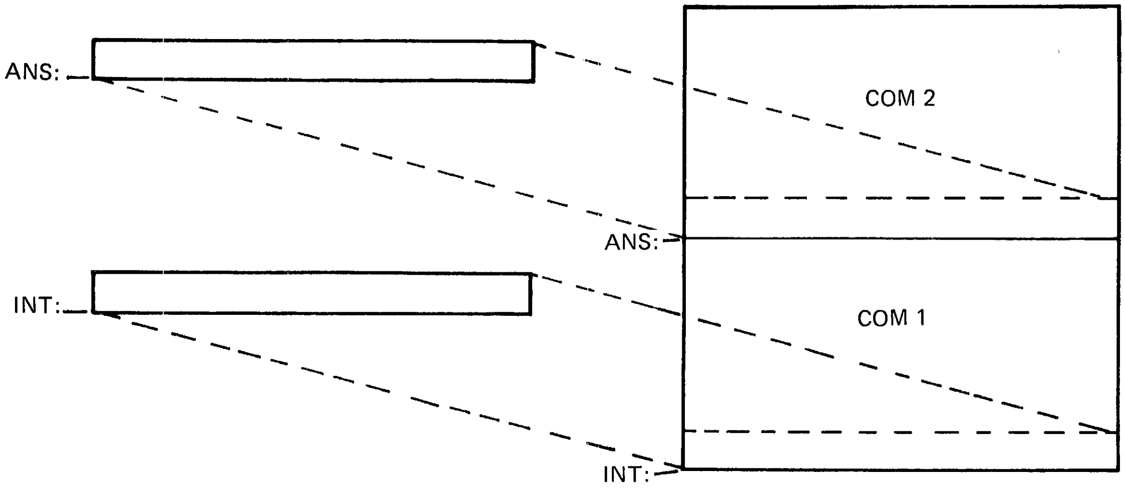


Figure 3-11 Assigning Symbolic References Within a Common

Once you have assembled MCOM1 and MCOM2, you can build them with the following Task Builder command sequences:

```
>TKB
TKB>MCOM1,MCOM1/-SP=MCOM1
TKB>/
ENTER OPTIONS:
TKB>RESCOM=MACCOM/RW
TKB>//
```

```
>TKB
TKB>MCOM2,MCOM2/-SP=MCOM2
TKB>/
ENTER OPTIONS:
TKB>RESCOM=MACCOM/RW
TKB>//
```

Under options in both of these command sequences, the RESCOM option tells the Task Builder that these programs intend to reference a common data area named MACCOM and that the tasks require read/write access to it.

Because the RESCOM option is used, the Task Builder expects to find the image file and the symbol definition file for the common on device SY: under the UFD that corresponds to the terminal UIC. In addition, because the optional APR specification was omitted from the RESCOM option, the Task Builder allocates virtual address space for the common starting with APR7 in both tasks (the highest APR available in both tasks).

The Task Builder map for MCOM1 is shown in Figure 3-12. The map for MCOM2 is not essentially different from that of MCOM1 and is therefore not included here.

```
MCOM1.TSK;23   MEMORY ALLOCATION MAP TKB M36        PAGE 1
                    7-FEB-79   14:32


PARTITION NAME : GEN                                  ┐
IDENTIFICATION : 01                                   │
TASK  UIC      : [303,3]                              │
STACK    LIMITS: 000212 001211 001000 00512.          │Task
PRG XFR ADDRESS: 001566                               │Attributes
TOTAL ADDRESS WINDOWS: 2.                             │Section
TASK  IMAGE  SIZE  : 1120. WORDS                      │
TASK ADDRESS LIMITS: 000000 004273                    │
R-W DISK BLK LIMITS: 000002 000006 000005 00005.     ─┘

*** ROOT SEGMENT: MCOM1


R/W MEM  LIMITS: 000000 004273 004274 02236.
DISK BLK LIMITS: 000002 000006 000005 00005.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                               TITLE  IDENT  FILE
-------                                               -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001212 002630 01432.
                          001212 000574 00380. MCOM1  01     MCOM1.OBJ;2
COM1  :(RW,D,GBL,REL,OVR) 160000 002000 01024.
                          160000 000024 00020. MCOM1  01     MCOM1.OBJ;2
COM2  :(RW,D,GBL,REL,OVR) 162000 002000 01024.
                          162000 000002 00002. MCOM1  01     MCOM1.OBJ;2
LNC$D :(RW,D,GBL,REL,CON) 004042 000002 00002.
$DPB$$:(RW,I,LCL,REL,CON) 004044 000016 00014.
                          004044 000016 00014. MCOM1  01     MCOM1.OBJ;2
$$RESL:(RW,I,LCL,REL,CON) 004062 000024 00020.
$$RESM:(RW,I,LCL,REL,CON) 004106 000166 00118.


*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 2125.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 8198. WORDS (32. PAGES)
    SIZE OF WORK FILE: 1024. WORDS (4. PAGES)

    ELAPSED TIME:00:00:06
```

Figure 3-12 Task Builder Map for MCOM1.TSK

Note that the Task Builder has placed two window blocks in MCOM1's header. When MCOM1 is installed, the INSTALL processor will initialize these window blocks as follows:

- Window block 0 will describe the range of virtual addresses (the window) for MCOM1's task region.

- Window block 1 will describe the window for the shared region MACCOM.

3.1.6 Example 2: Building and Linking to a Device Common in MACRO-11

A device common is a special type of common that occupies physical addresses on the I/O page. When mapped into the virtual address space of a task, a device common permits the task to manipulate peripheral device registers directly.

NOTE

> Because any access to the I/O page is potentially hazardous to the running system, you must exercise extreme caution when working with device commons.

The remaining text in this section and the figures associated with it illustrate the development and use of a device common. Figure 3-13 shows an assembly listing for a position-independent device common named TTCOM. When installed, TTCOM will map the control and data registers of the console terminal. Its physical base address will be 777500.

```
        .TITLE TTCOM
        .PSECT TTCOM
        .=.+60
RCSR::  .BLKW   1
RBUF::  .BLKW   1
XCSR::  .BLKW   1
XBUF::  .BLKW   1
        .END
```

Figure 3-13 Assembly Listing for TTCOM

The PDP-11 Peripherals Handbook defines the control and data register addresses for the console terminal. In Figure 3-13, the register addresses and the symbol names that correspond to them are as follows:

Register	Address	Symbol
Keyboard Status	777560	RCSR
Keyboard Data	777562	RBUF
Printer Status	777564	XCSR
Printer Data	777566	XBUF

The double colon (::) following each symbol in Figure 3-13 establishes the symbol as global. The first symbol, RCSR, is offset from the beginning of TTCOM by 60(8) bytes. Each symbol thereafter is one word removed from the symbol that precedes it. Thus, when TTCOM is installed at 777500, each symbol will be located at its proper address.

Once you have assembled TTCOM, you can build it using the following Task Builder command sequence:

```
> TKB
TKB> LB:[1,1]TTCOM/-HD/PI,LB:[1,1]TTCOM/-WI,LB:[1,1]TTCOM=TTCOM
TKB> /
ENTER OPTIONS:
TKB> STACK=0
TKB> PAR=TTCOM:0:100
TKB> //
```

This command sequence directs the Task Builder to create a common image named TTCOM.TSK and a symbol definition file named TTCOM.STB. The Task Builder will place both files on device LB: under UFD [1,1]. The command sequence also specifies that the Task Builder is to spool a map listing to the line printer. The -WI switch specifies an 80 column line printer listing format.

NOTE

> For the command sequence above to work in a multiuser protection system, it must be input from a privileged terminal.

Under options, STACK=0 suppresses the stack area in the common's image file.

The PAR option specifies that the device common will reside within a partition of the same name as the common. As with the data common in Example 1 (Section 3.1.5), this is a requirement of the RSX-11M system; in an RSX-11M-PLUS system it is not. The PAR option also specifies that the base of the common is 0 and that it is 100(8) bytes long.

The Task Builder map for TTCOM that results from the command sequence above is shown in Figure 3-14. The task attributes section of this map indicates that the common is position independent and that no header is associated with it. The common's image and symbol definition file reside on device LB: under UFD [1,1].

The map in Figure 3-14 shows the global symbols defined in the common with their relative offsets into the common region. You establish the virtual base address for the common and the virtual addresses for the symbols within it when you build the tasks that link to the common.

You establish the physical addresses for the common with the MCR command, SET. The keyword that you use with the SET command depends on which system you are running. If your system is an RSX-11M system, use the command:

>SET /MAIN=TTCOM:7775:1~~:COM~~DEV

If your system is an RSX-11M-PLUS system, use the command:

```
>SET /PAR=TTCOM:7775:1:DEV
```

Both sequences create a main partition named TTCOM that begins at physical address 777500. The partition is one 64-byte block long, (100(8) bytes). The arguments COM and DEV identify the partition type. With the common built and the partition for it created, you can install TTCOM.

You can establish the partition for a device common at any time in both the RSX-11M and the RSX-11M-PLUS systems. Partitions created to accommodate a device common are not a system generation consideration because they represent areas of physical address space above memory and therefore cannot conflict with memory partitions.

```
TTCOM.TSK;1   MEMORY ALLOCATION MAP  TKB           PAGE 1
                    9-NOV-78   14:25


PARTITION NAME : TTCOM                                  ┐
IDENTIFICATION :                                        │
TASK  UIC      : [1,1]                                  │ TASK
TASK ATTRIBUTES: -HD,PI                                 │ ATTRIBUTES
TOTAL ADDRESS WINDOWS: 1.                               │ SECTION
TASK  IMAGE  SIZE  : 32. WORDS                          │
TASK ADDRESS LIMITS: 000000 000067                      │
R-W DISK BLK LIMITS: 000003 000003 000001 00001.       ┘

*** ROOT SEGMENT: TTCOM


R/W MEM  LIMITS: 000000 000067 000070 00056.
DISK BLK LIMITS: 000002 000002 000001 00001.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                       TITLE  IDENT  FILE
-------                                       -----  -----  ----

. BLK.:(RW,I,LCL,REL,CON) 000000 000000 00000.
TTCOM :(RW,I,LCL,REL,CON) 000000 000070 00056.
                          000000 000070 00056. TTCOM  01     TTCOM.OBJ;1


GLOBAL SYMBOLS:

RBUF  000062-R  RCSR  000060-R  XBUF  000066-R  XCSR  000064-R



*** TASK BUILDER STATISTICS:

     TOTAL WORK FILE REFERENCES: 220.
     WORK  FILE  READS: 0.
     WORK  FILE WRITES: 0.
     SIZE OF CORE POOL: 2062. WORDS (8. PAGES)
     SIZE OF WORK FILE: 768. WORDS (3. PAGES)

     ELAPSED TIME:00:00:02
```

Figure 3-14 Task Builder Map for TTCOM

Figure 3-15 shows an assembly listing for a demonstration program named TEST. When built and installed, TEST will print the letters A through Z on the console terminal by directly accessing the console terminal status and data registers. It will access the status and data registers through the device common TTCOM.

```
         .TITLE TEST
         .IDENT /01/
         .MCALL EXIT$S

START:   MOV     #15,R0     ; START WITH A CARRIAGE RETURN
         CALL    OUTBYT     ; PRINT IT
         MOV     #12,R0     ; THEN A LINE FEED
         CALL    OUTBYT     ; PRINT IT
         MOV     #101,R0    ; FIRST LETTER IS AN "A"
         MOV     #26.,R1    ; NUMBER OF LETTERS TO PRINT
OUTPUT:  CALL    OUTBYT     ; PRINT CURRENT LETTER
         DEC     R1         ; ONE LESS TIME
         BNE     OUTPUT     ; AGAIN
         MOV     #15,R0     ; ANOTHER CARRIAGE RETURN
         CALL    OUTBYT
         MOV     #12,R0     ; ANOTHER LINE FEED
         CALL    OUTBYT
         EXIT$S
OUTBYT:  TSTB    XCSR       ; OUTPUT BUFFER READY?
         BPL     OUTBYT     ; IF NOT WAIT
         MOV     R0,XBUF    ; MOVE CHARACTER TO OUTPUT BUFFER
         INC     R0         ; INITIALIZE NEXT LETTER
         RETURN
         .END    START
```

Figure 3-15 Assembly Listing for TEST

Once you have assembled TEST, you can build it with the following Task Builder command sequence:

```
>TKB
TKB>TEST,TEST/-WI/MA=TEST
TKB>/
ENTER OPTIONS:
TKB>COMMON=TTCOM:RW:1
TKB>//
```

Under options, the COMMON option in this command sequence tells the Task Builder that TEST intends to access the device common TTCOM and, that TEST will have read/write access to it. It also directs the Task Builder to reserve APR 1 for mapping the common into TEST's virtual address space.

The Task Builder map that results from the command sequence above is shown in Figure 3-16.

This map contains a global symbols section. The Task Builder included it because the MA switch was applied to the memory allocation file at task-build time. Note that the global symbols in this section, which were defined in TTCOM, now have virtual addresses assigned to them. The addresses assigned by the Task Builder are the result of the APR 1 specification in the COMMON= keyword during the task build.

It is important to remember that programs like TEST, which access the I/O page, take complete control of the registers they reference. Therefore, coding errors in such programs can disable the devices they reference and can even make it impossible for the device drivers to regain control of the device. If this happens, you must reboot the system.

```
TEST.TSK;2   MEMORY ALLOCATION MAP TKB           PAGE 1
                       9-NOV-78 14:35


PARTITION NAME : GEN                                         ┐
IDENTIFICATION : 01                                          │
TASK  UIC      : [301,356]                                   │
STACK    LIMITS: 000212 001211 001000 00512.                 │ Task
PRG XFR ADDRESS: 001212                                      │ Attributes
TOTAL ADDRESS WINDOWS: 2.                                    │ Section
TASK  IMAGE  SIZE  : 384. WORDS                              │
TASK ADDRESS LIMITS: 000000 001317                           │
R-W DISK BLK LIMITS: 000002 000003 000002 00002.            ┘

*** ROOT SEGMENT: TEST


R/W MEM  LIMITS: 000000 001317 001320 00720.
DISK BLK LIMITS: 000002 000003 000002 00002.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                        TITLE  IDENT  FILE
-------                                        -----  -----  ----

. BLK.:(RW,I,LCL,REL,CON) 001212 000104 00068.
                          001212 000104 00068. .MAIN. 01     TEST.OBJ;1
TTCOM :(RW,I,LCL,REL,CON) 020000 000070 00056.
                          020000 000070 00056. TTCOM  01     TTCOM.STB;1


GLOBAL SYMBOLS:

RBUF   020062-R   RCSR   020060-R   XBUF   020066-R   XCSR   020064-R


*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 220.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 2062. WORDS (8. PAGES)
    SIZE OF WORK FILE: 768. WORDS (3. PAGES)

    ELAPSED TIME:00:00:04
```

Figure 3-16 Memory Allocation Map for TEST

3.1.7 Example 3: Building and Linking to a Resident Library in MACRO-11

Resident libraries consist of subroutines that are shared by two or more tasks. When such tasks reside in physical memory simultaneously, resident libraries provide a considerable memory savings because the subroutines within the library appear in memory only once.

The text in this section and the figures associated with it illustrate the development and use of a resident library, called LIB.

Figure 3-17 shows five FORTRAN callable subroutines:

- An integer addition routine, AADD
- An integer subtraction routine, SUBB
- An integer multiplication routine, MULL
- An integer division routine, DIVV
- A register save and restore coroutine, SAVAL

These subroutines are contained in a single source file, LIB.MAC. When assembled and built, they will constitute an example of a resident library. FORTRAN callable routines were used in this example so that the library can be accessed by either FORTRAN or MACRO-11 programs.

```
        .TITLE  LIB
        .IDENT  /01/


        .PSECT  AADD,RO,I,GBL,REL,CON

;** FORTRAN CALLABLE SUBROUTINE TO ADD TWO INTEGERS

AADD::  CALL    $SAVAL          ; SAVE R0-R5
        MOV     @2(R5),R0       ; FIRST OPERAND
        MOV     @4(R5),R1       ; SECOND OPERAND
        ADD     R0,R1           ; SUM THEM
        MOV     R1,@6(R5)       ; STORE RESULT
        RETURN                  ; RESTORE REGISTERS AND RETURN



        .PSECT  SUBB,RO,I,GBL,REL,CON

;** FORTRAN CALLABLE SUBROUTINE TO SUBTRACT TWO INTEGERS

SUBB::  CALL    $SAVAL          ; SAVE R0-R5
        MOV     @2(R5),R0       ; FIRST OPERAND
        MOV     @4(R5),R1       ; SECOND OPERAND
        SUB     R1,R0           ; SUBTRACT SECOND FROM FIRST
        MOV     R0,@6(R5)       ; STORE RESULT
        RETURN                  ; RESTORE REGISTERS AND RETURN



        .PSECT  MULL,RO,I,GBL,REL,CON
```

Figure 3-17 Source Listing for Resident Library LIB.MAC

```
;** FORTRAN CALLABLE SUBROUTINE TO MULTIPLY TWO INTEGERS


MULL::  CALL    $SAVAL          ; SAVE R0-R5
        MOV     @2(R5),R0       ; FIRST OPERAND
        MOV     @4(R5),R1       ; SECOND OPERAND
        MUL     R0,R1           ; MULTIPLY
        MOV     R1,@6(R5)       ; STORE RESULT
        RETURN                  ; RESTORE REGISTERS AND RETURN




        .PSECT  DIVV,RO,I,GBL,REL,CON

;** FORTRAN CALLABLE SUBROUTINE TO DIVIDE TWO INTEGERS


DIVV::  CALL    $SAVAL          ; SAVE REGS R0-R5
        MOV     @2(R5),R3       ; FIRST OPERAND
        MOV     @4(R5),R1       ; SECOND OPERAND
        CLR     R2              ; LOW ORDER 16 BITS
        DIV     R1,R2           ; DIVIDE
        MOV     R2,@6(R5)       ; STORE RESULT
        RETURN                  ; RESTORE REGISTERS AND RETURN




        .PSECT  SAVAL,RO,I,GBL,REL,CON

;**ROUTINE TO SAVE REGISTERS

$SAVAL::
        MOV     R4,-(SP)        ;SAVE R4
        MOV     R3,-(SP)        ;SAVE R3
        MOV     R2,-(SP)        ;SAVE R2
        MOV     R1,-(SP)        ;SAVE R1
        MOV     R0,-(SP)        ;SAVE R0
        MOV     12(SP),-(SP)    ;COPY RETURN
        MOV     R5,14(SP)       ;SAVE R5
        CALL    @(SP)+          ;CALL THE CALLER
        MOV     (SP)+,R0        ;RESTORE R0
        MOV     (SP)+,R1        ;RESTORE R1
        MOV     (SP)+,R2        ;RESTORE R2
        MOV     (SP)+,R3        ;RESTORE R3
        MOV     (SP)+,R4        ;RESTORE R4
        MOV     (SP)+,R5        ;RESTORE R5
        RETURN
        .END
```

Figure 3-17 (Cont.) Source Listing for Resident Library LIB.MAC

Once you have assembled LIB, you can build it with the following Task Builder command sequence:

```
     TKB>LIB/PI/-HD,LIB/-WI,LIB=LIB
     TKB>/
     ENTER OPTIONS:
     TKB>STACK=0
     TKB>PAR=LIB:0:200
     TKB>//
```

This command sequence instructs the Task Builder to build a position-independent (/PI), headerless (/-HD) library image named LIB.TSK. It instructs the Task Builder to create a map file LIB.MAP and to output an 80 column listing (/-WI) to the line printer. It also specifies that the Task Builder is to create a symbol definition file, LIB.STB. The Task Builder will create all three files, LIB.TSK, LIB.MAP, LIB.STB on device SY: under the UFD that corresponds to the terminal UIC.

Under options, STACK=0 suppresses the stack area within the resident library's image.

The PAR option tells the Task Builder that the resident library will reside within a partition of the same name as the library. As with all shared regions, this is a requirement in an RSX-11M system; in an RSX-11M-PLUS system it is not. In addition, the PAR option specifies that the base of the library is 0 and that it is 200(8) bytes long. (For more information on the switches and options used in this example, refer to Chapter 6.)

Figure 3-18 shows the Task Builder map that results from the command sequence above.

Note in the global symbols section of the map in Figure 3-18 that the Task Builder has assigned offsets to the symbols for each library function. When the task that links to this library is built, the Task Builder will assign virtual addresses to these symbols.

The program MAIN in Figure 3-19 exercises the routines in the resident library LIB.TSK. When you assemble and build it, MAIN will call upon the library routines to add, subtract, multiply, and divide the integers contained in the labels OP1 and OP2 within the program. MAIN will print the results of each operation to device TI:.

```
LIB.TSK;12      MEMORY ALLOCATION MAP  TKB M36         PAGE 1
                      7-FEB-79   14:47



PARTITION NAME : LIB                                     ┐
IDENTIFICATION : 01                                      │
TASK  UIC      : [303,3]                                 │ TASK
TASK ATTRIBUTES: -HD,PI                                  │ ATTRIBUTES
TOTAL ADDRESS WINDOWS: 1.                                │ SECTION
TASK  IMAGE  SIZE  : 64. WORDS                           │
TASK ADDRESS LIMITS: 000000 000163                       │
R-W DISK BLK LIMITS: 000003 000002 000000 00000.        ┘

*** ROOT SEGMENT: LIB


R/W MEM  LIMITS: 000000 000163 000164 00116.
DISK BLK LIMITS: 000002 000002 000001 00001.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                            TITLE  IDENT  FILE
-------                                            -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 000000 000000 00000.
AADD  :(RO,I,GBL,REL,CON) 000000 000024 00020.
                          000000 000024 00020. LIB    01     LIB.OBJ;2
DIVV  :(RO,I,GBL,REL,CON) 000024 000026 00022.
                          000024 000026 00022. LIB    01     LIB.OBJ;2
MULL  :(RO,I,GBL,REL,CON) 000052 000024 00020.
                          000052 000024 00020. LIB    01     LIB.OBJ;2
SAVAL :(RO,I,GBL,REL,CON) 000076 000042 00034.
                          000076 000042 00034. LIB    01     LIB.OBJ;2
SUBB  :(RO,I,GBL,REL,CON) 000140 000024 00020.
                          000140 000024 00020. LIB    01     LIB.OBJ;2



GLOBAL SYMBOLS:

AADD   000000-R  MULL   000052-R SUBB   000140-R
DIVV   000024-R  $SAVAL 000076-R



*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 376.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 8198. WORDS (32. PAGES)
    SIZE OF WORK FILE: 768. WORDS (3. PAGES)

    ELAPSED TIME:00:00:04
```

Figure 3-18 Task Builder Map for LIB.TSK

```
        .TITLE  MAIN
        .IDENT  /01/

;+
;**MAIN - CALLING ROUTINE TO EXERCISE THE ARITHMETIC ROUTINES
;         FOUND IN THE RESIDENT LIBRARY, LIB.TSK.
;-

        .MCALL  QIOW$S,EXIT$S

OP1:    .WORD   1                       ; OPERAND 1
OP2:    .WORD   1                       ; OPERAND 2
ANS:    .BLKW   1                       ; RESULT

OUT:    .BLKW   100.                    ; FORMAT MESSAGE
FORMAT: .ASCIZ  /THE ANSWER = %D./
        .EVEN

        .ENABL  LSB

START:
        MOV     #ANS,-(SP)              ; TO CONTAIN RESULT
        MOV     #OP2,-(SP)              ; OPERAND 2
        MOV     #OP1,-(SP)              ; OPERAND 1
        MOV     #3,-(SP)                ; PASSING 3 ARGUMENTS
        MOV     SP,R5                   ; ADDRESS OF ARGUMENT BLOCK
        CALL    AADD                    ; ADD TWO OPERANDS
        CALL    PRINT                   ; PRINT RESULTS
        MOV     SP,R5                   ; ADDRESS OF ARGUMENT BLOCK
        CALL    SUBB                    ; SUBTRACT SUBROUTINE
        CALL    PRINT                   ; PRINT RESULTS
        MOV     SP,R5                   ; ADDRESS OF ARGUMENT BLOCK
        CALL    MULL                    ; MULTIPLY SUBROUTINE
        CALL    PRINT                   ; PRINT RESULTS
        MOV     SP,R5                   ; ADDRESS OF ARGUMENT BLOCK
        CALL    DIVV                    ; DIVIDE SUBROUTINE
        CALL    PRINT                   ; PRINT RESULTS
        EXIT$S

;+
;** PRINT - PRINT RESULT OF OPERATION.
;-

PRINT:  MOV     #OUT,R0                 ; ADDRESS OF SCRATCH AREA
        MOV     #FORMAT,R1              ; FORMAT SPECIFICATION
        MOV     #ANS,R2                 ; ARGUMENT TO CONVERT
        CALL    $EDMSG                  ; FORMAT MESSAGE
        QIOW$S  #IO.WVB,#5,#1,,,,<#OUT,R1,#40>
        RETURN                          ; RETURN FROM SUBROUTINE
        .END    START
```

Figure 3-19 Source Listing for MAIN.MAC

Once you have assembled MAIN, you can use the following Task Builder command sequence to build it:

```
     TKB>MAIN,MAIN/MA/-WI/-SP=MAIN
     TKB>/
     ENTER OPTIONS:
     TKB>RESLIB=LIB/RO:3
     TKB>//
```

This command sequence instructs the Task Builder to build a task file named MAIN.TSK on device SY: under the UFD that corresponds to the terminal UIC. It also specifies that the Task Builder is to create a map file MAIN.MAP. The MA switch requests an extended map format. In this example, /MA was applied to the device specification so that the Task Builder would include in the map for the task the symbols within the library LIB. The negated form of the wide listing switch (/-WI) was appended to the map specification to obtain an 80-column map format. In this example, the Task Builder will not output a map listing to the line printer

Under options, the RESLIB option specifies that the task MAIN is to access the library LIB and that it requires read-only access to LIB. The Task Builder will use APR3 to map the library.

The Task Builder map that results from this command sequence is shown in Figure 3-20.

```
MAIN.TSK;15     MEMORY ALLOCATION MAP  TKB M36          PAGE 1
                        30-APR-79   10:33



PARTITION NAME : GEN                                   ┐
IDENTIFICATION : 01                                    │
TASK  UIC      : [303,3]                               │
STACK    LIMITS: 000212 001211 001000 00512.           │ TASK
PRG XFR ADDRESS: 001552                                │ ATTRIBUTES
TOTAL ADDRESS WINDOWS: 2.                              │ SECTION
TASK  IMAGE  SIZE  : 1120. WORDS                       │
TASK ADDRESS LIMITS: 000000 004213                     │
R-W DISK BLK LIMITS: 000002 000006 000005 00005.      ─┘

*** ROOT SEGMENT: MAIN


R/W MEM  LIMITS: 000000 004213 004214 02188.
DISK BLK LIMITS: 000002 000006 000005 00005.
```

Figure 3-20 Task Builder Map for MAIN.TSK

```
MEMORY ALLOCATION SYNOPSIS:

SECTION                                            TITLE  IDENT  FILE
-------                                            -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001212 002564 01396.
                          001212 000530 00344. MAIN   01     MAIN.OBJ;1
                          001742 001016 00526. EDTMG  12     SYSLIB.OLB;40
                          002760 000216 00142. CBTA   04.3   SYSLIB.OLB;40
                          003176 000126 00086. CDDMG  00     SYSLIB.OLB;40
                          003324 000074 00060. CATB   03     SYSLIB.OLB;40
                          003420 000110 00072. C5TA   02     SYSLIB.OLB;40
                          003530 000246 00166. EDDAT  02     SYSLIB.OLB;40
AADD  :(RO,I,GBL,REL,CON) 060000 000024 00020.
                          060000 000024 00020. LIB    01     LIB.STB;13
DIVV  :(RO,I,GBL,REL,CON) 060024 000026 00022.
                          060024 000026 00022. LIB    01     LIB.STB;13
LNC$D :(RW,D,GBL,REL,CON) 003776 000002 00002.
                          003776 000002 00002. EDTMG  12     SYSLIB.OLB;40
MULL  :(RO,I,GBL,REL,CON) 060052 000024 00020.
                          060052 000024 00020. LIB    01     LIB.STB;13
SAVAL :(RO,I,GBL,REL,CON) 060076 000042 00034.
                          060076 000042 00034. LIB    01     LIB.STB;13
SUBB  :(RO,I,GBL,REL,CON) 060140 000024 00020.
                          060140 000024 00020. LIB    01     LIB.STB;13
$$RESL:(RW,I,LCL,REL,CON) 004000 000024 00020.
                          004000 000024 00020. SAVRG  03     SYSLIB.OLB;40
$$RESM:(RW,I,LCL,REL,CON) 004024 000166 00118.
                          004024 000066 00054. ARITH  03.02  SYSLIB.OLB;40
                          004112 000100 00064. DARITH 0005   SYSLIB.OLB;40



GLOBAL SYMBOLS:

AADD   060000-R SAVAL  060076-R $CBDSG 002774-R $CBTMG 003016-R
DIVV   060024-R SUBB   060140-R $CBOMG 003002-R $CBVER 003002-R
IO.WVB 011000   $CBDAT 002760-R $CBOSG 003010-R $CDDMG 003176-R
MULL   060052-R $CBDMG 002766-R $CBTA  003040-R $CDTB  003324-R


MAIN.TSK;15     MEMORY ALLOCATION MAP  TKB M36         PAGE 2
MAIN                  30-APR-79   10:33



$COTB  003332-R $DDIV  004150-R $EDMSG 002036-R $SAVRG 004000-R
$C5TA  003420-R $DIV   004054-R $LNCNT 003776-R $TIM   003652-R
$DAT   003574-R $DMUL  004112-R $MUL   004024-R


*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 2518.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 8200. WORDS (32. PAGES)
    SIZE OF WORK FILE: 1024. WORDS (4. PAGES)

    ELAPSED TIME:00:00:19
```

Figure 3-20 (Cont.) Task Builder Map for MAIN.TSK

This map contains a global symbols section. Note that the symbols within the library now have virtual addresses assigned to them and that these addresses begin at 60000(8) -- the virtual base address of APR 3. The Task Builder's allocation of virtual address space for MAIN.TSK is represented diagrammatically in Figure 3-21.

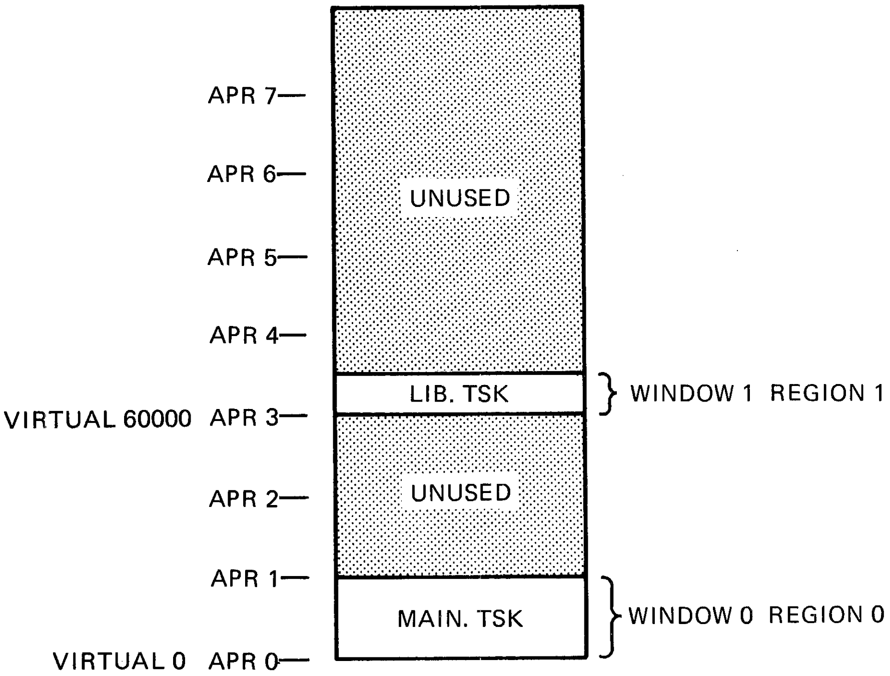


Figure 3-21 Allocation of Virtual Address Space for MAIN.TSK

The library LIB is position independent and can therefore be mapped anywhere in the referencing task's virtual address space. APR 3 was used in this example to contrast this mapping arrangement with the mapping of MACCOM in the virtual address space of task MCOM1 in Example 1 (Section 3.1.5). If the optional APR parameter in the RESLIB option above had been left blank, the Task Builder would have allocated the highest available APR to map the library.

As described in earlier sections of this chapter, program section names within position-independent shared regions must normally be unique with respect to program section names within tasks that reference them. When a shared region is a position-independent resident common and you explicitly declare the program section names within it, avoiding program section name conflicts is an easy matter. However, when a shared region is a position-independent resident library that contains calls to routines within an object module library (SYSLIB, for example), conflicts may develop that are not apparent to you. The problem arises when the position-independent resident library and one or more tasks that link to it contain calls to separate routines residing within the same program section of an object module library.

When the Task Builder resolves a call from within a module it is processing to a routine within an object module library, it places the routine from the library into the image it is building. It also enters into its internal table the name of the program section in the object module library within which the routine resides. If a position-independent resident library contains a call to a routine within a given program section of SYSLIB, for example, and then subsequently a task that links to the resident library contains a call to a different routine within the same program section of SYSLIB, both the resident library and the referencing task will contain the program section name. When you build the referencing task, the library's .STB file will contain the program section name and a program section conflict will develop. (Refer to Section 3.1.4 for additional information on the sequence in which the Task Builder processes tasks and the potential program section name conflicts that can result.)

This situation and one possible solution to it can be illustrated with Example 3. When this example was first created, only the arithmetic routines were included in the source file of the resident library (LIB.MAC in Figure 3-17). The system library coroutine ($SAVAL) was resolved from SYSLIB. Because the first instruction of each arithmetic routine called $SAVAL, the Task Builder included a copy of it in the resident library's image at task-build time. This turned out to be unsatisfactory because of a call to the SYSLIB routine $EDMSG (edit message) within the program MAIN that links to the resident library. Both routines ($SAVAL and $EDMSG) reside within the unnamed or blank program section (. BLK.) within SYSLIB. Therefore, a program section name conflict developed when MAIN was built.

To circumvent this problem, the source code for $SAVAL was included into the source file for the resident library under the explicitly declared program section name, SAVAL.

Another solution would have been to build the resident library absolute. In this case, the Task Builder would not have included program section names from the resident library into the symbol definition file for the library when the library was built.

It is important to note that the above program section name conflict develops only when two different routines residing within the same program section of an object module library are involved. It presents no problem when a resident library and a task that links to it contain a call to the same routine in an object module library. In that case, the Task Builder copies the routine and the program section name in which it resides into the resident library when the library is built. Then, when the task that calls the same routine is built, the Task Builder will resolve the reference to the routine in the resident library instead of in the object module library.

3.1.8 **Example 4: Building and Linking to a Supervisor-Mode Library in MACRO-11 (RSX-11M-PLUS Only)**

Supervisor-mode libraries are a special type of resident library that provide you with the means to effectively double the address space of your task and thereby extend the physical memory to which your task has access. Supervisor-mode libraries are particularly useful when used to accommodate large run-time systems.

Supervisor-mode libraries are mapped with the instruction space APRs of the processor's supervisor mode (Supervisor APR 0 through Supervisor APR 7). Once you have linked your task to a supervisor-mode library, a call from within your task to a global symbol within the library automatically causes a context switch from user mode to supervisor mode. Control of the processor is then assumed by the called library routine. When the library routine executes a return, control of the processor is transferred to a completion routine within the library. It is the completion routine's responsibility to perform the return context switch from supervisor mode to user mode. (Completion routines are described later in this section.)

When you build a task that links to a supervisor-mode library, the Task Builder replaces each call from the task to a routine within the library with a 4-word vector. This vector contains a transfer of control instruction to a routine ($SUPL) that switches the processor from user mode to supervisor mode. It also contains the address of the completion routine and the address of the entry point of the called library routine. (Refer to Figure B-14 in Appendix B.)

Figure 3-22 shows a typical mapping arrangement for a 14K word supervisor-mode library and a 24K word task that refers to it.

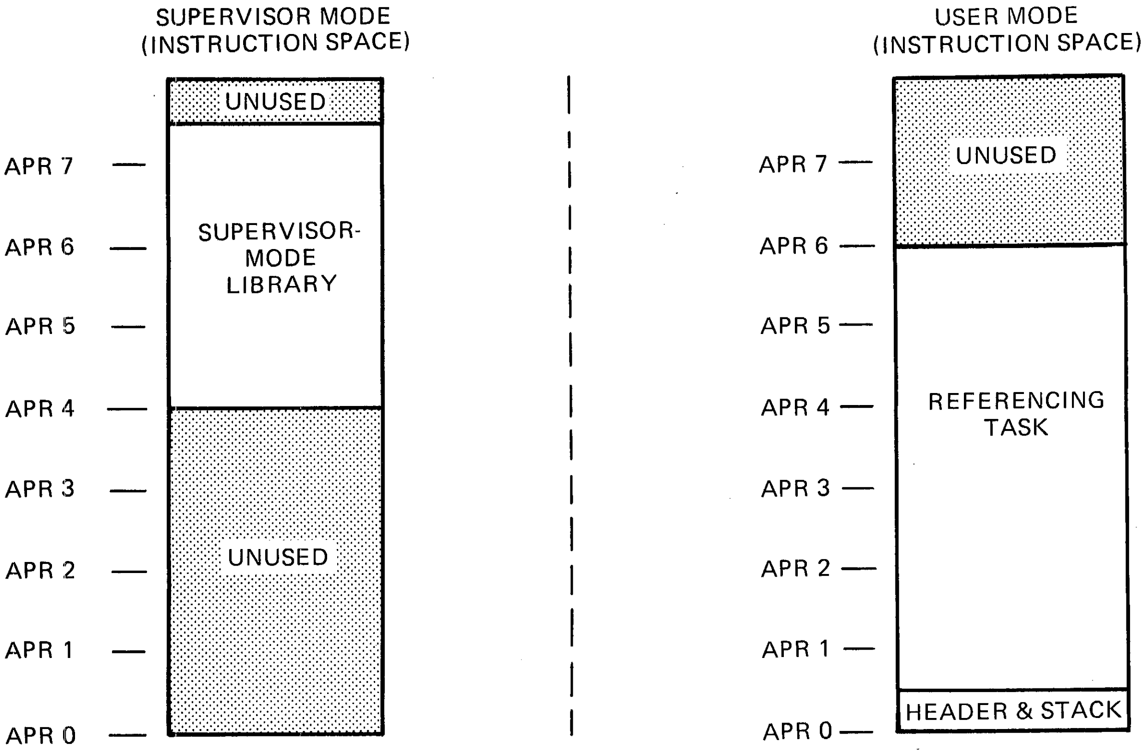


Figure 3-22 Typical Mapping for Supervisor-Mode Library

When the processor context switches from user mode to supervisor mode, the system copies the user-mode instruction APRs (which map the task image) into the supervisor-mode data space APRs. Therefore, when the processor is running in supervisor mode under control of a library routine, any data within the task image is available to the routine. For example, library routines that require parameters or pointers to parameters in registers or through low core impure area pointers can obtain the parameters transparently. Figure 3-23 shows a typical mapping arrangement when a supervisor-mode routine is in control of the processor.

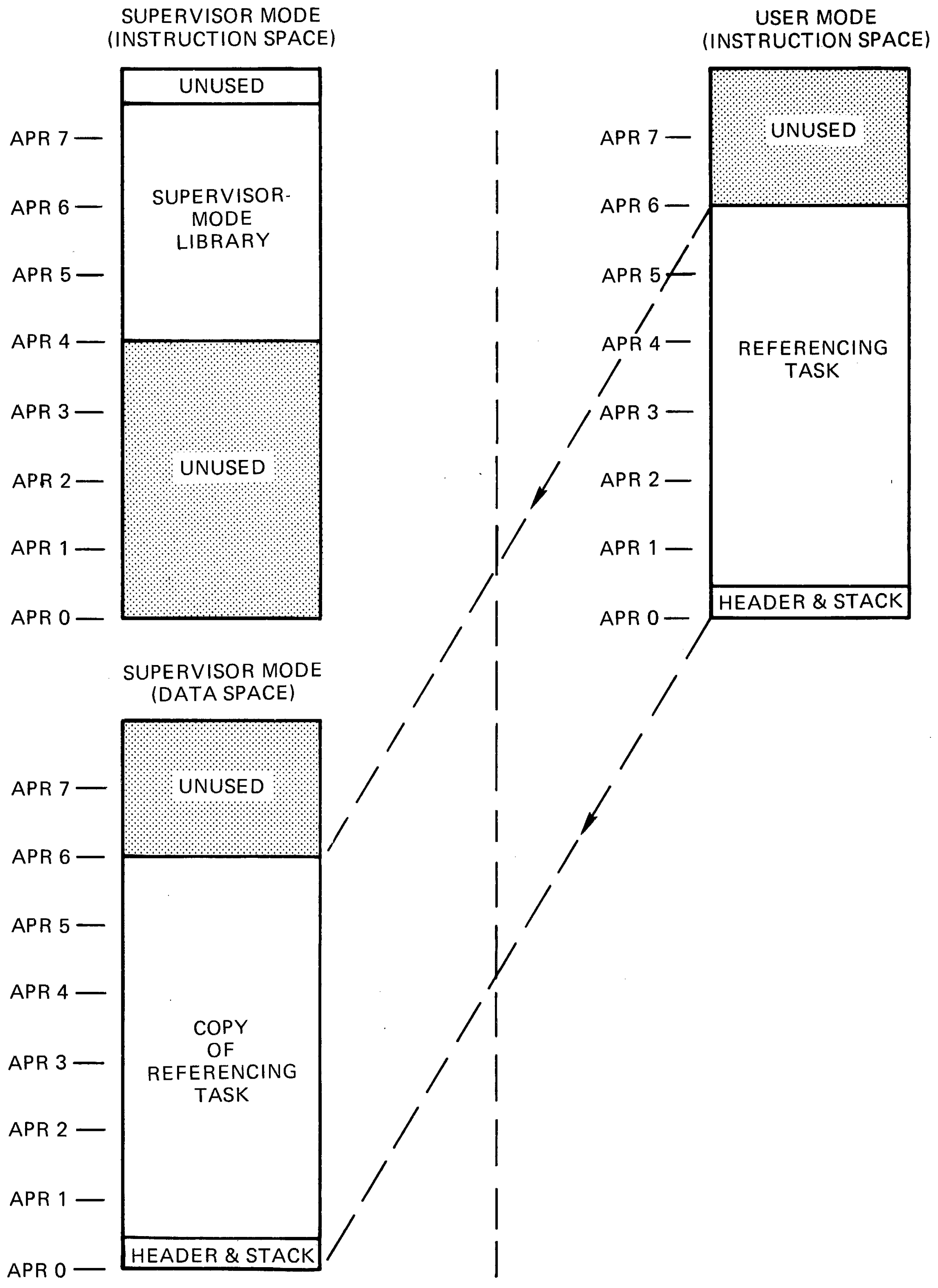


Figure 3-23 Task Mapping while Running in Supervisor Mode

Building a supervisor-mode library is essentially the same as building a conventional resident library. When you build a supervisor-mode library, you suppress the header by attaching /-HD to the task image file. During option input, you suppress the stack area by specifying STACK=0. You specify the partition in which the library is to reside and, optionally, the base address and length of the library with the PAR option.

You indicate to the Task Builder that you are building a supervisor-mode library with the CMPRT option. The argument for this option identifies the entry symbol of the completion routine. When the Task Builder processes this option, it places the completion routine entry point in the library's .STB file. (Refer to Chapter 6 for more information on the CMPRT option).

The following restrictions are placed on the contents of a supervisor-mode library.

1. Only subroutines using JSR PC, X should be used within the library.

2. The library must not contain subroutines that use the stack to pass parameters if tasks referring to the library call the same routines.

3. The library must not contain data of any kind. This includes: user data, buffers, I/O status blocks, and directive parameter blocks (the $S directive form can be used because the directive parameter block for this form of directive is pushed onto the stack at run time).

When you build a supervisor-mode library, you must include within it a completion routine that performs the following:

1. Transfers any condition code bits that are relevant to your user-mode task from the Processor Status Word (PSW) to the Processor Status Word on the stack. All condition code bits in the stacked PSW are set to 0 during the context switch from user to supervisor-mode.

2. Writes an appropriate value into the user stack pointer, because the user stack will not be context switched.

3. Executes an RTI instruction.

Following is an example of a completion routine from the system library, LB:[1,1]SYSLIB.OLB which returns the carry bit:

```
$COMPL::    ADC     2(SP)               ;TRANSFER CARRY BIT
            MOV     #6,-(SP)            ;CALCULATE USER SP VALUE
            ADD     SP,(SP)             ;
            MTPI    SP                  ;CHANGE USER STACK POINTER VALUE
            RTI                         ;RETURN TO CALLER
```

The system library contains two other completion routines:

- $CMPAL -- which returns status bits NZVC
- $CMPRV -- which sets up PS for privileged tasks

Figure 3-24 shows a module containing three routines: a sort routine (SORT::), a search routine (SEARCH::), and a completion routine ($COMPL::). When assembled and built, these routines will constitute an example of a supervisor-mode library.

```
        .TITLE  SUPLIB
        .IDENT  /01/

SORT::
        CALL    $SAVAL                  ; SAVE ALL REGISTERS
        TST     (R5)+                   ; SKIP OVER NUMBER OF ARGUMENTS
        MOV     (R5)+,R0                ; GET ADDRESS OF LIST
        MOV     (R5)+,R4                ; GET ADDRESS OF LENGTH OF LIST
        MOV     (R4),R4                 ; GET LENGTH OF LIST
        MOV     R0,R5                   ;
        DEC     R4                      ;
10$:
        MOV     R5,R0                   ; COPY
        MOV     R4,R3                   ; COPY LENGTH OF LIST
20$:
        TST     (R0)+                   ; MOVE POINTER TO NEXT ITEM
        CMP     (R5),(R0)               ; COMPARE ITEMS
        BLE     30$                     ; IF LE IN CORRECT ORDER
        MOV     (R5),R2                 ; SWAP ITEMS
        MOV     (R0),(R5)               ;
        MOV     R2,(R0)                 ;
30$:
        DEC     R3                      ; DECREMENT LOOP COUNT
        BNE     20$                     ; IF NE LOOP
        DEC     R4                      ; DECREMENT
        BEQ     40$                     ; IF EQ SORT COMPLETED
        TST     (R5)+                   ; GET POINTER TO NEXT ITEM TO BE COMPARED
        BR      10$
40$:
        RETURN

SEARCH::
        CALL    $SAVAL                  ; SAVE ALL THE REGISTERS
        CMP     #4,(R5)+                ; FOUR ARGUMENTS?
        BNE     20$                     ; IF NE NO
        MOV     (R5)+,R0                ; GET ADDRESS OF NUMBER TO LOCATE
        MOV     (R5)+,R1                ; ADDRESS OF LIST SEARCHING
        MOV     (R5)+,R2                ; GET ADDRESS OF LENGTH OF LIST
        MOV     (R2),R2                 ; GET LENGTH OF LIST
        MOV     (R5),R5                 ; ADDRESS OF RETURNED VALUE
        MOV     R2,R3                   ; COPY LENGTH
10$:
        CMP     (R0),(R1)+              ; IS THIS THE NUMBER?
        BEQ     30$                     ; IF EQ YES
        BMI     20$                     ; IF MI NUMBER NOT THERE
        DEC     R2                      ; DECREMENT LOOP COUNT
        BNE     10$                     ; IF NE NOT AT END OF LIST
20$:
        MOV     #-1,(R5)                ; END OF LIST PASS BACK ERROR
        RETURN
30$:
        SUB     R2,R3                   ; NUMBER FOUND - GET INDEX INTO LIST
        INC     R3                      ;
        MOV     R3,(R5)                 ; RETURN INDEX
        RETURN
```

Figure 3-24 Source Listing for SUPLIB.MAC

```
$COMPL::
        ADC     2(SP)           ; COMPLETION ROUTINE
        MOV     #6,-(SP)        ;
        ADD     SP,(SP)         ;
        MTPI    SP              ;
        RTI                     ; RETURN TO USER MODE

        .END
```

Figure 3-24 (Cont.) Source Listing for SUPLIB.MAC

Once you have assembled SUPLIB, you can build it with the following Task Builder command string:

```
TKB>SUPLIB/-HD,SUPLIB/MA/-SP,SUPLIB=SUPLIB
TKB>/
ENTER OPTIONS:
TKB>STACK=0
TKB>CMPRT=$COMPL
TKB>PAR=SUPLIB:160000:2000
TKB>GBLXCL=$SAVAL
TKB>//
```

This command sequence directs the Task Builder to create a headerless image file (/-HD) named SUPLIB.TSK and to create an extended map file (/MA) named SUPLIB.MAP. Because /-SP is appended to the map file, the Task Builder will not output it to the line printer. It also specifies that the Task Builder is to create a .STB file named SUPLIB.STB.

Under options, STACK=0 suppresses the stack area in SUPLIB's task image. The CMPRT option identifies the global symbol of the completion routine within SUPLIB. The Task Builder will place this global symbol in a special entry in the library's .STB file. The PAR option specifies to the Task Builder that SUPLIB will reside within a partition of the same name as the library. This is not a requirement. The PAR option also specifies to the Task Builder that SUPLIB will have a base address of 160000, and that it will be 2000(8) bytes long.

The GBLXCL option directs the Task Builder to exclude from SUPLIB's .STB file the global symbol $SAVAL. $SAVAL is a system library coroutine that saves the contents of all general registers. It uses the stack to pass parameters. When SUPLIB is built, the Task Builder will resolve the reference to $SAVAL by including the $SAVAL routine into SUPLIB's image file. Since it is known that the task TSUP (described below) will be linked to SUPLIB at a later time, and that TSUP also contains a call to $SAVAL, the symbol $SAVAL must be excluded from SUPLIB's .STB file .o prevent the Task Builder from resolving the call in TSUP to the $SAVAL routine in SUPLIB. The net result of excluding the symbol from SUPLIB's .STB file is that the Task Builder will include separate copies of $SAVAL in SUPLIB and in the task that links to it TSUP. (For more information on the switches and options used in this example, refer to Chapter 6.)

Suppose the symbol $SAVAL were not excluded from SUPLIB's .STB file. When the Task Builder built TSUP, instead of resolving the reference to SYSLIB, it would resolve the reference to the routine existing within SUPLIB. When TSUP ran, the call to $SAVAL within it would cause a context switch from user mode to supervisor mode. $SAVAL would execute but, because it is a coroutine, it would attempt a direct call back to TSUP (which resides in user mode) instead of returning to user mode through a completion routine.

This illegal return call would fail and cause the system to trap.

Note also, that SUPLIB is built absolute. That is, the PI (position-independent) switch is not attached to the library image file. As written, SUPLIB must be absolute to prevent a program section name conflict between SUPLIB and the task that links to it, TSUP. Even though the symbol $SAVAL was excluded from SUPLIB's .STB file, the program section in which $SAVAL resides in SYSLIB was not. When the Task Builder resolves the reference to $SAVAL in TSUP, it will place the routine into TSUP's image file and the program section name in which it resides will be placed into the Task Builder's internal section table. If, when TSUP is built, the program section name were allowed to remain in the .STB file, a conflict would develop. (Refer to Section 3.1.4 for additional information.)

The map that results from the above command sequence is shown in Figure 3-25. Note that the virtual addresses for the symbols SEARCH, SORT, $COMPL, and the entry point for the system library coroutine $SAVAL have already been established. The Task Builder establishes the virtual addresses for these symbols when it builds SUPLIB because SUPLIB is absolute. If SUPLIB were built position independent, the Task Builder would defer the assignment of virtual addresses for these symbols until the tasks that link to the library are built.

The program TSUP in Figure 3-26 uses the sort and search routines in the example in Figure 3-24. When you link TSUP to SUPLIB, install and run it, TSUP will prompt for a number by printing ARRAY= on your terminal. When you type a number, TSUP will place the number you have typed into an array and prompt you again in the same manner. It will continue to prompt you until it has prompted you 10 times or until you type a 0, whichever comes first. After you have input 10 numbers or typed a 0, TSUP will use the sort routine in SUPLIB to sort the numbers in ascending order. It will then print them on your console terminal. Once it has printed the numbers, it will prompt for a number with the following message:

```
     NUMBER TO SEARCH FOR?
```

When you respond by typing a number, TSUP will use the search routine in SUPLIB to search the array for the number. If the number is in the array, TSUP will print it on your terminal. If the number is not in the array, TSUP will report that fact.

TSUP uses three system library routines: $SAVAL (save all registers coroutine), $EDMSG (edit message routine), and $CDTB (decimal to binary conversion routine). These routines are described in the IAS/RSX-11 System Library Routines Reference Manual. The Executive directives used by TSUP (QIOW$, DIR$, and QIOW$S) are described in the RSX-11M/M-PLUS Executive Reference Manual.

```
SUPLIB.TSK;17  MEMORY ALLOCATION MAP   TKB M35         PAGE 1
                29-DEC-78   09:18



PARTITION NAME : SUPLIB                                          ┐
IDENTIFICATION : 01                                              │
TASK  UIC      : [301,356]                                       │
TASK ATTRIBUTES: -HD                                             │TASK
TOTAL ADDRESS WINDOWS: 1.                                        │ATTRIBUTES
TASK  IMAGE  SIZE  : 96. WORDS                                   │SECTION
TASK ADDRESS LIMITS: 160000 160213                               │
R-W DISK BLK LIMITS: 000003 000002 000000 00000.                 ┘

*** ROOT SEGMENT: SUPLIB


R/W MEM  LIMITS: 160000 160213 000214 00140.
DISK BLK LIMITS: 000002 000002 000001 00001.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                            TITLE  IDENT  FILE
-------                                            -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 160000 000214 00140.
                          160000 000152 00106. SUPLIB 01     SUPLIB.OBJ;1
                          160152 000042 00034. SAVAL  00     SYSLIB.OLB;6



GLOBAL SYMBOLS:

SEARCH 160056-R  SORT   160000-R  $COMPL 160134-R  $SAVAL 160152-R



*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 229.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 2076. WORDS (8. PAGES)
    SIZE OF WORK FILE: 768. WORDS (3. PAGES)

    ELAPSED TIME:00:00:05
```

Figure 3-25 Task Builder Map for SUPLIB.TSK

```
         .TITLE  TSUP
         .IDENT  /01/


         .MCALL  QIOW$,DIR$,QIOW$S

WRITE:   QIOW$   IO.WVB,5,1,,,,<OUT,,40>
READIN:  QIOW$   IO.RVB,5,1,,,,<OUT,5>


IARRAY:  .BLKW   12.
LEN:     .BLKW   1
IART:    .BLKW   1
INDEX:   .BLKW   1
OUT:     .BLKW   100.
ARGBLK:
EDBUF:   .BLKW   10.

FMT1:    .ASCIZ  /%2SARRAY(%D)=/
FMT2:    .ASCIZ  /%N%2SNUMBER TO SEARCH FOR?/
FMT3:    .ASCIZ  /%N%2S%D WAS FOUND IN ARRAY(%D)/
FMT4:    .ASCIZ  /%N%2S%D WAS NOT IN ARRAY/
FMT5:    .ASCIZ  /%2SARRAY (%D)=%D/

         .EVEN
START:
         MOV     #IARRAY,R0       ; GET ADDRESS OF ARRAY
         MOV     #10,R1           ; SET LENGTH OF ARRAY
5$:
         CLR     (R0)+            ; INITIALIZE ARRAY
         DEC     R1               ; LOOP
         BNE     5$
         MOV     #IARRAY,R0       ;
         MOV     #INDEX,R2
10$:
         MOV     #FMT1,R1         ; FORMAT SPECIFICATION (ADDRESS
                                  ; OF INPUT STRING)
         MOV     (R2),EDBUF       ; GET INDEX
         INC     EDBUF            ;
         CALL    PRINT            ; PRINT MESSAGE
         CALL    READ             ; READ INPUT
         MOV     IART,(R0)+       ; PUT BINARY KEYBOARD INPUT INTO ARRAY
         BEQ     20$              ; ZERO MARKS END OF INPUT
         INC     (R2)             ;
         CMP     (R2),#10.
         BNE     10$              ; IF NE YES
20$:
         MOV     (R2),LEN         ; CALCULATE LENGTH OF ARRAY
         MOV     #ARGBLK,R5       ; GET ADDRESS OF ARGUMENT BLOCK
         MOV     #2,(R5)+         ; NUMBER OF ARGUMENTS
         MOV     #IARRAY, (R5)+   ; PUT ADDRESS OF ARRAY
         MOV     #LEN,(R5)        ;
         MOV     #ARGBLK,R5       ;
         CALL    SORT             ; SORT ARRAY
         CLR     R2               ;
         MOV     #IARRAY,R0       ; GET ARRAY ADDRESS
```

Figure 3-26 Source Listing for TSUP.MAC

```
30$:
        INC     R2                  ; INCREMENT INDEX
        MOV     R2,EDBUF            ; GET INDEX FOR PRINT
        MOV     (R0)+,EDBUF+2       ; GET CONTENTS OF ARRAY
        MOV     #FMT5,R1            ; GET ADDRESS OF FORMAT SPECIFICATION
        CALL    PRINT               ;
        CMP     R2,LEN              ; MORE TO PRINT?
        BLT     30$                 ; IF LE YES
        MOV     #FMT2,R1            ; GET ADDRESS OF FORMAT SPECIFICATION
        CALL    PRINT               ; OUTPUT MESSAGE
        CALL    READ                ; READ RESPONSE
        MOV     #ARGBLK,R5          ;
        MOV     #4,(R5)+            ; SET NUMBER OF ARGUMENTS
        MOV     #IART,(R5)+         ; SET ADDRESS OF NUMBER LOOKING FOR
        MOV     #IARRAY,(R5)+       ; SET ADDRESS OF ARRAY
        MOV     #LEN,(R5)+          ; SET ADDRESS OF LEN OF ARRAY
        MOV     #INDEX,(R5)         ; ADDRESS OF RESULT
        MOV     #ARGBLK,R5          ;
        CALL    SEARCH              ; SEARCH FOR NUMBER IN IART
        TST     INDEX               ; WAS NUMBER FOUND?
        BLT     40$                 ; IF LT NO
        MOV     IART,EDBUF          ; GET NUMBER LOOKING FOR
        MOV     INDEX,EDBUF+2       ; GET ARRAY NUMBER
        MOV     #FMT3,R1            ; GET FORMAT ADDRESS
        CALL    PRINT               ;
        BR      100$                ; DONE
40$:
        MOV     #FMT4,R1            ; GET FORMAT ADDRESS
        MOV     IART,EDBUF          ; GET NUMBER
        CALL    PRINT
100$:
        CALL    $EXST               ; EXIT WITH STATUS

PRINT:
        CALL    $SAVAL              ; SAVE ALL REGISTERS
        MOV     #OUT,R0 ; ADDRESS OF OUTPUT BLOCK
        MOV     #EDBUF,R2           ; START ADDRESS OF ARGUMENT BLOCK
        CALL    $EDMSG              ; FORMAT MESSAGE
        MOV     R1,WRITE+Q.IOPL+2 ; PUT LENGTH OF OUTPUT
                                    ; BLOCK INTO PARAMETER BLOCK
        DIR$    #WRITE              ; WRITE OUTPUT BLOCK
        RETURN


READ:
        CALL    $SAVAL              ; SAVE ALL REGISTERS
        DIR$    #READIN ; READ REQUEST
        MOV     #OUT,R0 ; GET KEYBOARD INPUT
        CALL    $CDTB               ; CONVERT KEYBOARD INPUT TO BINARY
        MOV     R1,IART             ; PUT INPUT INTO BUFFER
        RETURN

        .END    START
```

Figure 3-26 (Cont.) Source Listing for TSUP.MAC

Once you have assembled TSUP, you can build it with the following Task Builder command sequence:

```
TKB>TSUP,TSUP/MA/-WI/-SP=TSUP
TKB>/
ENTER OPTIONS:
TKB>RESSUP=SUPLIB/SV
TKB>//
```

This command sequence directs the Task Builder to build a task image file TSUP.TSK and to create an extended (/MA) 80 column (/-WI) map file named TSUP.MAP. Because /-SP is appended to the map file, the Task Builder will not output the map file to the line printer.

Under options, the RESSUP option tells the Task Builder that the task intends to access a supervisor-mode library and that context switching vectors are required. The Task Builder expects to find a library image file SUPLIB.TSK and a symbol definition file SUPLIB.STB, on device LB: under the UFD that corresponds to the terminal UIC. In addition, the Task Builder expects to find a special entry in the .STB file that contains the symbol definition for the supervisor-mode library's completion routine ($COMPL). This entry was created by the Task Builder when SUPLIB was built as a result of the CMPRT option. (Refer to Chapter 6 for more information on the switches and options used in this example.) A portion of the map that results from the Task Builder command sequence above is shown in Figure 3-27.

Note under global symbols in Figure 3-27 that the Task Builder has changed the virtual addresses for symbols SEARCH, SORT, and $SAVAL while leaving the virtual address of symbol $COMPL the same as it was when SUPLIB was built. The new virtual addresses for the first three symbols are addresses to the context switching vectors that the Task Builder placed in TSUP's code. The Task Builder did not change the virtual address of $COMPL because it is referenced only from within the library. Therefore, calls to it do not constitute an initial processor-mode context switch.

Finally, note that building a resident library as a supervisor-mode library in no way precludes its use as a "standard" user-mode resident library. A given resident library might be mapped by one task as a supervisor-mode library while simultaneously being mapped by another as a user-mode library.

```
TSUP.TSK;6      MEMORY ALLOCATION MAP  TKB M35          PAGE 1
                       29-DEC-78   16:39



PARTITION NAME : GEN                                       ┐
IDENTIFICATION : 01                                        │
TASK  UIC      : [301,356]                                 │
STACK    LIMITS: 000212 001211 001000 00512.               │ TASK
PRG XFR ADDRESS: 002046                                    │ ATTRIBUTES
TOTAL ADDRESS WINDOWS: 2.                                  │ SECTION
TASK  IMAGE  SIZE  : 1312. WORDS                           │
TASK ADDRESS LIMITS: 000000 005017                         │
R-W DISK BLK LIMITS: 000002 000007 000006 00006.          ─┘

*** ROOT SEGMENT: TSUP


R/W MEM  LIMITS: 000000 005017 005020 02576.
DISK BLK LIMITS: 000002 000007 000006 00006.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                             TITLE  IDENT  FILE
-------                                             -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001212 003312 01738.
                          001212 001234 00668.  MAIN   01     TSUP.OBJ;1
                          .
                          .
                          .
$$SUPL:(RW,I,LCL,REL,CON) 004760 000040 00032.
                          004760 000040 00032.  $SUPL  01     SYSLIB.OLB;6



GLOBAL SYMBOLS:

IO.RVB 010400     $CBOSG 003632-R  $C5TA  004146-R  $MUL   004552-R
IO.WVB 011000     $CBTA  003662-R  $DAT   004322-R  $SAVAL 003540-R
SEARCH 004740-R   $CBTMG 003640-R  $DDIV  004676-R  $SAVRG 004526-R
SORT   004750-R   $CBVER 003624-R  $DIV   004602-R  $SUPL  004766-R
$CBDAT 003602-R   $CDDMG 004020-R  $DMUL  004640-R  $TIM   004400-R
$CBDMG 003610-R   $CDTB  002446-R  $EDMSG 002632-R
$CBDSG 003616-R   $COMPL 160134    $EXST  003522-R
$CBOMG 003624-R   $COTB  002454-R  $LNCNT 004524-R



*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 2544.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 2076. WORDS (8. PAGES)
    SIZE OF WORK FILE: 1024. WORDS (4. PAGES)

    ELAPSED TIME:00:00:12
```

Figure 3-27 Task Builder Map for TSUP.TSK

3.2 EXAMPLE 5: BUILDING A MULTIUSER TASK (RSX-11M-PLUS ONLY)

A multiuser task is a task that shares the pure (read-only) portion of its code with two or more copies of the impure (read/write) portion of its code. When the system receives an initial run request for a multiuser task, a copy of both the read-only and read/write portions of the task are read into physical memory. As long as the task is running, all subsequent run requests for it result in the system duplicating only the read/write portion of the task in physical memory. Thus, multiuser tasks are memory efficient.

You designate a task as multiuser when you build it by applying the MU switch to the task image file. This switch directs the Task Builder to create two regions for the task. One region (region 0) will contain the read-write portion of the task; the other region (region 1) will contain the read-only portion of the task.

As with all other tasks, the Task Builder uses a program section's access code to determine its placement within a multiuser task's image. It divides address space into read/write and read-only sections. Unlike a single user task, however, in a multiuser task, the read-only portion of the task is hardware protected. In addition, the Task Builder separates the read/write portions of a multiuser task from the read-only portions and places them in separate regions at opposite ends of the task's address space. It allocates the low address APRs to the read/write portion (which includes the task's header and stack area) and the highest available APRs to the read-only portion. Figure 3-28 illustrates this allocation.

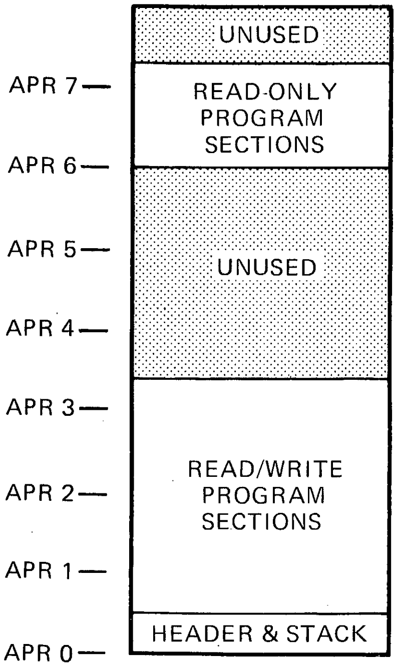


Figure 3-28 Allocation of Program Sections in a Multiuser Task

If neither the read-only nor the read/write portion of the task contain memory-resident overlays the Task Builder will allocate two window blocks in the header of the task. When the task is installed, the INSTALL processor will initialize these window blocks as follows:

- Window block 0 will describe the range of virtual addresses (the window) for the read/write portion of the task. This region will always contain the task's header.
- Window block 1 will describe the range of virtual addresses for the read-only portion.

Figure 3-29 below shows the window-to-region relationship of a multiuser task.

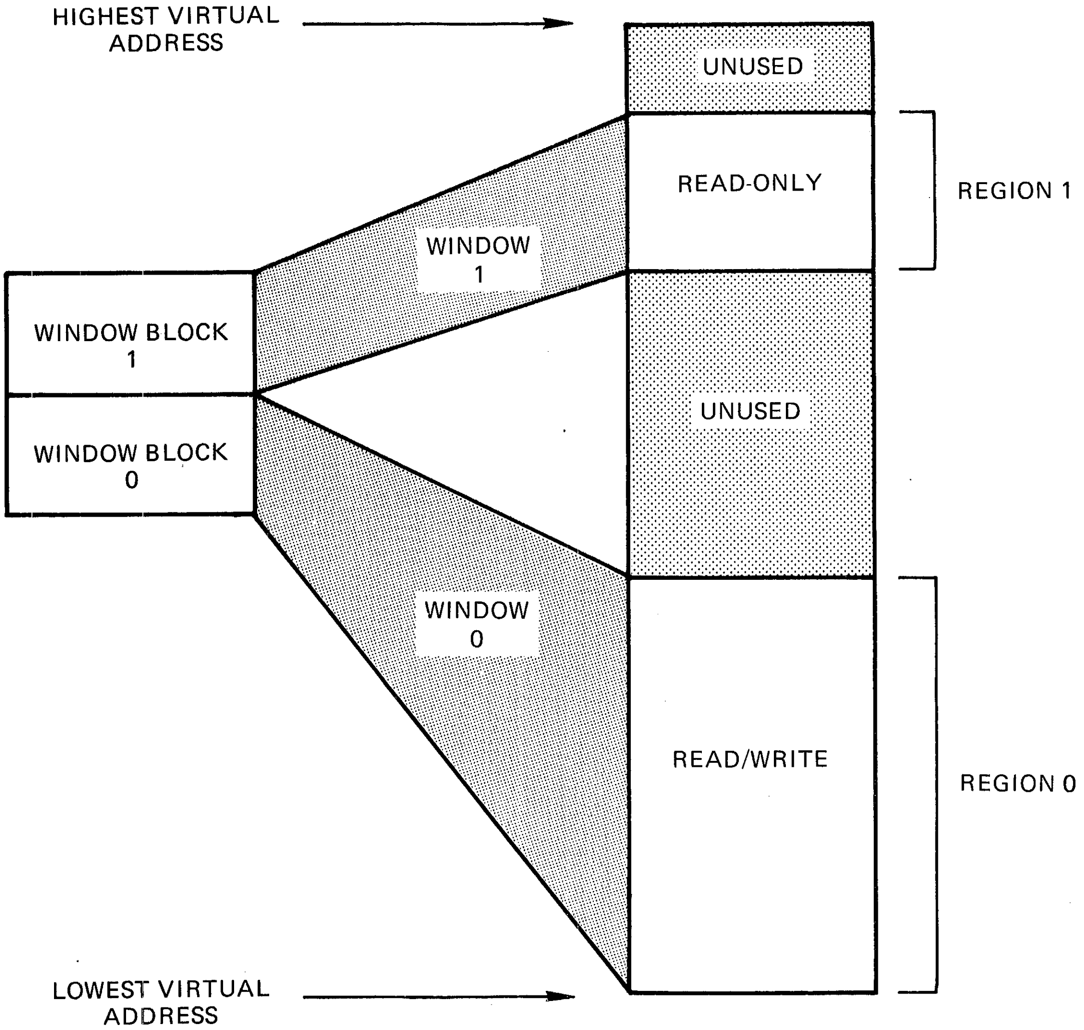


Figure 3-29 Windows for a Multiuser Task

If a multiuser task is an overlaid task, the read-only portion of the task can be made up of the following:

- The read-only program sections of the root segment

- Branches of an overlay structure if the complete branch is memory resident and read-only

- A co-tree structure if the entire co-tree is memory resident and read-only.

(Overlaid tasks are described in Chapter 4.)

Finally, the disk image of a multiuser task is somewhat different from that of a single-user task. The read-only portion of the task is placed at the end of the disk image. The relative block number of the read-only portion and the number of blocks it occupies appears in the label block. The read-only portion of the image is described in the first library descriptor of the LIBRARY REQUEST section of the label block. (Refer to Appendix B for more information on the task image data structures.

The remainder of the text in this section and the figures associated with it illustrate the development of a multiuser task. This example was created by concatenating into a single file the resident library file (LIB.MAC) and the task that links to it (MAIN.MAC) from Example 4. It is not intended to represent a typical multiuser task application. However, it does illustrate the Task Builder's allocation of program sections in a multiuser task and that is its primary value. The concatenated source file, named ROTASK.MAC, for this example is shown in Figure 3-30.

```
        .TITLE  ROTASK
        .IDENT  /01/


        .MCALL  QIOW$S,EXIT$S

OP1:    .WORD   1                       ; OPERAND 1
OP2:    .WORD   1                       ; OPERAND 2
ANS:    .BLKW   1                       ; RESULT

OUT:    .BLKW   100.                    ; FORMAT MESSAGE
FORMAT: .ASCIZ  /THE ANSWER = %D,/
        .EVEN

START:
        MOV     #ANS,-(SP)              ; TO CONTAIN RESULT
        MOV     #OP2,-(SP)              ; OPERAND 2
        MOV     #OP1,-(SP)              ; OPERAND 1
        MOV     #3  ,-(SP)              ; PASSING 3 ARGUMENTS
        MOV     SP,R5                   ; ADDRESS OF ARGUMENT BLOCK
        CALL    AADD                    ; ADD TWO OPERANDS
        CALL    PRINT                   ; PRINT RESULTS
        MOV     SP,R5                   ; ADDRESS OF ARGUMENT BLOCK
        CALL    SUBB                    ; SUBTRACT SUBROUTINE
        CALL    PRINT                   ; PRINT RESULTS
        MOV     SP,R5                   ; ADDRESS OF ARGUMENT BLOCK
        CALL    MULL                    ; MULTIPLY SUBROUTINE
        CALL    PRINT                   ; PRINT RESULTS
        MOV     SP,R5                   ; ADDRESS OF ARGUMENT BLOCK
        CALL    DIVV                    ; DIVIDE SUBROUTINE
        CALL    PRINT                   ; PRINT RESULTS
        EXIT$S
```

Figure 3-30 Source Listing for ROTASK.MAC

```
;+
;** PRINT - PRINT RESULT OF OPERATION.
;-

PRINT:  MOV     #OUT,R0                   ; ADDRESS OF SCRATCH AREA
        MOV     #FORMAT,R1                ; FORMAT SPECIFICATION
        MOV     #ANS,R2                   ; ARGUMENT TO CONVERT
        CALL    $EDMSG                    ; FORMAT MESSAGE
        QIOW$S  #IO.WVB,#5,#1,,,,<#OUT,R1,#40>
        RETURN                            ; RETURN FROM SUBROUTINE


;** FORTRAN CALLABLE SUBROUTINE TO ADD TWO INTEGERS


        .PSECT  AADD,RO,I,GBL,REL,CON

AADD::  CALL    $SAVAL           ; SAVE R0-R5
        MOV     @2(R5),R0        ; FIRST OPERAND
        MOV     @4(R5),R1        ; SECOND OPERAND
        ADD     R0,R1            ; SUM THEM
        MOV     R1,@6(R5)        ; STORE RESULT
        RETURN                   ; RESTORE REGISTERS AND RETURN

;** FORTRAN CALLABLE SUBROUTINE TO SUBTRACT TWO INTEGERS


        .PSECT  SUBB,RO,I,GBL,REL,CON


SUBB::  CALL    $SAVAL           ; SAVE R0-R5
        MOV     @2(R5),R0        ; FIRST OPERAND
        MOV     @4(R5),R1        ; SECOND OPERAND
        SUB     R1,R0            ; SUBTRACT SECOND FROM FIRST
        MOV     R0,@6(R5)        ; STORE RESULT
        RETURN                   ; RESTORE REGISTERS AND RETURN

;** FORTRAN CALLABLE SUBROUTINE TO DIVIDE TWO INTEGERS


        .PSECT  DIVV,RO,I,GBL,REL,CON


DIVV::  CALL    $SAVAL           ; SAVE REGS R0-R5
        MOV     @2(R5),R3        ; FIRST OPERAND
        MOV     @4(R5),R1        ; SECOND OPERAND
        CLR     R2               ; LOW ORDER 16 BITS
        DIV     R1,R2            ; DIVIDE
        MOV     R2,@6(R5)        ; STORE RESULT
        RETURN                   ; RESTORE REGISTERS AND RETURN

;** FORTRAN CALLABLE SUBROUTINE TO MULTIPLY TWO INTEGERS

        .PSECT  MULL,RO,I,GBL,REL,CON

MULL::  CALL    $SAVAL           ; SAVE R0-R5
        MOV     @2(R5),R0        ; FIRST OPERAND
        MOV     @4(R5),R1        ; SECOND OPERAND
        MUL     R0,R1            ; MULTIPLY
        MOV     R1,@6(R5)        ; STORE RESULT
        RETURN                   ; RESTORE REGISTERS AND RETURN
        .END    START
```

Figure 3-30 (Cont.) Source Listing for ROTASK.MAC

Once you have assembled ROTASK, you can build it with the following command sequence:

```
TKB>ROTASK/MU,ROTASK/-WI/-SP=ROTASK
TKB>/
ENTER OPTIONS:
TKB>ROPAR=RDONLY
TKB>//
```

This command sequence directs the Task Builder to build a multiuser (/MU) task image named ROTASK.TSK and to create an 80 column (/-WI) map file named ROTASK.MAP. Because /-SP is attached to the map file, the Task Builder will not output a map to the line printer.

Under options, the ROPAR option specifies that the system is to load the read-only portion of the task into a partition named RDONLY. Specifying a separate partition for the task's read-only region is not a system requirement. The system will load the read/write portion into partition GEN. The system will not load either region until it receives a run request for the task.

The map that results from this command sequence is shown in Figure 3-31. Note that the Task Builder has added one field to the task attributes section of this map describing the disk block limits of the read-only portion of the task. It has also added a field to the root segment portion of the map that describes the memory limits of the read-only portion of the task.

Finally, note that the Task Builder has allocated space for all the program sections with the read-only attribute beginning with the highest available APR (in this case, APR 7).

```
ROTASK.TSK;6   MEMORY ALLOCATION MAP  TKB M35        PAGE 1
                   6-JAN-79   13:58

PARTITION NAME : GEN                                 ┐
IDENTIFICATION : 01                                  │
TASK  UIC      : [301,356]                           │
STACK    LIMITS: 000212 001211 001000 00512.         │
PRG XFR ADDRESS: 001552                              │ TASK
TASK ATTRIBUTES: MU                                  │ ATTRIBUTES
TOTAL ADDRESS WINDOWS: 2.                            │ SECTION
TASK  IMAGE  SIZE  : 1120. WORDS                     │
TASK ADDRESS LIMITS: 000000 004217                   │
R-W DISK BLK LIMITS: 000002 000006 000005 00005.     │
R-O DISK BLK LIMITS: 000007 000007 000001 00001.    ┘

*** ROOT SEGMENT: ROTASK


R/W MEM  LIMITS: 000000 004217 004220 02192.
R-O MEM  LIMITS: 160000 160177 000200 00128.
DISK BLK LIMITS: 000002 000006 000005 00005.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                         TITLE  IDENT  FILE
-------                                         -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001212 002570 01400.
                          001212 000530 00344. ROTASK
 01      ROTASK.OBJ;6
AADD  :(RO,I,GBL,REL,CON) 160000 000024 00020.
                          160000 000024 00020. ROTASK
 01      ROTASK.OBJ;6
DIVV  :(RO,I,GBL,REL,CON) 160024 000026 00022.
                          160024 000026 00022. ROTASK
 01      ROTASK.OBJ;6
LNC$D :(RW,D,GBL,REL,CON) 004002 000002 00002.
MULL  :(RO,I,GBL,REL,CON) 160052 000024 00020.
                          160052 000024 00020. ROTASK
 01      ROTASK.OBJ;6
SUBB  :(RO,I,GBL,REL,CON) 160076 000024 00020.
                          160076 000024 00020. ROTASK
 01      ROTASK.OBJ;6
$$RESL:(RW,I,LCL,REL,CON) 004004 000024 00020.
$$RESM:(RW,I,LCL,REL,CON) 004030 000166 00118.


GLOBAL SYMBOLS:

AADD   160000-R  DIVV   160024-R  MULL   160052-R  SUBB   160076-R


*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 2365.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 2076. WORDS (8. PAGES)
    SIZE OF WORK FILE: 1024. WORDS (4. PAGES)

    ELAPSED TIME:00:00:06
```

Figure 3-31 Task Builder Map for ROTASK.TSK

3.3 EXAMPLE 6: BUILDING A TASK THAT CREATES A DYNAMIC REGION

In all the examples of tasks shown thus far in this chapter, the Task Builder has automatically constructed and placed in the header of the task all of the window blocks necessary to map all of the regions of the task's image. The INSTALL processor has been responsible for initializing the window blocks when the task was installed. In all the examples, this has been possible because both the Task Builder and the INSTALL processor have had all the information concerning the regions available to them.

When a task creates regions while it is running (dynamic regions), the information concerning the regions is not available to either the Task Builder on the INSTALL processor. Therefore, when the Task Builder builds such a task, it does not automatically create window blocks for the dynamic regions. It creates only the window blocks necessary to map the task region (the region containing the header and stack) and any shared regions that the task references.

Dynamic regions are created and mapped with Executive directives that are imbedded in the task's code. When you build a task that creates dynamic regions, you must explicitly specify to the Task Builder how many window blocks (in excess of those created by the Task Builder for the task region and any shared regions) it is to place in the task's header. The Executive will initialize these window blocks when it processes the region and mapping directives. In all (including window blocks for the task region and shared regions), you can inlcude as many as eight window blocks to a task in an RSX-11M system and as many as 16 in an RSX-11M-PLUS system.

The text in the remainder of this section and the figures associated with it illustrate the development of a task that creates dynamic regions. Figure 3-32 shows a task (DYNAMIC.MAC) that creates a 128 word dynamic region. This task simply creates an unnamed region, maps to it, and fills it with an ascending sequence of numbers beginning at the region's base and moving upwards. When the region is full, DYNAMIC detaches from it and prints the following message on your terminal:

```
DYNAMIC IS NOW EXITING
```

The region is automatically deleted on detach.

All of the Executive directives used by DYNAMIC (RDBBK$, WDBBK$, DTRG$S, EXIT$S, CRRG$S, CRAW$S, QIOW$S, and QIOW$C) to create and manipulate the region are described in the RSX-11M/M-PLUS Executive Reference Manual.

```
        .TITLE  DYNAMIC
        .IDENT  /V01/

        .MCALL  RDBBK$,WDBBK$,DTRG$S,EXIT$S,CRRG$S,CRAW$S
        .MCALL  QIOW$C,QIOW$S

        .NLIST  BEX

;       REGION DESCRIPTOR BLOCK
;       WORD 0  SIZE OF REGION IN 32 DECIMAL WORD BLOCKS
;       WORD 1  REGION NAME
;       WORD 2      ""
;       WORD 3  NAME OF SYSTEM CONTROLLED PARTITION IN
;       WORD 4  WHICH REGION WILL BE CREATED
;       WORD 5  STATUS WORD
;       WORD 6  PROTECTION WORD

RDB:    RDBBK$  128.,,GEN,<RS.MDL!RS.ATT!RS.DEL!RS.RED!RS.WRT>,170017

;       WINDOW DESCRIPTOR BLOCK
;       WORD 0  APR TO BE USED TO MAP REGION
;       WORD 1  SIZE OF WINDOW IN 32-WORD BLOCKS
;       WORD 2  REGION ID
;       WORD 3  OFFSET INTO REGION TO START MAPPING
;       WORD 4  LENGTH IN 32-WORD BLOCKS TO MAP
;       WORD 5  STATUS WORD

WDB:    WDBBK$  7,128.,0,0,,<WS.MAP!WS.WRT>

MES1:   .ASCIZ  /DYNAMIC IS NOW EXITING/
        S1 = . - MES1
ERR1:   .ASCII  /CREATE REGION FAILED/
        SIZ1 = . - ERR1
ERR2:   .ASCII  /CREATE ADDRESS WINDOW FAILED/
        SIZ2 = . - ERR2
ERR3:   .ASCII  /DETACH REGION FAILED/
        SIZ3 = . - ERR3
        .EVEN
        .PAGE
        .ENABL  LSB
START:
        CRRG$S  #RDB                ; CREATE A 128 WORD UNNAMED REGION
        BCS     1$                  ; FAILED TO CREATE REGION
        MOV     RDB+R.GID,WDB+W.NRID ; COPY REGION ID INTO WINDOW BLOCK
        CRAW$S  #WDB                ; CREATE ADDR WINDOW AND MAP
        BCS     2$                  ; FAILED TO CREATE ADDR WINDOW
        MOV     WDB+W.NBAS,R0       ; BASE ADDR OF CREATED REGION
        MOV     WDB+W.NSIZ,R2       ; NUMBER OF 32. WORDS IN REGION
        .REPT   5                   ; MULTIPLY
        ASL     R2                  ; BY
        .ENDR                       ; 32.
        MOV     #1,R1               ; INITIAL VALUE TO PLACE IN REGION
20$:    MOV     R1,(R0)+            ; MOVE VALUE INTO REGION
        INC     R1                  ; NEXT VALUE TO PLACE IN REGION
        DEC     R2                  ; ONE LESS WORD LEFT
        BGT     20$                 ; TO FILL IN
        DTRG$S  #RDB                ; DETACH AND DELETE REGION
        BCS     3$                  ; DETACH FAILED
        QIOW$C  IO.WVB,5,1,,,,<MES1,S1,40>
        EXIT$S                      ;
```

Figure 3-32 Source Listing for DYNAMIC.MAC

```
;
;       ERROR ROUTINES
;
1$:     MOV     #ERR1,R0        ; CREATE FAILED
        MOV     #SIZ1,R1        ; SIZ OF MESSAGE
        BR      6$              ; WRITE MESSAGE
2$:     MOV     #ERR2,R0        ; CREATE ADDRESS WINDOW FAILED
        MOV     #SIZ2,R1        ; SIZE OF MESSAGE
        BR      6$
3$:     MOV     #ERR3,R0        ; DETACH FAILED
        MOV     #SIZ1,R1        ; SIZE OF MESSAGE
6$:     QIOW$S  #IO.WVB,#5,#1,,,,<R0,R1,#40>
        EXIT$S
        .END    START
```

Figure 3-32 (Cont.) Source Listing for DYNAMIC.MAC

Once you have assembled DYNAMIC, you can build it with the following Task Builder command sequence:

```
     TKB>DYNAMIC,DYNAMIC/-WI/-SP=DYNAMIC
     TKB>/
     ENTER OPTIONS:
     TKB>WNDWS=1
     TKB>//
```

This command sequence directs the Task Builder to create a task image named DYNAMIC.TSK and an 80 column (/-WI) map file named DYNAMIC.MAP on device SY: under the terminal UIC. Because /-SP is attached to the map file, the Task Builder will not output the file to the line printer.

Under options, the WNDWS option directs the Task Builder to create one window block over and above that required to map the task region. Note that one window block must be created for each region the task expects to be mapped to simultaneously.

The map that results from this command sequence is shown in Figure 3-33.

Note that creating dynamic regions always involves the assumption that there will be enough room in the partition named in the task's region descriptor block to create the region when the task is run. In this example, if DYNAMIC were to be run in a system whose partition GEN was not large enough to accommodate the region it creates, the CREATE REGION directive would fail.

```
DYNAMIC.TSK;1  MEMORY ALLOCATION MAP  TKB M35          PAGE 1
                    6-JAN-79  14:13



PARTITION NAME : GEN                                 ┐
IDENTIFICATION : V01                                 │
TASK  UIC      : [301,356]                           │
STACK    LIMITS: 000212 001211 001000 00512.         │ TASK
PRG XFR ADDRESS: 001406                              │ ATTRIBUTES
TOTAL ADDRESS WINDOWS: 2.                            │ SECTION
TASK  IMAGE  SIZE  : 480. WORDS                      │
TASK ADDRESS LIMITS: 000000 001673                   │
R-W DISK BLK LIMITS: 000002 000003 000002 00002.    ┘

*** ROOT SEGMENT: DYNAMI


R/W MEM  LIMITS: 000000 001673 001674 00956.
DISK BLK LIMITS: 000002 000003 000002 00002.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                            TITLE  IDENT  FILE
-------                                            -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001212 000430 00280.
                          001212 000430 00280. DYNAMI V01    DYNAMIC.OBJ;1
$DPB$$:(RW,I,LCL,REL,CON) 001642 000030 00024.
                          001642 000030 00024. DYNAMI V01    DYNAMIC.OBJ;1



*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 518.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 2076. WORDS (8. PAGES)
    SIZE OF WORK FILE: 768. WORDS (3. PAGES)

    ELAPSED TIME:00:00:03
```

Figure 3-33 Task Builder Map for DYNAMIC.TSK

3.4 VIRTUAL PROGRAM SECTIONS

A virtual program section is a special Task Builder storage allocation facility that permits you to create and refer to large data structures by means of the mapping directives. Virtual program sections are supported in the Task Builder through the VSECT option and in FORTRAN through a set of FORTRAN-callable subroutines that issue the necessary mapping directives at runtime. With the Task Builder VSECT option you can specify the following parameters for a relocatable program section or FORTRAN common block that you have defined in your object module:

- Base virtual address
- Virtual length (window size)
- Physical length

By specifying the base address, you can align the program section on a 4K address boundary as required by the mapping directives. Thereafter, references within the program need only point to the base of the program section or to the first element in the common block, to ensure proper boundary alignment.

By specifying the window size, you can fix the amount of virtual address space that the Task Builder allocates to the program section. If the allocation made by a module causes the total size to exceed this limit, the allocation wraps around to the beginning of the window.

By specifying the physical size, you can allocate, before runtime, the physical memory that the program section will be mapped into at runtime. The Task Builder allocates this physical memory within an area that precedes the task image. This area is called the mapped array area.

The physical length parameter is optional. If you intend to allocate physical memory at runtime through the Create Region directive, you can specify a value of 0.

Note that when you specify a nonzero value for the physical memory parameter, the resulting allocation affects only the task's memory image, not its disk image.

Note also that the Task Builder will attach the virtual attribute to a relocatable program section you have specified in the VSECT option only if the section is defined in your task through the common statement. For example:

```
     TKB>VSECT=VIRT:160000:20000:2000
```

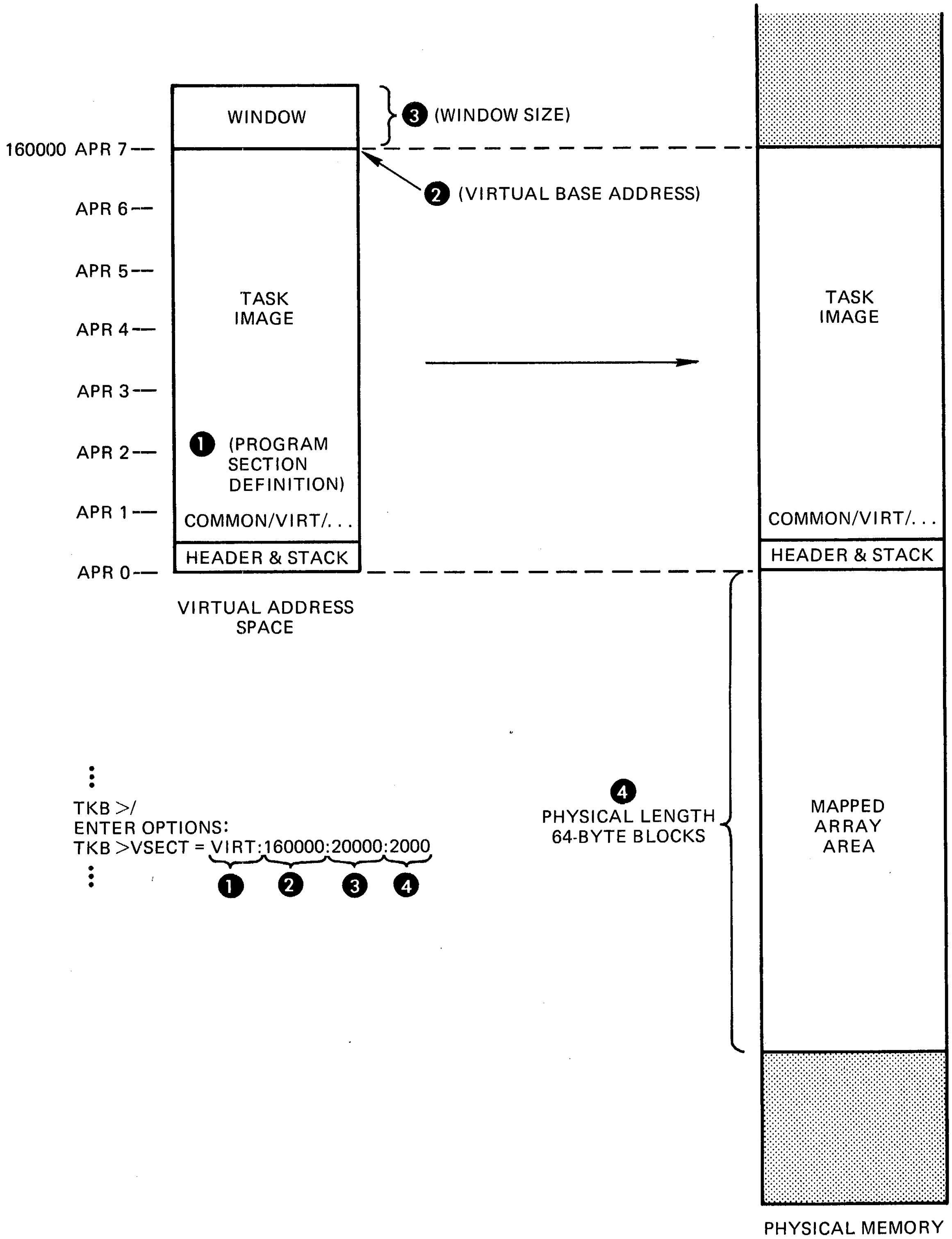
In this example, virtual program section VIRT is allocated with a window size of 4K words and a base virtual address of 160000. In physical memory, 32K words are reserved for mapping the section at runtime.

Assume the program is written in FORTRAN, and includes the following statement:

```
     COMMON /VIRT/ARRAY(4)...
```

This statement generates a program section to which the Task Builder attaches the virtual attribute. A reference to the first element of the section, ARRAY(1), is translated by the Task Builder to the virtual address 160000.

Figure 3-34 shows the effect of this use of the VSECT option.



```
      .
      .
      .
TKB>/
ENTER OPTIONS:
TKB>VSECT=VIRT:160000:20000:2000
      .
      .
      .
```

Figure 3-34 VSECT Option Usage

As mentioned previously, the Task Builder restricts the amount of virtual address space allocated to the section to a value that is less than or equal to the window size, wrapping around to the base if the window size is exceeded.

This process is illustrated in the following example, in which three modules, A, B, and C, each contain a program section named VIRT that is 3000 words long. A window size of 4K words has been set through the VSECT option. If the program section has the concatenate attribute, the Task Builder allocates memory to each module as follows:

Module	Low Limit	Length	High Limit
A	160000	14000	174000
B	174000	14000	170000
C	170000	14000	164000

The address limits for modules B and C illustrate the effect of address wrap around when a component of the total allocation exceeds the window boundary. Note that the addresses generated will be properly aligned with the contents of physical memory if the virtual section is remapped in increments of the window size.

3.4.1 FORTRAN Run-Time Support for Virtual Program Sections

FORTRAN supports subroutines to make use of the mapping directives. FORTRAN also supports calls to the following subroutines, which are related to virtual program sections:

Subroutine	Function
ALSCT	Allocates a portion of physical memory for use as a virtual section
RLSCT	Releases all physical memory allocated to a virtual section

As mentioned earlier, the effect of one or more VSECT= declarations at task-build time is to create a pool of physical memory below the task image (the mapped array area). Before a virtual section is referred to, the task must allocate a portion of this memory through a call to ALSCT. When space is no longer required, it is released through a call to RLSCT.

Note that these subroutines issue no mapping directives. They allocate and release space using region and window descriptor arrays that you supply. The resulting physical offsets are used in the task's subsequent calls, that perform the actual mapping.

The subroutine ALSCT is called to allocate physical memory to a virtual program section as follows:

```
CALL ALSCT (ireg,iwnd[,ists])
```

ireg

A one-dimensional integer array that is 9 words long. Elements 1 through 8 of the array contain a region descriptor for the physical memory to be mapped. The descriptor has the following format:

ireg(1)	Region ID
ireg(2)	Size of region in units of 64-byte blocks
ireg(3)	Name of region in Radix-50 format (first three characters)
ireg(4)	(Second three characters)
ireg(5)	Name of main partition containing region
ireg(6)	The name is in Radix-50 format
ireg(7)	Region status word
ireg(8)	Region protection code
ireg(9)	Thread word: this element links window descriptors that are used to map portions of the region. It is maintained by the subroutine.

The elements of the array that you set up consist of ireg(1), and ireg(3) through ireg(8). The thread word, ireg(9), must be zero on the initial call; thereafter, the subroutine maintains it.

When your task makes an allocation, ireg(1) and ireg(2) must be 0 on the initial call. In this case, ALSCT obtains and stores the region size in ireg(2). When the allocation is being made from a separate region, the caller must supply both region ID and size. Elements 3 through 8 are not referred to by the subroutine but must be set up by the caller as required by the applicable system directives. For a detailed description of these parameters, refer to the RSX-11M/M-PLUS Executive Reference Manual.

iwnd

A one-dimensional array that is 11 words long. The first 8 words contain a window descriptor in the following format:

iwnd(1)	Base APR in bits 8 through 15; the Executive Sets bits 0 through 7 when the appropriate mapping directives are issued
iwnd(2)	Virtual base address
iwnd(3)	Window size in units of 64-byte blocks
iwnd(4)	Region ID
iwnd(5)	Offset into the region, in units of 64-byte blocks
iwnd(6)	Length to map, in units of 64-byte blocks
iwnd(7)	Status word
iwnd(8)	Address of send/receive buffer

iwnd(9) Base offset of physical block allocated to section in units of 64-byte blocks

iwnd(10) Length of block in units of 64-byte blocks (supplied by caller); set to maximum block offset by subroutine

iwnd(11) Thread word: this element links window descriptors that are used to map other portions of the region. It is maintained by the subroutine

The following array elements are supplied as output from the subroutine:

iwnd(4), iwnd(5), iwnd(9), iwnd(10), and iwnd(11)

The remaining elements must be set up as required by the Executive directives. Consult the RSX-11M/M-PLUS Executive Reference Manual for a detailed description of these parameters.

ists

receives the result of the call. One of the following values is returned:

+1 Block successfully allocated. In this case, the region and window descriptor arrays are set up as described above.

-200. Insufficient physical memory was available for allocating the block

The subroutine RLSCT is called to deallocate the physical memory assigned to a virtual section as follows:

```
CALL RLSCT (ireg,iwnd)
```

ireg

A one-dimensional integer array that is 9 words long. The contents of the array are the same as those described for subroutine ALSCT.

iwnd

A one-dimensional integer array that is 11 words long. The contents of the array are the same as those described for subroutine ALSCT.

Upon return, element iwnd(10) is the length of the deallocated region in units of 64-byte blocks.

The procedure for using these subroutines can be summarized as follows:

- You allocate storage in the program for one window descriptor per VSECT, and for a single region descriptor.
- Your task calls the subroutine ALSCT to reserve physical memory to which the program section will be mapped.

- Your task issues the mapping directives to map the virtual address space into a portion of the physical memory. It is the task's responsibility to ensure that the physical memory to be mapped is always within the limits defined by iwnd(9) and iwnd(10).
- When the space is no longer required, the task unmaps it and releases it with a call to RLSCT.

3.4.2 Example 7: Building a Program that Uses a Virtual Program Section

Figure 3-35 shows the FORTRAN source file for a task named VSECT.FTN. It illustrates the use of the ALSCT FORTRAN subroutine. When you build, install, and run VSECT, it will allocate the mapped array area below its header, create a 4K-word window, and map to the area through the window. ALSCT will then initialize the area and prompt for an array subscript at your terminal by printing:

```
SUBSCRIPT?
```

When you input a subscript, it will respond with ELEMENT= and the contents of the array element for the subscript you typed. VSECT will continue to prompt until you type CTRL/Z. Upon receiving a CTRL/Z VSECT will exit.

Once you have compiled VSECT, you can build it with the following Task Builder command sequence:

```
TKB>VSECT,VSECT/-SP=VSECT,LB:[1,1]FOROTS/LB
TKB>/
ENTER OPTIONS:
TKB>WNDWS=1
TKB>VSECT=VIRT:160000:20000:200
TKB>//
```

This command sequence directs the Task Builder to create a task image file named VSECT.TSK and a short (by default) map file VSECT.MAP. Because /-SP is appended to the map file, the Task Builder will not output the map to the line printer.

The library switch (/LB) specifies that the Task Builder is to search the FORTRAN run time library FOROTS.OLB to resolve any undefined references in the input module VSECT.OBJ. Because the library switch was applied to the FORTRAN library file without arguments, the Task Builder extracts from the library and includes in the task image, any modules in which references are defined.

Under options, the WNDWS option directs the Task Builder to add a window block to the header in the task image. This window block will be initialized by the Executive when it processes the mapping directives within the task.

The VSECT option directs the Task Builder to establish for the program section named VIRT a base address of 160000 (APR 7) and a length of 20000(8) bytes (4K words). The program section VIRT is defined within the task through the FORTRAN COMMON statement. The VSECT option also specifies that the Task Builder is to allocate 200 64-byte blocks of physical memory in the task's mapped array area below the task's header. (For more information on the switches and options used in this example, refer to Chapter 6)

The map that results from this command sequence is shown in Figure 3-36.

```
C
C       VSECT.FTN
C
        INTEGER *2 SUB,IRDB(9),IWDB(11),DSW
        INTEGER *2 IARRAY(4096)
        COMMON /VIRT/IARRAY
        IWDB (1) = "3400                 !USE APR 7 FOR WINDOW
        IWDB (3) = 128                   !WINDOW SIZE = 128*32 WORDS = 4K
        IWDB (5) = 0                     !OFFSET
        IWDB (7) = "402                  !STATUS = WS.64B!WS.WRT
C
C       ALLOCATE 4K MAPPED ARRAY TO IWDB,IRDB
C
        CALL ALSCT (IRDB,IWDB,DSW)
        IF (DSW .NE. 1) GOTO 100
C
C       CREATE A 4K ADDRESS WINDOW
C
        CALL CRAW (IWDB,DSW)
        IF (DSW .NE. 1) GOTO 200
C
C       MAP 4K MAPPED ARRAY
C
        CALL MAP (IWDB,DSW)
        IF (DSW .NE. 1) GOTO 300
        DO 1 I=1,4096
1       IARRAY (I) = I
C
C       MAPPED ARRAY IS INITIALIZED, PROMPT FOR A SUBSCRIPT
C
3       WRITE (5,5)
5       FORMAT ('$SUBSCRIPT?')
        READ (5,4,END=1000)SUB
4       FORMAT (I7)
        WRITE (5,6)IARRAY (SUB)
6       FORMAT ( ' ELEMENT = ',I7)
        GOTO 3
C
C       ERROR ROUTINES
C
100     WRITE (5,101)DSW
101     FORMAT (' ERROR FROM ALSCT. ERROR = ',I7)
        GOTO 1000
200     WRITE (5,201)DSW
201     FORMAT (' ERROR FROM CREATING ADDRESS WINDOW. ERRROR = ',I7)
        GOTO 1000
300     WRITE (5,301)DSW
301     FORMAT (' ERROR FROM MAPPING. ERROR = ',I7)
1000    CALL EXIT
        END
```

Figure 3-35 Source Listing for VSECT.FTN

```
VSECT.TSK;1     MEMORY ALLOCATION MAP  TKB M35          PAGE 1
                      8-JAN-79   11:41



PARTITION NAME : GEN                                         ┐
IDENTIFICATION : $FORT                                       │
TASK  UIC      : [301,356]                                   │
STACK    LIMITS: 000216 001215 001000 00512.                 │
PRG XFR ADDRESS: 001216                                      │ TASK
TOTAL ADDRESS WINDOWS: 2.                                    │ ATTRIBUTES
MAPPED  ARRAY  AREA: 4096. WORDS                             │ SECTION
TASK  IMAGE  SIZE  : 9440. WORDS                             │
TOTAL  TASK  SIZE  : 13536. WORDS                            │
TASK ADDRESS LIMITS: 000000 044653                           │
R-W DISK BLK LIMITS: 000002 000046 000045 00037.            ─┘

*** ROOT SEGMENT: VSECT


R/W MEM  LIMITS: 000000 044653 044654 18860.
DISK BLK LIMITS: 000002 000046 000045 00037.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                              TITLE  IDENT  FILE
-------                                              -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001216 000000 00000.
                          001216 001020 00528. .MAIN. $FORT  VSECT.OBJ;1
                            .
                            .
                            .
VIRT  :(RW,D,GBL,REL,OVR) 001216 020000 08192.
                          001216 020000 08192. .MAIN. $FORT  VSECT.OBJ;1
                            .
                            .
                            .
GLOBAL SYMBOLS:

BAH$   003506-R  FOO$   003532-R  MOL$IS 002736-R  TVQ$   003144-R
                            .
                            .
                            .


*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 16257.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 4188. WORDS (16. PAGES)
    SIZE OF WORK FILE: 3584. WORDS (14. PAGES)

    ELAPSED TIME:00:00:32
```

Figure 3-36 Task Builder Map for VSECT.TSK

3.5 EXAMPLE 8: PRIVILEGED TASKS

There are two classes of tasks in the RSX-11M/M-PLUS systems: privileged and nonprivileged. The majority of tasks running on any one system are nonprivileged.

The distinction between privileged and nonprivileged tasks is primarily a distinction of system access capabilities. Because, in an unmapped system, all tasks have access to all of memory, this distinction is not hardware enforceable. Therefore, if your system is unmapped your task must be responsible for observing the access rules of your system.

In a mapped system, privileged tasks have special device and memory access rights that nonprivileged tasks do not have. A privileged task can, with certain exceptions, access the Executive routines and data structures; a nonprivileged task cannot. Some privileged tasks have automatic I/O page mapping available to them; nonprivileged tasks do not. Finally, a privileged task can bypass system security features while a nonprivileged task cannot.

Because of their special access rights, privileged tasks are potentially hazardous to a running system. A privileged task with coding errors can corrupt the Executive or unintentionally disable peripheral devices. Moreover, problems caused by such a privileged task can be obscure and difficult to isolate. For these reasons, you must exercise caution when developing and running a privileged task.

You designate a task as privileged with the PR (privileged) Task Builder switch (this switch is described in Chapter 6). The Task Builder allocates address space for a privileged task based on the memory management APR that you specify as an argument to this switch. Three arguments are acceptable to the Task Builder: 0, 4, and 5. The choice of which of these arguments to specify is based on the considerations described below.

When you specify 4 or 5, the Task Builder automatically reserves APR 7 for mapping the I/O page. Moreover, the Task Builder makes the Executive available to your task by reserving the APRs necessary to map the Executive into your task's virtual address space. Therefore, if your task requires access to the Executive, you must specify an argument of either 4 or 5.

The choice between APR 4 and 5 is dictated by the size of the Executive area. If the Executive is 16K words or less, you should specify an argument of 4. The Task Builder applies a bias of 100000 (16K) to all addresses within your task.

If the Executive is 20K words, you must specify an argument of 5. The Task Builder applies a bias of 120000 (20K) to all addresses within your task.

The mapping for APR 4 and 5 is shown in Figure 3-37.

When you specify an argument of 4, there will be 12K words of address space between the beginning of the task and the start of the mapping for the I/O page. If your task expects to access the I/O page, it must not exceed this 12K word limit. If it does, the Task Builder will be forced to reserve APR 7 to map the task instead of the I/O page.

When you specify an argument of 5, there will be 8K words of address space between the beginning of the task and the start of the mapping for the I/O page. In this case, the task must not be greater than 8K words if it expects to access the I/O page.

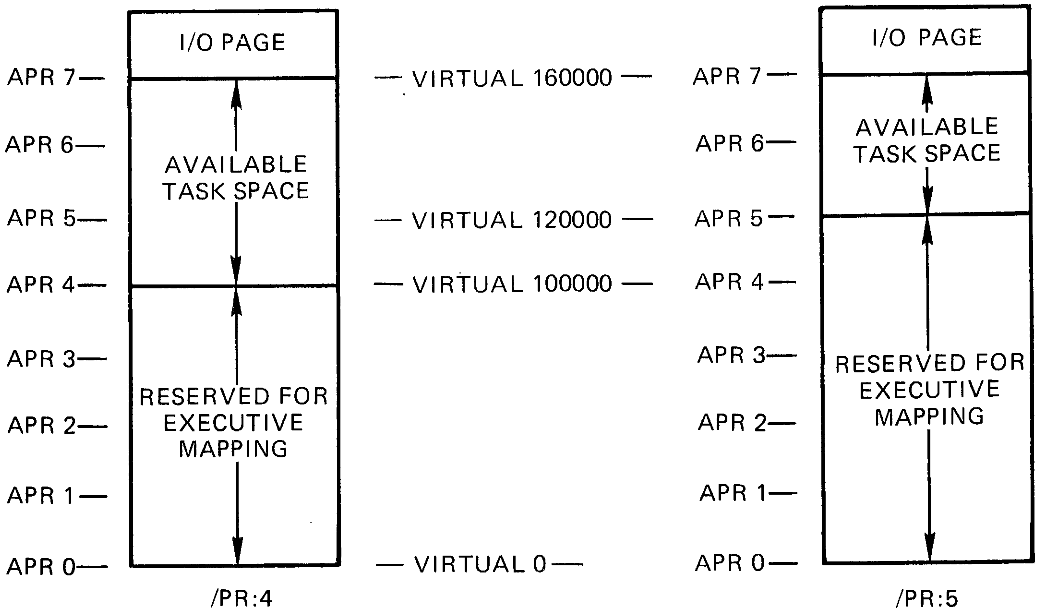


Figure 3-37 Mapping for /PR:4 and /PR:5

When a task overlaps the I/O page, the Task Builder does not necessarily generate an error message. Before the Task Builder generates an error message, a task designated to be mapped with APR 4 must be greater than 16K words; a task designated to be mapped with APR 5 must be greater than 12K words. Only when you install a task that overlaps the I/O page does the INSTALL processor generate the following message:

```
INS--WARNING--PRIVILEGED TASK OVERMAPS THE I/O PAGE
```

While this is not a fatal error message, you should consider the condition to be fatal if your task expects to access the I/O page.

When you specify an argument of 0, the Task Builder reserves APR 0 for mapping your task. Virtual address space begins at virtual address 0 and extends upward as far as 32K words. Your task cannot access the Executive routines or data structures, and the Task Builder does not automatically reserve an APR to map the I/O page.

A task mapped with APR 0 can access the I/O page through a device common (refer to Chapter 3 for a description of device commons).

The MACRO-11 source program PRIVEX.MAC in Figure 3-38 illustrates one possible use of a privileged task.

NOTE

> The nature of privileged tasks is such that you must have a working knowledge of system concepts to understand the operation of one or to write one. If this example deals with Executive functions that are unfamiliar to you, you may prefer to skip this section and return to it at a later time.

If you assemble, build, and install PRIVEX into your system, it will scan the system device tables and examine the UCBs of all non-pseudo devices on your system. It will determine whether each device is attached by a task and print on your terminal the names of all attached devices on your system with the name of each attached program.

PRIVEX accesses two Executive routines, $SWSTK (switch stack) and $SCDVT (scan device tables). The routine $SWSTK switches the processor to system state (Kernel mode). This switch to system state is necessary because it inhibits all other processes from modifying the Executive data structures until PRIVEX is finished with them. The double semicolons (;;) indicate the portion of the task that is running in system state.

The routine $SCDVT performs the actual scanning of the device tables. It returns to PRIVEX each time it accesses a new UCB.

PRIVEX also calls the system library routine $EDMSG (edit message) to format the data it has retrieved from the device tables. This routine is documented in the IAS/RSX-11 System Library Routines Reference Manual.

```
;  MACRO  LIBRARY CALLS
        .TITLE PRIVEX
        .IDENT /01/

        .MCALL  ALUN$C,EXIT$S,QIOW$S

;
;  LOCAL DATA
;

        .NLIST  BEX

ATTMES: .ASCIZ  /%2A%P: IS ATTACHED BY %2R/      ;
BUFMES: .ASCIZ  /BUFFER OVERFLOW/                ;

        .LIST   BEX

QIOBUF: .BLKB   132.                            ;MESSAGE OUTPUT BUFFER

        .EVEN

;
;  BUFFER INTO WHICH INFORMATION IS STORED AT SYSTEM STATE FOR
;  PRINTING AT USER STATE.  AN ENTRY IS FOUR WORDS LONG:
;
;
;          ADDRESS IN DCB OF THE TWO ASCII CHARACTER DEVICE NAME
;
;                      BINARY UNIT NUMBER
;
;                FIRST RAD50 WORD OF NAME OF ATTACHED TASK
;
;                SECOND RAD50 WORD OF NAME OF ATTACHED TASK
;
```

Figure 3-38 Source Code for PRIVEX

```
;
;   THE BUFFER IS TERMINATED BY A
;
;         0  = ALL UNITS IN THE SYSTEM HAVE BEEN EXAMINED
;         -1 = THE BUFFER WAS FILLED BEFORE ALL UNITS COULD BE EXAMINED
;

BUFFER: .BLKW   4*200.+1          ;
BUFEND=.-2                        ;ADDRESS OF LAST WORD OF BUFFER

START:   MOV      #BUFFER,R2      ;GET ADDRESS OF INFORMATION BUFFER
         CLR      (R2)            ;ASSUME NO UNITS ARE ATTACHED
         CLR      R1              ;INITIALIZE CURRENT DCB ADDRESS

;
;   "CALL $SWSTK,FORMAT" SWITCHES TO SYSTEM STATE.  ALL REGISTERS
;   ARE PRESERVED ACROSS THE TRANSITION FROM USER MODE TO KERNEL
;   MODE.  BEING IN SYSTEM STATE LOCKS OTHER PROCESSES OUT OF THE
;   EXECUTIVE (GUARANTEES THAT THE DATA BEING EXAMINED WILL NOT
;   CHANGE WHILE IT IS BEING EXAMINED).  A "RETURN" WILL GIVE
;   CONTROL TO "FORMAT" AND WILL RESTORE THE CONTENTS OF THE
;   REGISTERS TO THEIR VALUES BEFORE THE "CALL $SWSTK".
;
         CALL     $SWSTK,FORMAT   ;SWITCH TO SYSTEM STATE
         MOV      #$SCDVT,-(SP)   ;;GET ADDRESS OF SCAN DEVICE TABLES
                                  ;;COROUTINE
20$:     CALL     @(SP)+          ;;GET NEXT NONPSEUDO DEVICE UCB
                                  ;;  ADDRESS
         BCS      100$            ;;IF CS NO MORE UCBS

;
;   AT THIS POINT:
;        R3 - ADDRESS OF THE DEVICE CONTROL BLOCK
;        R4 - ADDRESS OF THE STATUS CONTROL BLOCK
;        R5 - ADDRESS OF THE UNIT CONTROL BLOCK
;

         CMP      R1,R3           ;;IS THIS A NEW DCB?
         BEQ      40$             ;;IF EQ NO
         MOV      R3,R1           ;;REMEMBER THIS DCB
         CLR      R0              ;;FORM LOWEST UNIT NUMBER ON
         BISB     D.UNIT(R3),R0   ;;  THIS DCB
40$:     MOV      U.ATT(R5),R4    ;;IS A TASK ATTACHED?
         BEQ      60$             ;;IF EQ NO
                                  ;;IF NE R4 IS TCB ADDRESS
         CMP      #BUFEND,R2      ;;ANY MORE ROOM IN BUFFER?
         BLOS     80$             ;;IF LOS NO
         ADD      #D.NAM,R3       ;;FORM ADDRESS OF DEVICE NAME
         MOV      R3,(R2)+        ;;SAVE IT IN BUFFER
         MOV      R0,(R2)+        ;;SAVE UNIT NUMBER
         MOV      T.NAM(R4),(R2)+ ;;SAVE NAME OF ATTACHED TASK
         MOV      T.NAM+2(R4),(R2)+ ;;
         CLR      (R2)            ;;ASSUME NO MORE ATTACHED UNITS
60$:     INC      R0              ;;INCREMENT UNIT NUMBER
         BR       20$             ;;
80$:     CALL     @(SP)+          ;;GET $SCDVT TO CLEAN OFF STACK
         BCC      80$             ;;
         COM      (R2)            ;;SHOW BUFFER OVERFLOW
100$:    RETURN                   ;;RETURN TO USER STATE AT FORMAT

         .ENABL   LSB
```

Figure 3-38 (Cont.) Source Code for PRIVEX

```
FORMAT: TST     (R2)                ;ANY MORE INFORMATION IN BUFFER?
        BEQ     EXIT                ;IF EQ NO
        CMP     #-1,(R2)            ;OVERFLOWED BUFFER?
        BNE     40$                 ;IF NE NO
        MOV     #BUFMES,R1          ;GET ADDRESS OF OVERFLOW MESSAGE
        CALL    PRINT               ;PRINT IT
EXIT:   EXIT$S                      ;
40$:    MOV     #ATTMES,R1          ;GET ADDRESS OF TEMPLATE
        CALL    PRINT               ;FORMAT AND PRINT THE INFORMATION
        BR      FORMAT              ;

        .DSABL  LSB

;
;  PRINT - FORMAT AND PRINT A MESSAGE
;
;  INPUTS:
;       R1 - ADDRESS OF AN $EDMSG INPUT STRING
;       R2 - ADDRESS OF AN $EDMSG PARAMETER BLOCK
;
;  OUTPUTS:
;       R2 - ADDRESS OF NEXT PARAMETER IN THE $EDMSG PARAMETER BLOCK
;       R0, R1, R3, R4 ARE DESTROYED
;       R5 IS PRESERVED
;

PRINT:  MOV     #QIOBUF,R0          ;GET ADDRESS OF OUTPUT BUFFER
        MOV     R0,R3               ;SAVE FOR QIOW$S
        CALL    $EDMSG              ;FORMAT MESSAGE INTO OUTPUT BUFFER

;
;  REMOVE LEADING ZEROS FROM UNIT NUMBER
;

        MOV     R3,R0               ;POINT AT OUTPUT BUFFER
        TST     (R0)+               ;INCREMENT BY TWO (POINT PAST
                                    ;  DEVICE NAME)
        MOV     R0,R4               ;REMEMBER THIS SPOT
20$:    DEC     R1                  ;ASSUME NEXT BYTE IS A LEADING ZERO
                                    ;  (REDUCE LENGTH OF MESSAGE)
        CMPB    #'0,(R0)+           ;IS IT?
        BEQ     20$                 ;IF EQ YES -- IGNORE IT
        INC     R1                  ;COUNTERACT TOO MUCH DECREMENTING
        CMPB    #':,-(R0)           ;WAS THE BYTE A COLON (WAS THE UNIT
                                    ;  NUMBER ZERO)?
        BNE     40$                 ;IF NE NO
        MOVB    #'0,(R4)+           ;ADD A ZERO UNIT NUMBER
        INC     R1                  ;INCREASE LENGTH OF MESSAGE
40$:    MOVB    (R0)+,(R4)+         ;TACK ON REST OF MESSAGE
        BNE     40$                 ;IF NE NOT DONE

;
;  PRINT THE MESSAGE ON LUN "OUTLUN" (DEFINED BY THE TASK BUILD FILE)
;  AND WAIT USING EVENT FLAG 1
;

        QIOW$S  #IO.WVB,#OUTLUN,#1,,,,<R3,R1,<#' >> ;
        RETURN

        .END    START
```

Figure 3-38 (Cont.) Source Code for PRIVEX

PRIVEX.MAC should be assembled with the following assembler command string:

```
     MAC>PRIVEX,PRIVEX/-SP=[1,1]EXEMC/ML,[200,200]RSXMC/PA:1,[301,311]PRIVEX
```

The file EXEMC is the Executive macro library and the file RSXMC is the Executive prefix file. The switches used in the command string are described in the IAS/RSX-11 MACRO-11 Programmer's Reference Manual.

The Task Builder command sequence for PRIVEX is as follows:

```
     >TKB
     TKB> PRIVEX/PR:5,PRIVEX/-SP=PRIVEX
     TKB> [2,54]RSX11M.STB,[1,1]EXELIB/LB
     TKB> /
     ENTER OPTIONS:
     TKB> UNITS=1                    ;DEFINE NUMBER OF LUNS
     TKB> GBLDEF=OUTLUN:1            ;DEFINE LUN ON WHICH TO PRINT MESSAGES
     TKB> ASG=TI0:1                  ;ASSIGN LUN TO DEVICE
     TKB> //
     >
```

This command sequence directs the Task Builder to build PRIVEX as a privileged task and to add a bias of 120000 to all locations within it. APR 5 was chosen in this example because the Executive in the system on which this example was originally built is 20K words long. If the Executive in your system is 16K words or less, you can use /PR:4 when you build the task.

In the options section of the Task Builder command sequence, the UNITS=1 option specifies that PRIVEX is going to use only one logical unit. The GBLDEF=OUTLUN:1 option defines the symbol OUTLUN as being equal to 1, and the ASG=TI0:1 option associates device TI0: with logical unit 1.

The Task Builder map for PRIVEX is shown in Figure 3-39. The GLOBAL SYMBOL SECTION has been shortened to save space. Note that the task's address limits begin at virtual address 120000. The diagram in Figure 3-40 illustrates how the Task Builder allocates virtual address space for the program.

```
PRIVEX.TSK;2   MEMORY ALLOCATION MAP TKB M32           PAGE 1
                     7-OCT-78   16:10


PARTITION NAME :  GEN                                  ┐
IDENTIFICATION :  01                                   │
TASK  UIC      :  [301,311]                            │
STACK    LIMITS:  120146 121145 001000 00512.          │ Task
PRG XFR ADDRESS:  124526                               │ Attributes
TASK ATTRIBUTES:  PR                                   │ Section
TOTAL ADDRESS WINDOWS:  1.                             │
TASK  IMAGE  SIZE  :  1856. WORDS                      │
TASK ADDRESS LIMITS:  120000 127147                    │
R-W DISK BLK LIMITS:  000002 000011 000010 00008.     ┘

*** ROOT SEGMENT:PRIVEX


R/W MEM LIMITS:  120000 127147 007150 03688.
DISK BLK LIMITS: 000002 000011 000010 00008.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                            TITLE   IDENT   FILE
--------                                           -----   -----   ----
. BLK.:(RW,I,LCL,REL,CON) 121146 005654 02988.
                          121146 003656 01966.  PRIVEX 01        PRIVEX.OBJ;2
LNC$D :(RW,D,GBL,REL,CON) 127022 000002 00002.
$$RESL:(RW,I,LCL,REL,CON) 127024 000024 00020.
$$RESM:(RW,I,LCL,REL,CON) 127050 000100 00064.

GLOBAL SYMBOLS:

AS.DEL 000010    CI.PWF 177776    D.RS80 177660    D.VKRB 000010
G.STAT 000003    IO.CLN 003400    KINDR7 172316
AS.EXT 000004    DV.ISP 002000    D.RS81 177657    D.VOUT 000004
HI$DIC 000115    IO.DET 002000    KISAR0 172360
                        .
                        .
                        .
$TT56  033102    $VTDCB 033406    .DB3   020630    .MM1  021620
.TT22  025634    .TT43  027314


*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES:  185418.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL:  9454.  WORDS (36.  PAGES)
    SIZE OF WORK FILE:  8448.  WORDS (33.  PAGES)

    ELAPSED TIME:00:00:30
```

Figure 3-39 Task Builder Map for PRIVEX

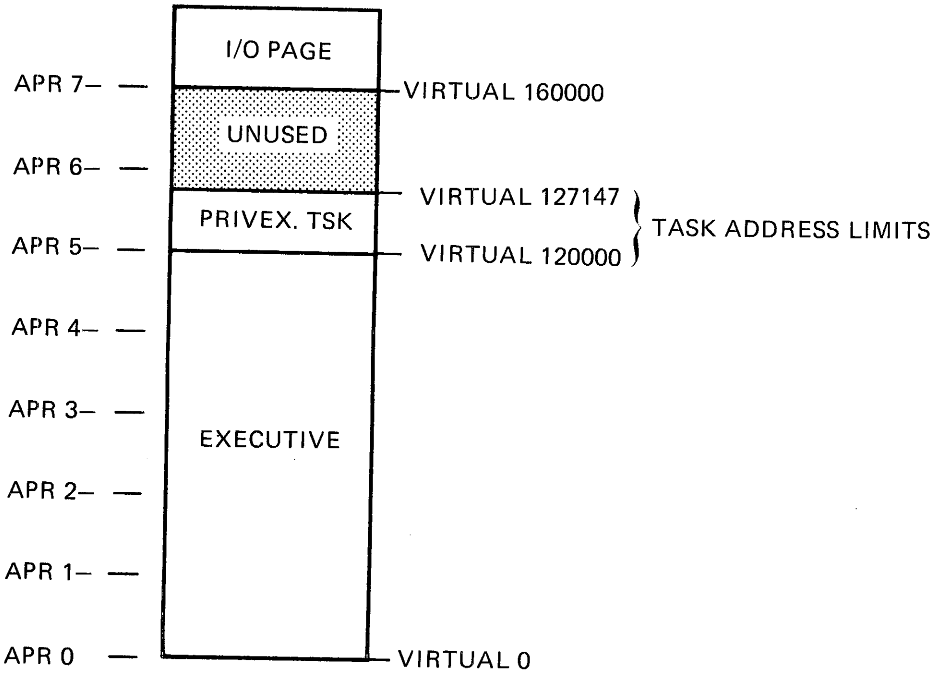


Figure 3-40 Allocation of Virtual Address Space for PRIVEX

CHAPTER 4

OVERLAY CAPABILITY

The Task Builder provides you with the means to reduce the memory and/or virtual address space requirements of your task by using tree-like overlay structures created with the Overlay Description Language (ODL). You can specify two kinds of overlay segments: those that reside on disk and those that reside permanently in memory.

4.1 OVERLAY STRUCTURES

To create an overlay structure, you divide a task into a series of segments consisting of:

- A single root segment, which is always in memory and,
- Any number of overlay segments, which either 1) reside on disk and share virtual address space and physical memory with one another (disk-resident overlays); or 2) reside in memory and share only virtual address space with one another (memory-resident overlays)[1]

Segments consist of one or more object modules which in turn consist of one or more program sections. Segments that overlay each other must be logically independent; that is, the components of one segment cannot reference the components of a segment with which it shares virtual address space. In addition to the logical independence of the overlay segments, the general flow of control within the task must be considered when creating overlay segments.

You must also consider the kind of overlay segment to create at a given position in the structure, and how to construct it. Dividing a task into disk-resident overlays saves physical space, but introduces the overhead activity of loading these segments each time they are needed, but not present in memory. Memory-resident overlays, on the other hand, are loaded from disk only the first time they are referenced. Thereafter, they remain in memory and are referenced by remapping.

Several large classes of tasks can be handled effectively by an overlay structure. For example, a task that moves sequentially through a set of modules is well suited to the use of an overlay structure. A task that selects one of a set of modules according to the value of an item of input data is also well suited to the use of an overlay structure.

[1] Note that memory-resident overlays can be used only if the hardware has a memory management unit, and if support for the memory management directives has been included in the system on which the task is to run.

4.1.1 Disk-Resident Overlay Structures

Disk-resident overlays conserve virtual address space and physical memory by sharing them with other overlays. Segments that are logically independent need not be present in memory at the same time. They, therefore, can occupy a common physical area in memory (and, therefore, common virtual address space) whenever either needs to be used.

The use of disk-resident overlays is shown in this section by an example, task TK1, which consists of four input files. Each input file consists of a single module with the same name as the file. The task is built by the command:

```
>TKB TK1=CNTRL,A,B,C
```

In this example, the modules A, B, and C are logically independent; that is:

A does not call B or C and does not use the data of B or C.

B does not call A or C and does not use the data of A or C.

C does not call A or B and does not use the data of A or B.

A disk-resident overlay structure can be defined in which A, B, and C are overlay segments that occupy the same storage area in physical memory. The flow of control for the task will be as follows:

CNTRL calls A and A returns to CNTRL.

CNTRL calls B and B returns to CNTRL.

CNTRL calls C and C returns to CNTRL.

CNTRL calls A and A returns to CNTRL.

In this example, the loading of overlays occurs only four times during the execution of the task. Therefore, the virtual address space and physical memory requirements of the task can be reduced without unduly increasing the overhead activity.

The effect of the use of an overlay structure on the allocation of virtual address space and physical memory for task TK1 is described in the following paragraphs.

The lengths of the modules are:

Module	Length (in octal)
CNTRL	20000 bytes
A	30000 bytes
B	20000 bytes
C	14000 bytes

Figure 4-1 shows the virtual address space and physical memory requirements as a result of building TK1 as a single-segment task on a system with memory management hardware.

The virtual address space and physical memory requirement to build TK1 as a single-segment task is 104000(8) bytes.

In contrast, Figure 4-2 shows the virtual address space and physical memory required to build TK1 as a result of using the overlay capability and building it as a multisegment task.

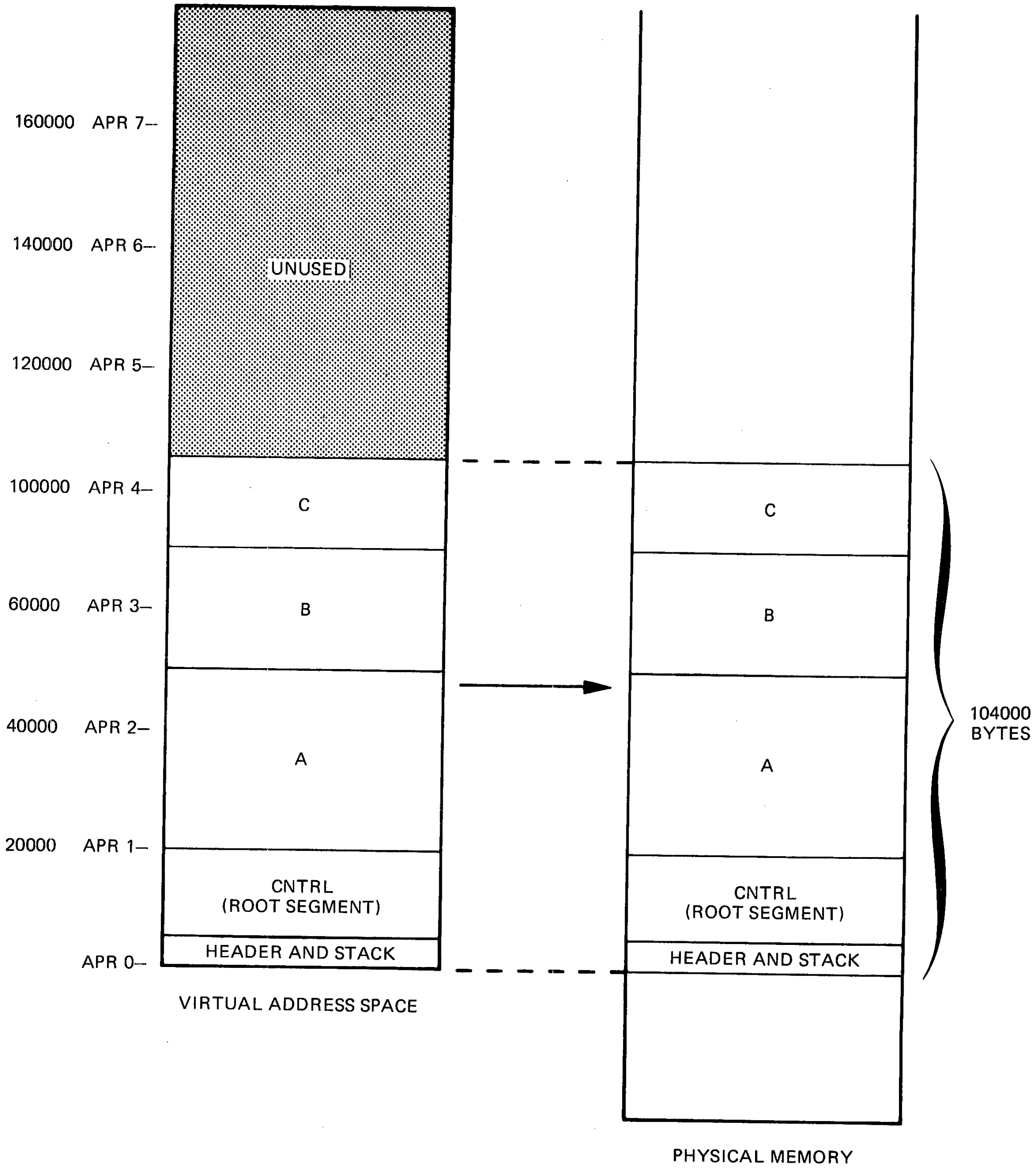


Figure 4-1 TK1 Built as a Single-Segment Task

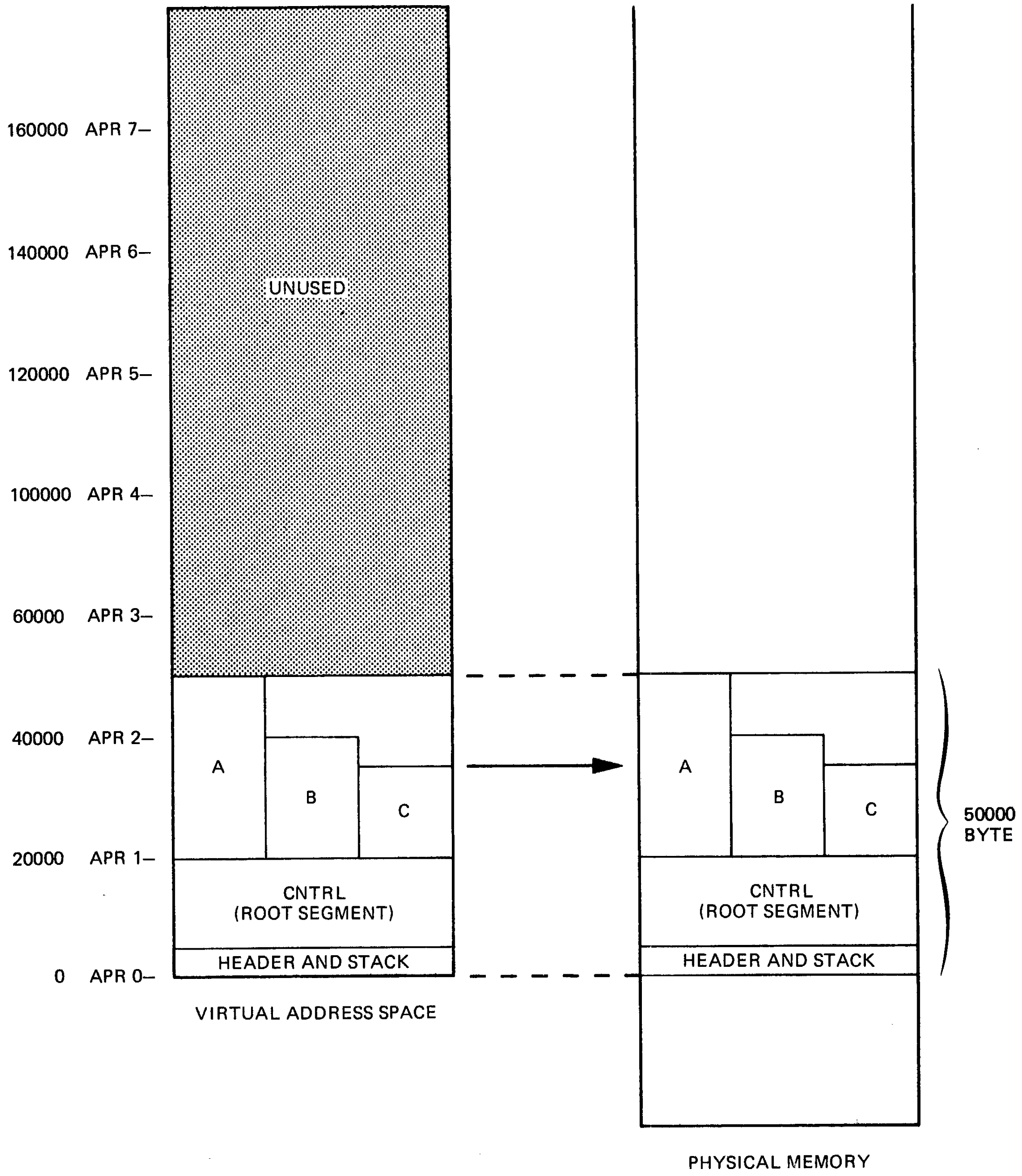


Figure 4-2 TK1 Built as a Multisegment Task

The multisegment task requires 50000(8) bytes.

NOTE

> In addition to module storage, storage is required for overhead in handling the overlay structures. This overhead is not reflected in this example.

In the use of the overlay capability, the amount of virtual address space and physical memory required for the task is determined by the length of the root segment and the length of the longest overlay segment. Overlay segments A and B in this example are much longer than overlay segment C. If A and B are divided into sets of logically independent modules, task storage requirements can be further reduced. Segment A can be divided into a control program (A0) and two overlays (A1 and A2). Segment A2 can then be divided into the main part (A2) and two overlays (A21 and A22). Similarly, segment B can be divided into a control module (B0) and two overlays (B1 and B2).

Figure 4-3 shows the virtual address space and physical memory required for the task produced by the additional overlays defined for A and B.

As a single-segment task, TK1 requires 104000 bytes of virtual address space and physical memory. The first overlay structure reduces the requirement by 34000 bytes. The second overlay structure further reduces the requirement by 14000 bytes.

The vertical and horizontal lines in the diagrams of Figures 4-2 and 4-3 represent the stqte of virtual address space and physical memory at various times during the calling sequence of TK1. For example, in Figure 4-3 the leftmost vertical line in both diagrams shows virtual address space and physical memory, respectively, when CNTRL, A0, and A1 are loaded. The next vertical line shows virtual address space and physical memory when CNTRL, A0, A2, and A21 are loaded, and so on.

The horizontal lines in the diagrams of Figures 4-2 and 4-3 indicate segments that share virtual address space and physical memory. For example, in Figure 4-3, the uppermost horizontal line of the task region in both diagrams shows A1, A21, A22, B1, B2, and C, all of which can use the same virtual address space and physical memory. The next horizontal line shows A1, A2, B1, B2, and C, and so on.

4.1.2 Memory-Resident Overlay Structures (Not Supported on RSX-11S)

The Task Builder provides for the creation of overlay segments that are loaded from disk only the first time they are referenced. Thereafter, they reside in memory. Memory-resident overlays share virtual address space just as disk-resident overlays do but, unlike disk-resident overlays, memory-resident overlays do not share physical memory. Instead, they reside in separate areas of physical memory, each segment aligned on a 32-word boundary. Memory-resident overlays save time for a running task because they do not need to be copied from a secondary storage device each time they are to overlay other segments. "Loading" a memory-resident overlay, reduces to mapping a set of shared virtual addresses to the unique physical area of memory containing the overlaying segment.

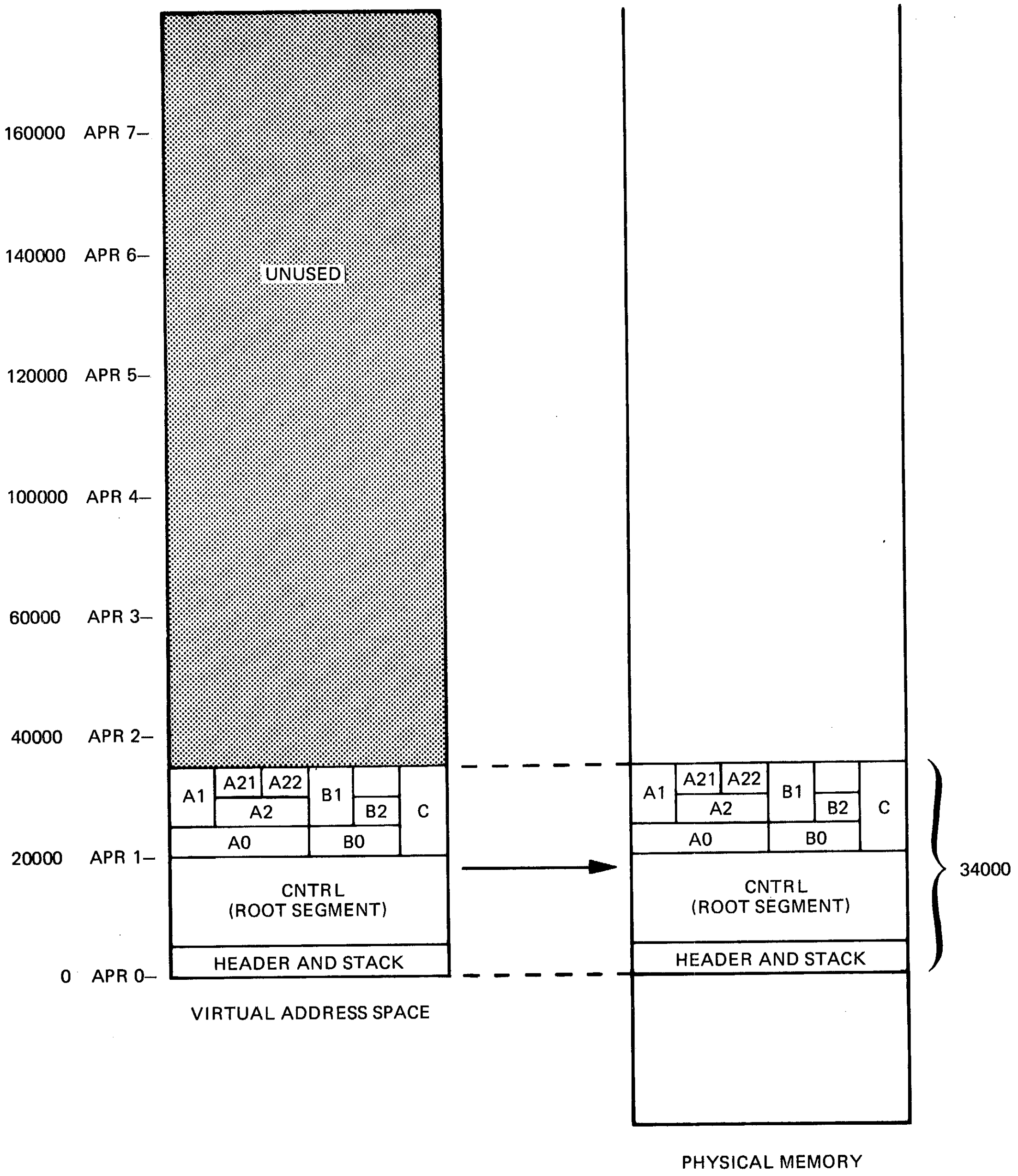


Figure 4-3 TK1 Built with Additional Overlay Defined

The use of memory-resident overlays is shown in this section by an example, task TK2, which consists of four input files. Each input file consists of a single module with the same name as the file. The task is built by the command:

```
>TKB TK2=CNTRL,D,E,F
```

In this example, the modules D, E, and F are logically independent; that is:

D does not call E or F and does not use the data of E or F.

E does not call D or F and does not use the data of D or F.

F does not call D or E and does not use the data of D or E.

A memory-resident overlay structure can be defined in which D, E, and F are overlay segments that occupy separate physical memory locations but which occupy the same virtual address space. The flow of control for the task will be as follows:

CNTRL calls D and D returns to CNTRL.

CNTRL calls E and E returns to CNTRL.

CNTRL calls F and F returns to CNTRL.

The effect of the use of a memory-resident overlay structure on the allocation of virtual address space and physical memory for task TK2 is described in the following paragraphs.

The lengths of the modules are:

Module	Length (in octal)
CNTRL	20000
D	10000
E	14000
F	12000

Figure 4-4 shows the virtual address space and physical memory requirements as a result of building TK2 as a single-segment task on a system with memory management hardware.

The virtual address space and physical memory requirements when TK2 is built as a single-segment task is 56000(8) bytes.

If TK2 is built using the Task Builder's memory-resident overlay capability, the relationship of virtual address space to physical memory changes, as shown in Figure 4-5.

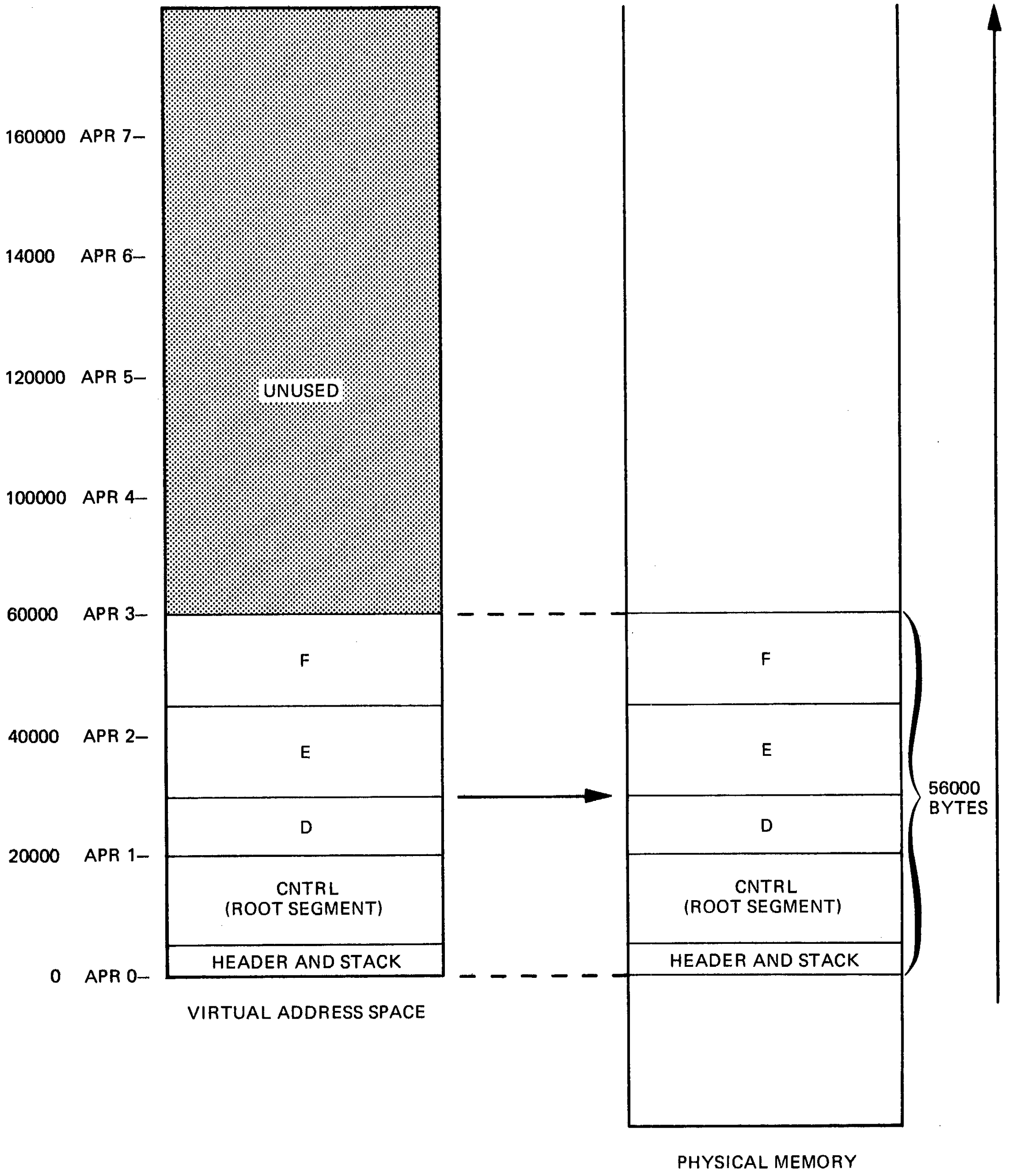


Figure 4-4 TK2 Built as a Single-Segment Task

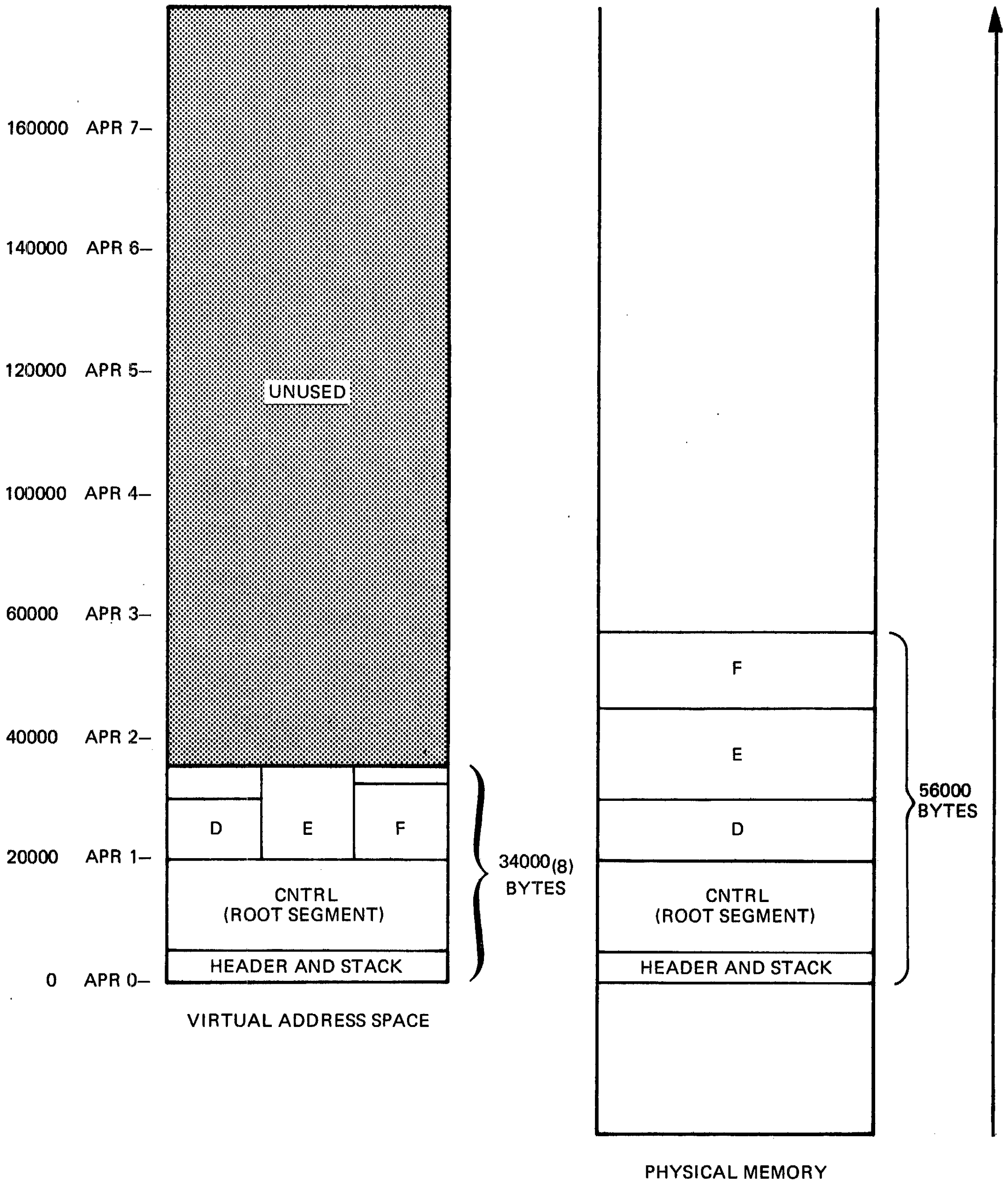


Figure 4-5 TK2 Built as a Memory-Resident Overlay

The physical memory requirements for TK2 do not change(56000(8) bytes), but the virtual address space requirements have been reduced to 34000(8) bytes. This represents a savings in virtual address space of 22000(8) bytes.

NOTE

> In addition to module storage, storage is required for overhead in handling the overlay structures. This overhead is not reflected in this example.

In Figure 4-5, the vertical and horizontal lines in the virtual address space diagram represent the state of virtual address space at various times during the calling sequence of TK2. The leftmost vertical line shows virtual address space when CNTRL and D are loaded and mapped. The next vertical line shows virtual address space when CNTRL and E are loaded and mapped, and the third vertical line shows virtual address space when CNTRL and F are loaded and mapped.

The uppermost horizontal line of the task region shows that segments D, E, and F share virtual address space.

When TK2 is activated, the Executive loads TK2's root segment into physical memory. The Executive loads segments D, E, and F into memory as they are called. Once all segments in the structure have been called, "loading" of the overlay segments reduces to the remapping of virtual address space to the physical locations in memory where the overlay segments permanently reside. Figures 4-6 and 4-7 illustrate the relationship between virtual address space and physical memory for task TK2 during four time periods:

- TIME 1 (Figure 4-6A) - TK2 is run and the system loads the root segment (CNTRL) into physical memory and maps to it.
- TIME 2 (Figure 4-6B) - CNTRL calls segment D. The system loads segment D into physical memory and maps to it. Segment D returns to CNTRL.
- TIME 3 (Figure 4-7A) - CNTRL calls segment E. The system loads segment E into physical memory, unmaps from segment D, and maps to segment E. Segment E returns to CNTRL.
- TIME 4 (Figure 4-7B) - CNTRL calls segment F. The system loads segment F into physical memory, unmaps from segment E, and remaps to segment F. Segment F returns to CNTRL

Figure 4-6A Time 1

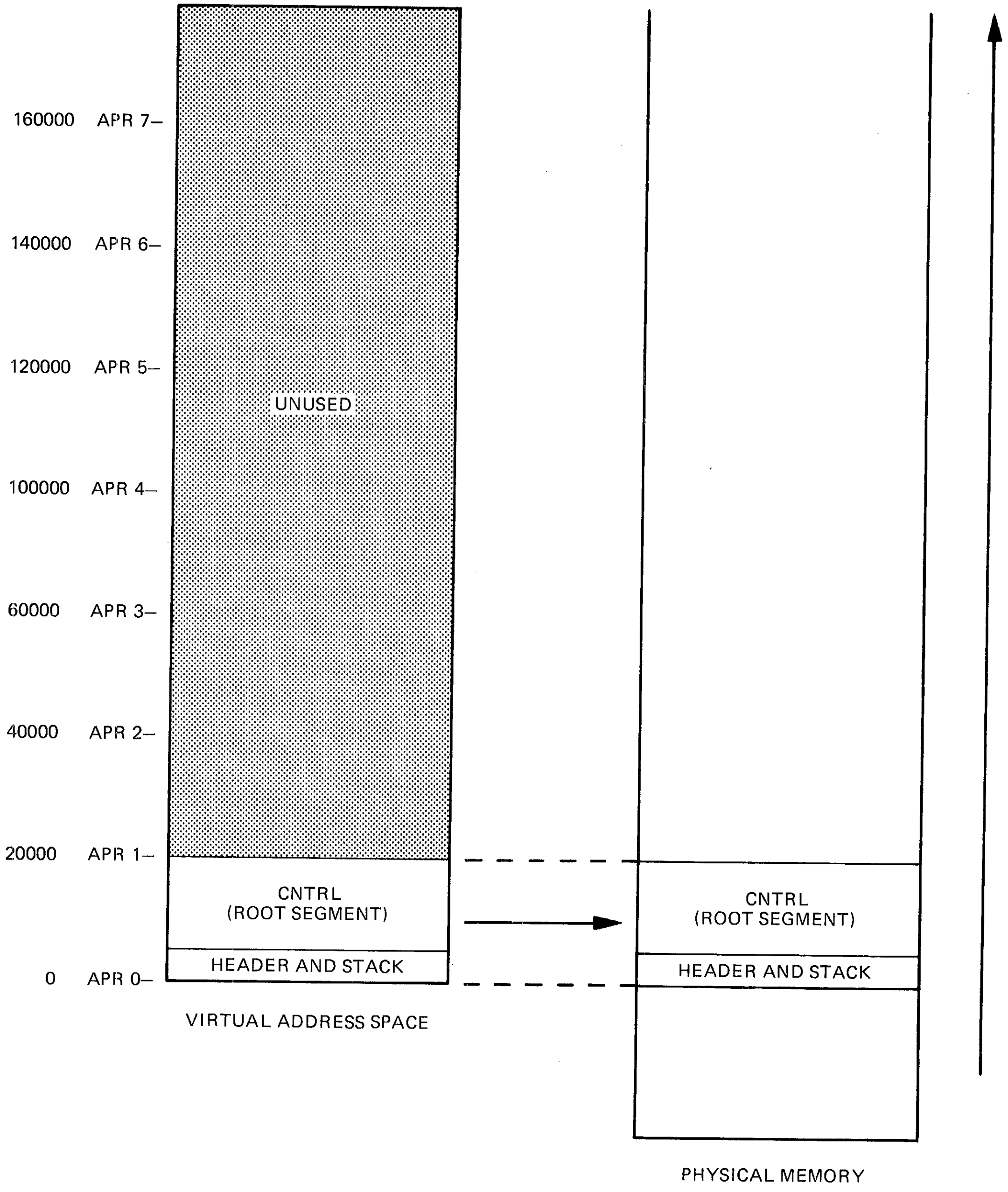


Figure 4-6 Relationship Between Virtual Address Space and Physical Memory -- Time 1 and Time 2

Figure 4-6B Time 2

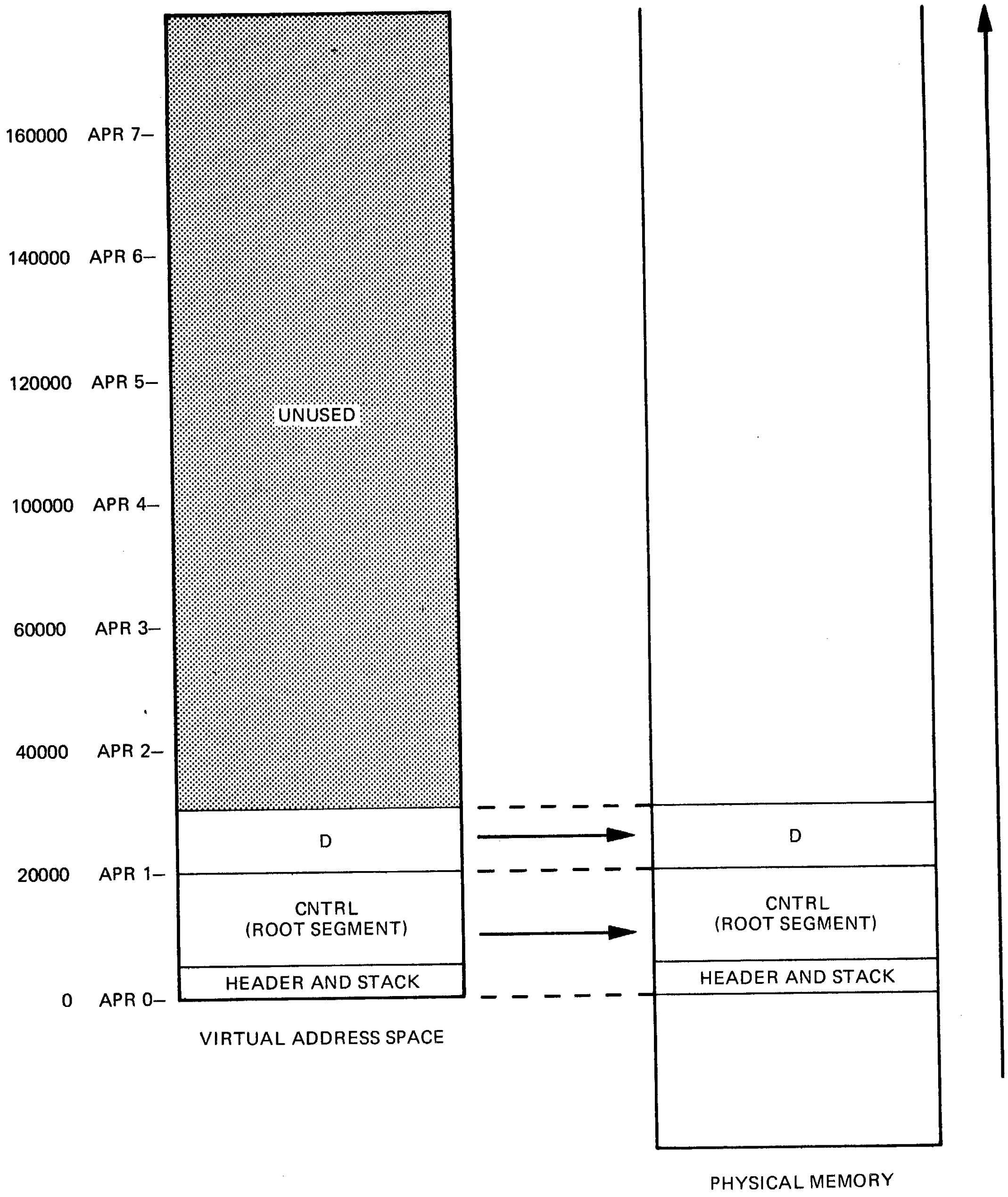


Figure 4-6 (Cont.) Relationship Between Virtual Address Space and Physical Memory -- Time 1 and Time 2

Figure 4-7A Time 3

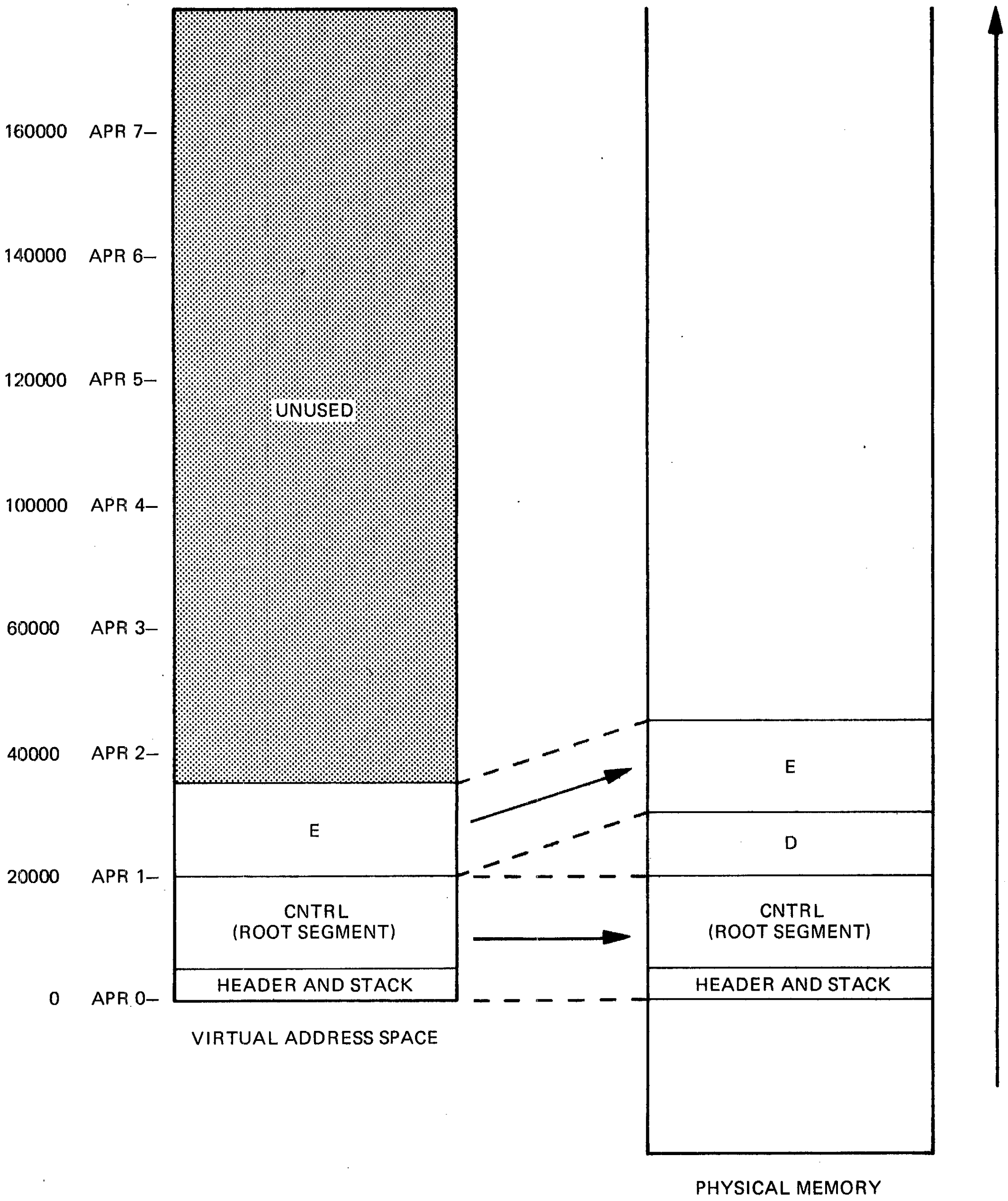


Figure 4-7 Relationship Between Virtual Address Space and Physical Memory -- Time 3 and Time 4

Figure 4-7B Time 4

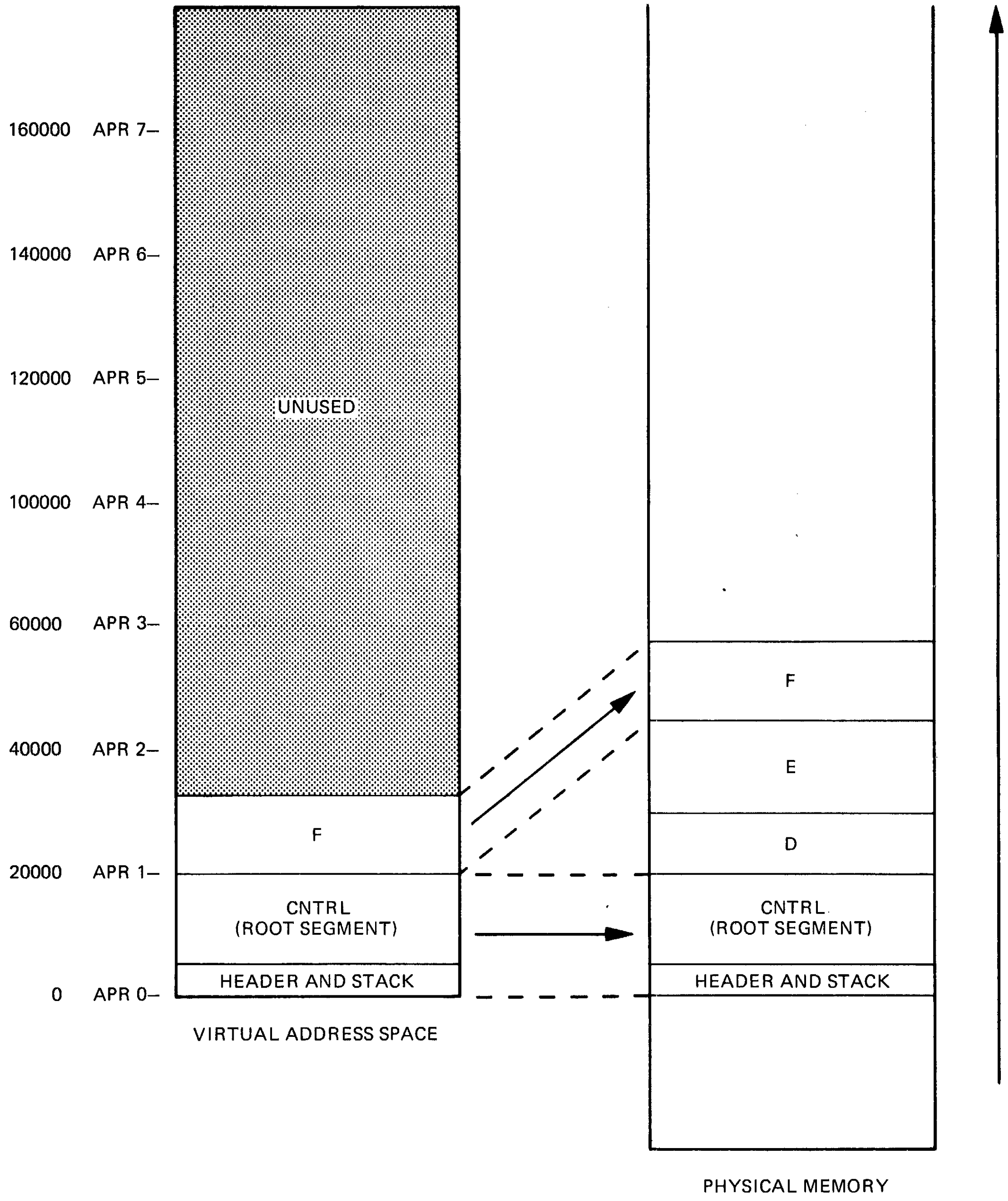


Figure 4-7 (Cont.) Relationship Between Virtual Address Space and Physical Memory -- Time 3 and Time 4

It is important to be careful in choosing whether to have memory-resident overlays in a structure. Careless use of these segments can result in inefficient allocation of virtual address space. This is because the Task Builder allocates virtual address space in blocks of 4K words. Consequently, the length of each overlay segment should approach that limit if you are to minimize waste. (A segment that is one word longer than 4K words, for example, will be allocated 8K words of virtual address space. All but one word of the second 4K words will be unusable.)

You can also conserve physical memory by maintaining control over the contents of each segment. The inclusion of a module in several memory-resident segments that overlay one another causes physical memory to be reserved for each extra copy of that module. Common modules, including those from the system object module library (SYSLIB), should be placed in a segment that can be accessed from all referencing segments.

The primary criterion for choosing to have memory-resident overlays is the need to save virtual address space when disk-resident overlays are either undesirable (because they would slow the system down unacceptably), or impossible (because the segments are part of a resident library or other shared region that must permanently reside in memory).

Memory-resident overlays can help you use large systems to better advantage because of the time savings realized when a large amount of physical memory is available. Resident libraries, in particular, can benefit from the virtual address space saved when they are divided into memory-resident segments.

4.2 OVERLAY TREE

The arrangement of overlay segments within the virtual address space of a task can be represented schematically as a tree-like structure. Each branch of the tree represents a segment. Parallel branches denote segments that overlay one another and therefore have the same virtual address; these segments must be logically independent. Branches connected end to end represent segments that do not share virtual address space with each other; these segments need not be logically independent.

The Task Builder provides an overlay description language (ODL) for representing an overlay structure consisting of one or more trees (the ODL is described in Section 4.4).

The allocation of virtual address space for TK1 (see Section 4.1.1) can be represented by the single overlay tree shown in Figure 4-8.

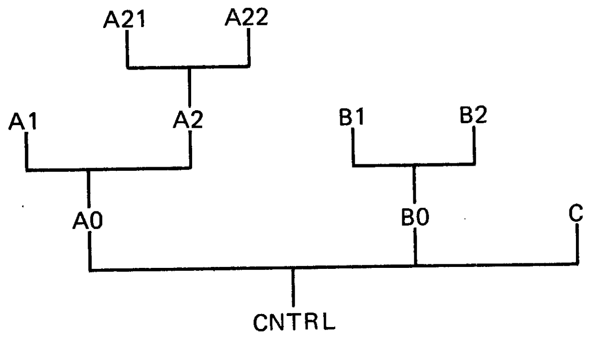


Figure 4-8 Overlay Tree for TK1

The tree has a root (CNTRL) and three main branches (A0, B0, and C). It also has six leaves (A1, A21, A22, B1, B2, and C).

The tree has as many paths as it has leaves. The path down is defined from the leaf to the root. For example:

A21-A2-A0-CNTRL

The path up is defined from the root to the leaf. For example:

CNTRL-B0-B1

Knowing the properties of the tree and its paths is important to understanding the overlay loading mechanism and the resolution of global symbols.

4.2.1 Loading Mechanism

Modules can call other modules that exist on the same path. The module CNTRL (Figure 4-8) is common to every path of the tree and, therefore, can call and be called by every module in the tree. The module A2 can call the modules A21, A22, A0, and CNTRL; but A2 cannot call A1, B1, B2, B0, or C.

When a module in one overlay segment calls a module in another overlay segment, the called segment must be in memory and mapped, or must be brought into memory. The methods for loading overlays are described in Chapter 5.

4.2.2 Resolution of Global Symbols in a Multisegment Task

In resolving global symbols for a multisegment task, the Task Builder performs the same activities that it does for a single-segment task. The rules defined in Chapter 2 for resolving global symbols in a single-segment task apply also in this case, but the scope of the global symbols is altered by the overlay structure.

In a single-segment task, any module can refer to any global definition. In a multisegment task, however, a module can only refer to a global symbol that is defined on a path that passes through the called segment.

The following points, illustrated in the tree diagram in Figure 4-9, describe the two distinct cases of multiply defined symbols and ambiguously defined symbols.

In a single-segment task, if you define two global symbols with the same name, the symbols are multiply defined, and an error message is produced.

In a multisegment task, you can define two global symbols with the same name if they are on separate paths, and not referenced from a segment that is common to both.

If you define a global symbol more than once on separate paths, but they are referenced from a segment that is common to both, the symbol is ambiguously defined. If you define a global symbol more than once on a single path, it is multiply defined.

The Task Builder's procedure for resolving global symbols is summarized as follows:

1. The Task Builder selects an overlay segment for processing.

2. The Task Builder scans each module in the segment for global definitions and references.

3. If the symbol is a definition, the Task Builder searches all segments on paths that pass through the segment being processed, and looks for references that must be resolved.

4. If the symbol is a reference, the Task Builder performs the tree search as described in step 3, looking for an existing definition.

5. If the symbol is new, the Task Builder enters it in a list of global symbols associated with the segment.

Overlay segments are selected for processing in an order corresponding to their distance from the root. That is, the Task Builder processes the segment farthest from the root first, before processing an adjoining segment.

When the Task Builder processes a segment, its search for global symbols proceeds as follows:

- The segment being processed
- All segments toward the root
- All segments away from the root
- All co-trees (see Section 4.5)

Figure 4-9 illustrates the resolution of global symbols in a multisegment task.

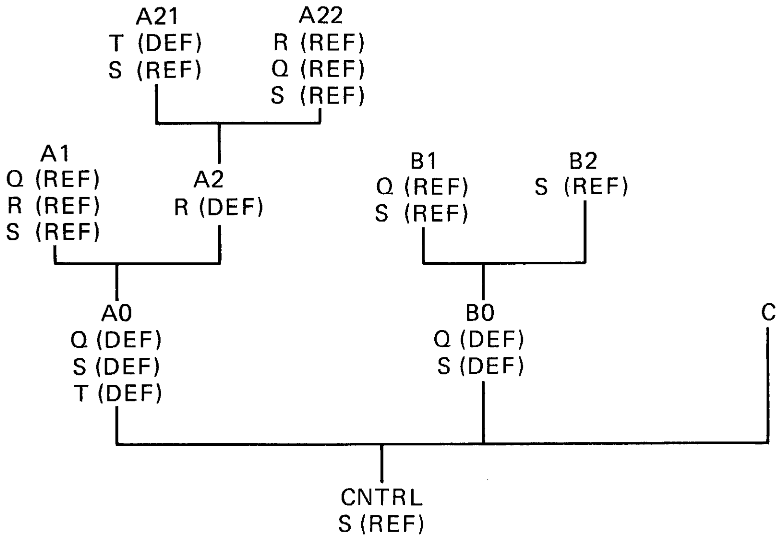


Figure 4-9 Resolution of Global Symbols in a Multisegment Task

The following notes discuss the resolution of references in Figure 4-9:

1. The global symbol Q is defined in both segment A0 and segment B0. The references to Q in segment A22 and in segment A1 are resolved by the definition in A0. The reference to Q in B1 is resolved by the definition in B0. The two definitions of Q are distinct in all respects and occupy different overlay paths.

2. The global symbol R is defined in segment A2. The reference to R in A22 is resolved by the definition in A2 because there is a path to the reference from the definition (CNTRL-A0-A2-A22). The reference to R in A1, however, is undefined because there is no definition for R on a path through A1.

3. The global symbol S is defined in both segment A0 and segment B0. References to S from segments A1, A21, or A22 are resolved by the definition in A0, and references to S in B1 and B2 are resolved by the definition in B0. However, the reference to S in CNTRL cannot be resolved because there are two definitions of S on separate paths through CNTRL. The global symbol S is ambiguously defined.

4. The global symbol T is defined in both segment A21 and segment A0. Since there is a single path through the two definitions (CNTRL-A0-A2-A21), the global symbol T is multiply defined.

4.2.3 Resolution of Global Symbols from the Default Library

The process of resolving global symbols may require two passes over the tree structure. The global symbols discussed in the previous section are included in user-specified input modules that the Task Builder scans in the first pass. If any undefined symbols remain, the Task Builder initiates a second pass over the structure in an attempt to resolve such symbols by searching the default object module library (normally LB0:[1,1]SYSLIB.OLB). The Task Builder reports any undefined symbols remaining after its second pass.

When multiple tree structures (co-trees) are defined, as described in Section 4.5, any resolution of global symbols across tree structures during a second pass can result in multiple or ambiguous definitions. In addition, such references can cause overlay segments to be inadvertently displaced from memory by the overlay loading routines, thereby causing run-time failures. To eliminate these conditions, the tree search on the second pass is restricted to:

- The segment in which the undefined reference has occurred
- All segments in the current tree that are on a path through the segment
- The root segment

When the current segment is the main root, the tree search is extended to all segments. You can unconditionally extend the tree search to all segments by including the FU (full) switch in the task image file specification. (Refer to Chapter 6 for a description of the FU switch.)

4.2.4 Allocation of Program Sections in a Multisegment Task

One of a program section's attributes indicates whether the program section is local (LCL) to the segment in which it is defined or is global (GBL).

Local program sections with the same name can appear in any number of segments. The Task Builder allocates virtual address space for each local program section in the segment in which it is declared. Global program sections that have the same name, however, must be resolved by the Task Builder.

When a global program section is defined in several overlay segments along a common path, the Task Builder allocates all virtual address space for the program section in the overlay segment closest to the root.

FORTRAN common blocks are translated into global program sections with the overlay (OVR) attribute. In Figure 4-10, the common block COMA is defined in modules A2 and A21. The Task Builder allocates the virtual address space for COMA in A2 because that segment is closer to the root than the segment that contains A21.

If the segments A0 and B0 use a common block COMAB, however, the Task Builder allocates the virtual address space for COMAB in both the segment that contains A0 and the segment that contains B0. A0 and B0 cannot communicate through COMAB. When the overlay segment containing B0 is loaded, any data stored in COMAB by A0 is lost.

You can specify the allocation of program sections explicitly. If A0 and B0 need to share the contents of COMAB, you can force the allocation of this program section into the root segment by the use of the .PSECT directive of the Task Builder's overlay description language, described in Section 4.4.

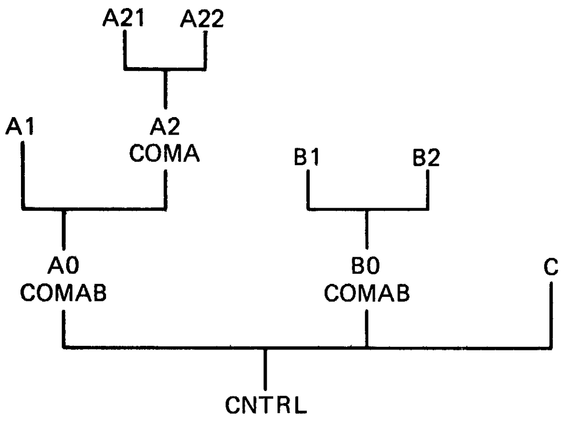


Figure 4-10 Resolution of Program Sections for TK1

4.3 OVERLAY DATA STRUCTURES AND RUN-TIME ROUTINES

When the Task Builder constructs an overlaid task, it builds additional data structures and adds them to the task image. It also includes into the task image a number of system library routines (called overlay run-time routines). The data structures contain information about the overlay segments and describe the relationship of each segment in the tree to the other segments in the tree. The overlay run-time routines use the data structures to facilitate the loading of the segments and to provide the necessary linkages from one segment to another at run time.

The Task Builder links the majority of data structures and all of the overlay run-time routines into the root segment of the task. The number and type of data structures, and the functions the routines perform, depend on two considerations:

- Whether the task is built to use the Task Builder's autoload or manual load facilities
- Whether the overlay segment is memory resident or disk resident

These considerations have a marked impact on the size and operation of the task. Chapter 5 describes the Task Builder's autoload and manual load facilities and describes the methods for loading overlays. Appendix B describes the data structures and their contents in detail.

The contents of the root segment for a task with an overlay structure are discussed briefly in the following paragraphs.

Depending on the considerations above, some or all of the following data structures are required by the overlay run-time routines:

- Segment tables
- Autoload vectors
- Window descriptors
- Region descriptors

Figure 4-11 shows a typical overlay root segment structure.

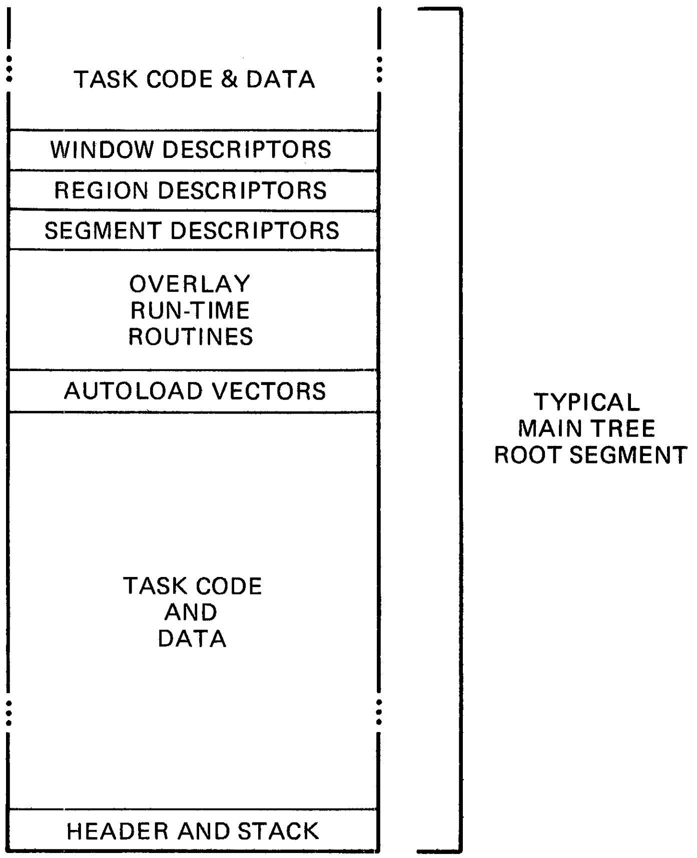


Figure 4-11 Typical Overlay Root Segment Structure

There is a segment descriptor for every segment in the task. The descriptor contains information about the load address, the length of the segment, and the tree linkage.

When you build an overlaid task autoloadable, autoload vectors appear in the root segment and in every segment that calls modules in another segment located farther away from the root of the tree.

Window descriptors are allocated whenever a memory-resident overlay structure is defined for the task. The descriptor contains information required by the Create Address Window system directive (CRAW$). One descriptor is allocated for each memory-resident overlay segment.

Region descriptors are allocated whenever a task is linked to a shared region containing memory-resident overlays. The descriptor contains information required by the Attach Region system directive (ATRG$).

4.4 OVERLAY DESCRIPTION LANGUAGE

The Task Builder provides a language, called the Overlay Description Language (ODL), that allows you to describe the overlay structure of a task. An overlay description is a text file consisting of a series of ODL directives, one directive per line. You enter this file in a Task Builder command line, and identify it as an ODL file by specifying the MP switch (see Chapter 6) to the file name. If you specify an ODL file to the Task Builder, it must be the only input file you specify.

An ODL line takes the form:

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

A label is required only for the .FCTR directive (see Section 4.4.2).

The ODL directives are listed below and described in Sections 4.4.1 through 4.4.6:

- .ROOT and .END
- .FCTR
- ! (exclamation point operator)
- .NAME
- .PSECT
- @ (at sign; indirect command file specifier)

Directives act upon argument-list which consists of named input files, overlay segments, program sections, and lines in the ODL file itself. Operators group these named task elements, or attach attributes to them.

If the name belongs to a file, you can enter a complete file specification. Defaults for omitted parts of the file specification are as described in Chapters 1 and 6, except that the default device is SY0:, and the default UFD is taken from the terminal UIC.

In addition, the following restrictions apply to argument-lists:

- You can only use the dot character (.) in a file name.
- Comments cannot appear on a line ending with a file name.

4.4.1 .ROOT and .END Directives

Each overlay description must begin with one .ROOT directive and end with one .END directive. The .ROOT directive tells the Task Builder where to start building the tree, and the .END directive tells the Task Builder where the input ends.

The arguments of the .ROOT directive use three operators to express concatenation, overlaying, and memory residency.

- The hyphen (-) operator indicates the concatenation of virtual address space. For example, X-Y means that sufficient virtual address space will be allocated to contain segment X and segment Y simultaneously. The Task Builder allocates segment X and segment Y in sequence.
- The exclamation point (!) operator indicates memory-residency of overlays. (This operator is discussed in Section 4.4.3.)
- The comma (,) operator, appearing within parentheses, indicates the overlaying of virtual address space. For example, Y,Z means that virtual address space can contain either segment Y or segment Z. If no exclamation point (!) precedes the left parenthesis, segment Y and segment Z also share physical memory.

  The comma (,) operator is also used to define multiple tree structures (as described in Section 4.5.1).

You use parentheses to delimit a group of segments that start at the same virtual address. The number of nested parenthetical groups cannot exceed 16.

For example:

```
.ROOT X-(Y,Z-(Z1,Z2))
.END
```

These directives describe the tree and its corresponding virtual address space shown in Figure 4-12:

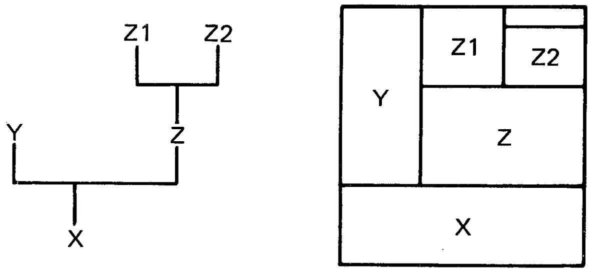


Figure 4-12 Tree and Virtual Address Space Diagram

To create the overlay description for the task TK1 in Figure 4-3 (Section 4.1.1), you could create a file called TFIL.ODL that contains the directives:

```
.ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0-(B1,B2),C)
.END
```

To build the task with that overlay structure, you would type:

```
>TKB TK1=TFIL/MP
```

The MP switch in the command string above tells the Task Builder that there is only one input file (TFIL.ODL), and that this file contains an overlay description for the task.

4.4.2 .FCTR Directive

The .FCTR directive allows you to build large, complex trees and represent them clearly.

The .FCTR directive has a label at the beginning of the ODL line that is pointed to by a reference in a .ROOT or another .FCTR statement. The label must be unique with respect to module names and other labels. The .FCTR directive allows you to extend the tree description beyond a single line, and thus allows you to provide a clearer description of the overlay. (There can be only one .ROOT directive.)

For example, to simplify the tree given in the file TFIL (described in Section 4.4.1), you could use the .FCTR directive in the overlay description as follows:

```
            .ROOT CNTRL-(AFCTR,BFCTR,C)
AFCTR:      .FCTR A0-(A1,A2-(A21,A22))
BFCTR:      .FCTR B0-(B1,B2)
            .END
```

The label BFCTR is used in the .ROOT directive to designate the argument of the .FCTR directive, B0-(B1,B2). The resulting overlay description is easier to interpret than the original description. The tree consists of a root, CNTRL, and three main branches. Two of the main branches have sub-branches.

The .FCTR directive can be nested to a level of 16. For example, you could further modify TFIL as follows:

```
            .ROOT CNTRL-(AFCTR,BFCTR,C)
AFCTR:      .FCTR A0-(A1,A2FCTR)
A2FCTR:     .FCTR A2-(A21,A22)
BFCTR:      .FCTR B0-(B1,B2)
            .END
```

4.4.3 Exclamation Point Operator

The exclamation point operator allows you to specify overlay segments that will reside in memory rather than on disk (see Section 4.1.2). You specify memory residency by placing an exclamation point (!) immediately before the left parenthesis enclosing the segments to be affected. The overlay description for task TK2 in Figure 4-4 (Section 4.1.2) is as follows:

```
.ROOT CNTRL-!(D,E,F)
.END
```

In the example above, segments D, E, and F are declared resident in separate areas of physical memory. The single starting virtual address for D, E, and F is determined by the Task Builder, by rounding the octal length of segment CNTRL up to the next 4K boundary. The physical memory allocated to segments D, E, and F is determined by rounding the actual length of each segment to the next 32-word boundary (256-word boundary if the CM switch is in effect), and adding this value to the total memory required by the task.

The exclamation point operator applies only to segments at the first level inside a pair of parentheses; segments in parentheses nested within that level are not affected. It is therefore possible to define an overlay structure that combines the space-saving attributes of disk-resident overlays with the speed of memory-resident overlays. For example:

```
.ROOT A-!(B1-(B2,B3),C)
.END
```

In this example, B1 and C are declared memory resident by the exclamation point operator. B2 and B3 are declared disk resident, however, because no exclamation point operator precedes the parentheses enclosing them.

Note that while a memory-resident overlay can call a disk-resident overlay, the converse is not legal; that is, you cannot use an exclamation point for segments emanating from a disk-resident segment. For example, you cannot build the following structure:

```
.ROOT A-(B1-!(B2,B3),C)    ; this overlay description is illegal
.END
```

In this example, B1 is declared disk resident, so it is illegal to use the exclamation point to declare B2 and B3 memory resident.

4.4.4 .NAME Directive

The .NAME directive allows you to specify a name for a segment, and then to assign attributes to the segment. The name must be unique with respect to file names, program section names, .FCTR labels, and other segment names used in the overlay description. The chief uses of this directive are:

1. To name a segment uniquely that is to be loaded through the manual load facility (see Chapter 5)

2. To permit a segment that does not contain executable code to be loaded through the autoload mechanism

The format of the .NAME directive is:

```
.NAME segname[,attr][,attr]
```

segname

A 1- to 6-character name; this name can consist of the Radix-50 characters A-Z, 0-9, and $ (the period (.) cannot be used).

attr

One of the following:

GBL	The name is entered in the segment's global symbol table. The GBL attribute makes it possible to load nonexecutable overlay segments by means of the autoload mechanism (see Chapter 5).

NODSK No disk space is allocated to the named segment.

If a data overlay segment has no initial values, but will have its contents established by the running task, no space for the named segment on disk need be reserved. If the code attempts to establish initial values for data in a segment for which no disk space is allocated (a segment with the NODSK attribute), the Task Builder gives a fatal error.

NOGBL The name is not entered in the segment's global symbol table.

If the GBL attribute is not present, NOGBL is assumed.

DSK Disk storage is allocated to the named segment.

If the NODSK attribute is not present, DSK is assumed.

The attributes described are not attached to a segment until the name is used in a .ROOT or .FCTR statement that defines an overlay segment. When multiple segment names are applied to a segment, the attributes of the last name given are in effect.

In the following modified ODL file for TK1 (Figure 4-3 of Section 4.1.1), the three main branches, A0, B0, and C, are provided with names by specifying them in the .NAME directive and using them in the .ROOT directive. The default attributes NOGBL and DSK are in effect for BRNCH1 and BRNCH3, but BRNCH2 has the complementary attributes (GBL and NODSK) that will cause the Task Builder to enter the name BRNCH2 into the segment's global symbol table and to allocate disk space for the segment to be suppressed. BRNCH2 contains uninitialized storage to be utilized at runtime.

```
          .NAME BRNCH1
          .NAME BRNCH2,GBL,NODSK
          .NAME BRNCH3
          .ROOT CNTRL-!(BRNCH1-AFCTR,*BRNCH2-BFCTR,BRNCH3-C)
AFCTR:    .FCTR A0-(A1,A2-(A21,A22))
BFCTR:    .FCTR B0-*!(B1,B2)
          .END
```

(The asterisk (*) is the autoload indicator; it is discussed in Chapter 5.)

The data overlay segment BRNCH2 is loaded by including the following statement in the program.

```
CALL BRNCH2
```

This action is immediately followed by an automatic return to the next instruction in the program.

You can also use segment names in making patches with the ABSPAT and GBLPAT options (see Chapter 6).

NOTE

> In the absence of a unique .NAME specification, the Task Builder establishes a segment name, using the first program section file, or library module name occurring in the segment.

4.4.5 .PSECT Directive

You can use the .PSECT directive to directly specify the placement of a global program section in an overlay structure. The name of the program section (a 1- to 6-character name consisting of the Radix-50 characters A-Z, 0-9, and $) and its attributes are given in the .PSECT directive. Thus, you can use the name to indicate to the Task Builder the segment to which the program section will be allocated. An example of the use of .PSECT is given in the modified version of task TK1 (the original version is shown in Figure 4-3 in Section 4.1.1) shown below.

In this example, TK1 has a disk-resident overlay structure. The example assumes that the programmer was careful about the logical independence of the modules in the overlay segment, but failed to take into account the requirement for logical independence in multiple executions of the same overlay segment.

The flow of task TK1 can be summarized as follows. CNTRL calls each of the overlay segments, and the overlay segment returns to CNTRL in the order A, B, C, A. Module A is executed twice. The overlay segment containing A must be reloaded for the second execution.

Module A uses a common block named DATA3. The Task Builder allocates DATA3 to the overlay segment containing A. The first execution of A stores some results in DATA3. The second execution of A requires these values. In this disk-resident overlay structure, however, the values calculated by the first execution of A are overlaid. When the segment containing A is read in for the second execution, the common block is in its initial state.

To permit the two executions of A to communicate, a .PSECT directive is used to force the allocation of DATA3 into the root. The indirect command file for TK1, TFIL.ODL, is modified as follows:

```
                .PSECT DATA3,RW,GBL,REL,OVR
                .ROOT CNTRL-DATA3-(AFCTR,BFCTR,C)
     AFCTR:     .FCTR A0-(A1,A2-(A21,A22))
     BFCTR:     .FCTR B0-(B1,B2)
                .END
```

The attributes RW, GBL, REL, and OVR are described in Chapter 2.

4.4.6 Indirect Command Files

The Overlay Description Language processor can accept ODL text indirectly, that is, specified in an indirect command file. If an at sign (@) appears as the first character in an ODL line, the processor will read text from the file specified immediately after the at sign. The processor accepts the ODL text from the file as input, at the point in the overlay description where the file is specified.

For example, suppose you create a file, called BIND.ODL, that contains the text:

```
     B:    .FCTR B1-(B2,B3)
```

This text can be replaced by a line beginning with @BIND, at the position where the text would have appeared:

```
                   Indirect                       Direct

               .ROOT A-(B,C)                 .ROOT A-(B,C)
     C:        .FCTR C1-(C2,C3)         C:   .FCTR C1-(C2,C3)
     @BIND                              B:   .FCTR B1-(B2,B3)
               .END                          .END
```

The Task Builder allows two levels of indirection.

4.5 MULTIPLE-TREE STRUCTURES

You can define more than one tree within an overlay structure. These multiple tree structures consist of a main tree and one or more co-trees. The root segment of the main tree is loaded by the Executive when the task is made active, while segments within each co-tree are loaded through calls to the overlay run-time routines. Except for this distinction, all overlay trees have identical characteristics: a root segment that resides in memory, and two or more overlay segments.

The main property of a structure containing more than one tree is that storage is not shared among trees. Any segment in a tree can be referred to from another tree without displacing segments from the calling tree. Routines that are called from several main tree overlay segments, for example, can overlay one another in a co-tree. The same considerations in deciding whether to create memory-resident overlays or disk-resident overlays in a single-tree structure apply in building a structure containing co-trees.

4.5.1 Defining a Multiple-Tree Structure

Multiple-tree structures are specified within the Overlay Description Language by extending the function of the comma operator. As described in Section 4.4, this operator, when included within parentheses, defines a pair of segments that share storage. The inclusion of the comma operator outside all parentheses delimits overlay trees. The first overlay tree thus defined is the main tree. Subsequent trees are co-trees. For example:

```
               .ROOT      X,Y
     X:        .FCTR      X0-(X1,X2,X3)
     Y:        .FCTR      Y0-(Y1,Y2)
               .END
```

In this example, two overlay trees are specified: 1) a main tree containing the root segment X0 and three overlay segments, and 2) a co-tree consisting of root segment Y0 and two overlay segments. The Executive loads segment X0 into memory when the task is activated. The task then loads the remaining segments through calls to the overlay run-time routines.

A co-tree must have a root segment to establish linkage with its own overlay segments. However, co-tree root segments need not contain code or data and, therefore, can be 0 length. You can create a segment of this type, called a null segment, by means of the .NAME directive. The previous example is modified, as shown below, to move file Y0.OBJ to the root and include a null segment.

```
        .ROOT     X,Y
X:      .FCTR     X0-Y0-(X1,X2,X3)
        .NAME     YNUL
Y:      .FCTR     YNUL-(Y1,Y2)
        .END
```

The null segment YNUL is created by use of the .NAME directive, and replaces the co-tree root that formerly contained Y0.OBJ.

4.5.2 Multiple-Tree Example

The following example illustrates the use of multiple trees to reduce the size of the task.

In this example, the root segment CNTRL of task TK1 (described in Section 4.1.1) has had two routines added to it: CNTRLX and CNTRLY. The routines are logically independent of each other, and both are approximately 4000(8) bytes long. However, the routines have been placed in the root segment of TK1 instead of being overlaid because both routines must be accessed from modules on all paths of the tree. In a single-tree overlay structure, the root segment is the only segment common to all paths of the tree. The schematic diagram for the modified structure is shown in Figure 4-13.

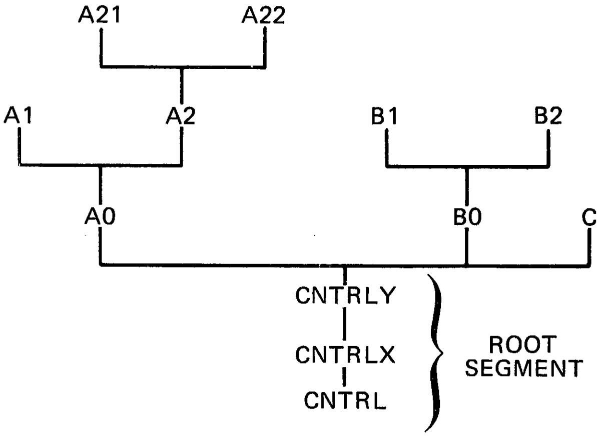


Figure 4-13 Overlay Tree for Modified TK1

One possible overlay description for this structure is shown below.

```
             .ROOT CNTRL-CNTRLX-CNTRLY-(AFCTR,BFCTR,C)
    AFCTR:   .FCTR A0-(A1,A2FCTR)
    A2FCTR:  .FCTR A2-(A21,A22)
    BFCTR:   .FCTR B0-(B1,B2)
             .END
```

Because TK1 consists of disk-resident overlays and the new routines are concatenated within the overlay structure, the new routines add 10000(8) bytes to both the virtual address space and physical memory requirements of the task. However, the added routines consume more virtual address space than might be expected, as shown in Figure 4-14.

The expansion of TK1's virtual address space requirements caused the task to extend 4000(8) bytes beyond the next highest 4K word boundary (APR 2). Because the Executive must use an additional mapping register (APR2) the apparent cost in virtual address space above APR 2 of 4000(8) bytes is in fact 20000(8) bytes. (Compare the diagram in Figure 4-14 with the diagram in Figure 4-3.) The shaded portion of the unused virtual address space in Figure 4-14 represents the portion of virtual address space that is allocated but is unusable as allocated.

Small tasks, such as TK1, are seldom adversely affected by the inefficient allocation of virtual address space, but larger tasks may be. For example, a large task that contains code to create dynamic regions (see Chapter 3) or that contains Executive directives to extend its task region (see the RSX-11M/M-PLUS Executive Reference Manual) will require at least 4K words of virtual address space to map each region. In such a task, the use of co-trees can often save virtual address space and can, therefore, be of paramount importance. TK1 can be modified to reflect this.

As noted earlier, the routines CNTRLX and CNTRLY are logically independent. Logical independence is a primary requirement for all segments that overlay each other. However, CNTRLX and CNTRLY cannot be structured into either of the main branches of TK1's tree because it is further required that the routines be accessible from modules on all paths of the tree. Therefore, the only way CNTRLX and CNTRLY can be overlaid and still meet all of these requirements is through a co-tree structure. Figure 4-15 shows the schematic representation of TK1 as a co-tree structure.

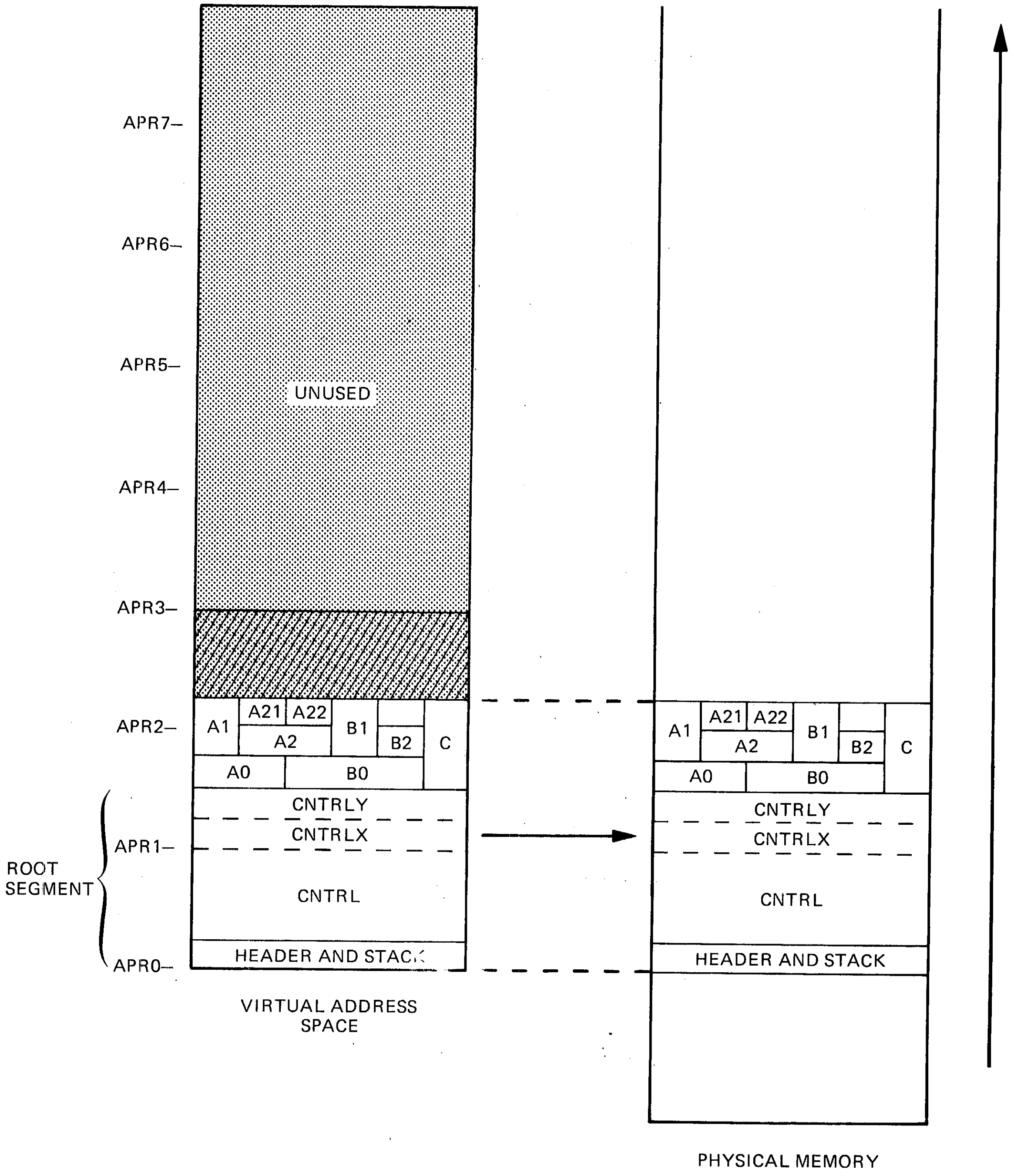


Figure 4-14 Virtual Address Space and Physical Memory for Modified TK1

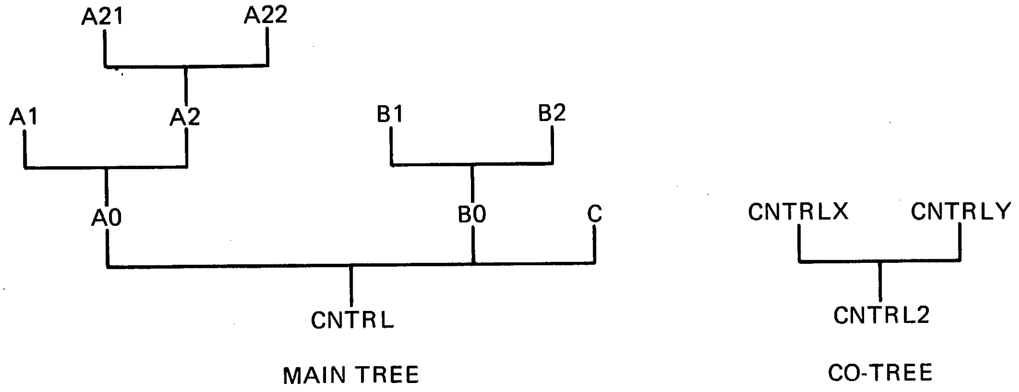


Figure 4-15 Overlay Co-Tree for Modified TK1

The root segment CNTRL2 of the co-tree is a null segment. It contains no code or data and has a length of 0. As noted earlier, the root segment is required by the Task Builder in order to establish linkage with the overlay segments. One possible overlay description for building TK1 as a two-tree structure is shown below.

```
          .NAME CNTRL2
          .ROOT CNTRL-(AFCTR,BFCTR,C),CNTRL2-(CNTRLX,CNTRLY)
AFCTR:    .FCTR A0-(A1,A2FCTR)
A2FCTR:   .FCTR A2-(A21,A22)
BFCTR:    .FCTR B0-(B1,B2)
          .END
```

The co-tree is defined in the .ROOT directive by placing the comma operator outside all parenthesis (immediately before CNTRL2). The .NAME directive creates the null root segment. Figure 4-16 shows the new relationship between virtual address space and physical memory.

The diagrams in Figure 4-16 illustrate the savings (4000(8) bytes) in both virtual address space and physical memory that is realized by overlaying CNTRLX and CNTRLY. What may be more important in some applications, however, is that the top of TK1's task region has dropped below the 4K boundary of APR 2. TK1 has gained 4K words of potentially usable virtual address space.

NOTE

> The numbers used in this example have been simplified for illustrative purposes. In addition, the storage required for overhead in handling the overlay structures is not reflected in this example.

Because the null root CNTRL2 is 0 bytes long, it does not require any virtual address space or physical memory and, therefore, does not appear in the diagrams in Figure 4-16.

Finally, you can define any number of co-trees. Additional co-trees can access all modules in the main tree and other co-trees.

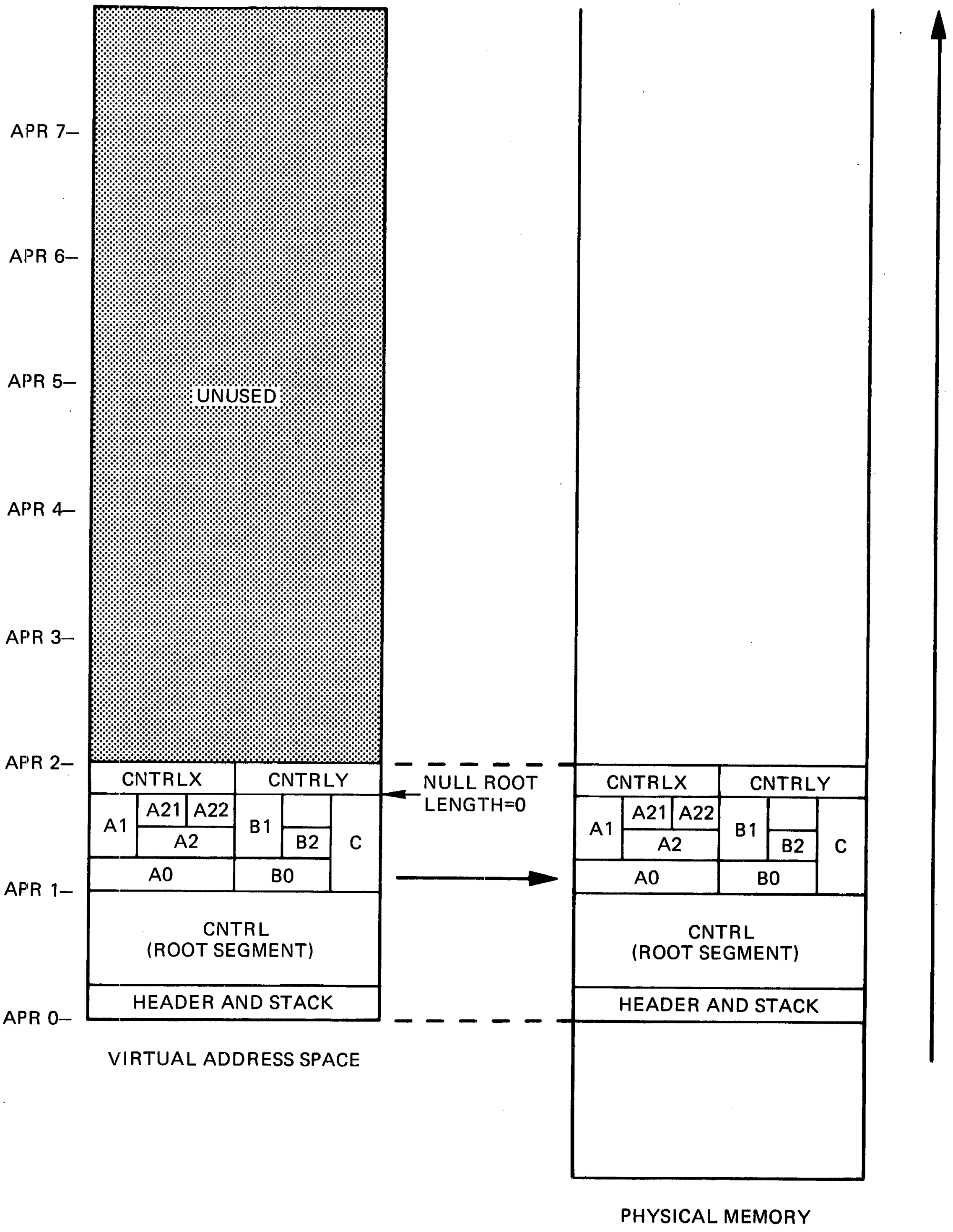


Figure 4-16 Virtual Address Space
and Physical Memory for TK1 as a Co-Tree

4.6 OVERLAYING PROGRAMS WRITTEN IN A HIGH-LEVEL LANGUAGE

Programs written in a high-level language usually require the use of a large number of library routines in order to execute. Unless care is taken when overlaying such programs, the following problems can occur:

- Task Builder throughput may be drastically reduced because of the number of library references in each overlay segment.
- Library references from the default object module library, that are resolved across tree boundaries can result in unintentional displacement of segments from memory at runtime.
- Attempts to task build such programs can result in multiple and ambiguous symbol definitions when a co-tree structure is defined.

The following procedures are effective in solving these problems:

- Task Builder throughput can be increased if you link commonly used library routines into the main root segment.
- Ambiguous and multiple definitions, and cross-tree references can be eliminated by using the NOFU switch (the Task Builder default) to restrict the scope of the default library search.

If sufficient memory is available, the object time system can be effectively placed in the root segment by building a memory-resident library. This also reduces total system memory requirements if other tasks are also currently using the library.

If a memory-resident library cannot be built, you can force library modules into the root by preparing a list of the appropriate global references and linking the object module into the root segment.

For other ways to reduce task size, you should consult the user's guide for the language you are using.

4.7 EXAMPLE 9: BUILDING AN OVERLAY

The text in this section and the figures associated with it illustrate the building of an overlay structure. For this example, the routines of the resident library LIB.TSK and the task that refer to it, MAIN.TSK (from Example 4, Chapter 3), are assembled as separate modules and built as an overlaid task. This task is built first with disk-resident overlays and then with memory-resident overlays. The disk-resident version of the task is named OVR.TSK and the memory-resident version is named RESOVR.TSK.

> NOTE
>
> This example is intended to provide you with a working illustration of the Overlay Description Language. It does not reflect the most efficient use of it.

Two alterations were made to each of the routines for this example:

- A .TITLE and .END assembler directive was added to each routine to establish it as an unique module.
- The following assembler directive was added to each arithmetic routine to increase its allocation:

  ```
  .BLKW 1024.*3
  ```

  This was done to make the Task Builder allocation of address space more obvious for documentation purposes.

The operation of the overlaid task is identical to that of Example 4 in Chapter 3. The routines and their titles as a result of the .TITLE directives are as follows:

- The integer addition routine is named ADDOV
- The integer subtraction routine is named SUBOV
- The integer multiplication routine is named MULOV
- The integer division routine is named DIVOV
- The register save and restore routine is named SAVOV
- The print routine is named PRNOV
- The main calling routine is named ROOTM

The lengths of the modules are:

Module	Length (in octal)
ADDOV	14024 bytes
SUBOV	14024 bytes
MULOV	14024 bytes
DIVOV	14026 bytes
SAVOV	4042 bytes
PRNOV	4260 bytes
ROOTM	4104 bytes

The flow of control for OVR.TSK is as follows:

ROOTM calls ADDOV and ADDOV returns to ROOTM

ROOTM calls PRNOV to print the result and PRNOV returns to ROOTM

ROOTM calls SUBOV and SUBOV returns to ROOTM

ROOTM calls PRNOV to print the result and PRNOV returns to ROOTM

ROOTM calls DIVOV and DIVOV returns to ROOTM

ROOTM calls PRNOV to print the result and PRNOV returns to ROOTM

ROOTM calls MULOV and MULOV returns to ROOTM

ROOTM calls PRNOV to print the result and PRNOV returns to ROOTM

The print routine (contained in module PRNOV) is called between each arithmetic operation by the control routine (contained in module ROOTM). To avoid loading it into physical memory each time it is called, PRNOV can be placed in the root segment of the task. In addition, each arithmetic routine calls SAVOV. Therefore, SAVOV must be on a path common to all segments in the tree. It too is placed in the root segment of the task. One possible overlay configuration for this task is shown in Figure 4-17.

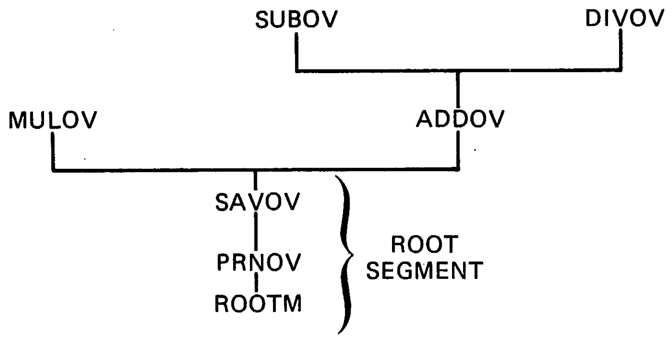


Figure 4-17 Overlay Tree of Virtual Address Space for OVR.TSK

To build this overlay, first create an ODL file (OVERTREE.ODL) that contains its description:

```
.ROOT    ROOTM-PRNOV-SAVOV-*(MULOV,ADDOV-(SUBOV,DIVOV))

.END
```

Then, after you have modified the modules and assembled them, you can build the task with the following command line:

```
TKB> OVR,OVR/-SP=OVRTREE/MP
```

This command instructs the Task Builder to build a task image OVR.TSK and to create a map file, OVR.MAP, under the UFD that corresponds to the terminal UIC. The negated spool switch (/-SP) inhibits the Task Builder from spooling the map file to the line printer.

The overlay switch (/MP) attached to the input file tells the Task Builder that the input file is an ODL file. Therefore, this file will be the only input file specified. Refer to Chapter 6 for a description of the switches used in this example.

A portion of the map that results from this task build is shown in Figure 4-18.

```
PARTITION NAME :  GEN
IDENTIFICATION :  01
TASK  UIC      :  [303,3]
STACK    LIMITS:  000176 001175 001000 00512.          Task
PRG XFR ADDRESS:  010010                               Attributes
TOTAL ADDRESS WINDOWS:  1.  ❷                          Section
TASK  IMAGE  SIZE  :  10496.  WORDS
TASK ADDRESS LIMITS:  000000 050753
R-W DISK BLK LIMITS:  000002 000106 000105 00069.

OVR.TSK;11 OVERLAY DESCRIPTION:

BASE    TOP       LENGTH
----    ---       ------
000000  020677  020700  08640.    ROOTM
020700  034723  014024  06164.      MULOV
020700  034723  014024  06164.      ADDOV
034724  050747  014024  06164.        SUBOV
034724  050753  014030  06168.        DIVOV
                                .
      ❶     ❺      ❹❸          .
                                .
*** ROOT SEGMENT:  ROOTM

R/W MEM  LIMITS: 000000 020677 020700 08640.
DISK BLK LIMITS: 000002 000022 000021 00017.

MEMORY ALLOCATION SYNOPSIS:

SECTION                                      TITLE  IDENT  FILE
-------                                      -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001176 002034 01052.
ANS   :(RW,D,GBL,REL,OVR) 003232 004006 02054.
                          003232 004006 02054. ROOTM  01     ROOTM.OBJ;10
                                .
                                .
                                .

GLOBAL SYMBOLS:

AADD    007276-R  DIVV    007316-R  PRINT  014274-R  SUBB    007306-R
ANS     007232-R  MULL    007266-R  SAVAL  020366-R
                                .
                                .
                                .
*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 7768.
    WORK  FILE  READS:  0.
    WORK  FILE WRITES:  0.
    SIZE OF CORE POOL:  8198.  WORDS (32.  PAGES)
    SIZE OF WORK FILE:  3328.  WORDS (13.  PAGES)

    ELAPSED TIME:00:00:27
```

Figure 4-18 Map File for OVR.TSK

Figure 4-19 shows the allocation of virtual address space for OVR.TSK. The circled numbers in Figure 4-18 correspond to the circled numbers in Figure 4-19.

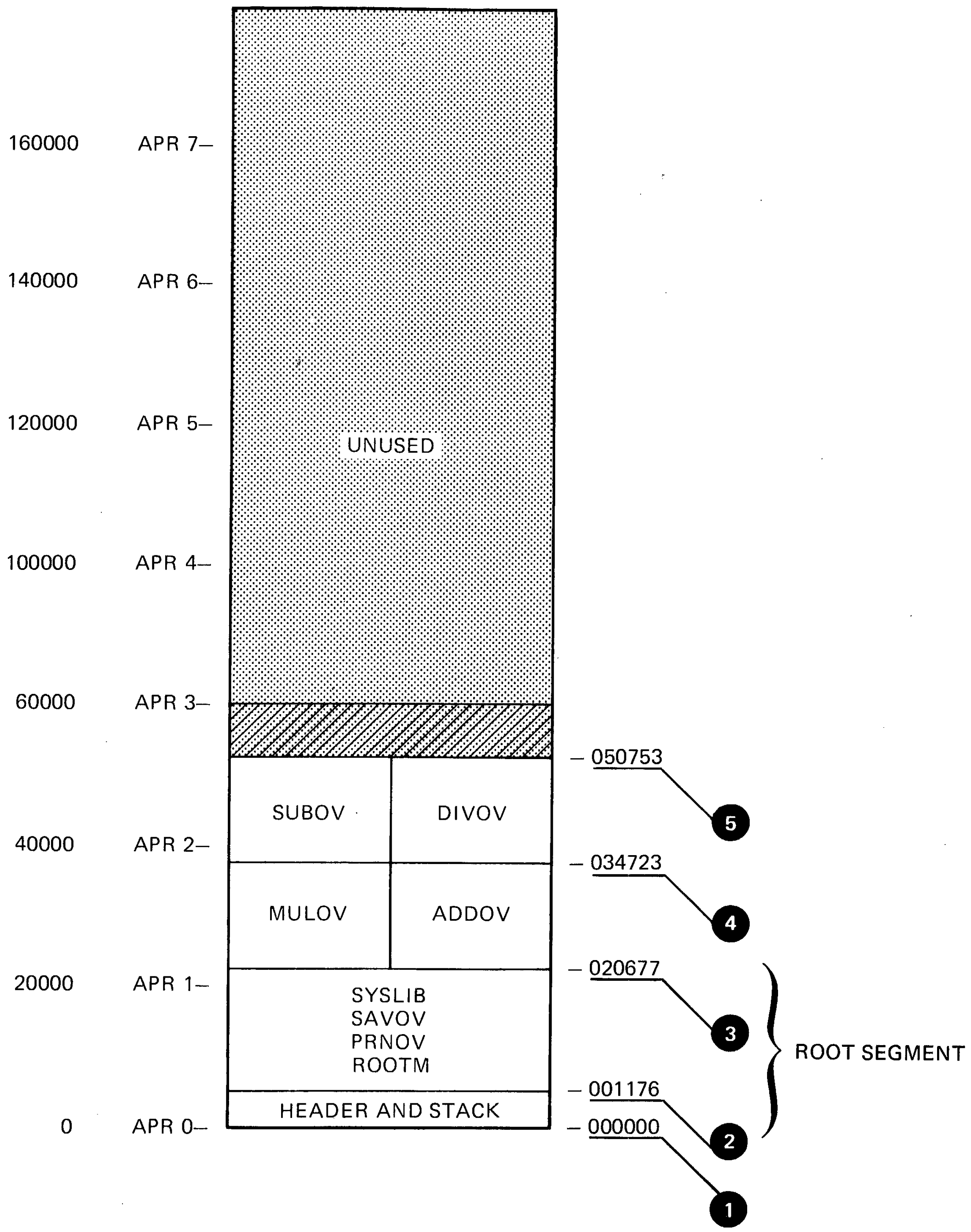


Figure 4-19 Allocation of Virtual Address Space for OVR.TSK

Note that the root segment for OVR.TSK (ROOTM) has expanded with task building while the segments containing the arithmetic routines have not. Before task building, the sum of the modules (in octal bytes) that comprise the root segment is:

4104 + 4260 + 4042 = 14,426 bytes

After task building, the root segment is 20,677(8) bytes long. The Task Builder has added a header, a stack area, and the overlay run-time routines to it. The segments containing the arithmetic routines have not changed. If there had been calls from segments nearer the root to segments up tree, the Task Builder would have added data structures to the calling segments as well. (Refer to Chapter 5 for a description of the overlay loading methods.)

You can build OVR as a memory-resident overlay by simply adding the memory-resident operator (!) to the ODL file for OVR as shown below:

```
.ROOT    ROOTM-PRNOV-SAVOV-*!(MULOV,ADDOV-!(SUBOV,DIVOV))
.END
```

For this example, the name of the ODL file and the task image file have been changed to RESOVR.ODL to distinguish it from the disk resident version. You can build RESOVR with the following command line:

```
TKB>RESOVR,RESOVR/-SP=RESOVR/MP
```

This command directs the Task Builder to build a task named RESOVR.TSK and to create a map file named RESOVR.MAP. The negated spooling switch (/-SP) inhibits spooling of the map file.

The MP switch on the input file tells the Task Builder that the file is an ODL file and that it will be the only input file for this task build. Refer to Chapter 6 for a description of the switches used in this example.

A portion of the map that results from this task build is shown in Figure 4-20.

Figure 4-21 shows the allocation of virtual address space for RESOVR.TSK. The circled numbers in Figure 4-20 correspond to the numbers in Figure 4-21.

```
PARTITION NAME : GEN
IDENTIFICATION : 01
TASK  UIC      : [303,3]
STACK    LIMITS: 000236 001235 001000 00512.          Task
PRG XFR ADDRESS: 010226                                Attributes
TOTAL ADDRESS WINDOWS:  3. ❷                           Section
TASK  IMAGE  SIZE  : 16896.  WORDS
TASK ADDRESS LIMITS: 000000 077777
R-W DISK BLK LIMITS: 000002 000107 000106 00070.

RESOVR.TSK;2 OVERLAY DESCRIPTION:

BASE ❶  TOP    ❸  LENGTH
----    ---       ------
000000  021377  021400  08960.    ROOTM
040000  054077  014100  06208.      MULOV
040000  054077  014100  06208.      ADDOV
060000  074077  014100  06208.        SUBOV
060000  074077  014100  06208.        DIVOV
                  .
     ❻❹     ❼❺    .
                  .
*** ROOT SEGMENT: ROOTM

R/W MEM  LIMITS: 000000 021377 021400 08960.
DISK BLK LIMITS: 000002 000023 000022 00018.

MEMORY ALLOCATION SYNOPSIS:

SECTION                                        TITLE  IDENT  FILE
-------                                        -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001236 002034 01052.
ANS   :(RW,D,GBL,REL,OVR) 003272 004006 02054.
                          003272 004006 02054. ROOTM  01     ROOTM.OBJ;10
                          .
                          .
                          .

GLOBAL SYMBOLS:

AADD   007336-R  DIVV   007356-R  PRINT  014512-R  SUBB   007346-R
ANS    007272-R  MULL   007326-R  SAVAL  020604-R
                          .
                          .
                          .

*** TASK BUILDER STATISTICS:

     TOTAL WORK FILE REFERENCES:  7840.
     WORK FILE READS  :  0.
     WORK FILE WRITES :  0.
     SIZE OF CORE POOL:  8198.  WORDS (32.  PAGES)
     SIZE OF WORK FILE:  3328.  WORDS (13.  PAGES)

     ELAPSED TIME:00:00:24
```

Figure 4-20 Map File for RESOVR.TSK

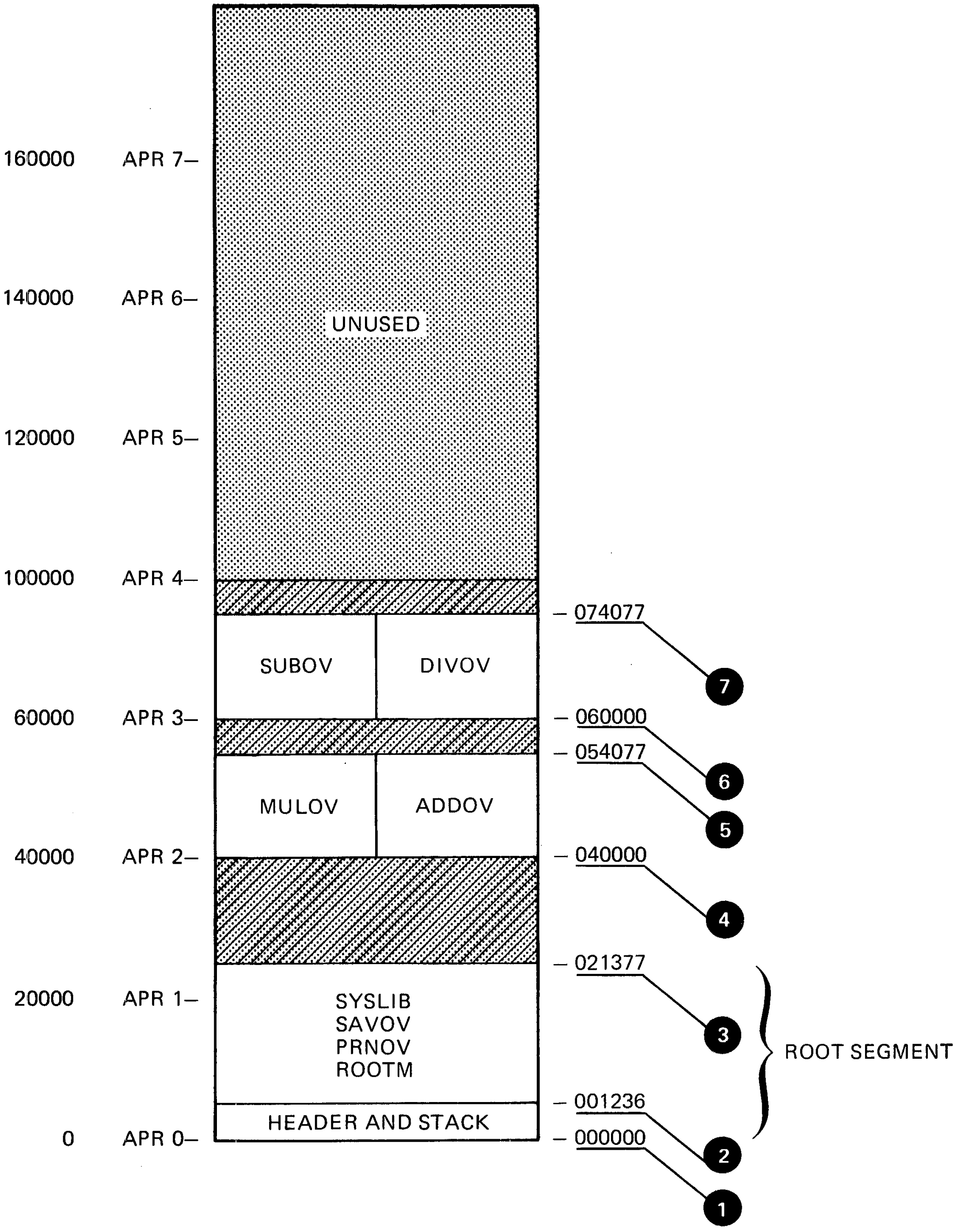


Figure 4-21 Allocation of Virtual Address Space for RESOVR.TSK

Note that the Task Builder allocates virtual address space for each level of overlay segment on a 4K word boundary. When built as a disk-resident overlay, this structure requires 12K words of virtual address space; when built as a memory-resident overlay structure, it requires 16K words of virtual address space. As noted earlier, you must be careful when using memory-resident overlays to ensure that virtual address space is used efficiently.

Finally, note in Figure 4-20 that the Task Builder has allocated three window blocks to map RESOVR.TSK. Each level of the overlay in a memory-resident overlay requires a separate window block to map it. In a disk-resident overlay, a single window block maps the entire structure regardless of how many segment levels there are within the structure. This consideration can be important when you are building an overlaid task that either creates dynamic regions or that accesses a resident library or common because of the extra window blocks required to use these features.

4.8 SUMMARY OF THE OVERLAY DESCRIPTION LANGUAGE

1. An overlay structure consists of one or more trees. Each tree contains at least one segment. A segment is one or more modules containing one or more program sections that can be loaded by a single disk access.

   A tree can have only one root segment, but it can have any number of overlay segments.

2. An ODL file is a text file consisting of a series of overlay description directives, one directive per line. You enter this file in the Task Builder command line, and identify it as an ODL file by attaching the MP switch to the file name. If you enter an ODL file in the Task Builder command line, it must be the only input file you specify.

3. The overlay description language provides five directives for specifying the tree representation of the overlay structure:

   a. .ROOT and .END -- There can be only one .ROOT and one .END directive; the .END directive must be the last directive because it terminates input.
   b. .PSECT
   c. .FCTR
   d. .NAME

   .PSECT, .FCTR, and .NAME can be used in any order in the ODL file.

4. You define the tree structure using the hyphen (-), comma (,), and exclamation point (!) operators, and by using parentheses.

   a. The hyphen operator (-) indicates that its arguments are to be concatenated and thus are to coexist in memory.

   b. The comma operator (,) within parentheses indicates that its arguments are to overlay each other either physically, if disk resident, or virtually, if memory resident.

   c. The comma operator not within parentheses delimits overlay trees.

d. The parentheses group segments that begin at the same point in memory. For example:

```
.ROOT A-B-(C,D-(E,F))
```

This ODL command line defines an overlay structure with a root segment consisting of the modules A and B. In this structure, there are four overlay segments: C, D, E, and F. The outer pair of parentheses indicates that the overlay segments C and D start at the same virtual address; and similarly, the inner parentheses indicate that E and F start at the same virtual address.

e. The exclamation point operator (!) immediately before a left parenthesis declares the enclosed segments to be memory resident. Nested segments in parentheses are not affected by an exclamation point operator at a level closer to the root.

5. The .ROOT directive defines the beginning overlay structure. The arguments of the .ROOT directive are one or more of the following:

   a. File specifications as described in Chapter 1

   b. Factor labels

   c. Segment names

   d. Program-section names

6. The .END directive terminates input

7. The .FCTR directive provides a means for replacing text by a symbolic reference (the factor label). This replacement is useful for two reasons:

   a. The .FCTR directive extends the text of the .ROOT directive to more than one line and thus allows complex trees to be represented.

   b. The .FCTR directive allows the overlay description to be written in a form that makes the structure of the tree more apparent.

   For example:

```
        .ROOT A-(B-(C,D),E-(F,G),H)
        .END
```

   Using the .FCTR directive this overlay description can be written as follows:

```
        .ROOT A-(F1,F2,H)
F1:     .FCTR B-(C,D)
F2:     .FCTR E-(F,G)
        .END
```

   The second representation makes it clear that the tree has three main branches.

8. The .PSECT directive provides a means for directly specifying the segment in which a program section is placed. It accepts the name of the program section and its attributes. For example:

   ```
   .PSECT ALPHA,CON,GBL,RW,I,REL
   ```

   ALPHA is the program section name and the remaining arguments are the program section's attributes (program section attributes are described in Chapter 2).

   The program section name (composed of the characters A-Z, 0-9, and $) must appear first in the .PSECT directive, but the attributes can appear in any order, or can be omitted. If an attribute is omitted, a default condition is assumed. The defaults for program section attributes are RW, I, LCL, REL, and CON.

   In the example above, therefore, you need only specify the attributes that do not correspond to the defaults: .PSECT ALPHA,GBL

9. The .NAME directive provides you with the means to designate a segment name for use in the overlay description, and to specify segment attributes. This directive is useful for creating a null segment, naming a segment that is to be loaded manually, or naming a nonexecutable segment that is to be autoloadable. (Refer to Chapter 5 of this manual for a description of manually loaded and automatically loaded segments.) If you do not use the .NAME directive, the Task Builder uses the name of the first file, program section, or library module in the segment to identify the segment.

   The .NAME directive creates a segment name as follows:

   ```
   .NAME segname,attr,attr
   ```

   where segname is the designated name (composed of the characters A-Z, 0-9, and $), and attr is an optional attribute taken from the following: GBL, NODSK, NOGBL, DSK. The defaults are NOGBL and DSK. The defined name must be unique with respect to the names of program sections, segments, files, and factor labels.

10. You can define a co-tree by specifying an additional tree structure in the .ROOT directive. The first overlay tree description in the .ROOT directive is the main tree. Subsequent overlay descriptions are co-trees. For example:

    ```
    .ROOT A-B-(C,D-(E,F)),X-(Y,Z),Q-(R,S,T)
    ```

    The main tree in this example has the root segment consisting of files A.OBJ and B.OBJ. Two co-trees are defined; the first co-tree has the root segment X and the second co-tree has the root segment Q.

CHAPTER 5

OVERLAY LOADING METHODS

The RSX-11M/M-PLUS systems provide two methods for loading disk-resident and memory-resident overlays:

- Autoload -- the Overlay Run-time routines are automatically called to load segments you have specified
- Manual Load -- you include in the task explicit calls to the Overlay Run-Time routines.

When you build an overlaid task, you must decide which one of these methods to use, because both cannot be used in the same task.

The loading process depends on the kind of overlay:

- Disk resident -- a segment is loaded from disk into a shared area of physical memory, writing over whatever was present.
- Memory resident -- a segment is "loaded" by mapping a set of shared virtual addresses to a unique unshared area of physical memory, where the segment has been made permanently resident (after having been initially brought in from the disk).

With the autoload method, the Overlay Run-Time routines handle loading and error recovery. Overlays are automatically loaded by being referenced through a transfer-of-control instruction (CALL, JMP, or JSR). No explicit calls to the Overlay Run-Time routines are needed.

In the manual load method, you handle loading and error recovery explicitly. Manual loading saves space and gives you full control over the loading process, including the ability to specify whether loading is to be done synchronously or asynchronously.

In the manual load method, you must provide for loading the overlay segments of both the main tree and the root segments, as well as the overlay segments, of the co-trees. Once loaded, the root segment of a co-tree remains in memory.

5.1 AUTOLOAD

To specify the autoload method, you use the autoload indicator, an asterisk (*). You place this indicator in the ODL description of the task at the points where loading must occur. The execution of a transfer-of-control instruction to an autoloadable segment up-tree (farther away from the root) automatically initiates the autoload process.

5.1.1 Autoload Indicator

The autoload indicator (*) marks as autoloadable the segment or other task element (as defined below). If you apply the autoload indicator to an ODL statement enclosed in parentheses, every task element within the parentheses is marked as autoloadable. Placing the autoload indicator at the outermost level of parentheses in the ODL description marks every module in the overlay segments as autoloadable.

If, in the TK1 example of Chapter 4, Section 4.1.1, segment C consisted of a set of modules C1, C2, C3, C4, and C5, the tree diagram would be as shown in Figure 5-1.

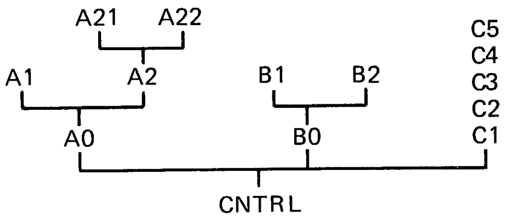


Figure 5-1 Details of Segment C of TK1

Placing the autoload indicator at the outermost level of parentheses ensures that, regardless of the flow of control within the task, a module will be properly loaded when it is called. The ODL description for task TK1 would be:

```
        .ROOT CNTRL-*(AFCTR,BFCTR,CFCTR)
AFCTR:  .FCTR A0-(A1,A2-(A21,A22))
BFCTR:  .FCTR B0-(B1,B2)
CFCTR:  .FCTR C1-C2-C3-C4-C5
        .END
```

Also, when the root segment of a co-tree is not a null segment, you must mark the co-tree's root segment (CNTRL2) as well as its outermost level of parentheses to ensure that all modules of the co-tree are properly loaded. For example, if the co-tree root (CNTRL2) of the multiple tree example, Section 4.5.2, had contained code or data it would have been marked as follows:

```
.ROOT CNTRL-*(AFCTR,BFTCR,CFCTR),*CNTRL2-*(CNTRLX,CNTRLY)
  .
  .
  .
```

You can apply the autoload indicator to the following elements:

- File names - to make all the components of the file autoloadable.
- Portions of ODL tree descriptions enclosed in parentheses - to make all the elements within the parentheses autoloadable, including elements within any nested parentheses.
- Program section names - to make the program section autoloadable. The program section must have the instruction (I) attribute.

- Segment names defined by the .NAME directive - to make all components of the segment autoloadable.

- .FCTR label names - to make the first component of the factor autoloadable. All elements specified in the .FCTR statement are autoloadable if they are enclosed in parentheses.

In the following example, two .PSECT directives and a .NAME directive are introduced into the ODL description for TK1. Autoload indicators are applied as follows:

```
                .ROOT CNTRL-(*AFCTR,*BFCTR,*CFCTR)
     AFCTR:     .FCTR A0-*ASUB1-ASUB2-*(A1,A2-(A21,A22))
     BFCTR:     .FCTR (B0-(B1,B2))
     CFCTR:     .FCTR CNAM-C1-C2-C3-C4-C5
                .NAME CNAM,GBL
                .PSECT ASUB1,I,GBL,OVR
                .PSECT ASUB2,I,GBL,OVR
                .END
```

The following notes are keyed to the example above.

1. The autoload indicator is applied to each factor name; therefore:

   - *AFCTR=*A0

   - *BFCTR=*(B0-(B1-B2))

   - *CFCTR=*CNAM

   CNAM, however, is an element defined by a .NAME directive. Therefore, all components of the segment to which the name applies are made autoloadable, that is, C1, C2, C3, C4, and C5.

2. The autoload indicator is applied to the name of a program section with the instruction (I) attribute (*ASUB1), so program section ASUB1 is made autoloadable.

3. The autoload indicator is applied to a portion of the ODL description enclosed in parentheses:

   *(A1,A2-(A21,A22))

   Thus, every element within the parentheses is made autoloadable (that is, files A1, A2, A21, and A22).

The net effect of this ODL description is to make every element except program section ASUB2 autoloadable.

5.1.2 **Path Loading**

The autoload method uses path loading; that is, a call from one segment to another segment up-tree (farther away from the root) ensures that all the segments on the path from the calling segment to the called segment will reside in physical memory and be mapped. Path loading is confined to the tree in which the called segment resides. A call from a segment in one tree to a segment in another tree results in the loading of all segments on the path in the second tree from the root to the called module.

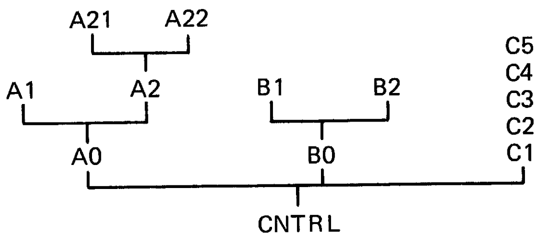


Figure 5-2 Path-Loading Example

In Figure 5-2, if CNTRL calls A22, all the modules between the CNTRL and A2 are loaded. In this case, modules A0 and A2 are loaded.

With the autoload method, the Overlay Run-Time routines keep a record of the segments that are loaded and mapped, and issue disk-load requests only for segments that are not in memory. If CNTRL calls A2 after calling A1, A0 is not loaded again because it is already in memory and mapped.

A reference from one segment to another segment down-tree (closer to the root) is resolved directly. For example, A2 can immediately access A0 because A0 was path loaded in the call to A2.

5.1.3 Autoload Vectors

To resolve a reference up-tree to a global symbol in an autoloadable segment, the Task Builder generates an autoload vector for the referenced global symbol. The reference in the code is changed to a definition that points to an autoload vector entry. The format for the autoload vector is shown in Figure 5-3.

JSR PC,SUB
$AUTO
SEGMENT DESCRIPTOR ADDR.
ENTRY POINT ADDRESS

Figure 5-3 Autoload Vector Format

In the figure, a transfer-of-control instruction to the global symbol executes the call to the autoload routine, $AUTO.

An exception to the procedure for generating autoload vectors is made in the case of a program section with the data (D) attribute. References from a segment to a global symbol up-tree in a program section with the data (D) attribute are resolved directly.

Because the Task Builder can obtain no information about the flow of control within the task, it often generates more autoload vectors than are necessary. However, your knowledge of the flow of control within your task, and knowledge of path loading, can help you determine where to place the autoload indicators. By placing the autoload indicators only at the points where loading is actually required, you can minimize the number of autoload vectors generated for the task.

In the following example, all the calls to overlays originate in the root segment. That is, no module in an overlay segment calls outside its segment. The root segment CNTRL has the following contents:

```
PROGRAM CNTRL
CALL A1
CALL A21
CALL A2
CALL A0
CALL A22
CALL B0
CALL B1
CALL B2
CALL C1
CALL C2
CALL C3
CALL C4
CALL C5
END
```

If you place the autoload indicator at the outermost level of parentheses, 13 autoload vectors are generated for this task; however, because A2 and A0 are loaded by path loading to A21, the autoload vectors for A2 and A0 are unnecessary. Moreover, the call to C1 loads the segment that contains C2, C3, C4, and C5; therefore, autoload vectors for C2 through C5 are unnecessary.

You can eliminate the unnecessary autoload vectors by placing the autoload indicator only at the points where explicit loading is required, as follows:

```
            .ROOT CNTRL-(AFCTR,*BFCTR,CFCTR)
AFCTR:      .FCTR A0-(*A1,A2-*(A21,A22))
BFCTR:      .FCTR (B0-(B1,B2))
CFCTR:      .FCTR *C1-C2-C3-C4-C5
            .END
```

With this ODL description, the Task Builder generates seven autoload vectors -- for A1, A21, A22, B0, B1, B2, and C1.

5.1.4 Autoloadable Data Segments

You can make overlay segments that contain no executable code autoloadable, as follows. First, you must include a .NAME directive and specify the GBL attribute, as described in Section 4.4.4. For example:

```
      .ROOT A-*(B,C)
      .NAME BNAME,GBL
B:    .FCTR BNAME-BFIL
      .END
```

The global symbol BNAME is created and entered into the symbol table of segment BNAME. Since this segment is marked to be autoloaded, root segment A calls segment BNAME as follows:

```
CALL BNAME
```

The segment is autoloaded and an immediate return to inline code occurs.

The data of BFIL must be placed in a program section with the data (D) attribute to suppress the creation of autoload vectors.

5.2 MANUAL LOAD

If you decide to use the manual load method to load segments, you must include in your program explicit calls to the $LOAD routine. These load requests must supply the name of the segment to be loaded. In addition, they can include information necessary to perform asynchronous load requests, and to handle load request failures.

The $LOAD routine does not path load. A call to $LOAD loads only the segment named in the request. The segment is read in from disk and mapped regardless of its previous status.

A MACRO-11 programmer calls the $LOAD routine directly. A FORTRAN programmer calls $LOAD using the FORTRAN subroutine MNLOAD.

5.2.1 MACRO-11 Manual Load Calling Sequence

A MACRO-11 programmer calls $LOAD as follows:

```
        MOV     #PBLK,R0
        CALL    $LOAD
```

PBLK is the address of a parameter block with the following format:

```
PBLK:   .BYTE   length,event-flag
        .RAD50  /seg-name/
        .WORD   [i/o-status]
        .WORD   [ast-trp]
```

length

The length of the parameter block (3 to 5 words).

event-flag

The event flag number, used for asynchronous loading. If the event-flag number is 0, synchronous loading is performed.

seg-name

The name of the segment to be loaded: a 1- to 6-character Radix-50 name, occupying two words.

i/o-status

The address of the I/O status doubleword. Standard QIO status codes apply.

ast-trp

The address of an asynchronous trap service routine to which control is transferred at the completion of the load request.

The condition code C-list is set or cleared on return, as follows:

- If condition code C=0, the load request was accepted.
- If condition code C=1, the load request was unsuccessful.

For a synchronous load request, the return of the condition code C=0 means that the desired segment is loaded and is ready to be executed. For an asynchronous load request, the return of the code C=0 means that the load request was successfully queued to the device driver, but the segment is not necessarily in memory. Your program must ensure that loading has been completed, by waiting for the specified event flag before calling any routines or accessing any data in the segment.

5.2.2 FORTRAN Manual Load Calling Sequence

To use the manual load mechanism in a FORTRAN program, your program must refer to the $LOAD routine by means of the MNLOAD subroutine. The subroutine call has the form:

```
    CALL MNLOAD(seg-name[,event-flag][,i/o-status][,ast-trp][,ld-ind])
```

seg-name

A 2-word real variable containing the segment name in Radix-50 format.

event-flag

An optional integer event flag number used for an asynchronous load request. If the event flag number is 0, the load request is synchronous.

i/o-status

An optional 2-word integer array containing the I/O status doubleword, as described for the QIO directive in the <u>RSX-11M/M-PLUS Executive Reference Manual</u>.

ast-trp

An optional asynchronous trap subroutine entered at the completion of a request. MNLOAD requires that all pending traps specify the same subroutine.

ld-ind

An optional integer variable containing the results of the subroutine call. One of the following values is returned:

+1 Request was successfully executed.

-1 Request had bad parameters or was not successfully executed.

You can omit optional arguments. The following calls are legal:

Call	Effect
CALL MNLOAD (SEGA1)	Load segment named in SEGA1 synchronously.
CALL MNLOAD (SEGA1,0,,,LDIND)	Load segment named in SEGA1 synchronously and return success indicator to LDIND.

Call	Effect
CALL MNLOAD (SEGA1,1,IOSTAT,ASTSUB,LDIND)	Load segment named in SEGA1 asynchronously, transferring control to ASTSUB upon completion of the load request; store the I/O Status doubleword in IOSTAT, and the success indicator in LDIND.

The following example uses the program CNTRL, previously discussed in Section 5.1. In this example, there is sufficient processing between the calls to the overlay segments to make asynchronous loading effective. The autoload indicators are removed from the ODL description and the FORTRAN programs are recompiled with explicit calls to the MNLOAD subroutine, as follows:

```
PROGRAM CNTRL
EXTERNAL ASTSUB
DATA SEGA1 /6RA1    /
DATA SEGA21 /6RA21  /
    .
    .
    .
CALL MNLOAD (SEGA1,1,IOSTAT,ASTSUB,LDIND)
    .
    .
    .
CALL A1
    .
    .
    .
CALL MNLOAD (SEGA21,1,IOSTAT,ASTSUB,LDIND)
    .
    .
    .
CALL A21
    .
    .
    .

END
SUBROUTINE ASTSUB
DIMENSION IOSTAT(2)

    .
    .
    .
END
```

When the AST trap routine is used, the I/O status doubleword is automatically supplied to the dummy variable IOSTAT.

5.3 ERROR HANDLING

If you select the manual load method, you must provide error handling routines that diagnose load errors and provide appropriate recovery.

If you use the autoload mechanism, a simple recovery procedure is provided that checks the Directive Status Word (DSW) for an error indication. If the DSW indicates that no system dynamic storage is available, the routine issues a Wait for Significant Event directive and tries again; if the problem is not dynamic storage, the recovery procedure generates a synchronous breakpoint trap. If the task services the trap and returns without altering the state of the program, the request will be retried.

A more comprehensive user-written error recovery subroutine can be substituted for the system-provided routine if the following conventions are observed:

1. The error recovery routine must have the entry point name $ALERR.

2. The contents of all registers must be saved and restored.

On entry to $ALERR, R2 contains the address of the segment descriptor that could not be loaded. Before recovery action can be taken, the routine must determine the cause of the error by examining the following words in the sequence indicated:

1. $DSW — The Directive Status Word may contain an error status code, indicating that the Executive rejected the I/O request to load the overlay segment.

2. N.OVPT — The contents of this location, offset by N.IOST, point to a 2-word I/O status block containing the results of the load overlay request returned by the device driver. The status code occupies the low-order byte of word 0. For example, for a device not ready condition, the code will be IE.DNR. (For more information on these codes refer to the _IAS/RSX-11 I/O Operations Reference Manual_.)

5.4 GLOBAL CROSS-REFERENCE OF AN OVERLAID TASK

This section illustrates a global cross-reference that has been created for an overlaid task. The task consists of a root segment containing the module ROOT.OBJ, and two overlay segments composed of modules OVR1 and OVR2. The overlay description of the file is as follows:

```
    .ROOT   ROOT-(OVR1,*OVR2)
```

Only segment OVR2 is autoloadable. Figure 5-4 shows the resulting cross-reference listing.

As shown, the global symbol OVR1 is defined in module OVR1, and a single nonautolaodable, up-tree reference is made to this symbol by the module ROOT, as indicated b the circumflex. Note that segment OVR1 cannot be evaded because of the restriction against mixing manual load and autoload in the same task.

```
OVRTST     CREATED BY   TKB     ON 1-OCT-76 AT 12:04       PAGE 1

GLOBAL CROSS REFERENCE                                     CREF     V01

SYMBOL   VALUE       REFERENCES...

N.ALER   000010        AUTO     # OVRES
N.IOST   000004        OVCTL    # OVRES
N.MRKS   000016      # OVRES
N.OVLY   000000        OVCTL    # OVRES
N.OVPT   000054        AUTO       OVCTL    # VCTDF
N.RDSG   000014      # OVRES
N.STBL   000002      # OVRES
N.SZSG   000012      # OVRES
OVR1     002014-R    # OVR1     ^ ROOT
OVR2     002014-R    * OVR2     @ ROOT
ROOT     001176-R    0 ROOT
$ALBP1   001320-R    # AUTO
$ALBP2   001416-R    # AUTO
$ALERR   001246-R    # ALERR      OVDAT
$AUTO    001302-R    # AUTO
$DSW     000046        ALERR    # VCTDF
$MARKS   001546-R    # OVCTL
$OTSV    000052      # VCTDF
$SAVRG   001452-R      AUTO     # SAVRG
$VEXT    000056      # VCTDF
.FSRPT   000050      # VCTDF
.NALER   001442-R    # OVDAT
.NIOST   001436-R    # OVDAT
.NMRKS   001450-R    # OVDAT
.NOVLY   001432-R    # OVDAT
.NOVPT   000042      # OVDAT
.NRDSG   001446-R    # OVDAT
.NSTBL   001434-R    # OVDAT
.NSZSG   001444-R    # OVDAT


OVRTST     CREATED BY   TKB     ON 1-OCT-76 AT 12:04       PAGE 2

SEGMENT CROSS REFERENCE                                    CREF     V01

SEGMENT NAME   RESIDENT MODULES

OVR1           OVR1
OVR2           OVR2
ROOT           ALERR    AUTO    OVCTL    CVDAT    OVRES    ROOT    SAVRG
               VCTDF
```

Figure 5-4 Sample Overlaid Cross-Reference Listing

As shown, the global symbol OVR1 is defined in module OVR1, and a single nonautoloadable, up-tree reference is made to this symbol by the module ROOT, as indicated by the circumflex. Note that segment OVR1 cannot be evaded because of the restriction against mixing manual load and autoload in the same task.

The asterisk preceding the module OVR2 indicates that the global symbol OVR2 is an autoload symbol and is referenced from the module ROOT through an autoload vector, as shown by the at sign (@) character.

Down-tree references to the global symbol ROOT are made from modules OVR1 and OVR2. These references are resolved directly.

The segment cross-reference shows the segment name and modules in each overlay.

CHAPTER 6

SWITCHES AND OPTIONS

You use switches and options to control the construction of your task image. This chapter provides detailed reference information on all the Task Builder switches and options.

6.1 SWITCHES

The syntax for a file specification, as given in Chapter 1, is:

```
dev:[group,member]filename.type;version/swl/sw2.../swn
```

Optionally, you can conclude a file specification with one or more switches (swl,sw2,...swn). When you do not specify a switch, the Task Builder establishes a default setting for it.

You designate a switch by a 2- to 4-character code preceded by a slash (/). If you precede the 2- to 4-character code with a minus sign (-) or the letters NO, the Task Builder negates the function of the two characters. For example, the Task Builder recognizes the following settings for the switch CP (checkpointable):

/CP	The task is checkpointable
/-CP	The task is not checkpointable
/NOCP	The task is not checkpointable

In some cases, two particular switches cannot both be used in a file specification. When sach a conflict occurs, the Task Builder selects the overriding switch according to the following table:

Switch	Switch	Overriding Switch
AC (Ancillary Control Processor)	PR (Privileged)	AC
EA (Extended Arithmetic Element)	FP (Floating Point Processor)	FP
CC (Concatenated object file)	LB (Library file)	LB

For example:

```
MCR>TKB IMG5=IN6,IN5/LB/CC
```

The Task Builder assumes that the input file IN5 is a library file. It searches the file for undefined global references. It does not include in the task image all of the modules in IN5.

The switches that the Task Builder recognizes are given in alphabetical order in Table 6-1. Sections 6.1.1 through 6.1.31 give detailed descriptions of each switch, in alphabetical order, including:

- The switch format
- The file(s) to which the switch can be applied
- A description of the effect of the switch on the Task Builder
- The default assumption made if the switch is not present

Table 6-1
Task Builder Switches

Format	Meaning	Applies to File	Default
AC[:n]	Task is an ancillary control processor	.TSK	-AC
AL	Task can be checkpointed to space allocated in the task image file	.TSK	-AL
CC	Input file consists of concatenated object modules	.OBJ	CC
CM	Memory-resident overlays are aligned on 256-word physical boundaries	.TSK	-CM
CP	Task is checkpointable	.TSK	-CP
CR	A global cross-reference listing is appended to the memory allocation file	.MAP	-CR
DA	Task contains a debugging aid	.TSK, .OBJ	-DA
DL	Specified library file is a replacement for the system object module library	.OLB	-DL
EA	Task uses extended arithmetic element	.TSK	-EA
FP	Task uses the floating-point processor	.TSK	-FP

[1] The default is /MA for an input file, and /-MA for system and resident library .STB files.

[2] The default for the memory management switch is /MM if the host system has memory management hardware and /-MM if the host system does not have memory management hardware.

(continued on next page)

Table 6-1 (Cont.)
Task Builder Switches

Format	Meaning	Applies to File	Default
FU	All cotree overlay segments are searched for matching definition or reference when modules from the default object module library are being processed	.TSK	-FU
HD	Task image includes a header	.TSK, .STB	HD
LB	Input file is a library file	.OLB	-LB
MA	Map file includes information from the file	.MAP, .TSK	MA or -MA[1]
MM	System has memory management	.TSK	MM or -MM[2]
MP	Input file contains an overlay description	.ODL	-MP
MU	Task is a multiuser task	.TSK	-MU
PI	Task is position independent	.TSK, .STB	-PI
PM	Postmortem Dump is requested	.TSK	-PM
PR[:n]	Task has privileged access rights	.TSK	-PR
RO	Memory-resident overlay operator (!) is enabled	.TSK	RO
SE	Messages can be directed to the task by means of the Executive SEND directive	.TSK	SE
SH	Short memory allocation file is requested	.MAP	SH
SL	Task is slaved to an initiating task	.TSK	-SL
SP	Spool map output	.MAP	SP
SQ	Task program sections are allocated sequentially	.TSK	-SQ
SS	Selective Search for global symbols	.OBJ	-SS
TR	Task is to be traced	.TSK	-TR
WI	Memory allocation file is printed at a width of 132 characters	.MAP	WI
XT[:n]	Task Builder exits after n diagnostics	.TSK	-XT

This page left blank intentionally

6.1.1 /AC[:n] -- Ancillary Control Processor

File

Task image

Syntax

```
file.TSK/AC:0=file.OBJ
     or
file.TSK/AC:4=file.OBJ
     or
file.TSK/AC:5=file.OBJ
```

Description

Your task is an ancillary control processor; that is, it is a privileged task that extends certain Executive functions. For example, the system task F11ACP is an ancillary control processor that receives and processes FILES-11 related input and output requests on behalf of the Executive.

Effect

Your task is privileged. The Task Builder sets the AC attribute flag and the privileged attribute flag in your task's label block flag word.

The value of n is an octal number that specifies the first KT-11 Active Page Register that you want the Executive to use to map your task's image when your task is running in user mode. Legal values are 0, 4, and 5. If you do not specify n, the Task Builder assumes a value of 5.

If you do not explicitly specify that your task is to run on a mapped system (through the MM switch) and it is not otherwise implied (the Task Builder is not running in a system with KT-11 hardware), the Task Builder merely tests the value of the switch for validity, but otherwise ignores it.

Default

/-AC

NOTE

You should not use /AC and /PR on the same command line.

AL

6.1.2 /AL -- Allocate Checkpoint Space

File

Task image

Syntax

```
file.TSK/AL=file.OBJ
```

Description

Your task is checkpointable. The system will checkpoint it to space in your task's image file.

Effect

Your task is checkpointable. This switch directs the Task Builder to allocate additional space in your task image file to contain the checkpointed task image.

Default

/-AL

NOTE

It does not make sense to use /CP and /AL in the same command line.

6.1.3 /CC -- Concatenated Object Modules

File

Input

Syntax

```
file.TSK=file.OBJ/-CC
```

Description

This switch controls the way the Task Builder extracts modules from your input file.

Effect

By default, the Task Builder includes in your task image all the modules of your input file. If you negate this switch (as in the "Syntax" section above), the Task Builder includes only the first module of your input file.

Default

/CC

CM

6.1.4 /CM -- Compatibility Mode Overlay Structure

File

Task image

Syntax

```
file.TSK/CM=file.OBJ
```

Description

Your task will be built in compatibility mode.

Effect

The Task Builder aligns memory-resident overlay segments on 256-word boundaries for compatibility with other implementations of the mapping directives.

Default

/-CM

6.1.5 /CP -- Checkpointable

File

Task image

Syntax

```
file.TSK/CP=file.OBJ
```

Description

Your task is checkpointable. The system will checkpoint it to space that you have allocated in the system checkpoint file on the system disk. This switch assumes that you have allocated the checkpoint space through the MCR command ACS. (Refer to the RSX-11M/M-PLUS MCR Operations Manual.)

Effect

The system writes your task to the system checkpoint file on secondary storage when its physical memory is required by a task of higher priority.

Default

/-CP

NOTE

It does not make sense to use /CP and /AL in the same command line.

6.1.6 /CR -- Cross-Reference

File

Memory allocation (map)

Syntax

```
file.TSK,file.MAP/CR=file.OBJ
```

Description

This switch determines whether or not a cross reference listing is added to your map file.

Effect

The Task Builder creates a special work file (file.CRF) which contains segment, module, and global symbol information. The Task Builder then calls the cross reference processor (CRF) to process the file. CRF creates a cross reference listing from the information contained in the file, and then deletes file.CRF. (Refer to Appendix D, RSX-11 Utilities Manual for more information on CRF.

The cross-reference listing and its contents are described in the "Example" section below.

NOTE

In order for this switch to be effective, CRF must be installed in your system.

Default

/-CR

Example

Figure 6-1 shows a cross reference listing for task OVR. The numbers in the following text correspond to the circled numbers in the listing.

1. The cross-reference page header gives the name of the memory allocation file, the originating task (TKB), the date and time the memory allocation file was created, and the cross-reference page number.

2. The cross-reference list contains an alphabetic listing of each global symbol along with its value and the name of each referencing module. When a symbol is defined in several segments within an overlay structure, the last defined value is printed. Similarly, if a module is loaded in several segments within the structure, the module name will be displayed more than once within each entry.

The suffix -R is appended to the value if the symbol is relocatable.

Prefix symbols accompanying each module name define the type of reference as follows:

Prefix Symbol	Reference Type
blank	Module contains a reference that is resolved in the same segment or in a segment toward the root.
^	Module contains a reference that is resolved directly in a segment away from the root or in a co-tree.
@	Module contains a reference that is resolved through an autoload vector.
#	Module contains a non-autoloadable definition.
*	Module contains an autoloadable definition.

3 The segment cross-reference lists the name of each overlay segment and the modules that compose it. If the task is a single segment task, this section does not appear.

```
OVR        CREATED BY  TKB    ON 11-APR-79 AT 09:27     PAGE 1    1
GLOBAL CROSS REFERENCE                                  CREF  V01

SYMBOL  VALUE       REFERENCES...

AADD    034700-R  * ADDOV   @ ROOTM
ANS     007232-R    PRNOV   # ROOTM
DIVV    050724-R  * DIVOV   @ ROOTM
IO.WVB  011000      PRNOV                                              2
MULL    034700-R  * MULOV   @ ROOTM
PRINT   014274-R  # PRNOV     ROOTM
SAVAL   020366-R    ADDOV     DIVOV     MULOV    # SAVOV     SUBOV
SUBB    050724-R  @ ROOTM   * SUBOV
$EDMSG  001272      PRNOV

OVR        CREATED BY  TKB    ON 11-APR-79 AT 09:27     PAGE 2

SEGMENT CROSS REFERENCE                                 CREF  V01

SEGMENT NAME    RESIDENT MODULES

ADDOV           ADDOV                                                  3
DIVOV           DIVOV
MULOV           MULOV
ROOTM           PRNOV    ROOTM    SAVOV
SUBOV           SUBOV
```

Figure 6-1 Cross Reference Listing for OVR.TSK

DA

6.1.7 /DA -- Debugging Aid

File

Task image or input

Syntax

```
file.TSK/DA=file.OBJ
     or
file.TSK=file.OBJ,file.OBJ/DA
```

Description

Your task includes a debugging aid that will control its execution.

Effect

If you apply this switch to your task image file, the Task Builder automatically includes the system debugging aid LB0:[1,1]ODT.OBJ into your task image.

The Task Builder causes control to be passed to the debugging program when task execution is initiated.

If you apply this switch to one of your input files, the Task Builder assumes that the file is a debugging aid that you have written. Such debugging programs can trace a task, printing out relevant debugging information, or monitor the task's performance for analysis.

In either case, /DA has the following effects on your task image:

- The transfer address of the debugging aid overrides the task transfer address.
- The Task Builder initializes the header of your task so that, on initial task load, registers R0 through R4 contain the following values:

  R0 - Transfer address of task

  R1 - Task name in Radix-50 format (word #1)

  R2 - Task name (word #2)

  R3 - The first three of six RAD50 characters representing the version number of your task. The Task Builder derives this number from the first .IDENT directive it encounters in your task. If no .IDENT directive is in your task, this value will be 0.

  R4 - The second three RAD50 characters representing the version number of your task.

Default

/-DA

6.1.8 /DL -- Default Library

File

Input

Syntax

```
file.TSK=file.OBJ,file.OLB/DL
```

Description

Your input file is a replacement for the system object module library.

Effect

The library file you have specified replaces the file LB0:[1,1]SYSLIB.OLB as the library file that the Task Builder searches to resolve undefined global references. The default device for the replacement file is SY0:. The Task Builder refers to it only when undefined symbols remain after it has processed all the files you have specified. You can apply the DL switch to only one input file.

Default

/-DL

EA

6.1.9 /EA -- Extended Arithmetic Element

File

Task image

Syntax

```
file.TSK/EA=file.OBJ
```

Description

Your task uses the KE11-A Extended Arithmetic Element.

Effect

The Task Builder allocates three words in your task's header for saving the state of the extended arithmetic element.

Default

/-EA

NOTE

You should not use /EA and /FP on the same command line.

6.1.10 /FP -- Floating Point

File

Task image

Syntax

```
file.TSK/FP=file.OBJ
```

Description

Your task uses the floating-point processor.

Effect

The Task Builder allocates 25 words in your task's header for saving the state of the floating-point processor.

Default

/-FP on RSX-11M systems
/FP on RSX-11M-PLUS systems

Notes

1. You should not use /FP and /EA on the same command line.

2. In an RSX-11M system, the FP switch will be effective only if the Executive supports the Floating Point Processor.

3. In an RSX-11M system if a task that uses the Floating Point Processor is built without the FP switch, the task will run correctly until a second task that uses the Floating Point Processor is run. Then both tasks will either crash or produce incorrect results. For information on changing the Task Builder's defaults, refer to Appendix E.

FU

6.1.11 /FU -- Full Search

File

Task image

Syntax

```
file.TSK/FU=file.ODL/MP
```

Description

This switch controls the Task Builder's search for undefined symbols when it is processing modules from the default library.

Effect

When the Task Builder processes modules from the default object module library, and it encounters undefined symbols within those modules, it normally limits its search for definitions to the root of the main tree and to the current tree. Thus, unintended global references between co-tree overlay segments are eliminated. When the FU switch is appended to the task image file of an overlaid task, the Task Builder searches all co-tree segments for a matching definition or reference.

Default

/-FU

6.1.12 /HD -- Header

File

Task image or symbol definition

Syntax

```
file.TSK/-HD,,file.STB=file.OBJ
     or
file.TSK,,file.STB/-HD=file.OBJ
```

Description

When negated this switch directs the Task Builder to exclude a header from your task image.

Effect

The Task Builder does not construct a header in your task image. You use the negated form of this switch when you are building commons, resident libraries, and loadable drivers.

Default

/HD

LB

6.1.13 /LB -- Library File

File

Input

Syntax

```
file.TSK=file.OBJ,file.OLB/LB
     or
file.TSK=file.OBJ,file.OLB/LB:mod-1:mod-2...:mod-8
     or
file.OBJ=file.OLB/LB:mod-1:mod-2,file.OLB/LB
```

Description

The file to which this switch is attached is an object module library file. The Task Builder's interpretation of this switch depends upon the form you use. There are three forms:

1. Without arguments (the first syntax given above)

2. With arguments (the second syntax given above)

3. Both with and without arguments (the third syntax given above)

Effect

If you apply this switch without arguments, the Task Builder assumes that your input file is a library file of relocatable object modules. The Task Builder searches the file immediately to resolve undefined references in any modules preceding the library specification and extracts from the library, for inclusion in the task image, any modules that contain definitions for such references.

If you apply the switch with arguments, the Task Builder extracts from the library the modules named as arguments of the switch regardless of whether or not the modules contain definitions for unresolved references.

If you want the Task Builder to search an object module library file both to resolve global references and to select named modules for inclusion in your task image, you must name the library file twice: once, with the modules you want included in your task image listed as arguments of the LB switch; and a second time, with the LB switch and no arguments.

The position of the library file within the Task Builder command sequence is important. The following rules apply:

1. The library file must follow to the right of the input file(s) that contain references to be defined in the library. For example:

   ```
   TKB>file.TSK=infile1.OBJ,lib.OLB/LB
   ```

The command above illustrates the correct usage of the LB switch; the following command illustrates **incorrect** usage:

```
TKB>file.TSK=lib.OLB/LB,file1.OBJ
```

2. If you are using the Task Builder's multiple line input, and you specify a given library more than once during the command sequence, you must attach the LB switch to the library file each time you specify the library. For example:

```
>TKB
TKB>file.TSK=file1.OBJ,file2.OBJ,lib.OLB/LB
TKB>file3.OBJ,file4.OBJ,lib.OLB/LB
//
```

3. When you are building an overlay structure, the Task Builder limits the number of input files you can specify to 1. Therefore, you must specify object module libraries for an overlay structure within the Overlay Description Language (ODL) file for the structure. To do this, you must use the .FCTR directive to specify the library. For example:

```
          .ROOT CNTRL-LIB(AFCTR,BFCTR,C)
AFCTR:    .FCTR A0-LIB(A1,A2-(A21,A22))
BFCTR:    .FCTR B0-LIB(B1,B2)
LIB:      .FCTR LB:[303,3]LIBOBJ.OLB/LB
          .END
```

The technique used in the ODL file above allows you to control the placement of object module library routines into the segments of your overlay structure. (For more information on overlaid tasks, see Chapter 4.)

NOTES

1. You should not use the LB switch and the CC switch in the same command sequence.

2. You can use the SS switch in conjunction with the LB switch (with or without arguments) to perform a selective search for global definitions.

Default

/-LB

MA

6.1.14 /MA -- Map Contents of File

File

Input or memory allocation

Syntax

```
file.TSK,file.MAP=file.OBJ,file.OBJ/-MA
     or
file.TSK,file.MAP/MA=file.OBJ
```

Description

The Task Builder is to include information from your input file in the memory allocation output file.

Effect

If you negate this switch and apply it to an input file, the Task Builder will exclude from the map and cross-reference listings all global symbols defined or referred to in the file. In addition, the Task Builder will not list the file in the "file contents" section of the map.

If you apply this switch to the map file, the Task Builder will include in the map file the names of routines it has added to your task from SYSLIB. It will also include in the map file information contained in the symbol definition file of any shared region referred to by the task.

Default

/MA for input files.

/-MA for system library and resident library STB files.

6.1.15 /MM -- Memory Management

File

Task image

Syntax

```
file.TSK/MM=file.OBJ
     or
file.TSK/-MM=file.OBJ
```

Description

The system on which your task is to run has memory management hardware.

Effect

The Task Builder can build a task image for a mapped or unmapped system independently of the mapping status of the system on which your task is being built. If you specify /-MM, the Task Builder assumes an unmapped system.

Default

/MM or /-MM. When you do not apply /MM to your task image file, the Task Builder allocates memory according to the mapping status of the system on which your task is being built.

NOTE

When you negate this switch (/-MM), it suppresses the Task Builder's recognition of the memory-resident overlay operator (!). The Task Builder checks the operator for correct syntax but it does not create any resident overlay segments.

MP

6.1.16 /MP -- Overlay Description

File

Input

Syntax

```
file.TSK=file.ODL/MP
```

Description

Your input file is an Overlay Description Language (ODL) file.

Effect

The Task Builder receives all the input file specifications from this file. It allocates virtual address space as directed by the overlay description. If you use the Task Builder's multiline command format (see section 1.3), the Task Builder automatically requests option information at the console terminal by displaying

```
ENTER OPTIONS:.
```

NOTES

1. If you use the multiline command format when you specify an ODL file, the Task Builder automatically prompts for option input. Therefore, you must not use the single slash (/) to direct the Task Builder to switch to option input mode when you have specified /MP on your input file.

2. When you specify /MP on the input file for your task, it must be the only input file that you specify. Furthermore, the input file must have a file type of .ODL.

Default

/-MP

6.1.17 /MU -- Multiuser

File

Input

Syntax

```
file.TSK/MU=file.OBJ
```

Description

Your task is a multiuser task.

Effect

The Task Builder separates your task's read-only and read/write program sections. It then places the read-only program sections in your task's upper virtual address space and the read/write program sections in your task's lower virtual address space.

Default

/-MU

PI

6.1.18 /PI -- Position Independent

File

Task image or symbol definition

Syntax

```
file.TSK/PI=file.OBJ
     or
file.TSK,,file.STB/PI=file.OBJ
```

Description

Your shared region contains only position-independent code or data.

Effect

The Task Builder sets the Position-Independent Code (PIC) attribute flag in the label block flag word of your shared region.

Default

/-PI

6.1.19 /PM -- Postmortem Dump

File

Task image

Syntax

```
file.TSK/PM=file.OBJ
```

Description

If your task terminates abnormally, the system automatically lists the contents of the memory image.,

Effect

The Task Builder sets the Postmortem Dump flag in your task's label flag word.

Notes

1. If your task issues an ABRT$ (abort task) directive, the system will not dump the task image even though the Task Builder has set the Postmortem Dump flag in your task's label flag word. In this case, the system assumes that a Postmortem Dump is not necessary since you know why your task was aborted.

2. The PMD utility must be installed in your system and be able to get into physical memory for this switch to be effective.

Default

/-PM

PR

6.1.20 /PR[:n] -- Privileged

File

Task image

Syntax

```
file.TSK/PR:0=file.OBJ
     or
file.TSK/PR:4=file.OBJ
     or
file.TSK/PR:5=file.OBJ
```

Description

Your task is privileged with respect to memory and device access rights. If you specify PR:0, your task does not have access to the I/O page or the Executive. However, if you specify PR:4 or PR:5, your task does have access to the I/O page and the Executive, in addition to its own partition.

Effect

The Task Builder sets the Privileged Attribute flag in your task's label block flag word.

The value of n is an octal number that specifies the first Active Page register that you want the Executive to use to map your task image when your task is running in user mode. Legal values are 0, 4, and 5. If you do not specify one of these values, the Task Builder assumes a value of 5.

If you do not explicitly specify that your task is to run on a mapped system, (through the MM switch) and it is not otherwise implied (by the presence of KT-11 hardware on the system upon which the Task Builder is running), the Task Builder merely tests the value of the switch for validity, but otherwise ignores it. Privileged tasks are described in Chapter 2.

Default

/-PR

NOTE

You should not use /PR and /AC on the same command line.

6.1.21 /RO -- Resident Overlay

File

Task image

Syntax

```
file.TSK/-RO=file.ODL/MP
```

Description

The Task Builder's recognition of the memory-resident overlay operator (!) is enabled.

Effect

The memory-resident overlay operator (!), when present in the overlay description file, indicates to the Task Builder that it is to construct a task image that contains one or more memory-resident overlay segments. If you negate this switch (as in the "Syntax" section above), the Task Builder checks the operator for correct syntactical usage, but otherwise ignores it. With the memory-resident overlay operator thus disabled, the Task Builder builds a disk-resident overlay from the overlay description file.

Default

/RO

6.1.22 /SE -- Send

File

Task image

Syntax

```
file.TSK/-SE=file.OBJ
```

Description

This switch determines whether or not messages can be directed to your task by means of the Executive Send directive. (Refer to the RSX-11M/M-PLUS Executive Reference Manual for information on the Send directive)

Effect

By default, messages can be directed to your task by means of the Executive Send directive. If you negate this switch (as in the "Syntax" section above), the system inhibits the queuing of messages to your task.

Default

/SE

6.1.23 /SH -- Short Map

File

Memory allocation (map)

Syntax

```
file.TSK,file.MAP/-SH=file.OBJ
```

Description

The Task Builder produces the short version of the memory allocation file.

Effect

The Task Builder does not produce the "file contents" section of the memory allocation file.

Default

/SH

Example

The memory allocation file consists of the following items:

1. Page Header
2. Task Attributes Section
3. Overlay Description (if applicable)
4. Root Segment Allocation
5. Tree Segment Description (if applicable)
6. Undefined References (if applicable)
7. Task Builder Statistics

An example of the memory allocation file (map) is shown in Figure 6-2. The numbered and lettered items in the notes following the figure correspond to the numbers and letters in Figure 6-2.

```
OVR.TSK;25    MEMORY ALLOCATION MAP  TKB M36          PAGE 1  ]❶ PAGE HEADER
                    13-APR-79   09:10


TASK    NAME      :ⓐ                                              ┐
PARTITION NAME  : GENⓑ                                             │
IDENTIFICATION  : 01ⓒ                                              │
TASK  UIC       : [303,3]ⓓ                                         │
TASK PRIORITY   :ⓔ                                                 │
STACK    LIMITS: 000176 001175 001000 00512.ⓕ                      │
ODT XFR ADDRESS:ⓖ                                                  │
PRG XFR ADDRESS: 010010ⓗ                                           │
TASK ATTRIBUTES:ⓘ                                                  │ ❷ TASK ATTRIBUTES
TOTAL ADDRESS WINDOWS: 1.ⓙ                                         │      SECTION
MAPPED    ARRAY       :ⓚ                                           │
TASK   EXTENSION     :  ⓛ                                          │
TASK  IMAGE  SIZE  : 10496. WORDSⓜ                                 │
TOTAL  TASK  SIZE  :ⓝ                                              │
TASK ADDRESS LIMITS: 000000 050753ⓞ                                │
R-W DISK BLK LIMITS: 000002 000106 000105 00069.ⓟ                  │
R-O DISK BLK LIMITS:ⓠ                                              ┘

OVR.TSK;25 OVERLAY DESCRIPTION:                                    ┐
                                                                   │
BASE    TOP        LENGTH                                          │
----    ---        ------                                          │
000000  020677  020700  08640.    ROOTM                            │ ❸   OVERLAY
020700  034723  014024  06164.      MULOV                          │   DESCRIPTION
020700  034723  014024  06164.      ADDOV                          │
034724  050747  014024  06164.        SUBOV                        │
034724  050753  014030  06168.        DIVOV                        ┘

OVR.TSK;25   MEMORY ALLOCATION MAP  TKB M36          PAGE 2                  ┐
ROOTM                  13-APR-79   09:10                                     │
                                                                             │
*** ROOT SEGMENT: ROOTMⓐ                                                     │
                                                                             │
R/W MEM  LIMITS: 000000 020677 020700 08640.ⓑ                                │
DISK BLK LIMITS: 000002 000022 000021 00017.ⓒ                                │
                                                                             │
MEMORY ALLOCATION SYNOPSIS:                                                  │
                                                                             │
SECTION                                         TITLE  IDENT  FILE           │
-------                                         -----  -----  ----           │
. BLK.:(RW,I,LCL,REL,CON) 001176 002034 01052.ⓓ                              │
ANS   :(RW,D,GBL,REL,OVR) 003232 004006 02054.                               │
                          003232 004006 02054.  ROOTM   01    ROOTM.OBJ;1ⓔ   │
                                .                                            │
                                .                                            │
                                .                                            │ ❹ ROOT SEGMENT
GLOBAL SYMBOLS:                                                              │     ALLOCATION
                                                                             │
AADD    007276-R  DIVV    007316-R  PRINT  014274-R  SUBB    007306-Rⓕ       │
ANS     007232-R  MULL    007266-R  SAVAL  020366-R                          │
                                                                             │
FILE: ROOTM.OBJ;1  TITLE: ROOTM   IDENT: 01ⓖ                                 │
    <ANS    >: 003232 007237 004006 02054.ⓗ                                  │
       ANS     007232-Rⓘ                                                     │
    <MAIN   >: 010010 010105 000076 00062.ⓙ                                  │
                                .                                            │
                                .                                            │
                                .                                            │
                                                                             │
************                                                                 │
                                                                             │
UNDEFINED REFERENCES:ⓚ                                                       ┘
```

Figure 6-2 Memory Allocation File (Map) Example

```
OVR.TSK;25    MEMORY ALLOCATION MAP  TKB M36          PAGE 4
MULOV                  13-APR-79   09:10


*** SEGMENT: MULOV


R/W MEM  LIMITS: 020700 034723 014024 06164.
DISK BLK LIMITS: 000023 000037 000015 00013.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                             TITLE  IDENT  FILE
-------                                             -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 020700 000000 00000.
MULL  :(RO,I,GBL,REL,CON) 020700 014024 06164.
                          020700 014024 06164.  MULOV  01     MULOV.OBJ;1   ❺ TREE SEGMENT
                                                                              DESCRIPTION

GLOBAL SYMBOLS:

MULL   034700-R


FILE: MULOV.OBJ;1  TITLE: MULOV    IDENT: 01
    <MULL  >: 020700 034723 014024 06164.
       MULL   034700-R
                        .
                        .
                        .


************

UNDEFINED REFERENCES:

*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 8178.ⓐ
    WORK  FILE  READS: 0.}ⓑ                                                 ❻ TASK BUILDER
    WORK  FILE WRITES: 0.}                                                     STATISTICS
    SIZE OF CORE POOL: 8200. WORDS (32. PAGES)ⓒ
    SIZE OF WORK FILE: 3328. WORDS (13. PAGES)ⓓ

    ELAPSED TIME:00:00:28ⓔ
```

Figure 6-2(Cont.) Memory Allocation File (Map) Example

Notes to Figure 6-2:

❶ The Page Header shows the name of the task image file and the overlay segment name (if applicable), along with the date, time, and version of the Task Builder that created the map.

❷ The Task Attribute Section contains the following information:

ⓐ Task Name -- The name specified in the TASK option. If you do not use the TASK option, the Task Builder suppresses this field.

ⓑ Partition Name -- The partition specified in the PAR option. If you do not specify a partition, the default is partition GEN.

ⓒ Identification -- The task version as specified in the .IDENT assembler directive. If you do not specify the task identification, the default is 01.

ⓓ Task UIC -- The task UIC as specified in the UIC option. If you do not specify the UIC, the default is the terminal UIC.

ⓔ Task Priority -- The priority of the task as specified in the PRI option. If you do not specify PRI, the default is 50.

ⓕ Stack Limits -- The low and high octal addresses of the stack, followed by its length in octal and decimal bytes.

ⓖ ODT Transfer Address -- the starting address of the ODT debugging aid. If you do not specify the ODT debugging aid, this field is suppressed.

ⓗ Program Transfer Address -- The address of the symbol specified in the .END directive of the source code of your task. If you do not specify a transfer address for your task, the Task Builder automatically establishes a tranfer address of 000001 for it. The Task Builder also suppresses this field in the map if you do not specify a transfer address.

ⓘ Task Attributes -- These attributes are listed only if they differ from the defaults. One or more of the following may be displayed:

AC	Ancillary control processor
AL	Task is checkpointable, and task image file contains checkpoint space allocation
CP	Task is checkpointable, and task image file will be checkpointed to system checkpoint file
DA	Task contains debugging aid
EA	Task uses KE11-A extended arithmetic element
FP	Task uses floating-point processor
-HD	Task image does not contain header
PI	Task contains position-independent code and data
PM	Postmortem Dump requested in the event of abnormal task termination

PR Task is privileged

-SE Messages addressed to the task through the SEND directive will be rejected by the Executive

SL Task can be slaved

TR Task initial PS word has T-bit enabled

(j.) Total Address Windows -- the number of window blocks allocated to the task.

(k.) Mapped Array -- the amount of physical memory (decimal words) allocated through the VSECT option or Mapped Array Declaration (GSD type 7, described in Section B.1.8 of Appendix B).

(l.) Task Extension -- the increment of physical memory (decimal words) allocated through the EXTTSK or PAR option.

(m.) Task Image Size -- the amount of memory (decimal words) required to contain your task's code. This number does not include physical memory allocated through the EXTTSK option.

(n.) Total Task Size -- the amount of physical memory (decimal words) allocated including mapped array area and task extension area.

(o.) Task Address Limits -- the lowest and highest virtual addresses allocated to the task, exclusive of virtual addresses allocated to VSECTs and shared regions.

(p.) Read/Write Disk Block Limits -- from left to right: the first octal relative disk block number of the task's header; the last octal relative disk block number of the task image; the total contiguous disk blocks required to accommodate the read/write portion of the task image in octal and decimal.

(q.) Read-Only Disk Block Limits -- from left to right: the first octal relative disk block of the multiuser task's read-only region; the last octal relative disk block number of the read-only region; the total contiguous disk blocks required to accommodate the read-only region in octal and decimal. This field appears only when you are building a multiuser task.

3 The Overlay Description shows, for each overlay segment in the tree structure of an overlaid task, the beginning virtual address (the base), the highest virtual address (the top), the length of the segment in octal and decimal bytes, and the segment name. Indenting is used to illustrate the ascending levels in the overlay structure. The Task Builder prints the Overlay Description only when an overlaid task is created.

4 The Root Segment Allocation -- This section has the following elements:

(a.) Root Segment -- The name of the root segment. If your task is a single-segment task, the entire task is considered to be the root segment.

(b.) Read/Write Memory Limits -- From left to right: the beginning virtual address of the root segment (the base), the virtual address of the last byte in the segment (the top), the length of the segment in octal and decimal bytes.

c. Disk Block Limits -- From left to right: the first relative block number of the beginning of the root segment, the last relative block number of the root segment, total number of disk blocks in octal, and the total number of disk blocks in decimal.

d. Memory Allocation Synopsis -- From left to right: the program section name, the program section attributes, starting virtual address of the program section, total length of the program section in octal and decimal bytes.

The program section shown as . BLK. in this field is the unnamed relocatable program section. Notice in this example that there are 636(8) bytes allocated to it (2034 bytes - 1176 bytes = 636 bytes). This allocation is the result of calls to routines that reside within the unnamed program section in SYSLIB. (For more information, see the description of the MA switch in Section 6.1.14.)

e. Module contributor -- This field lists the modules that have contributed to each program section. In this example, the program section ANS was defined in module ROOTM. The module version is 01 (as a result of the .IDENT assembler directive) and the file name from which the module was extracted is ROOTM.OBJ;1. If the program section ANS had been defined in more than one module, each contributing module and the file from which it was extracted would have been listed here.

NOTE

The absolute section, . ABS. is not shown because it appears in every module and always has a length of 0.

f. The global symbols section lists the global symbols defined in the segment. Each symbol is listed along with its octal value. A -R is appended to th value if the symbol is relocatable. The list is alphabetized in columns.

The file contents section (which is composed of the four fields listed below) is printed only if you specify /-SH in the Task Builder command sequence. The Task Builder creates this section for each segment in an overlay structure. It lists the following information:

g. Input file -- File name, module name as established by the .TITLE assembler directive, module version as established by the .IDENT assembler directive.

h. Program section -- Program section name, starting virtual address of the program section, ending virtual address of the program section, length in octal and decimal bytes.

i. Global symbol -- Global symbol names within each program section and their octal values. If the segment is autoloadable (see Chapter 5), this value will be the address of an autoload vector. The autoload vector in turn will contain the actual address of the symbol.

An -R is appended to the value if the symbol is relocatable.

(j.) Program section -- This field is identical to the field described in note g above.

(k.) Undefined References -- This field lists the undefined global symbols in the segment.

❺ The Tree Segment Description is printed for every overlay segment in an overlay structure. Its contents are the same for each overlay segment as the Root Segment Allocation is for the root segment.

❻ Task Builder Statistics lists the following information, which can be used to evaluate Task Builder performance:

(a.) Work File References -- The number of times that the Task Builder accessed data stored in its work file.

(b.) Work File Reads -- The number of times that the work file device was accessed to read work file data.

(c.) Work File Writes -- The number of times that the work file device was accessed to write work file data.

(d.) Size of Pool -- The amount of memory that was available for work file data and table storage.

(e.) Size of Work File -- The amount of device storage that was required to contain the work file.

(f.) Elapsed Time -- The amount of wall-clock time required to construct the task image and produce the memory allocation (map) file. Elapsed time is measured from the completion of option input to the completion of map output. This value excludes the time require to process the overlay description, parse the list of input file names, and create the cross-reference listing (if specified).

See Appendix E for a more detailed discussion of the work file.

6.1.24 /SL -- Slave

File

Task image

Syntax

```
file.TSK/SL=file.OBJ
```

Description

Your task is slaved to an initiating task.

Effect

The Task Builder attaches the slave attribute to your task. When your task successfully executes a Receive Data directive, the system gives the UIC and TI: device of the sending task to it. The slave task then assumes the indentity and privileges of the sending task.

This switch only applies to you if your system has multiuser protection. (Refer to your system generation manual for more information on multiuser protection and slave tasks.)

Default

/-SL

6.1.25 /SP -- Spool Map Output

File

Memory allocation (map)

Syntax

```
file.TSK,file.MAP/-SP=file.OBJ
```

Description

This switch determines whether or not the Task Builder calls the print spooler to spool your memory allocation (map) file after task build.

Effect

By default, when you specify a map file in a Task Builder command sequence, the Task Builder creates a map file on device SY0: and then has the file queued for listing on LP0:.

If you negate this switch (as shown in the "syntax" section above), the Task Builder will create the map file on device SY0: but will not call the print spooler to output it to LP0:

Default

/SP

SQ

6.1.26 /SQ -- Sequential

File

Task image

Syntax

```
file.TSK/SQ=file.OBJ
```

Description

The Task Builder constructs your task image from the program sections you specified, in the order that you input them.

Effect

The Task Builder does not reorder the program sections alphabetically. Instead, it collects all the references to a given program section from your input object modules, groups them according to their access code and, within these groups, allocates memory for them in the order that you input them.

You use this switch to satisfy any adjacency requirements that existing code may have when you are converting it to run under RSX-11. Use of this feature is otherwise discouraged for the following reasons:

- Standard library routines (such as FORTRAN I/O handling routines and FCS modules from SYSLIB) will not work properly.
- Sequential allocation can result in errors if you alter the order in which modules are linked.

Alternatively, you can achieve physical adjacency of program sections by selecting names alphabetically to correspond to the desired order.

Default

/-SQ

6.1.27 /SS -- Selective Search

File

Input

Syntax

```
file.TSK=file.OBJ,file.OBJ/SS
     or
file.TSK=file.OBJ,file.STB/SS
     or
file.TSK=file.OBJ,file.OLB/LB/SS
```

Description

The SS switch directs the Task Builder to include into its internal symbol table only those global symbols for which there is a previously undefined reference.

Effect

When processing an input file, the Task Builder normally includes into its internal symbol table each global symbol it encounters within the file whether or not there are references to it. When you attach the SS switch to an input file, the Task Builder checks each global symbol it encounters within that file against its list of undefined references. If the Task Builder finds a match, it includes the symbol into its symbol table.

Default

/-SS

Example

Assume that you are building a task named SEL.TSK. The task is composed of input files containing global entry points and references (calls) to them as shown in Table 6-2.

Table 6-2 Input Files for SEL.TSK

Input File Name	Global Definition	Global Reference
IN1		A
IN2	A B C	
IN3		C
IN4	A B C	

File IN2 and IN4 contain global symbols of the same name that represent entry points to different routines within their respective files. Assume that you want the Task Builder to resolve the reference to global symbol A in IN1 to the definition for A in IN2. Assume further, that you want the Task Builder to resolve the reference to global symbol C in IN3 to the definition for C in IN4. By selecting the sequence of the input files properly and applying the SS switch to files IN2 and IN4, the Task Builder will resolve the references correctly without conflict. The following command sequence illustrates the correct sequence:

```
TKB>SEL.TSK-IN1.OBJ,IN2.OBJ/SS,IN3.OBJ,IN4.OBJ/SS
```

The Task Builder processes input files from left to right, therefore, in processing the above command sequence, the Task Builder processes file IN1 first and encounters the reference to symbol A. There is no definition for A within IN1, therefore, the Task Builder marks A as undefined and moves on to process file IN2. Because the SS switch is attached to IN2, the Task Builder limits its search of IN2 to symbols it has previously listed as undefined, in this case, symbol A. The Task Builder finds a definition for A and places A in its symbol table. There are no undefined references to symbols B or C, so the Task Builder does not place either of these symbols in its symbol table.

NOTE

It is important to realize that the SS switch effects only the way the Task Builder constructs its internal symbol table. The routines for which symbols B and C are entry points will be included in the task image even though there are no references to them.

The Task Builder moves on to IN3. It encounters the references to symbol C. Since the Task Builder did not include symbol C from IN2 in its symbol table, it cannot resolve the reference to C in IN3. The Task Builder marks symbol C as undefined and moves on to IN4.

When the Task Builder processes IN4 it encounters the definition for C in that file and includes it in the table. Again, since the SS switch is attached to IN4, the Task Builder includes only C in its symbol table.

When the Task Builder has completed its processing of the above command sequence, it has constructed a task image composed of all of the code from all of the modules, IN1 through IN4. However, only symbols A from IN2 and C from IN4 will appear in its internal symbol table.

NOTE

The example above does not represent good programming practice. It is included here to illustrate the effect of the SS switch on the Task Builder during a search sequence.

The SS switch is particularly valuable when used to limit the size of the Task Builder's internal symbol table during the building of a privileged task that references the Executive's routines and data structures. By specifying the Executive's Symbol Definition File (STB) as an input file and applying the SS switch to it, the Task Builder will include into its internal symbol table only those symbols in the Executive that the task references. An example of a Task Builder command sequence that illustrates this is shown below:

```
TKB>OUTFILE.TSK/PR:5=INFILE.OBJ,RSX11M.STB/SS
```

The above command sequence directs the Task Builder to build a privileged task named OUTFILE.TSK from the input file INFILE.OBJ. The specification of the Executive's STB file as an input file with the SS switch applied to it directs Task Builder to extract from RSX11M.STB only those symbols for which there are references within OUTFILE.TSK.

TR

6.1.28 /TR - Traceable

File

Task image

Syntax

```
file.TSK/TR=file.OBJ
```

Description

Your task is traceable.

Effect

The Task Builder sets the T-bit in the initial PS word of your task. When your task is executed, a trace trap occurs on the completion of each instruction.

Default

/-TR

6.1.29 /WI -- Wide Listing Format

File

Memory allocation (map)

Syntax

```
file.TSK,file.MAP/-WI=file.OBJ
```

Description

This switch controls the width of your map file.

Effect

By default, the Task Builder formats a map file 132 columns wide. When you negate this switch (as in the "Syntax" section above), the Task Builder formats the map file 80 columns wide.

Default

/WI

XT

6.1.30 /XT[:n] -- Exit on Diagnostic

File

Task image

Syntax

```
file.TSK/XT:4=file.OBJ
```

Description

More than n errors are not acceptable.

Effect

The Task Builder exits after encountering n errors. The number of errors can be specified as a decimal or octal number, using the convention:

n.	indicates a decimal number (the decimal point must be included).
#n or n	indicates an octal number.

If you do not specify n, the Task Builder assumes that n is 1.

Default

/-XT

6.2 OPTIONS

Task Builder options provide you with the means to give the Task Builder information about the characteristics of your task.

These options which are listed in Table 6-2 can be divided into seven categories. The identifying abbreviation and a brief description of each category are listed below:

1. contr — You use control options to affect Task Builder execution. ABORT is the only member of this category. You can direct the Task Builder to abort the task-build with this option.

2. ident — You use identification options to identify your task's characteristics. You can specify the name of your task, its priority, user identification code, and partition with options in this category.

3. alloc — You use allocation options to modify your task's memory allocation. With the options in this category, you can change the size of your task's stack and program sections. When you write programs in a high-level language, you can change the size of your work areas and buffers and declare the virtual base address and size of program sections. Finally, you can declare the number of additional window blocks (if any) that your task requires.

4. share — You use storage sharing options to indicate to the Task Builder that your task intends to access a shared region.

5. device — You use device specifying options to specify the number of units required by your task, and the assignment of logical unit numbers to physical devices.

6. alter — You use the content altering options to define a global symbol and value, or to introduce patches in your task image.

7. synch — You use synchronous trap options to define synchronous trap vectors.

Some Task Builder options are of interest to all users of the system; others are of interest only to high-level language programmers; and still others are of interest only to MACRO-11 programmers. Table 6-3 lists all the options alphabetically, and gives a brief description of each.

Table 6-3
Task Builder Options

Option	Meaning	Interest[1]	Category
ABORT	Directs TKB to terminate a task build	H,M	contr
ABSPAT	Declares absolute patch values	M	alter
ACTFIL	Declares number of files open simultaneously	H	alloc
ASG	Declares device assignment to logical units	H,M	device
CMPRT[2]	Declares completion routine for supervisor-mode library	H,M	ident
COMMON LIBR	Declare task's intention to access a memory-resident shared region	H,M	share
EXTSCT	Declares extension of a program section	H,M	alloc
EXTTSK	Declares extension of the amount of memory owned by a task	H,M	alloc
FMTBUF	Declares extension of buffer used for processing format strings at run time	H	alloc
GBLDEF	Declares a global symbol definition	M	alter
GBLPAT	Declares a series of patch values relative to a global symbol	M	alter
GBLREF	Declares a global symbol reference	H,M	alter
GBLXCL[2]	Declares global symbols to be excluded from a supervisor-mode library	H,M	alter
LIBR	Declares task's intention to access a memory-resident shared region	H,M	share
MAXBUF	Declares an extension to the FORTRAN record buffer	H	alloc
ODTV	Declares the address and size of the debugging aid SST vector	M	synch

[1] The user interest range is indicated as follows:

- H indicates options of interest to high-level language (such as FORTRAN) programmers
- M indicates options of interest to MACRO-11 programmers

[2] These options are applicable to RSX-11M-PLUS systems only.

(continued on next page)

Table 6-3 (Cont.)
Task Builder Options

Option	Meaning	Interest[1]	Category
PAR	Declares partition name and dimensions	H,M	ident
PRI	Declares priority	H,M	ident
RESCOM RESLIB	Declare task's intention to access a memory-resident shared region	H,M	share
RESSUP[2]	Declares task's intention to access a resident supervisor-mode library	H,M	share
ROPAR[2]	Declares partition in which read-only portion of multiuser task is to reside	H,M	ident
STACK	Declares the size of the stack	H,M	alloc
SUPLIB[2]	Declares task's intention to access a system-owned supervisor-mode library	H,M	share
TASK	Declares the name of the task	H,M	ident
TSKV	Declares the address of the task SST vector	M	synch
UIC	Declares the user identification code under which the task runs	H,M	ident
UNITS	Declares the maximum number of units	H,M	device
VSECT	Declares the virtual base address and size of a program section	H,M	alloc
WNDWS	Declares the number of additional address windows required by the task.	H,M	alloc

[1] The user interest range is indicated as follows:

- H indicates options of interest to high-level language (such as FORTRAN) programmers
- M indicates options of interest to MACRO-11 programmers

[2] These options are applicable to RSX-11M-PLUS systems only.

ABORT

6.2.1 ABORT -- Abort the Task-Build

You use the ABORT option when you discover that an earlier error in the terminal sequence will cause the Task Builder to produce an unusable task image.

The Task Builder, on recognizing the keyword ABORT, stops accepting input and restarts for another task-build.

Syntax

```
ABORT=n
```

n

An integer value. The integer is required to satisfy the general form of an option; however, the value is ignored in this case.

Default

none

NOTE

If you type a CTRL/Z at any time it will cause the Task Builder to stop accepting input and begin building the task.

The ABORT option is the only proper way for you to restart the Task Builder if you discover an error and decide you do not want the Task Builder output.

6.2.2 ABSPAT -- Absolute Patch

You use the ABSPAT option to declare a series of object level patches starting at a specified base address. You can specify up to 8 patch values.

Syntax

```
ABSPAT=seg-name:address:val1:val2...:val8
```

seg-name

The 1- to 6-character Radix-50 name of the segment.

address

The octal address of the first patch. The address can be on a byte boundary; however, two bytes are always modified for each patch: the addressed byte and the following byte.

val1

An octal number in the range of 0 through 177777 to be stored at address.

val2

An octal number in the range of 0 through 177777 to be stored at address+2

.
.
.

val8

An octal number in the range of 0 through 177777 to be stored at address 14.

NOTE

All patches must be within the segment address limits or the Task Builder will generate the following error message:

```
TKB--*DIAG*--LOAD ADDRESS OUT OF RANGE IN module name
```

ACTFIL

6.2.3 ACTFIL -- Number of Active Files

You use the ACTFIL option to declare the number of files that your task can have open simultaneously. For each active file that you specify, the Task Builder allocates approximately 512 bytes.

If you specify less than four active files (the default), the ACTFIL option saves space. If you want your task to have more than four active files, you must use the ACTFIL option to make the additional allocation.

You must include an Object Time System (OTS) and record I/O service routines (FCS or RMS-11) in your task image for the extension to take place. The program section that is extended has the reserved name $$FSR1.

Syntax

```
ACTFIL=file-max
```

file-max

A decimal integer indicating the maximum number of files that can be open at the same time.

Default

```
ACTFIL=4
```

6.2.4 ASG -- Device Assignment

The ASG option declares the physical device that is assigned to one or more logical units.

Syntax

```
     ASG=device-name:unit-num1:unit-num2...:unit-num8
```

device-name

A 2-character alphabetic device name followed by a 1- or 2-digit decimal unit number.

unit-num1
unit-num2
  .
  .
  .
unit-num8

Decimal integers indicating the logical unit numbers.

Default

```
     ASG=SY0:1:2:3:4,TI0:5,CL0:6
```

CMPRT

6.2.5 CMPRT -- Completion Routine

The CMPRT option is available on RSX-11M-PLUS systems only. You use this option to identify a task as a supervisor-mode library. It also identifies the entry point of the completion routine that the library will use to return control to your program in user mode.

You should define your completion routine in the root segment of your supervisor-mode library. When your library is an overlay structure with a root of 0, the completion routine must appear in all of the main branches of the overlay structure nearest to the root. The completion routine also must have the same virtual address in all branches.

When you specify a completion routine that does not appear in your code, the Task Builder expects to find the routine in the System Library (SYSLIB) on device LB: under UFD [1,1]. When it extracts the completion routine from the System Library, the Task Builder places the routine in the root of your task. This means that if you build an overlaid supervisor-mode library with a root of 0, the Task Builder will expand the root to accommodate the completion routine.

Syntax

```
CMPRT=name
```

name

A 1- to 6-character Radix-50 name identifying the completion routine.

Default

none

6.2.6 COMMON or LIBR -- System-Owned Resident Common or System-Owned Resident Library

The COMMON and LIBR options are functionally identical; they both declare that your task intends to access a system-owned shared region. However, by convention, the COMMON option is used to identify a shared region that contains only data, and the LIBR option is used to identify a shared region that contains only code.

The term "system-owned" means that the Task Builder expects to find the common or library named in the keyword and the symbol definition file associated with it under UFD [1,1] on device LB:.

Syntax

```
     COMMON=name:access-code[:apr]
          or
     LIBR=name:access-code[:apr]
```

name

The 1- to 6-character Radix-50 name specifying the common or library. The Task Builder expects to find a symbol definition file having the same name as the common or library with an extension of .STB under [1,1] of device LB:.

access-code

The code RW (read/write) or the code RO (read-only) indicating the type of access the task requires.

NOTE

A privileged task can issue QIOs to a resident common even though the task has been linked to the common with read-only access.

apr

An integer in the range of 1 through 7 that specifies the first Active Page Register (APR) that you want the Task Builder to reserve for the shared region. The Task Builder recognizes the APR only for a mapped system, you can specify it only for position-independent shared regions. If you omit the APR parameter and the shared region is position independent, the Task Builder will select the highest available APR to map the region. When a shared region is absolute, the base address of the region, and therefore, the APR the maps it, is determined by the PAR option when the region is built. Refer to PAR in Section 6.2.17.

Default

None

EXTSCT

6.2.7 EXTSCT -- Program Section Extension

You use the EXTSCT option to extend a program section.

If the program section to be extended has the attribute CON (concatenated), the Task Builder extends the section by the number of bytes you specify in the EXTSCT option. If the program section has the attribute OVR (overlay), the Task Builder will extend the section only if the length you specify in the EXTSCT option is greater than the length of the program section.

Syntax

```
EXTSCT=p-sect-name:extension
```

p-sect-name

A 1- to 6-character radix-50 name specifying the program section to be extended.

extension

An octal integer that specifies the number of bytes by which to extend the program section.

Example

In the following example, the program section BUFF is 200 bytes long.

```
EXTSCT=BUFF:250
```

The number of bytes the Task Builder extends the program section BUFF depends on the CON/OVR attribute:

- For CON, the extension is 250 bytes
- For OVR, the extension is 50 bytes

The Task Builder will extend the program section if it encounters the program section name in an input object file or in the overlay description file.

Default

None

6.2.8 EXTTSK -- Extend Task Memory

You use the EXTTSK option to direct the system to allocate additional memory for your task when it is installed in a system-controlled partition.

The amount of memory that the system allocates for your task is the sum of the task size plus the increment you specify (rounded up to the nearest 32-word boundary). If the task is built for a user-controlled partition, the allocation of task memory reverts to the partition size.

In an unmapped system, the Task Builder ignores the EXTTSK keyword.

NOTES

1. You should not use the EXTTSK option to extend a task containing memory resident overlays because the system will not map the extended area.

2. When you use the EXTTSK option to extend a checkpointable task that has been declared checkpointable with the AL switch, the check point file within the task image will be the size of the task plus the size of the extended task area.

Syntax

```
EXTTSK=length
```

length

A decimal number specifying the increase in task memory allocation (in words).

Default

The task is extended to the size specified in the PAR option (Section 6.2.17).

FMTBUF

6.2.9 **FMTBUF -- Format Buffer Size**

You use the FMTBUF option to declare the length of the internal working storage that you want the Task Builder to allocate within your task for the compilation of format specifications at runtime. The length of this area must equal or exceed the number of bytes in the longest format string to be processed.

Run-time compilation occurs whenever an array is referred to as the source of formatting information within a FORTRAN I/O statement. The program section that the Task Builder extends has the reserved name $$OBF1.

Syntax

```
FMTBUF=max-format
```

max-format

A decimal integer, larger than the default, that specifies the number of characters in the longest format specification.

Default

```
FMTBUF=132
```

6.2.10 GBLDEF -- Global Symbol Definition

You use the GBLDEF option to declare the definition of a global symbol.

The Task Builder considers this symbol definition to be absolute. It overrides any definition in your input object modules.

Syntax

```
GBLDEF=symbol-name:symbol-value
```

symbol-name

A 1- to 6-character Radix-50 name of the defined symbol.

symbol-value

An octal number in the range of 0 through 177777 assigned to the defined symbol.

Default

None

GBLPAT

6.2.11 GBLPAT -- Global Relative Patch

You use the GBLPAT option to declare a series of object level patch values starting at an offset relative to a global symbol. You can specify up to eight patch values.

Syntax

```
GBLPAT=seg-name:sym-name[+/-offset]:val1:val2...:val8
```

seg-name

The 1- to 6-character Radix-50 name of the segment.

sym-name

A 1- to 6-character Radix-50 name specifying the global symbol.

offset

An octal number specifying the offset from the global symbol.

val1

An octal number in the range of 0 through 177777 to be stored at the octal address of the first patch.

val2

An octal number in the range of 0 through 177777 to be stored at the first address+2.

.
.
.

val8

An octal number in the range of 0 through 177777 to be stored at the first address+14.

Default

None

NOTE

All patches must be within the segment address limits or the Task Builder will generate a fatal error.

6.2.12 GBLREF -- Global Symbol Reference

You use the GBLREF option to declare a global symbol reference. The reference originates in the root segment of the task. This keyword is used for memory-resident overlays of shared regions.

Syntax

```
GBLREF=symbol-name:symbol-name...:symbol-name
```

symbol-name

A 1- to 6-character name of a global symbol reference.

Default

None

GBLXCL

6.2.13 **GBLXCL -- Exclude Global Symbols**

The GBLXCL option keyword directs the Task Builder to exclude from the symbol definition file of a supervisor-mode library the symbol(s) specified in the option.

It is important to exclude from the symbol definition file of a Supervisor-mode library a symbol that is defined in the library if:

1. the symbol represents an entry point to a routine that uses the stack to pass parameters and,

2. the routine is called from the user-mode task linked to the library.

When the processor is switched from user to supervisor mode, two words are pushed onto the stack. If a user-mode task is permitted to call a routine in supervisor-mode that uses the stack to pass parameters, the two words pushed onto the stack will not be anticipated by the routine and both the called routine and the supervisor-mode completion routine will fail. (For more information on supervisor mode libraries, see Chapter 3.)

Syntax

```
GBLXCL=symbol-name:symbol-name...:symbol-name
```

symbol-name

The symbol(s) to be excluded.

Default

None

6.2.14 LIBR -- System-Owned Library

Refer to COMMON in Section 6.2.6.

MAXBUF

6.2.15 **MAXBUF -- Maximum Record Buffer Size**

You use the MAXBUF option to declare the maximum record buffer size required for any file used by the task.

If your task requires a maximum record size that exceeds the default buffer length, you must use this option to extend the buffer.

You must also include an Object Time System (OTS) in your task image for the extension to take place. The program section that is extended has the reserved name $$IOB1.

Syntax

```
MAXBUF=max-record
```

max-record

A decimal integer, larger than the default, that specifies the maximum record size in bytes.

Default

```
MAXBUF=132
```

6.2.16 ODTV -- ODT SST Vector

You use the ODTV option to declare that a global symbol is the address of the ODT Synchronous System Trap vector. You must define the global symbol in the main root segment of your task.

Syntax

```
ODTV=symbol-name:vector-length
```

symbol-name

A 1- to 6-character Radix-50 name of a global symbol.

vector-length

A decimal integer in the range of 1 through 32 specifying the length of the SST vector in words.

Default

None

PAR

6.2.17 PAR -- Partition

You use the PAR option to identify the partition for which your task is built.

In a mapped system, you can install your task in any system partition or user partition large enough to contain it. In an unmapped system, your task is bound to physical memory. Therefore, you must install your task in a partition starting at the same memory address as the partition for which it was built.

Syntax

```
PAR=pname[:base:length]
```

pname

The name of the partition.

base

The octal byte address defining the start of the partition.

length

The octal number of bytes contained in the partition.

In a mapped system, a length of 0 implies a system-controlled partition.

If the target system is mapped and you specify a partition length that is greater than the length of your task, the Task Builder will automatically extend the length of your task to match the length of the partition. This procedure is equivalent to using the EXTTSK keyword to increase the task memory. If your task size is greater than the partition size that you specify, TKB will generate the following error message:

```
TKB--*DIAG*-TASK HAS ILLEGAL MEMORY LIMITS
```

If you do not specify the base and length, the Task Builder will try to obtain that information from the system on which you are building your task. If you have specified a partition that resides in that system, the Task Builder can obtain the base and length.

The Task Builder binds the task to the addresses defined by the partition base. If the partition is user-controlled, the Task Builder verifies that the task does not exceed the length specification.

Default

```
PAR=GEN
```

6.2.18 PRI -- Priority

You use the PRI option to declare your task's execution priority.

On systems with multiuser protection, you cannot run a task at a priority that is greater than the system priority (50) unless it is installed or run from a privileged terminal. If you are working from a privileged terminal, and you do not override this option by specifying a different priority when you install your task, this priority is used.

Syntax

```
PRI=priority-number
```

priority-number

A decimal integer in the range of 1 through 250

Default

Established by Install; refer to the RSX-11M/M-PLUS MCR Operations Reference Manual.

RESCOM
RESLIB

6.2.19 RESCOM or RESLIB -- Resident Common or Resident Library

The RESCOM and RESLIB options are functionally identical; they both declare that your task intends to access a user-owned shared region. However, by convention the RESCOM option is used to identify a shared region that contains only data and the RESLIB option is used to identify a shared region that contains only code.

The term "user-owned" means that the resident common or library and the symbol definition file associated with it can reside under any UFD that you choose. You can specify the UFD and remaining portions of the file specification for both options. You must not place comments on the same line with either option.

Syntax

```
RESCOM=file-specification/access-code[:apr]
    or
RESLIB=file-specification/access-code[:apr]
```

file-specification

The memory image file of the resident common or resident library. The file specification format is discussed in Chapter 1.

access-code

The code RW (read/write) or the code RO (read-only), indicating the type of access required by the task.

NOTE

A privileged task can issue QIOs to a resident common even though the task has been linked to the common with read-only access.

apr

An integer in the range of 1 through 7 that specifies the first Active Page Register (APR) that you want the Task Builder to reserve for the common or library. The Task Builder recognizes the APR argument only for a mapped system. You can specify it only for position-independent shared regions. If the APR parameter is omitted and the shared region is position-independent, the Task Builder will select the highest available APR to map the region. When a shared region is absolute, the base address of the region, and therefore, the APR that maps it, is determined by the PAR option when the region is built. Refer to PAR in Section 6.2.17. You can specify it only for position-independent shared regions.

NOTES

1. The Task Builder expects to find a symbol definition file having the same name as the memory image file but with a file version of .STB, on the same device and under the same UFD as the memory image file.

2. Regardless of the version number you give in the file specification, the Task Builder uses the latest version of the .STB file.

Default

When you omit portions of the file-specification, the following defaults apply:

- UFD - taken from current terminal UIC
- device - SY0:
- file type - .TSK
- file version - latest

6.2.20 **RESLIB -- Resident Library**

Refer to RESCOM in Section 6.2.19.

6.2.21 RESSUP -- Resident Supervisor-Mode Library

You use the RESSUP option to declare that your task intends to access a user-owned, supervisor-mode library. The term "user-owned" means that the library and the symbol definition file associated with it can reside under any UFD that you choose. You can specify the UFD and remaining portions of the file specification. You must not place comments on the line with RESSUP.

Syntax

```
RESSUP=file-specification/[-]SV[:apr]
```

file-specification

The memory image file of the supervisor-mode library. The file specification has the standard RSX-11M/RSX-11M-PLUS format discussed in Chapter 1.

/[-]SV

The code SV for supervisor vectors or -SV for no supervisor vectors. If you specify SV, the Task Builder replaces calls to the supervisor-mode library within your task with context switching vectors. If you specify -SV, calls within your task to the supervisor-mode library are resolved directly and you must provide your own means for context switching.

NOTE

The elimination of supervisor vectors is useful if the supervisor-mode library contains threaded code.

apr

An integer in the range of 0 through 7 that specifies the first Supervisor Active Page Register that you want the Task Builder to reserve for your supervisor-mode library.

NOTES

1. The Task Builder expects to find a symbol definition file having the same name as the memory image file but with a file version of .STB, on the same device and under the same UFD as the memory image file.

2. Regardless of the version number you give in the file specification, the Task Builder uses the latest version of the .STB file.

3. When the CMPRT and RESSUP options appear in a command sequence together, the CMPRT option must be specified first.

Default

When you omit portions of the file specification, the following defaults apply:

- UFD - taken from the current terminal UIC
- device - SY0:
- file type - .TSK
- file version - latest

6.2.22 ROPAR -- Read-Only Partition -- RSX-11M-PLUS Only

You use this option to declare the partition in which the read-only portion of your multiuser task is to reside.

Syntax

```
ROPAR=parname
```

parname

The partition name in which your multiuser task is to reside.

Default

The partition in which the read/write portion of the task resides.

STACK

6.2.23 STACK -- Stack Size

You use the STACK option to declare the maximum size of the stack required by your task.

The stack is an area of memory that the MACRO-11 programmer uses for temporary storage, subroutine calls, and synchronous trap service linkage. The stack is referred to by hardware register 6 (SP, the stack pointer).

Syntax

```
STACK=stack-size
```

stack-size

A decimal integer specifying the number of words required for the stack.

Default

```
STACK=256
```

6.2.24 SUPLIB -- Supervisor-Mode Library -- RSX-11M-PLUS Only

You use this option to declare that your task intends to access a system-owned, supervisor-mode library. The term "system-owned" means that the Task Builder expects to find the supervisor-mode library and the symbol definition file associated with it under UFD [1,1] on device LB:.

Syntax

```
SUPLIB=name:[-]SV[:apr]
```

name

The 1- to 6-character Radix-50 name specifying the system-owned, supervisor-mode library. The Task Builder expects to find a symbol definition file having the same name as the library with a file version of .STB under [1,1] of device LB:.

:[-]SV

The code SV for supervisor vectors or -SV for no supervisor vectors. If you specify SV, the Task Builder replaces calls to the supervisor-mode library within your task with context switching vectors. If you specify -SV, calls within your task to the supervisor-mode library all resolved directly and you must provide your own means for context switching.

NOTE

The elimination of supervisor vectors is useful if the supervisor-mode library contains threaded code.

apr

An integer in the range of 0 through 7 that specifies the first Supervisor Active Page Register that the Task Builder is to reserve for the library.

NOTE

When the CMPRT and SUPLIB options appear in a command sequence together, the CMPRT option must be specified first.

Default

None

TASK

6.2.25 **TASK -- Task Name**

You use the TASK option to give your task an installed name different from its task image name.

Syntax

```
TASK=task-name
```

task-name

A 1- to 6-character name identifying your task.

Default

The first six characters of the task image file name are used to identify the task when the task is installed.

6.2.26 TSKV -- Task SST Vector

You use the TSKV option to declare that a global symbol is the address of the task Synchronous System Trap (SST) vector. You must define the global symbol in the main root segment of your task.

Syntax

```
TSKV=symbol-name:vector-length
```

symbol-name

A 1- to 6-character name of a global symbol.

vector-length

A decimal integer in the range of 1 through 32 specifying the length of the SST vector in words.

Default

None

UIC

6.2.27 UIC -- User Identification Code

You use the UIC option to declare the User Identification Code (UIC) for your task when you run it with a time-based schedule request.

Syntax

```
UIC=[group,member]
```

group

An octal number in the range of 1 through 377, or a decimal number in the range of 1 through 255. Decimal numbers must be followed by a decimal point (.).

member

An octal number in the range of 1 through 377, or a decimal number in the range of 1 through 255. Decimal numbers must be followed by a decimal point (.).

Default

The UIC that the Task Builder is running under (normally the terminal UIC).

6.2.28 UNITS -- Logical Unit Usage

You use the UNITS option to declare the number of logical units that are used by your task.

Syntax

```
UNITS=max-units
```

max-units

A decimal integer in the range of 0 through 250 specifying the maximum number of logical units. Note that in mapped systems the UNITS option creates tables that require dynamic memory. Therefore, large arguments can exhaust dynamic memory. (Refer to the system generation manual for more information.)

Default

```
UNITS=6
```

VSECT

6.2.29 VSECT -- Virtual Program Section

You use the VSECT option to specify the virtual base address, virtual length, and optionally, the physical memory allocated to the named program section. Refer to Section 3.4 for more information on virtual program sections.

Syntax

```
VSECT=p-sect-name:base:window[:physical-length]
```

p-sect-name

A 1- to 6-character program section name.

base

An octal value representing the virtual base address of the program section in the range of 0 through 177777. If you use the mapping directives the value you specify must be a multiple of 4K.

window

An octal value specifying the amount of virtual address space in bytes allocated to the program section. Base plus window must not exceed 177777 (octal).

physical-length

An octal value specifying the minimum amount of physical memory to be allocated to the section in units of 64-byte blocks. The Task Builder rounds this value up to the next 256-word limit. This value, when added to the task image size and any previous allocation, must not cause the total to exceed 2048K bytes. If you do not specify a length, the Task Builder assumes a value of 0.

Default

Physical-length defaults to 0.

6.2.30 WNDWS -- Number of Address Windows

The WNDWS option declares the number of address windows required by the task in addition to those needed to map the task image, and any mapped array or shared region. The number specified is equal to the number of simultaneously mapped regions the task will use.

Syntax

```
WNDWS=n
```

n

An integer in the range 1 through 7 in an RSX-11M system and 1 through 15 in an RSX-11M-PLUS system.

Default

```
WNDWS=0
```

CHAPTER 7

HOST AND TARGET SYSTEMS

7.1 INTRODUCTION

You can build a task on one system (the host), and run it on another (the target). For example, your installation might consist of one large computer system with mapping hardware, and several smaller unmapped systems. On the large system you could create and debug tasks, and then transfer them to the smaller systems to run.

For example, if you are developing a task named TK3, using the default partition of your host system, the Task Builder command could be:

```
>TKB TK3,TK3=SQ1,SQ2
```

When you are ready to move TK3 to a target system, you build it again, indicating the mapping status of the target system, and naming the partition in which the task is to reside:

```
>TKB
TKB>TK3/-MM,TK3=SQ1,SQ2
TKB>/
ENTER OPTIONS:
TKB>PAR=PART1:100000:40000
TKB>//
```

The resulting task image is ready to run on the unmapped target system.

You can transfer a task from the host system to the target system by following these steps:

1. Build the task image specifying the partition in which the task will run. If the target system is an unmapped system, specify the partition's base address and size.

2. Ensure that any shared regions accessed by the task are present in both systems.

3. If the target system and the host system do not have the same mapping status, set the Memory Management switch (/MM or /-MM) to reflect the mapping status of the target system.

The task code must not use any hardware options (FPP, EIS, EAE, etc.) that are not present on the target system. This is particularly important if you are a FORTRAN user because FORTRAN tasks often use mathematics routines that are hardware dependent. (Refer to the IAS/RSX-11 FORTRAN IV Installation Guide and the IAS/RSX-11 FORTRAN IV User's Guide for more information on FORTRAN requirements).

7.2 EXAMPLE: TRANSFERING A TASK FROM A HOST TO A TARGET SYSTEM

In this section, the resident library LIB and the task that refers to it MAIN (from Example 4, Chapter 3) are rebuilt to run on an unmapped system. To save space, only the Task Builder command sequences are shown.

Assuming that the target system has a partition within it named LIB, only two changes need be made to the original command sequence that builds the library:

1. The negated memory management switch (/-MM) must be attached to the image file specification

2. The partition base and length must be specified

The modified command sequence is as follows:

```
TKB> LIB/-HD/PI/-MM,LIB/-WI,LIB=LIB
TKB> /
ENTER OPTIONS:
TKB> STACK=0
TKB> PAR=LIB:136000:20000
TKB> //
```

If the target system does not contain a partition of the same name as the shared region you must change the name of the shared region to match the name of an existing partition in the target system. This is a requirement of RSX-11M; on RSX-11M-PLUS systems it is not.

Assuming that the target system has a partition named GEN and that the task MAIN is to run in that partition in the target system, three changes must be made to the command sequence that builds the task MAIN:

1. The negated memory management switch (/-MM) must be attached to the task image file specification

2. The APR parameter of the RESLIB keyword must be eliminated

3. The partition in which the task is to reside, its base address, and length must be explicitly specified

The modified command sequence is as follows:

```
TKB> MAIN/-MM,MAIN/MA/-WI=MAIN
TKB> /
ENTER OPTIONS:
TKB> RESLIB=LIB/RO
TKB> PAR=GEN:30100:40000
TKB> //
```

Figure 7-1 shows the map file of the resident library LIB for an unmapped system. LIB is bound to the partition base specified by the PAR keyword in the task-build command sequence. Note that the shared region is declared position independent even though it is bound to the partition base 136000. The position-independent declaration is not necessary in this example because the referencing task MAIN does not require the program section names within the library in order to refer to it. However, in applications involving tasks that require the program section names from the library, you must declare the library position-independent so that the Task Builder will place the program section names in the library's symbol definition file.

```
LIB.TSK;5   MEMORY ALLOCATION MAP  TKB M36          PAGE 1
                 22-JAN-79   10:49


PARTITION NAME : LIB
IDENTIFICATION : 01
TASK  UIC      : [303,3]
TASK ATTRIBUTES: -HD,PI
TOTAL ADDRESS WINDOWS: 1.
TASK  IMAGE  SIZE  : 64. WORDS
TASK ADDRESS LIMITS: 136000 136163
R-W DISK BLK LIMITS: 000003 000003 000001 00001.

*** ROOT SEGMENT: LIB


R/W MEM  LIMITS: 136000 136163 000164 00116.
DISK BLK LIMITS: 000002 000002 000001 00001.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                       TITLE  IDENT  FILE

. BLK.:(RW,I,LCL,REL,CON) 136000 000000 00000.
AADD  :(RO,I,GBL,REL,CON) 136000 000024 00020.
                          136000 000024 00020. LIB    01     LIB.OBJ;2
DIVV  :(RO,I,GBL,REL,CON) 136024 000026 00022.
                          136024 000026 00022. LIB    01     LIB.OBJ;2
MULL  :(RO,I,GBL,REL,CON) 136052 000024 00020.
                          136052 000024 00020. LIB    01     LIB.OBJ;2
SAVAL :(RO,I,GBL,REL,CON) 136076 000042 00034.
                          136076 000042 00034. LIB    01     LIB.OBJ;2
SUBB  :(RO,I,GBL,REL,CON) 136140 000024 00020.
                          136140 000024 00020. LIB    01     LIB.OBJ;2


GLOBAL SYMBOLS:

AADD   136000-R  MULL   136052-R  SUBB   136140-R
DIVV   136024-R  SAVAL  136076-R



*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 376.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 8200. WORDS (32. PAGES)
    SIZE OF WORD FILE: 768. WORDS (3. PAGES)

    ELAPSED TIME:00:00:05
```

Figure 7-1 Task Builder Map for LIB.TSK

Figure 7-2 shows the map file of the task MAIN for an unmapped system. The task is bound to the partition base 30100 and linked to the shared region LIB, which begins at 136000.

```
MAIN.TSK;6    MEMORY ALLOCATION MAP  TKB M36         PAGE 1
                        22-JAN-79   11:20


PARTITION NAME : GEN
IDENTIFICATION : 01
TASK  UIC      : [303,3]
STACK    LIMITS: 030312 031311 001000 00512.
PRG XFR ADDRESS: 031652
TOTAL ADDRESS WINDOWS: 2.
TASK  IMAGE  SIZE  : 1120 WORDS
TASK ADDRESS LIMITS: 030100 034313
R-W DISK BLK LIMITS: 000002 000006 000005 00005.

*** ROOT SEGMENT: MAIN



R/W MEM  LIMITS: 030100 034313 004214 02188
DISK BLK LIMITS: 000002 000006 000005 00005.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                          TITLE  IDENT  FILE

. BLK.:(RW,I,LCL,REL,CON) 031312 002564 01396.
                          031312 000530 00344. MAIN   01     MAIN.OBJ;1
                          .
                          .
                          .


GLOBAL SYMBOLS:

AADD    136000-R  SAVAL  136076-R  $CBDSG 033074-R  $CBTMG 033116-R
DIVV    136024-R  SUBB   136140-R  $CBOMG 033102-R  $CBVER 033102-R
IO.WVB  011000    $CBDAT 033060-R  $CBOSG 033110-R  $CDDMG 033276-R
MULL    136052-R  $CBDMG 033066-R  $CBTA  033140-R  $CDTB  033424-R
.
.
.


*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 2518.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 8200. WORDS (32. PAGES)
    SIZE OF WORK FILE: 1024. WORDS (4. PAGES)

    ELAPSED TIME:00:00:08
```

Figure 7-2 Task Builder Map for MAIN.TSK

CHAPTER 8

MEMORY DUMPS

The RSX-11M/M-PLUS Postmortem Dump task (PMD) generates Postmortem memory dumps of tasks that are abnormally terminated. In addition, PMD can produce edited dumps, called Snapshot Dumps, for tasks that are running. Section 8.1 describes Postmortem Dumps in general; Section 8.2 discusses the specific case of Snapshot dumps. Both types of dump are very useful debugging aids.

8.1 **POSTMORTEM DUMPS**

You can make a task eligible for a Postmortem Dump in any of three ways:

1. At task-build time, by specifying the PM switch for the task file. /-PM disables dumps; it is the default condition.

2. When you install a task by using the PMD switch to override the taskbuild option. /PMD=YES enables dumping; /PMD=NO disables dumping.

3. When you use the MCR command ABORT (described in the RSX-11M/M-PLUS MCR Operations Manual), by including the PMD switch in the command line to specify a dump.

You should install PMD in a 4K partition in which all other tasks are checkpointable. This allows the dump to be generated in a timely manner, and prevents the system from being locked up while the dump is being generated. PMD can dump either from memory or from the checkpoint image of your task. The PMD is sensitive to the location of the aborted task; therefore, if the aborted task is checkpointed during the dump, PMD switches to reading the checkpoint image. Once the task is checkpointed, PMD locks it out of memory until it has completed formatting the dump.

Dumps are always generated on the system disk under UFD [1,4]; therefore, to avoid errors from PMD, you must create a UFD for [1,4] before installing the task. When PMD finishes generating the dump, it attempts to queue the dump to the print spooler for subsequent printing. If no spooler is installed, the dump file is left on the disk and can be printed at a later time using the Peripheral Interchange Program (PIP; described in Chapter 4 of the RSX-11 Utilities Manual).

NOTE

> Dump files tend to be somewhat large. The dump of an 8K partition averages about 340 blocks. Therefore, if there is little space on the disk, it is important to print and delete the dump file without delay. The print spooler automatically deletes all files with the type .PMD after printing them.

Figure 8-1 shows the contents of a Postmortem and Snapshot Dump; the notes that follow the figure are keyed to figure and provide a description of the dumps contents. Snapshot Dumps are explained more fully in Section 8.2.

```
                         POST-MORTEM DUMP ❶

TASK: TT6 ❷                                     TIME: 5-OCT-76 15:06

PC: 000720 ❸      IOT EXECUTION ❸

REGS:      R0 - 000345   R1 - 074400   R2 - 000120   R3 - 140130
                                                                   ❹
           R4 - 000000   R5 - 000000   SP - 000304   PS - 170000

TASK STATUS:   MSG AST DST -CHK HLT STP REM MCR ❺

EVENT FLAG MASK FOR <1-16> 000001 ❻

CURRENT UIC: [007,001]    DSW: 1. ❼

PRIORITY: DEFAULT - 50.  RUNNING - 50.   I/O COUNT: 0.   TI DEVICE - TT6: ❽

LOAD DEVICE - DB0:     LBN: 1,160034 ❾


FLOATING POINT UNIT

     STATUS - 000000

     R0 - 000000  000000   000000  000000
     R1 - 000000  000000   000000  000000  ❿
     R2 - 000000  000000   000000  000000
     R3 - 000000  000000   000000  000000
     R4 - 000000  000000   000000  000000
     R5 - 000000  000000   000000  000000

LOGICAL UNITS


  UNIT  DEVICE      FILE STATUS
                                   ⓫
   1     DB0:
   2     DB0:
   3     DB0:
   4     DB0:

OVERLAY SEGMENTS LOADED AND RESIDENT LIBRARIES MAPPED

STARTING RELATIVE BLOCK: 000002   BASE: 000000   LENGTH: 001454  ⓬
STARTING RELATIVE BLOCK: 000004   BASE: 001454   LENGTH: 000264

TASK STACK

   ADDRESS CONTENTS ASCII RAD50  ⓭
    000304  000045    %      7
```

Figure 8-1 Sample Postmortem Dump (Truncated)

```
      PARTITION: GEN       VIRTUAL LIMITS: 000000 - 001777

000000  000304   000162   000001   067426  ! D6  B4   A Q08!
       304 000  162 000  001 000  026 157                  !D  r      o!
000010  003401   003401   170017   000352  !AD3 AD3 8PQ  E4!
       001 007  001 007  017 360  352 000                  !       p j !
000020  000304   000000   000000   000000  ! D6            !
       304 000  000 000  000 000  000 000                  !D          !
000030  000000   000000   000000   000000  !               !
       000 000  000 000  000 000  000 000                  !           !
000040  000000   140162   074106   000001  !    01Z SI0   A!
       000 000  162 300  106 170  001 000                  !   r@ Fx   !
000050  000000   000000   001104   000000  !         NT    !
       000 000  000 000  104 002  000 000                  !      D    !
000060  000373   000000   000000   000000  ! Fk            !
       373 000  000 000  000 000  000 000                  !           !
000070  000000   074150   000004   051646  !    SJX   D MON!
       000 000  150 170  004 000  246 123                  !   hx    &S!
000100  000000   051646   000000   051646  !    MON     MON!
       000 000  246 123  000 000  246 123                  !   &S    &S!
000110  000000   051646   000000   000001  !    MON       A!
       000 000  246 123  000 000  001 000                  !   &S      !
000120  067020   000000   001777   061404  !QXP      YW O3.!
       020 156  000 000  377 003  004 143                  ! n        c!
000130  000020   000000   000600   007406  !  P      IX BPF!
       020 000  000 000  200 001  006 017                  !           !
000140  170000   000720   000000   000000  !8P   KX        !
       000 360  320 001  000 000  000 000                  ! p P       !
000150  140130   000120   074400   000345  !01   B  SNP  E/!               14
       130 300  120 000  000 171  345 000                  !x@ P   y e !
000160  000000   000000   000000   000000  !               !
       000 000  000 000  000 000  000 000                  !           !

***  DUPLICATE THROUGH 000236  ***

000240  000000   000000   001110   000000  !         NX    !
       000 000  000 000  110 002  000 000                  !      H    !
00250   001454   000264   000000   000000  ! TL  DT        !
       054 003  264 000  000 000  000 000                  !,  4       !
000260  000001   001612   074360   003413  !  A  VZ SN  AEC!
       001 000  212 003  360 170  013 007                  !      px   !
00270   063014   131574   000000   000000  !PMD ...        !
       014 146  174 263  000 000  000 000                  ! f  3      !
000300  001051   000001   000045   050114  ! M3   A   7 L36!
       051 002  001 000  045 000  114 120                  !)     %  LP!
000310  000000   000001   000100   000304  !      A  AX  D6!
       000 000  001 000  100 000  304 000                  !      @  D !
000320  000524   000000   000000   000000  ! HT            !
       124 001  000 000  000 000  000 000                  !T          !
000330  000000   000000   000000   063014  !            PMD!
       000 000  000 000  000 000  014 146                  !          f!
000340  131574   047123   052120   052123  !... LUK MSX MS$!
       174 263  123 116  120 124  123 124                  ! 3 SN PT ST!
000350  000000   016746   177734   012746  !    D1N  7T CTF!
       000 000  346 035  334 377  346 025                  !   f  \  f !
000360  001037   104377   103456   005046  ! MW U61 UYF AX8!
       037 002  377 210  056 207  046 012                  !      .  & !
```

Figure 8-1 (Cont.) Sample Postmortem Dump (Truncated)

❶ Type of dump - Postmortem or Snapshot. If it is a Snapshot Dump, the dump identification is printed.

❷ The name of the task being dumped, and the date and time the dump was generated.

❸ The program counter at the time of the dump. If it is a Postmortem Dump, the reason the task was aborted is printed.

❹ The general registers, stack pointer, and processor status at the time of the dump.

❺ The task status flags at the time of the dump. See the description of ATL or TAL in the RSX-11M/M-PLUS MCR Operations Manual for the meaning of the flags.

❻ The task event flag mask word at the time of the dump. If the dump is a Snapshot Dump, the efn specified in the SNAP$ macro will be ON (see Section 8.2.2).

❼ The task UIC and the current value of the directive status word.

❽ The task's priority and default priority, number of outstanding I/O requests, and the terminal from which the task was initiated (TI:).

❾ The task load device and the logical block number for the start of the task image on the device.

❿ The floating-point unit (FPU) registers or the extended arithmetic element (EAE) registers if the task is using one of these hardware features. If the task is not using the FPU or EAE, these registers are not printed. If the task uses the FPU and does not specify /FP on the task image file, or if it uses the EAE unit and has not specified the EA switch, the registers are not printed. If the machine you are using has both an FPU and an EAE, PMD assumes you are using the FPU because it is the unit of choice for arithmetic computations.

⓫ The logical unit assignments at the time of the dump. UNIT is the logical unit number, and DEVICE is the device to which the logical unit is assigned. For Snapshot Dumps, the file names of any open files are displayed under FILE STATUS. Postmortem Dumps do not display this information because all of the files have been closed as a result of the I/O rundown on the aborted task.

⓬ The following are displayed: the overlay segments loaded and resident libraries mapped at the time of the dump; the relative block number of the segment; the base address; the length of the segment; and, for tasks using manual load, the segment names. For resident libraries, the library name is also displayed. The block number can be used to determine which segment is loaded, by reference to the memory allocation file generated by the Task Builder. The starting block number for each segment is the relative block number of the segment. By obtaining a match, the name of the segment in memory can be determined. Zero length segments are usually co-tree roots.

13 The task stack at the time of the dump. The address is displayed, along with the contents, in octal, ASCII, and Radix-50. Each word on the stack is dumped. If the stack pointer is above the initial value of the stack (H.ISP), only one word is dumped. The rest is dumped as part of the task image.

14 The task image itself. The partition being dumped and the limits of interest are displayed. For Postmortem Dumps, all address windows in use are dumped. For Snapshot Dumps, the virtual task limits that you request are displayed. The dump routine rounds the requested low limit down to the nearest multiple of eight bytes and rounds the requested high limit up to the nearest multiple of eight bytes. The dump image displays the virtual starting address of a 4-word block of memory, the data in both octal and Radix-50 on the first line, and byte octal and ASCII on the second line. A 4-word block that is repeated in a contiguous region of memory is printed once, and then noted by the message

```
*** DUPLICATE THROUGH xxxxxx ***
```

where xxxxxx indicates the last word that is duplicated. If the task was aborted, all address windows in use are dumped. If the dump is a Snapshot Dump, up to four contiguous blocks of memory can be dumped, if requested.

8.2 SNAPSHOT DUMP

Snapshot Dumps are edited dumps produced for running tasks. You can request a Snapshot Dump any number of times during the execution of a task. The information generated is under the control of the programmer.

Snapshot Dumps are generated by the following macros:

- SNPDF$ -- defines offsets in the Snapshot Dump Control Block, and defines control bits, which control the format of the dump
- SNPBK$ -- allocates the Snapshot Dump Control Block (see Table 8-1)
- SNAP$ -- causes a Snapshot Dump to be generated

SNPBK$ and SNAP$ issue calls to SNPDF$; so, you need not explicitly issue the SNPDF$ macro call. Sections 8.2.1 and 8.2.2 describe the SNPBK$ macro and the SNAP$ macro, respectively.

Label		Offset	
SB.CTL		0	CONTROL FLAGS
SB.DEV		2	DEVICE MNEMONIC
SB.UNT		4	UNIT NUMBER
SB.EFN		6	EVENT FLAG
SB.ID		10	SNAP IDENTIFICATION
SB.LM1	(L1)	12	MEMORY BLOCK 1 LIMITS
	(H1)	14	
	(L2)	16	MEMORY BLOCK 2 LIMITS
	(H2)	20	
	(L3)	22	MEMORY BLOCK 3 LIMITS
	(H3)	24	
	(L4)	26	MEMORY BLOCK 4 LIMITS
	(H4)	30	
SB.PMD		32	"PMD..." IN RADIX-50
		34	

Figure 8-2 Snapshot Dump Control Block Format

8.2.1 Format of the SNPBK$ Macro

The format of the SNPBK$ macro call is:

```
SNPBK$ dev,unit,ctl,efn,id,L1,H1,L2,H2,L3,H3,L4,H4
```

dev

The 2-character ASCII name of the device to which the dump is directed. If it is a directory device, the UFD [1,4] must be on the volume. The dump is written to the disk and then spooled to the line printer. If there is no print spooler, the file is left on the disk. If the device is not a directory device, the dump goes directly to the device.

unit

The unit number of the device to which the dump is directed.

ctl

The set of flags that control the format of the dump and the data to be printed. The flags are:

SC.HDR	Print the dump header (items 1 to 10 in Figure 8-1)
SC.LUN	Print information on all assigned LUNs (item 11)
SC.OVL	Print information about all loaded overlay segments (item 12)
SC.STK	Print the user stack (item 13)

SC.WRD	Print the requested memory in octal words and Radix-50 (item 14)
SC.BYT	Print the requested memory in octal bytes and ASCII (item 14)

efn
: The event flag to be used to synchronize your program and PMD.

id
: A number that identifies the Snapshot Dump. Because dumps can be requested at different times and under different conditions, this ID is used to identify the place or reason for the dump.

L1,L2,L3,L4
: The starting addresses of the memory blocks to be dumped.

H1,H2,H3,H4
: The ending addresses of the memory blocks to be dumped.

NOTE

> If no memory is to be dumped, each limit (L1,L2,L3,L4,H1,H2,H3,H4) should be 0.

Only one Snapshot Dump Control Block is allowed. It generates the global label ..SPBK.

NOTE

> Because SNPBK$ is used to allocate storage for the Snapshot Dump Control Block, all arguments except dev must be valid arguments for .WORD or .BYTE directives.

8.2.2 Format of the SNAP$ Macro

The format of the SNAP$ macro is:

```
SNAP$ ctl,efn,id,L1,H1,L2,H2,L3,H3,L4,H4
```

ctl
: The set of flags that control the format of the dump and the data to be printed. The flags are:

SC.HDR	Print the dump header
SC.LUN	Print information on all assigned LUNs
SC.STK	Print the user stack
SC.OVL	Print information about all loaded overlay segments
SC.WRD	Print the requested memory in octal words and Radix-50
SC.BYT	Print the requested memory in octal bytes and ASCII

efn

The event flag to be used to synchronize your program and PMD. A Wait-For-Single-Event-Flag directive is always generated to perform synchronization.

id

A number that identifies the Snapshot Dump. Because dumps can be requested at different times and under different conditions, this ID is used to identify the place or reason for the dump.

L1,L2,L3,L4

The starting addresses of memory blocks to be dumped.

H1,H2,H3,H4

The ending addresses of memory blocks to be dumped.

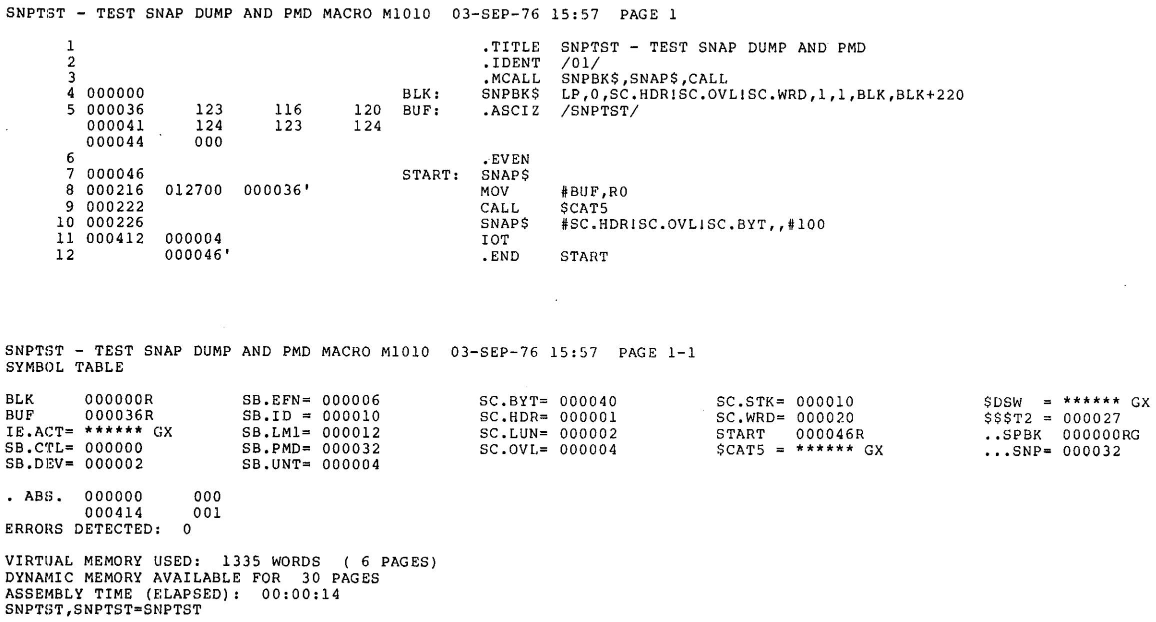
NOTES

1. If no memory is to be dumped, each limit (L1,L2,L3,L4,H1,H2,H3,H4) should be 0.
2. The control flags can be set in any combination; they are not mutually exclusive. Thus, any number of options can be obtained; for example, SC.HDR!SC.LUN!SC.WRD prints the header, LUNs, and the requested memory in word octal and Radix-50 mode.
3. Arguments should be specified only to override the information already in the Snapshot Dump Control Block.
4. Because SNAP$ generates instructions to move data into the Snapshot Dump Control Block, its arguments must be valid source operands for MOV instructions.

8.2.3 Example of a Snapshot Dump

The sample program shown in Figure 8-3 causes two Snapshot Dumps to be printed directly on LP0:. The first dump uses the parameters defined in the Snapshot Dump Control Block. The header is generated, and the data in relative locations BLK to BLK+220 is displayed, in word octal and Radix-50. The identification on the dump is 1.

The second dump causes the data in the locations BLK to BLK+220 to be displayed in byte octal and ASCII. A header is also generated. The dump identification is 64 (100 octal). Figures 8-4 and 8-5 show the dumps generated by the sample program.



```
SNPTST - TEST SNAP DUMP AND PMD MACRO M1010  03-SEP-76 15:57  PAGE 1

      1                                           .TITLE  SNPTST - TEST SNAP DUMP AND PMD
      2                                           .IDENT  /01/
      3                                           .MCALL  SNPBK$,SNAP$,CALL
      4 000000                            BLK:    SNPBK$  LP,0,SC.HDR!SC.OVL!SC.WRD,1,1,BLK,BLK+220
      5 000036      123      116      120 BUF:    .ASCIZ  /SNPTST/
        000041      124      123      124
        000044      000
      6                                           .EVEN
      7 000046                            START:  SNAP$
      8 000216  012700  000036'                   MOV     #BUF,R0
      9 000222                                    CALL    $CAT5
     10 000226                                    SNAP$   #SC.HDR!SC.OVL!SC.BYT,,#100
     11 000412  000004                            IOT
     12         000046'                           .END    START


SNPTST - TEST SNAP DUMP AND PMD MACRO M1010  03-SEP-76 15:57  PAGE 1-1
SYMBOL TABLE

BLK       000000R         SB.EFN= 000006         SC.BYT= 000040         SC.STK= 000010          $DSW   = ****** GX
BUF       000036R         SB.ID = 000010         SC.HDR= 000001         SC.WRD= 000020          $$$T2  = 000027
IE.ACT= ****** GX         SB.LM1= 000012         SC.LUN= 000002         START    000046R        ..SPBK   000000RG
SB.CTL= 000000            SB.PMD= 000032         SC.OVL= 000004         $CAT5  = ****** GX      ...SNP= 000032
SB.DEV= 000002            SB.UNT= 000004

. ABS.  000000      000
        000414      001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  1335 WORDS  ( 6 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  30 PAGES
ASSEMBLY TIME (ELAPSED):  00:00:14
SNPTST,SNPTST=SNPTST
```

Figure 8-3 Sample Program that Calls for Snapshot Dumps

```
                        SNAPSHOT DUMP  ID: 1

TASK: TT6                                        TIME: 5-OCT-76 15:06

PC: 000522

REGS:     R0 - 000000   R1 - 100104   R2 - 000000     R3 - 140130

          R4 - 000000   R5 - 000000   SP - 000304     PS - 170000

TASK STATUS:   MSG -CHK STP WFR REM MCR

EVENT FLAG MASK FOR <1-16> 000001

CURRENT UIC: [007,001]   DSW: 1.

PRIORITY: DEFAULT - 50.  RUNNING - 50.   I/O COUNT: 0.   TI DEVICE - TT6:

LOAD DEVICE - DB0:     LBN: 1,160034

FLOATING POINT UNIT

     STATUS - 000000

     R0 - 000000  000000  000000  000000
     R1 - 000000  000000  000000  000000
     R2 - 000000  000000  000000  000000
     R3 - 000000  000000  000000  000000
     R4 - 000000  000000  000000  000000
     R5 - 000000  000000  000000  000000

OVERLAY SEGMENTS LOADED AND RESIDENT LIBRARIES MAPPED

STARTING RELATIVE BLOCK: 000002   BASE: 000000   LENGTH: 001454


                    TASK IMAGE


      PARTITION: GEN       VIRTUAL LIMITS: 000304 - 000524

000300   001051   000001   000025   050114  ! M3    A    U L36!
000310   000000   000001   000001   000304  !       A    A  D6!
000320   000524   000000   000000   000000  ! HT                !
000330   000000   000000   000000   063014  !               PMD!
000340   131574   047123   052120   052123  !... LUK MSX MS$!
000350   000000   016746   177734   012746  !     D1N   7T CTF!
000360   001037   104377   103456   005046  ! MW U61 UYF AX8!
000370   012746   000304   012746   000336  !CTF  D6 CTF  EV!
000400   017646   000000   062766   000002  !EBV      PLV   B!
000410   000002   017666   000002   000002  !  B EB8    B   B!
000420   012746   0002507  104377   013435  !CTF  31 U61 UX/!
000430   005046   005046   005046   005046  !AX8 AX8 AX8 AX8!
000440   012746   000336   017646   000000  !CTF  EV EBV     !
000450   062766   000002   000002   017666  !PLV   B    B EB8!
000460   000002   000002   012746   003413  !  B    B CTF AEC!
000470   104377   103006   022737   177771  !U61 UQ0 FBO  8I!
000500   000046   001402   000261   000405  !  8  SJ  DQ  FU!
000510   016746   177576   012746   001051  !D1N  5F CTF  M3!
000520   104377   012700   000342   004767  !U61 CSH  EZ AW1!
```

Figure 8-4 Sample Snapshot Dump (in Word Octal and Radix-50)

```
                         SNAPSHOT DUMP  ID: 64

TASK: TT6                                    TIME: 5-OCT-76 15:06

PC: 000716

REGS:    R0 - 000345   R1 - 074400   R2 - 000120   R3 - 140130

         R4 - 000000   R5 - 000000   SP - 000304   PS - 170000

TASK STATUS:   MSG -CHK STP WFR REM MCR

EVENT FLAG MASK FOR <1-16> 000001

CURRENT UIC: [007001]   DSW: 1.

PRIORITY: DEFAULT - 50.  RUNNING - 50.   I/O COUNT: 0.   TI DEVICE - TT6:

LOAD DEVICE - DB0:     LBN: 1,160034


FLOATING POINT UNIT

     STATUS - 000000

     R0 - 000000  000000  000000  000000
     R1 - 000000  000000  000000  000000
     R2 - 000000  000000  000000  000000
     R3 - 000000  000000  000000  000000
     R4 - 000000  000000  000000  000000
     R5 - 000000  000000  000000  000000

OVERLAY SEGMENTS LOADED AND RESIDENT LIBRARIES MAPPED

STARTING RELATIVE BLOCK: 000002   BASE: 000000   LENGTH: 001454
STARTING RELATIVE BLOCK: 000004   BASE: 001454   LENGTH: 000264

                    TASK IMAGE


      PARTITION: GEN       VIRTUAL LIMITS: 000304 - 000524

000300  051 002  001 000  045 000  114 120     !)     %  LP!
000310  000 000  001 000  100 000  304 000     !      @  D !
000320  124 001  000 000  000 000  000 000     !T          !
000330  000 000  000 000  000 000  014 146     !          f!
000340  174 263  123 116  120 124  123 124     ! 3 SN PT ST!
000350  000 000  346 035  334 377  346 025     !   f  \  f !
000360  037 002  377 210  056 207  046 012     !      .  & !
000370  346 025  304 000  346 025  336 000     !f  D  f  ^ !
000400  246 037  000 000  366 145  002 000     !&     ve   !
000410  002 000  266 037  002 000  002 000     !   6       !
000420  346 025  107 005  377 210  035 207     !f  G       !
000430  046 012  046 012  046 012  046 012     !&  &  &  & !
000440  346 025  336 000  246 037  000 000     !f  ^  &    !
000450  366 145  002 000  002 000  266 037     !ve       6 !
000460  002 000  002 000  346 025  013 007     !      f    !
000470  377 210  006 206  337 045  371 377     !      _% y !
000500  046 000  002 003  261 000  005 001     !&     1    !
000510  346 035  176 377  346 025  051 002     !f     f  ) !
000520  377 210  300 025  342 000  367 011     !   @  b  w !
```

Figure 8-5 Sample Snapshot Dump (in Byte Octal and ASCII)

APPENDIX A

TASK BUILDER INPUT DATA FORMATS

An object module is the fundamental unit of input to the Task Builder. You create an object module by using any of the standard language processors (for example, MACRO-11 or FORTRAN) or by using the Task Builder itself (symbol definition file). The RSX-11M/M-PLUS librarian (LBR) gives you the capability to combine a number of object modules into a single library file.

An object module consists of variable length records of information that describe the contents of the module. These records guide the Task Builder in translating the object language into a task image. Six record (block) types are included in the object language:

- Declare global symbol directory (GSD) record (type 1)
- End of global symbol directory (GSD) record (type 2)
- Text information (TXT) record (type 3)
- Relocation directory (RLD) record (type 4)
- Internal symbol directory (ISD) record (type 5)
- End-of-module record (type 6)

The Task Builder requires at least five of these record types in each object module. The only record type that it does not require is the internal symbol directory.

The various record types are defined according to a prescribed format, as illustrated in Figure A-1. An object module must begin with a declare-GSD record and end with an end-of-module record. Additional declare-GSD records can occur anywhere in the file but must occur before an end-of-GSD record. An end-of-GSD record must appear before the end-of-module record, and at least one RLD record must appear before the first TXT record. Additional RLD and TXT records can appear anywhere in the file. The ISD records can appear anywhere in the file between the initial declare-GSD record and the end-of-module record.

Object module records are variable length and are identified by a record type code in the first byte of the record. The format of additional information in the record depends on the record type.

The following sections describe each of the six record types in greater detail. The outline of these sections is as follows

A.1 DECLARE GLOBAL SYMBOL DIRECTORY RECORD

The global symbol directory (GSD) record contains all the information required by the Task Builder to assign addresses to global symbols and to allocate the virtual address space required by a task.

GSD records are the only records processed by the Task Builder in its first pass; therefore, you can save significant time by placing all GSD records at the beginning of a module (because the Task Builder has to read less of the file).

GSD records contain nine types of entries:

- Module name (type 0)
- Control section name (type 1)
- Internal symbol name (type 2)
- Transfer address (type 3)
- Global symbol name (type 4)
- Program section name (type 5)
- Program version identification (type 6)
- Mapped array declaration (type 7)
- Completion routine name (type 10)

TASK BUILDER DATA FORMATS

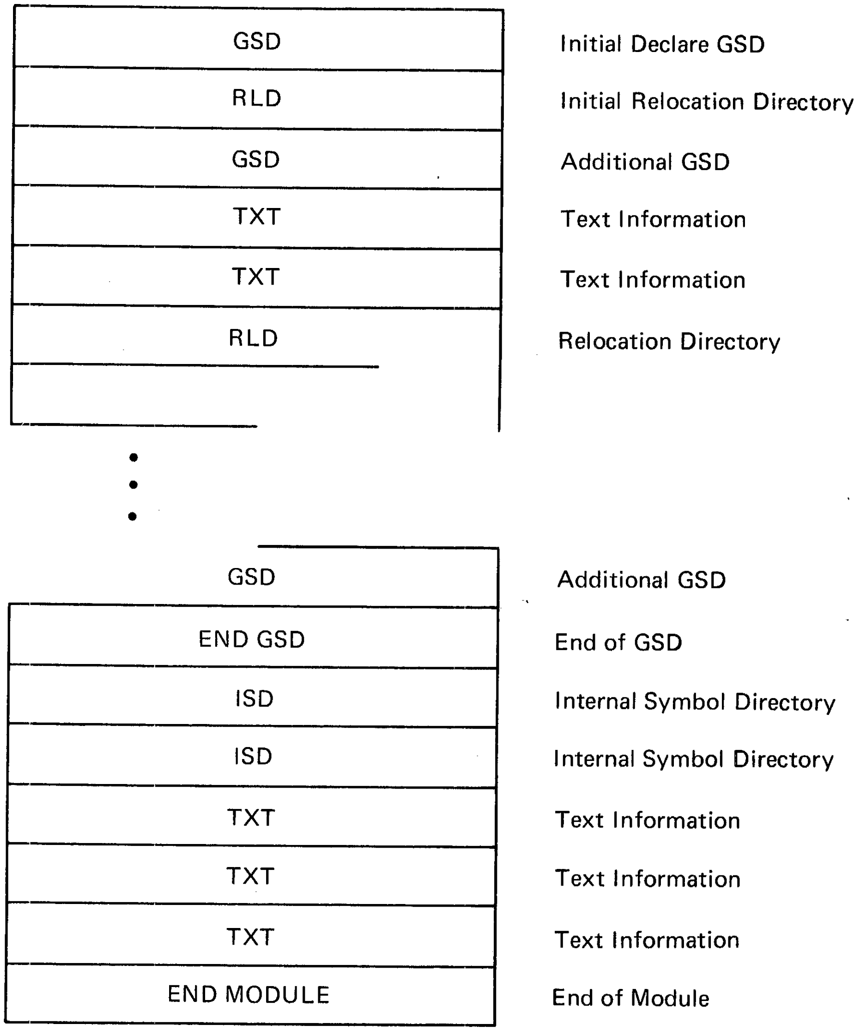


Figure A-1 General Object Module Format

Each entry type is represented by four words in the GSD record. As shown in Figure A-2, the first two words contain six Radix-50 characters, the third word contains a flag byte and the entry type identification, and the fourth word contains additional information about the entry.

<table>
<tr><td>0</td><td>RECORD TYPE = 1</td></tr>
<tr><td colspan="2">RAD50 NAME</td></tr>
<tr><td>ENTRY TYPE</td><td>FLAGS</td></tr>
<tr><td colspan="2">VALUE</td></tr>
<tr><td colspan="2">RAD50 NAME</td></tr>
<tr><td>TYPE</td><td>FLAGS</td></tr>
<tr><td colspan="2">VALUE</td></tr>
</table>

•
•
•

<table>
<tr><td colspan="2">RAD50 NAME</td></tr>
<tr><td>TYPE</td><td>FLAGS</td></tr>
<tr><td colspan="2">VALUE</td></tr>
<tr><td colspan="2">RAD50 NAME</td></tr>
<tr><td>TYPE</td><td>FLAGS</td></tr>
<tr><td colspan="2">VALUE</td></tr>
</table>

Figure A-2 Global Symbol Directory Record Format

A.1.1 Module Name (Type 0)

The module name entry declares the name of the object module. The name need not be unique with respect to other object modules (that is, modules are identified by file, not module name), but only one such declaration can occur in any given object module. Figure A-3 illustrates the module entry name format.

<table>
<tr><td colspan="2">MODULE NAME</td></tr>
<tr><td>ENTRY TYPE = 0</td><td>0</td></tr>
<tr><td colspan="2">0</td></tr>
</table>

Figure A-3 Module Name Entry Format

A.1.2 Control Section Name (Type 1)

Control sections, which include absolute sections (ASECTs), blank and named control sections (CSECTs), are replaced in RSX-11M by program sections (PSECTs). For compatibility with other systems, the Task Builder processes ASECTs and both forms of CSECTs. Section A.1.6 details the entry generated for a .PSECT directive.

ASECTs and CSECTs are defined in terms of .PSECT directives, as follows:

For a blank CSECT, a program section is defined with the following attributes:

```
.PSECT ,LCL,REL,CON,RW,I,LOW
```

For a named CSECT, the program section is defined as:

```
.PSECT name, GBL,REL,OVR,RW,I,LOW
```

For an ASECT, the program section is defined as:

```
.PSECT . ABS.,GBL,ABS,I,OVR,RW,LOW
```

The Task Builder processes ASECTs and CSECTs as program sections with the fixed attributes defined above. Figure A-4 illustrates the control section entry name format.

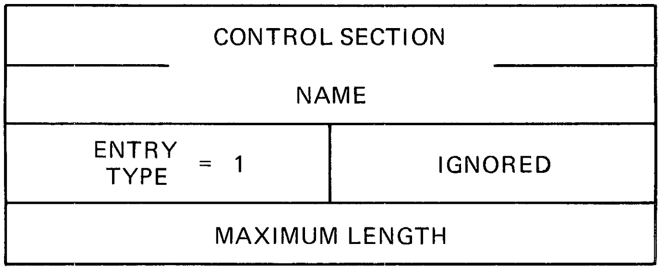


Figure A-4 Control Section Name Entry Format

A.1.3 Internal Symbol Name (Type 2)

The internal symbol name entry declares the name of an internal symbol (with respect to the module). The Task Builder does not support internal symbol tables; therefore, the detailed format of this entry is undefined. If the Task Builder encounters an internal symbol entry while reading the GSD, it ignores that entry. Figure A-5 illustrates the internal symbol name entry format.

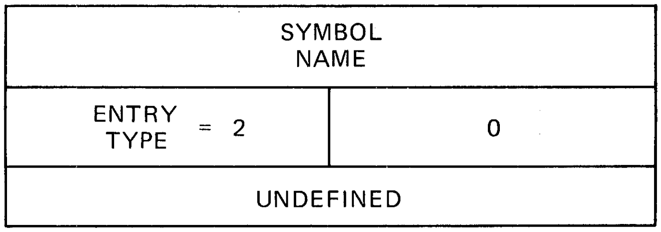


Figure A-5 Internal Symbol Name Entry Format

A.1.4 **Transfer Address (Type 3)**

The transfer address entry declares the transfer address of a module relative to a program section. The first two words of the entry define the name of the program section, and the fourth word defines the relative offset from the beginning of that program section. If a transfer address is not declared in a module, then a transfer address must not be included in the GSD, or a transfer address of 000001 relative to the default absolute program section (. ABS.) must be specified. Figure A-6 illustrates the transfer address entry format.

NOTE

> If the program section is absolute, the offset is the actual transfer address (if not 000001).

SYMBOL NAME	
ENTRY TYPE = 3	0
OFFSET	

Figure A-6 Transfer Address Entry Format

A.1.5 **Global Symbol Name (Type 4)**

The global symbol name entry declares either a global reference or a definition. Definition entries must appear after the declaration of the program section in which the global symbols are defined and before the declaration of another program section (see Section A.1.6). Global references can be used anywhere within the GSD.

As shown in Figure A-7, the first two words of the entry define the name of the global symbol. The flag byte declares the attributes of the symbol, and the fourth word defines the value of the symbol relative to the program section in which the symbol is defined.

SYMBOL NAME	
ENTRY TYPE = 4	FLAGS
VALUE	

Figure A-7 Global Symbol Name Entry Format

Table A-1 lists the bit assignments of the flag byte of the symbol declaration entry.

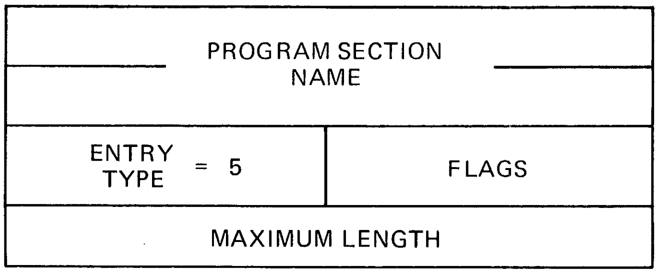
Table A-1
Symbol Declaration Flag Byte -- Bit Assignments

Bit Number and Name	Setting	Meaning
0 Weak Qualifier	0	The symbol has a strong definition and is resolved in the normal manner.
	1	The symbol has a weak definition or reference. The Task Builder ignores a weak reference (Bit 3 = 0). It also ignores a weak definition (Bit 3 = 1) unless a previous reference has been made.
1		Not used
2 Definition or Reference Type	0	Normal definition or reference.
	1	Library definition. If the symbol is defined in a resident library STB file, the base address of the library is added to the value and the symbol is converted to absolute (bit 5 is reset); otherwise, the bit is ignored.
3 Definition	0	Global symbol reference.
	1	Global symbol definition.
4		Not used.
5 Relocation	0	Absolute symbol value.
	1	Relative symbol value.
6		Not used.
7		Not used.

A.1.6 Program Section Name (Type 5)

The program section name entry declares the name of a program section and its maximum length in the module. It also uses the flag byte to declare the attributes of the program section.

GSD records must be constructed such that once a program section name has been declared, all global symbol definitions pertaining to it must appear before another program section name is declared. Global symbols are declared with symbol declaration entries. Thus, the normal format is a series of program section names each followed by optional symbol declarations. Figure A-8 illustrates the program section name entry format.



PROGRAM SECTION NAME	
ENTRY TYPE = 5	FLAGS
MAXIMUM LENGTH	

Figure A-8 Program Section Name Entry Format

Table A-2 lists the bit assignments of the flag byte of the program section name entry.

Table A-2
Program Section Name Flag Byte -- Bit Assignments

Bit Number and Name		Setting	Meaning
0	Memory Speed	0	The program section is to occupy low-speed (core) memory.
		1	The program section is to occupy high-speed (MOS/bipolar) memory.
1	Library program section	0	Normal program section.
		1	The program section is relocatable and refers to a resident library or common block.
2	Allocation	0	Program section references are to be concatenated with other references to the same program section to form the total memory allocated to the section.
		1	Program section references are to be overlaid. The total memory allocated to the program section is the largest request made by individual references to the same program section.

(continued on next page)

Table A-2 (Cont.)
Program Section Name Flag Byte -- Bit Assignments

Bit Number and Name	Setting	Meaning
3		Not used; reserved for future DIGITAL use.
4 Access	0	Program section has read/write access.
	1	Program section has read-only access.
5 Relocation	0	The program section is absolute and requires no relocation.
	1	The program section is relocatable and references to the control section must have a relocation bias added before they become absolute.
6 Scope	0	The scope of the program section is local. References to the same program section are collected only within the segment in which the program section is defined.
	1	The scope of the program section is global. References to the program section are collected across segment boundaries. The segment in which a global program section is allocated storage is determined either by the first module that defines the program section on a path, or by direct placement of a program section in a segment using the overlay description language .PSECT directive.
7 Type	0	The program section contains instruction (I) references.
	1	The program section contains data (D) references.

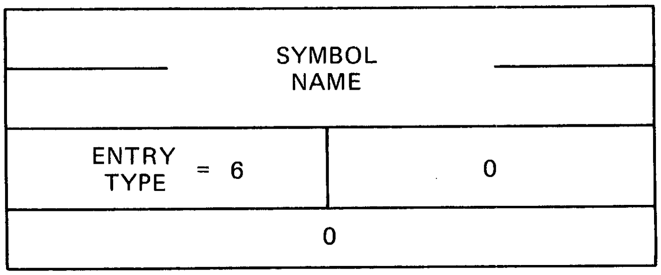
NOTE

The length of all absolute sections is zero.

A.1.7 **Program Version Identification (Type 6)**

The program version identification entry declares the version of the module. The Task Builder saves the version identification of the first module that defines a nonblank version. It then includes this identification on the memory allocation map and writes the identification in the label block of the task image file.

The first two words of the entry contain the version identification. The flag byte and fourth words are not used and contain no meaningful information. Figure A-9 illustrates the program version identification entry format.



SYMBOL NAME	
ENTRY TYPE = 6	0
0	

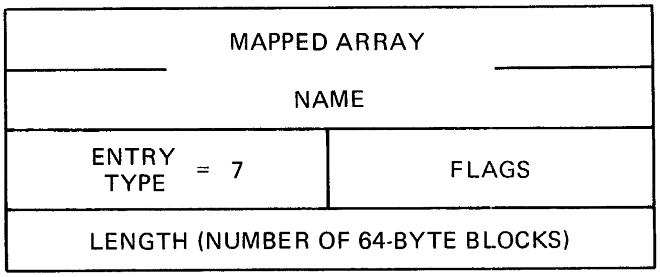
Figure A-9 Program Version Identification Entry Format

A.1.8 **Mapped Array Declaration (Type 7)**

The mapped array declaration entry allocates space within the mapped array area of task memory. The array name is added to the list of task program section names and may be referred to by subsequent RLD records. The length (in units of 64-byte blocks) is added to the task's mapped array allocation. The total memory allocated to each mapped array is rounded up to the nearest 512-byte boundary. The contents of the flag byte are reserved and assumed to be zero.

One additional window block is allocated whenever a mapped array is declared.

Figure A-10 illustrates the mapped array declaration entry format.



MAPPED ARRAY NAME	
ENTRY TYPE = 7	FLAGS
LENGTH (NUMBER OF 64-BYTE BLOCKS)	

Figure A-10 Mapped Array Declaration Entry Format

A.1.9 Completion Routine Definition (Type 10)

The completion routine definition declares the entry point for the completion routine of a supervisor-mode library. This data structure is created by the Task Builder and appears only in Symbol Definition files of supervisor-mode libraries.

As shown in Figure A-11, the first two words of the entry define the name of the entry point. The third word contains the entry type byte and the flag byte. The flag byte contains no meaningful information. The fourth word contains the symbol value.

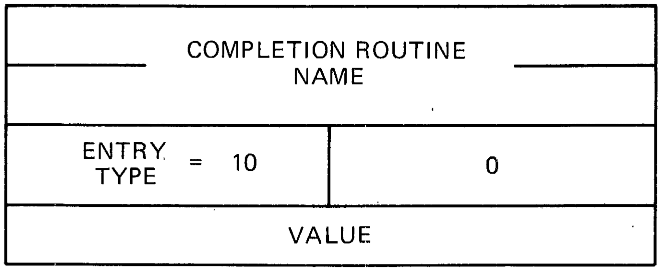


Figure A-11 Completion Routine Entry Format

A.2 END OF GLOBAL SYMBOL DIRECTORY RECORD

The end of global symbol directory (end-of-GSD) record declares that no other GSD records are contained further on in the module. There must be exactly one end-of-GSD record in every object module. As shown in Figure A-12, this record is one word long.

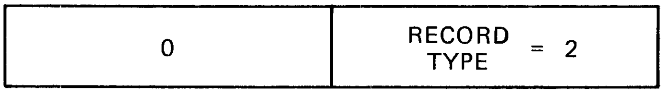


Figure A-12 End of Global Symbol Directory Record Format

A.3 TEXT INFORMATION RECORD

The text information (TXT) record contains a byte string of information that is to be written directly into the task image file. The record consists of a load address followed by the byte string.

TXT records can contain words and/or bytes of information whose final contents have not yet been determined. This information will be bound by a relocation directory record that immediately follows the text record (see Section A.4). If the TXT record needs no modification, then no relocation directory record is needed. Thus, multiple TXT records can appear in sequence before a relocation directory record.

The load address of the TXT record is specified as an offset from the current program section base. At least one relocation directory record must precede the first TXT record. This directory must declare the current program section.

The Task Builder writes a text record directly into the task image file and computes the value of the load address minus 4. This value is stored in anticipation of a subsequent relocation directory that modifies words and/or bytes contained in the TXT record. When added to a relocation directory displacement byte, this value yields the address of the word and/or byte to be modified in the task image.

Figure A-13 illustrates the TXT record format.

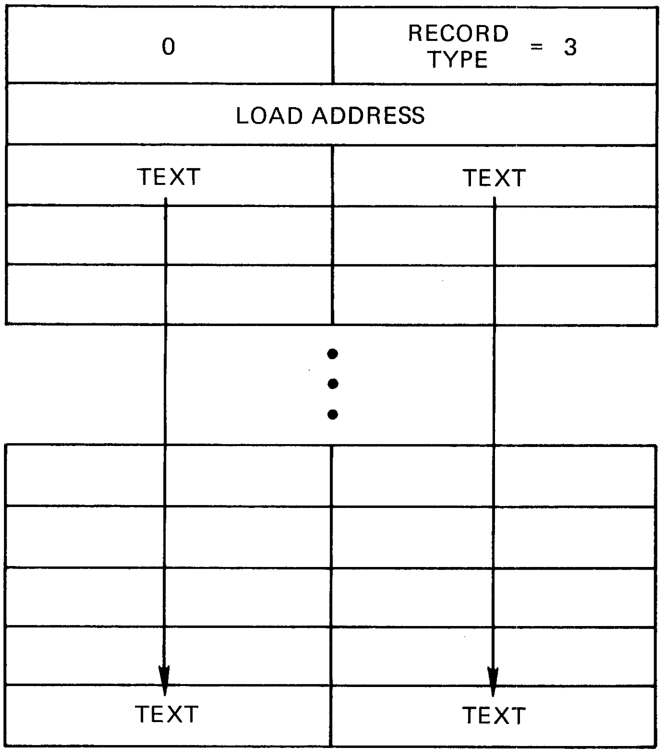


Figure A-13 Text Information Record Format

A.4 RELOCATION DIRECTORY RECORD

The relocation directory (RLD) record contains the information necessary to relocate and link the preceding TXT record. Every module must have at least one RLD record that precedes the first TXT record. The first RLD record does not modify a preceding TXT record; rather it defines the current program section and location. RLD records contain 15 types of entry, classified as relocation or location modification entries:

- Internal relocation (type 1)
- Global relocation (type 2)
- Internal displaced relocation (type 3)
- Global displaced relocation (type 4)
- Global additive relocation (type 5)

- Global additive displaced relocation (type 6)
- Location counter definition (type 7)
- Location counter modification (type 10)
- Program limits (type 11)
- Program section relocation (type 12)
- Program section displaced relocation (type 14)
- Program section additive relocation (type 15)
- Program section additive displaced relocation (type 16)
- Complex relocation (type 17)
- Resident library relocation (type 20)

Each type of entry is represented by a command byte which specifies the type of entry and the word/byte modification, followed by a displacement byte, and then by the information required for the particular type of entry. The displacement byte, when added to the value calculated from the load address of the preceding TXT record (see Section A.3), yields the virtual address in the image that is to be modified.

Table A-3 lists the bit assignments of the command byte of each RLD entry.

Table A-3
Relocation Directory Command Byte --
Bit Assignments

Bit Number and Name	Setting	Meaning
0-6 Entry Type		Potentially, 128 command types can be specified; currently, 15 are implemented.
7 Modification	0	The command modifies an entire word.
	1	The command modifies only 1 byte. The Task Builder checks for truncation errors in byte modification commands. If truncation is detected (that is, if the modification value is greater than 255), an error occurs.

Figure A-14 illustrates the RLD record format.

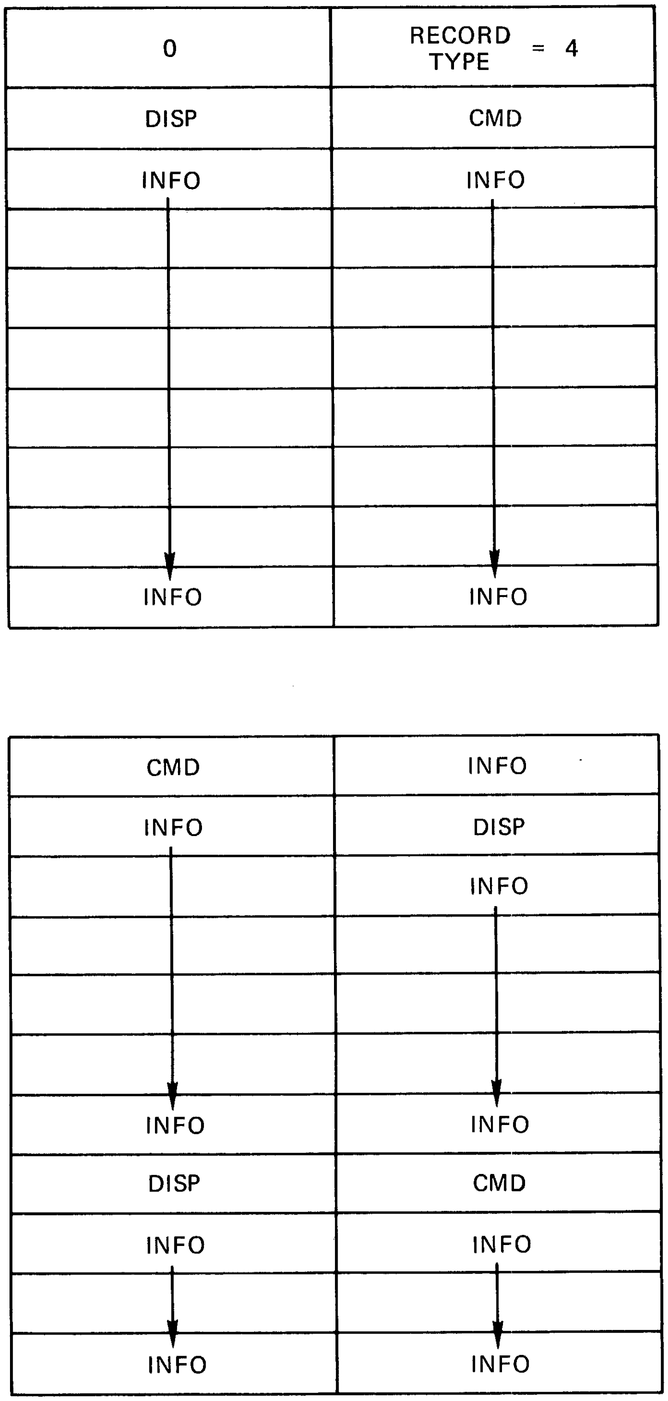


Figure A-14 Relocation Directory Record Format

A.4.1 Internal Relocation (Type 1)

The internal relocation entry relocates a direct pointer to an address within a module. The Task Builder adds the current program section base address to a specified constant and writes the result into the task image file at the calculated address (that is, a displacement byte is added to the value calculated from the load address of the preceding text block).

For example:

```
A:      MOV      #A,R0

           or

        .WORD    A
```

Figure A-15 illustrates the internal relocation entry format.

<table>
<tr><td>DISP</td><td>B</td><td>ENTRY TYPE = 1</td></tr>
<tr><td colspan="3">CONSTANT</td></tr>
</table>

Figure A-15 Internal Relocation Entry Format

A.4.2 Global Relocation (Type 2)

The global relocation entry relocates a direct pointer to a global symbol. The Task Builder obtains the definition of the global symbol and writes the result into the task image file at the calculated address.

For example:

```
        MOV      #GLOBAL,R0

           or

        .WORD    GLOBAL
```

Figure A-16 illustrates the global relocation entry format.

<table>
<tr><td>DISP</td><td>B</td><td>ENTRY TYPE = 2</td></tr>
<tr><td colspan="3">SYMBOL
NAME</td></tr>
</table>

Figure A-16 Global Relocation Entry Format

A.4.3 Internal Displaced Relocation (Type 3)

The internal displaced relocation entry relocates a relative reference to an absolute address from within a relocatable control section. The Task Builder subtracts the address plus 2 that the relocated value is to be written into from the specified constant and writes the result into the task image file at the calculated address.

For example:

```
        CLR      177550

           or

        MOV      177550,R0
```

Figure A-17 illustrates the internal displaced relocation entry format.

DISP	B	ENTRY TYPE = 3
CONSTANT		

Figure A-17 Internal Displaced Relocation Entry Format

A.4.4 Global Displaced Relocation (Type 4)

The global displaced relocation entry relocates a relative reference to a global symbol. The Task Builder obtains the definition of the global symbol, subtracts the address plus 2 that the relocated value is to be written into from the definition value, and writes the result into the task image file at the calculated address.

For example:

```
        CLR      GLOBAL

           or

        MOV      GLOBAL,R0
```

Figure A-18 illustrates the global displaced relocation entry format.

DISP	B	ENTRY TYPE = 4
SYMBOL NAME		

Figure A-18 Global Displaced Relocation Entry Format

A.4.5 Global Additive Relocation (Type 5)

The global additive relocation entry relocates a direct pointer to a global symbol with an additive constant. The Task Builder obtains the definition of the global symbol, adds the specified constant to the definition value, and writes the result into the task image file at the calculated address.

For example:

```
        MOV       #GLOBAL+2,R0

           or

        .WORD     GLOBAL-4
```

Figure A-19 illustrates the global additive relocation entry format.

<table>
<tr><td>DISP</td><td>B</td><td>ENTRY TYPE = 5</td></tr>
<tr><td colspan="3">SYMBOL NAME</td></tr>
<tr><td colspan="3">CONSTANT</td></tr>
</table>

Figure A-19 Global Additive Relocation Entry Format

A.4.6 Global Additive Displaced Relocation (Type 6)

The global additive displaced relocation entry relocates a relative reference to a global symbol with an additive constant. The Task Builder obtains the definition of the global symbol, adds the specified constant to the definition value, subtracts the address plus 2 that the relocated value is to be written into from the resultant additive value, and writes the result into the task image file at the calculated address.

For example:

```
        CLR       GLOBAL+2

           or

        MOV       GLOBAL-5,R0
```

Figure A-20 illustrates the global additive displaced relocation entry format.

<table>
<tr><td>DISP</td><td>B</td><td>ENTRY TYPE = 6</td></tr>
<tr><td colspan="3">SYMBOL NAME</td></tr>
<tr><td colspan="3">CONSTANT</td></tr>
</table>

Figure A-20 Global Additive Displaced Relocation Entry Format

A.4.7 Location Counter Definition (Type 7)

The location counter definition entry declares a current program section and location counter value. The Task Builder stores the control base as the current control section, adds the current control section base to the specified constant, and stores the result as the current location counter value.

Figure A-21 illustrates the location counter definition entry format.

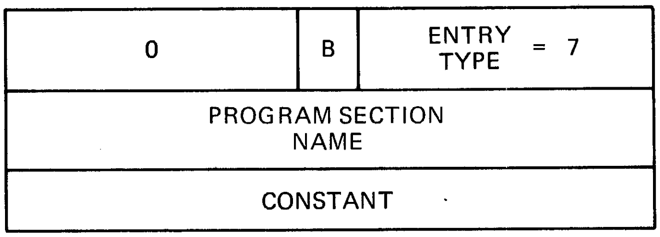


Figure A-21 Location Counter Definition Entry Format

A.4.8 Location Counter Modification (Type 10)

The location counter modification entry modifies the current location counter. The Task Builder adds the current program section base to the specified constant and stores the result as the current location counter.

For example:

```
.=.+N

   or

.BLKB    N
```

Figure A-22 illustrates the location counter modification entry format.

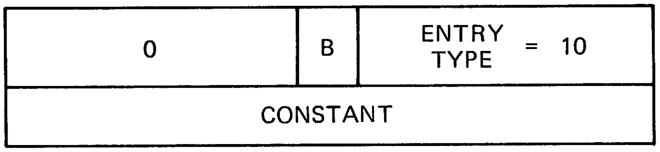


Figure A-22 Location Counter Modification Entry Format

A.4.9 Program Limits (Type 11)

The program limits entry is generated by the .LIMIT assembler directive. The Task Builder obtains the first address above the header (normally the beginning of the stack) and the highest address allocated to the task. It then writes these two addresses into the task image file at the calculated address and at the calculated address plus 2, respectively.

For example:

```
        .LIMIT
```

Figure A-23 illustrates the program limits entry format.

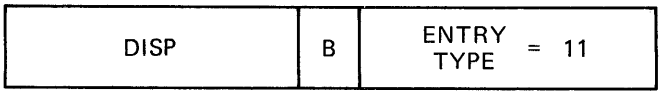


Figure A-23 Program Limits Entry Format

A.4.10 Program Section Relocation (Type 12)

The program section relocation entry relocates a direct pointer to the beginning address of another program section (other than the program section in which the reference is made) within a module. The Task Builder obtains the current base address of the specified program section and writes it into the task image file at the calculated address.

For example:

```
        .PSECT   A
B:
        .
        .
        .
        .PSECT   C
        MOV      #B,R0

           or

        .WORD    B
```

Figure A-24 illustrates the program section relocation entry format.

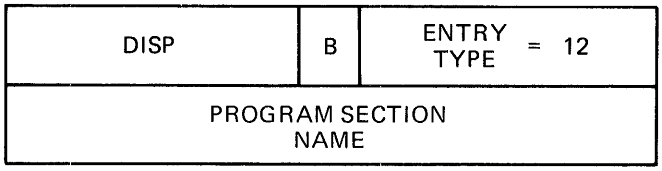


Figure A-24 Program Section Relocation Entry Format

A.4.11 Program Section Displaced Relocation (Type 14)

The program section displaced relocation entry relocates a relative reference to the beginning address of another program section within a module. The Task Builder obtains the current base address of the specified program section, subtracts the address plus 2 that the relocated value is to be written into from the base value, and writes the result into the task image file at the calculated address.

For example:

```
        .PSECT  A
B:
        .
        .
        .
        .PSECT  C
        MOV     B,R0
```

Figure A-25 illustrates the program section displaced relocation entry format.

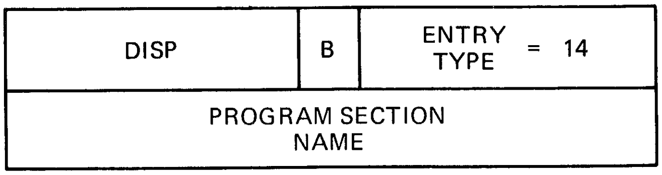


Figure A-25 Program Section Displaced Relocation Entry Format

A.4.12 Program Section Additive Relocation (Type 15)

The program section additive relocation entry relocates a direct pointer to an address in another program section within a module. The Task Builder obtains the current base address of the specified program section, adds this address to the specified constant, and writes the result into the task image file at the calculated address.

For example:

```
        .PSECT  A
B:
        .
        .
        .
C:
        .
        .
        .
        .PSECT  D
        MOV     #B+10,R0
        MOV     #C,R0

          or

        .WORD   B+10
        .WORD   C
```

Figure A-26 illustrates the program section additive relocation entry format.

<table>
<tr><td>DISP</td><td>B</td><td>ENTRY TYPE = 15</td></tr>
<tr><td colspan="3">PROGRAM SECTION NAME</td></tr>
<tr><td colspan="3">CONSTANT</td></tr>
</table>

Figure A-26 Program Section Additive Relocation Entry Format

A.4.13 Program Section Additive Displaced Relocation (Type 16)

The program section additive displaced relocation entry relocates a relative reference to an address in another program section within a module. The Task Builder obtains the current base address of the specified program section, adds this address to the specified constant, subtracts the address plus 2 that the relocated value is to be written into from the resultant additive value, and writes the result into the task image file at the calculated address.

For example:

```
        .PSECT  A
B:
        .
        .
        .
C:
        .
        .
        .
        .PSECT  D
        MOV     B+10,R0
        MOV     C,R0
```

Figure A-27 illustrates the program section additive displaced relocation entry format.

<table>
<tr><td>DISP</td><td>B</td><td>ENTRY TYPE = 16</td></tr>
<tr><td colspan="3">PROGRAM SECTION NAME</td></tr>
<tr><td colspan="3">CONSTANT</td></tr>
</table>

Figure A-27 Program Section Additive Displaced Relocation Entry Format

A.4.14 Complex Relocation (Type 17)

The complex relocation entry resolves a complex relocation expression. Such an expression is one in which any of the MACRO-11 binary or unary operations are permitted with any type of argument, regardless of whether the argument is an unresolved global symbol, is relocatable to any program section base, is absolute, or is a complex relocatable subexpression.

The RLD command word is followed by a string of numerically-specified operation codes and arguments. The operation codes each occupy one byte. The entire RLD command must fit in a single record. The following 15 operation codes are defined:

- No operation -- byte 0
- Addition (+) -- byte 1
- Subtraction (-) -- byte 2
- Multiplication (*) -- byte 3
- Division (/) -- byte 4
- Logical AND (&) -- byte 5
- Logical inclusive OR (!) -- byte 6
- Negation (-) -- byte 10
- Complement (^C) -- byte 11
- Store result (command termination) -- byte 12
- Store result with displaced relocation (command termination) -- byte 13
- Fetch global symbol -- byte 16. It is followed by 4 bytes containing the symbol name in Radix-50 representation.
- Fetch relocatable value -- byte 17. It is followed by 1 byte containing the sector number, and 2 bytes containing the offset within the sector.
- Fetch constant -- byte 20. It is followed by 2 bytes containing the constant.
- Fetch resident library base address -- byte 21. If the file is a resident library STB file, the library base address is obtained; otherwise, the base address of the task image is fetched.

The STORE commands indicate that the value is to be written into the task image file at the calculated address.

All operands are evaluated as 16-bit signed quantities using 2's complement arithmetic. The results are equivalent to expressions that the assembler evaluates internally. The following rules should be noted.

1. An attempt to divide by zero yields a zero result. The Task Builder issues a nonfatal diagnostic error message.

2. All results are truncated from the left to fit into 16 bits. No diagnostic error message is issued if the number is too large. If the result modifies a byte, the Task Builder checks for truncation errors as described in Section A.4.

3. All operations are performed on relocated (additive) or absolute 16-bit quantities. PC displacement is applied to the result only.

For example:

```
        .PSECT  ALPHA
A:
        .
        .
        .
        .PSECT  BETA
B:
        .
        .
        .
        MOV     #A+B-<G1/G2&^C<177120!G3>>,R1
```

Figure A-28 illustrates the complex relocation entry format.

DISP	B	ENTRY TYPE = 17
COMPLEX STRING		
12		

Figure A-28 Complex Relocation Entry Format

A.4.15 Resident Library Relocation (Type 20)

The library relocation entry relocates a direct pointer to an address within a resident library.

If the current file is a resident library symbol definition file (STB), the Task Builder obtains the base address of the library; adds this address to the specified constant; and writes the result into the task image file at the calculated address. If the file is not associated with a resident library, the Task Builder uses the task base address.

Figure A-29 illustrates the library relocation entry format.

DISP	B	ENTRY TYPE = 20
CONSTANT		

Figure A-29 Resident Library Relocation Entry Format

A.5 INTERNAL SYMBOL DIRECTORY RECORD

The internal symbol directory (ISD) record declares definitions of symbols that are local to a module. The Task Builder does not support this feature; therefore, the detailed record format is undefined. If the Task Builder encounters this type of record, it ignores that record.

Figure A-30 illustrates the ISD record format.

0	RECORD TYPE = 5
UNDEFINED	

Figure A-30 Internal Symbol Directory Record Format

A.6 END OF MODULE RECORD

The end-of-module record declares the end of an object module. There must be exactly one end-of-module record in every object module. As shown in Figure A-31, this record is one word long.

0	RECORD TYPE = 6

Figure A-31 End-of-Module Record Format

APPENDIX B

DETAILED TASK IMAGE FILE STRUCTURE

Figure B-1 illustrates how the Task Builder records a task image on disk. As noted in the following sections, many parts of the task disk image shown in this figure are optional and may not be recorded for every task image.

The following sections, which provide detailed information on the task image file structure, are organized as follows:

B.1 Label Block Group
B.2 Checkpoint Area
B.3 Header
    B.3.1 Low-memory Context
    B.3.2 Logical Unit Table Entry
B.4 Task Image
    B.4.1 Autoload Vectors
    B.4.2 Segment Descriptor
    B.4.3 Window Descriptor
    B.4.4 Region Descriptor
    B.4.5 Supervisor-Mode Vector

B.1 LABEL BLOCK GROUP

The label block group precedes the task on the disk and contains data that is needed by the system to install and load a task but need not reside in memory during task execution. This group consists of three parts:

- Task and resident library data (label block 0)
- Table of LUN assignments (label blocks 1 and 2)
- The segment load list (label block 3)

Table B-1 describes the task and resident library data. Figure B-2 illustrates how the Task Builder organizes this data in label block 0. The INSTALL processor verifies the task and resident library data when entering the tasks into the System Task Directory (STD) file. You can obtain the offsets shown in Figure B-2 by calling the LBLDF$ macro that resides in macro library LB:[1,1] EXEMC.MLB.

DETAILED TASK IMAGE FILE STRUCTURE

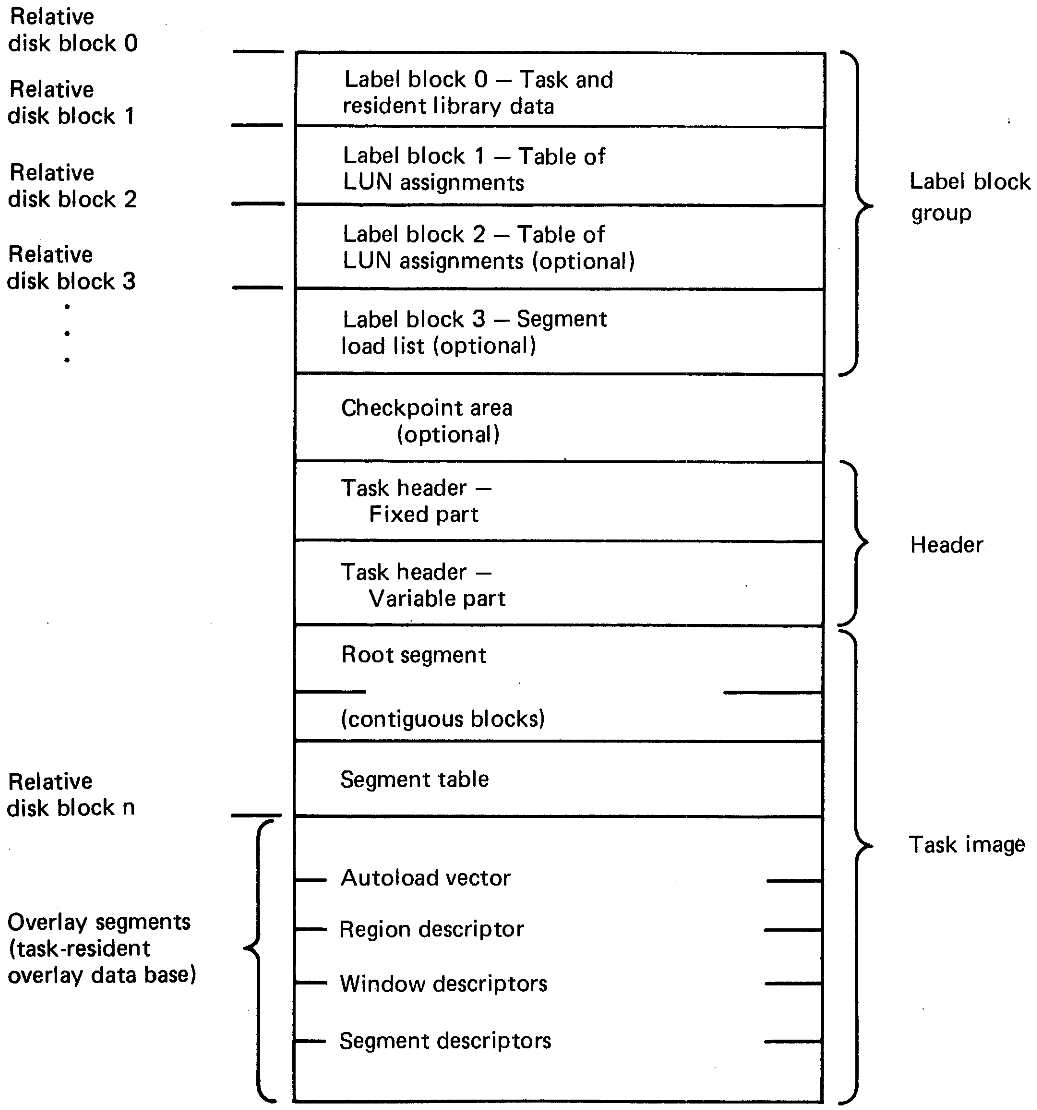

Relative disk block 0
Relative disk block 1
Relative disk block 2
Relative disk block 3
Label block 0 — Task and resident library data
Label block 1 — Table of LUN assignments
Label block 2 — Table of LUN assignments (optional)
Label block 3 — Segment load list (optional)
Label block group
Checkpoint area (optional)
Task header — Fixed part
Task header — Variable part
Header
Root segment
(contiguous blocks)
Segment table
Relative disk block n
Task image
Autoload vector
Region descriptor
Window descriptors
Segment descriptors
Overlay segments (task-resident overlay data base)

Figure B-1 Task Image on Disk

Table B-1
Task and Resident Library Data

Parameter	Definition
L$BTSK	Task name consisting of two words in Radix-50 format. This parameter is set by the TASK keyword.
L$BPAR	Partition name consisting of two words in Radix-50 format. This parameter is set by the PAR keyword.
L$BSA	Starting address of task. Marks the lowest task virtual address. This parameter is set by the PAR keyword.
L$BHGV	Highest virtual address mapped by address window 0.
L$BMXV	Highest task virtual address. When the task does not have memory-resident overlays, the value is set to L$BHGV.
L$BLDZ	Task load size in units of 64-byte blocks. This value represents the size of the root segment.
L$BMXZ	Task maximum size in units of 64-byte blocks. This value represents the size of the root segment plus any additional physical memory needed to contain task overlays.
L$BOFF	Task offset into partition in units of 64-byte blocks. This value represents the size of the mapped array area, which precedes the task's code and data in the partition.
L$BWND	Number of task window blocks less library window blocks -- low byte
L$BSYS	System ID -- high byte
L$BWND	Number of task windows (excluding resident libraries).
L$BSEG	Size of overlay segment descriptors (in bytes).
L$BFLG	Task flags word. The following flags are defined:
	Bit **Flag** **Meaning when Bit = 1**
	15 TS$PIC Task contains position-independent code (PIC)
	14 TS$NHD Task has no header
	13 TS$ACP Task is ancillary control processor
	12 TS$PMD Task generates post-mortem dump
	11 TS$SLV Task can be slaved

(continued on next page)

Table B-1 (Cont.)
Task and Resident Library Data

<table>
<tr><th>Parameter</th><th colspan="3">Definition</th></tr>
<tr><td>L$BFLG (Cont.)</td><td><b>Bit</b></td><td><b>Flag</b></td><td><b>Meaning when Bit = 1</b></td></tr>
<tr><td></td><td>10</td><td>TS$NSD</td><td>No SEND can be directed to task</td></tr>
<tr><td></td><td>9</td><td></td><td>(not used)</td></tr>
<tr><td></td><td>8</td><td>TS$PRV</td><td>Task is privileged</td></tr>
<tr><td></td><td>7</td><td>TS$CMP</td><td>Task is built in compatibility mode</td></tr>
<tr><td></td><td>6</td><td>TS$CHK</td><td>Task is not checkpointable</td></tr>
<tr><td></td><td>5</td><td>TS$RES</td><td>Task has memory-resident overlays</td></tr>
<tr><td></td><td>4</td><td>TS$SUP</td><td>Image linked as supervisor-mode library (RSX-11M-PLUS systems only)</td></tr>
<tr><td>L$BDAT</td><td colspan="3">Three words containing the task creation date as 2-digit integer values as follows:
Year since 1900
Month of year
Day of month</td></tr>
<tr><td>L$BLIB</td><td colspan="3">Resident library entries.</td></tr>
<tr><td>L$BPRI</td><td colspan="3">Task priority set by the PRI keyword.</td></tr>
<tr><td>L$BXFR</td><td colspan="3">Task transfer address. Used to initiate a bootable core image, for example, the resident executive.</td></tr>
<tr><td>L$BEXT</td><td colspan="3">Task extension size in units of 32-word blocks. This parameter is set by the EXTTSK keyword.</td></tr>
<tr><td>L$BSGL</td><td colspan="3">Relative block number of segment load list. Set to 0 if no list is allocated.</td></tr>
<tr><td>L$BHRB</td><td colspan="3">Relative block number of header.</td></tr>
<tr><td>L$BBLK</td><td colspan="3">Number of blocks in label block group.</td></tr>
<tr><td>L$BLUN</td><td colspan="3">Number of logical units.</td></tr>
<tr><td>L$BROB</td><td colspan="3">Relative block number of R/O image.</td></tr>
<tr><td>L$BROL</td><td colspan="3">R/O load size in 32-word blocks.</td></tr>
</table>

Label	Offset		
L$BTSK	0	Task	
	2	Name	
L$BPAR	4	Task	
	6	Partition	
L$BSA	10	Base address of task	
L$BHGV	12	Highest window 0 virtual address	
L$BMXV	14	Highest virtual address in task	
L$BLDZ	16	Load size in 64-byte blocks	
L$BMXZ	20	Maximum size in 64-byte blocks	
L$BOFF	22	Task offset into partition	
L$BWND/L$BSYS	24	System I.D. Number of window blocks*	
L$BSEG	26	Size of overlay segment descriptors	
L$BFLG	30	Task flag word	
L$BDAT	32	Task creation date — Year	
	34	— Month	
	36	— Day	
L$BLIB	40	Library/common	
	42	Name	R$LNAM
	44	Base address of library	R$LSA
	46	Highest address in first library window	R$LHGV
	50	Highest address in library	R$LMXV
	52	Library load size (64-byte blocks)	R$LLDZ
	54	Library maximum size (64-byte blocks)	R$LMXZ
	56	Library offset into region	R$LOFF
	60	Number of library window blocks	R$LWND
	62	Size of library segment descriptors	R$LSEG
	64	Library flag word	R$LFLG
	66	Library creation date — Year	R$LDAT
	70	— Month	
	72	— Day	
	⋮	⋮	
	344	0	
L$BPRI	346	Task priority	
L$BXFR	350	Task transfer address	
L$BEXT	352	Task extension (64-byte blocks)	
L$BSGL	354	Block number of segment load list	
L$BHRB	356	Block number of header	
L$BBLK	360	Number of blocks in label	
L$BLUN	362	Number of logical units	
L$BROB	364	Relative Block of R-O Image	
L$BROL	366	R-O Load Size	
		⋮	
		0	

Library Request (maximum of 7 14-word entries in RSX-11M systems and maximum of 15 14-word entries in RSX-11M+ systems)

*Less library window blocks.

Figure B-2 Label Block 0 -- Task and Resident Library Data

Table B-2 describes the contents of the resident shared region name block. The Task Builder constructs this block by referring to the disk image of the resident shared region. The format is identical to words 3 through 16 of the label group block.

Table B-2
Resident Library/Common Name Block Data

Parameter	Definition
R$LNAM	Shared region name consisting of 2 words in Radix-50 format.
R$LSA	Base virtual address of library or common.
R$LHGV	Highest address mapped by first library window.
R$LMXV	Highest virtual address in library or common.
R$LLDZ	Shared region load size in 64-byte blocks.
R$LMXZ	Library maximum size in 64-byte blocks. This value represents the size of the root segment plus the sum of all memory-resident overlays.
R$LOFF	Size of mapped array space allocated by resident library. This value is added to the mapped array area of the task.
R$LWND	Number of window blocks required by library.
R$LSEG	Size of library overlay segment descriptors in bytes.
R$LFLG	Library flags word. The following flags are defined: **Bit** **Meaning** 15 LD$ACC -- access intent (1=read/write, 0=read-only) 3 LD$SUP -- supervisor-mode library (1=yes) 2 LD$REL -- position-independent code (PIC) flag (1=PIC)
R$LDAT	Three words containing the shared region creation date in 2-digit integer values as follows: Year since 1900 Month of year Day of month

The table of LUN assignments, illustrated in Figure B-3, contains the name and logical unit number of each device assigned. Label block 2 (the second block of LUN assignments) is allocated only if the number of LUNs exceeds 128.

The Task Builder creates the segment load list if the image is a resident library that contains memory-resident overlays. Figure B-4 illustrates the segment load list. Each entry in the list gives the length, in bytes, of a memory-resident overlay segment.

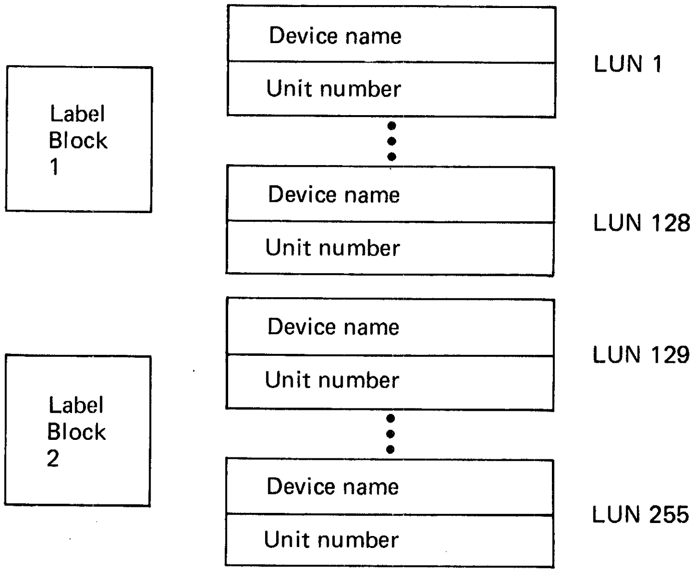


Figure B-3 Label Blocks 1 and 2 -- Table of LUN Assignments

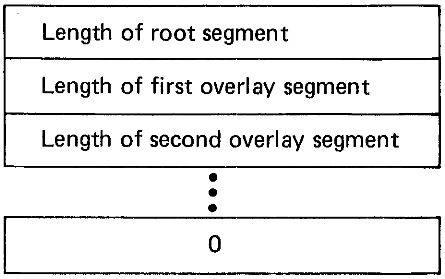

Length of root segment
Length of first overlay segment
Length of second overlay segment
0

Figure B-4 Label Block 3 -- Segment Load List

B.2 CHECKPOINT AREA

The checkpoint area is created by the AL switch (refer to Chapter 6). The checkpoint area will be as large as the task image plus any areas created by the EXTTSK, PAR, or VSECT options.

B.3 HEADER

As shown in Figure B-1, the task header starts on a block boundary and is immediately followed by the task image. The header is read into memory with the task image.

The header is divided into two parts: a fixed part as shown in Figure B-5 and a variable part as shown in Figure B-6. The offsets for the fixed part are defined by macro HDRDF$ residing in LB:[1,1]EXEMC.MLB.

The variable part of the header contains window blocks that describe the following:

- The task's virtual-to-physical mapping
- Logical unit data
- Task context

Although the header is fully accessible to the task, you should consider only the information in the low-memory context (H.DSW through H.VEXT) in the fixed part of the header to be accurate. In a mapped system, the Executive copies the header of an active task to protected memory. Subsequent Executive updates to the header are made to this copy, not to the header copy within the running task.

The following sections provide more detail on the low-memory context and on Logical Unit Table entries (the Logical Unit Table is part of the variable part of the header; see Figure B-6).

NOTE

> To save the identification, you should move the initial value set by the Task Builder to local storage. When the program is fixed in memory and being restarted without being reloaded, you must test the reserved program words for their initial values to determine whether the contents of R3 and R4 should be saved.
>
> The contents of R0, R1, and R2 are only set when a debugging aid is included in the task image.

B.3.1 **Low-Memory Context**

The low-memory context for a task consists of the Directive Status Word and the impure area vectors. The Task Builder recognizes the following global names:

Name	Meaning
.FSRPT	File Control Services work area and buffer pool vector
$OTSV	FORTRAN OTS work area vector
N.OVPT	Overlay run-time system work area vector
$VEXT	Vector extension area pointer

The only proper reference to these pointers is by symbolic name. The pointers are read-only. If you write into them, the result will be lost on the next context switch.

The impure area pointers contain the addresses of the storage used by the reentrant library routines listed above.

The address contained in the vector extension pointer locates an area of memory that can contain additional impure area pointers.

Label	Offset			
H.CSP	0	Current Stack Pointer (R6)		
H.HDLN	2	Header length		
H.EFLM	4	Event flag mask		
	6	Event flag address		
H.CUIC	10	Current UIC		
H.DUIC	12	Default UIC		
H.IPS	14	Initial PS		
H.IPC	16	Initial PC (R7)		
H.ISP	20	Initial Stack Pointer (R6)		
H.ODVA	22	ODT SST vector address		
H.ODVL	24	ODT SST vector length		
H.TKVA	26	Task SST vector address		
H.TKVL	30	Task SST vector length		
H.PFVA	32	Power fail AST control block		
H.FPVA	34	Floating-point AST control block		
H.RCVA	36	Receive AST control block		
H.EFSV	40	Address of event flag context		
H.FPSA	42	Address of floating-point context		
H.WND	44	Pointer to number of window blocks		
H.DSW	46	Directive Status Word		Low-Core Context
H.FCS	50	Address of FCS impure storage		
H.FORT	52	Address of FORTRAN impure storage		
H.OVLY	54	Address of overlay impure storage		
H.VEXT	56	Address of impure vectors		
H.SPRI/H.NML	60	Mailbox LUN	Swapping priority	
H.RRVA	62	Receive by reference AST control block		
	64	Reserved		
	66	Reserved		
	70	Reserved		
H.GARD	72	Header guard word pointer		
H.NLUN	74	Number of LUNs		

Figure B-5 Task Header, Fixed Part

H.LUN	LUN Table (2 words per LUN)

⋮

Number of window blocks		Offsets
Partition control block address		W.BPCB
Low virtual address limit		W.BLVR
High virtual address limit		W.BHVR
Address of attachment descriptor		W.BATT
Window size (in 32-word blocks)		W.BSIZ
Offset into partition (in 32-word blocks)		W.BOFF
Number of PDRs to Map	First PDR Address	W.BNPD/W.BFPD
Contents of last PDR		W.BLPD

⋮

Current PS	
Current PC	Initial Values
Current R5	Relative block number of header
Current R4	Ident. word #2
Current R3	Ident. word #1
Current R2	Task name word #2
Current R1	Task name word #1
Current R0	Program transfer address
Header guard word	

Figure B-6 Task Header, Variable Part

Figure B-7 illustrates the format of the vector extension area. Each location within this region contains the address of an impure storage area that is referred to by subroutines; these subroutines must be reentrant. Addresses below $VEXTA, referred to by negative offsets, are reserved for DIGITAL applications. Addresses above $VEXTA, referred to by positive offsets, are allocated for user applications.

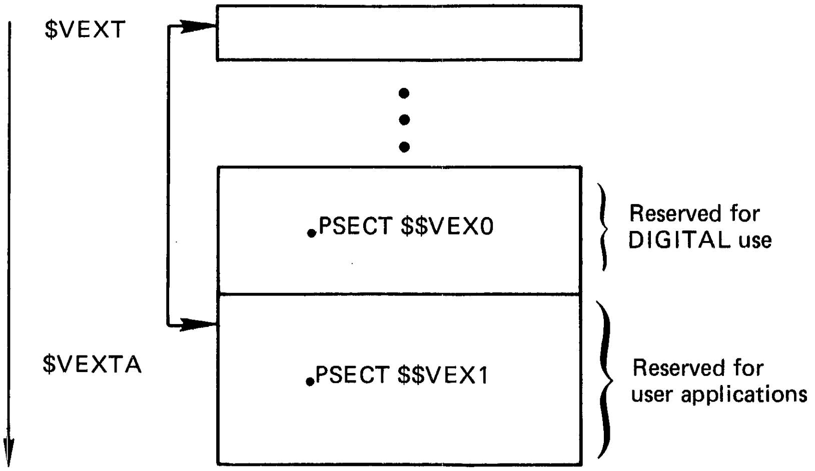


Figure B-7 Vector Extension Area Format

The program sections $$VEXO and $$VEX1 have the attributes D, GBL, RW, REL, and OVR.

The program section attribute OVR facilitates the definition of the offset to the vector and the initialization of the vector location at link time. For example:

```
                .GLOBL  $VEXTA      ;  MAKE SURE VECTOR AREA IS LINKED

                .PSECT  $$VEX1,D,GBL,RO,REL,OVR

BEG=.                               ;  POINT TO BASE OF POINTER TABLE

                .BLKW   N           ;  OFFSET TO CORRECT LOCATION
                                    ;  IN VECTOR AREA

LABEL:          .WORD   IMPURE      ;  SET IMPURE AREA ADDRESS
                                    ;  DEFINE OFFSET

OFFSET==LABEL-BEG

                .PSECT

IMPURE:

                  .
                  .
                  .
```

You should centralize all offset definitions within a single module from which the actual vector space allocation is made. Also, you should conditionalize the source to create two object modules: one that reserves the vector storage and, one that defines the global offsets which will be referred to by your resident library's subroutines.

Note that the sequence of instructions above intentionally redefines the global symbol. The Task Builder will report an error if this value differs from the centralized definition.

You can locate your vector through a sequence of instructions similar to the following:

```
MOV @#VEXT,R0           ; GET ADDRESS OF VECTOR EXTENSIONS
MOV OFFSET(R0),R0       ; POINT TO IMPURE AREA
.END
```

B.3.2 Logical Unit Table Entry

Figure B-8 illustrates the format of each entry in the Logical Unit Table.

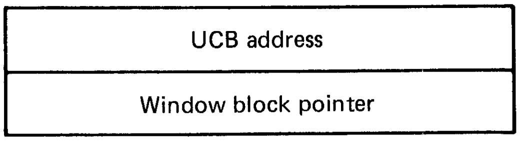


Figure B-8 Logical Unit Table Entry

The first word contains the address of the device unit control block in the Executive system tables. That block contains device-dependent information.

The second word is a pointer to the window block if the device is file structured.

The UCB address is set during task installation if a corresponding ASG parameter is specified at task build time. You can also set this word at run time with the Assign LUN Directive to the Executive.

The window block pointer is set when a file is opened on the device whose UCB address is specified by word 1. The window block pointer is cleared when the file is closed.

B.4 TASK IMAGE

The system reads the task image into memory beginning with the task header (see Figure B-1). The root segment of the task image is a set of contiguous disk blocks; it is therefore loaded with a single disk access.

Each overlay segment of the task image begins on a block boundary (see Figure B-1). The relative block number for the segment is placed in the segment table. Note that a given overlay segment occupies as many contiguous disk blocks as it needs to supply its space request. The maximum size for any segment, including the root, is 32K minus 32 words.

NOTE

One exception to the block boundary alignment of segments occurs when shared regions contain resident overlays. When this occurs, the image is compressed and, instead of being aligned on block boundaries, segments are aligned on 32-word boundaries. This facilitates the loading of regions.

Figure B-9 illustrates the structure and principal components of the task-resident overlay data base.

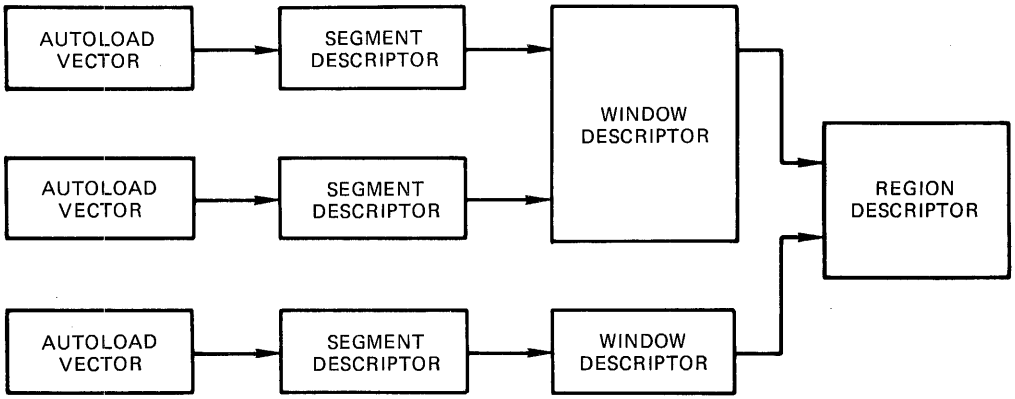


Figure B-9 Task-Resident Overlay Data Base

Autoload vectors are generated whenever a reference is made to an autoloadable entry point in a segment located farther away from the root than the segment making the reference.

One segment descriptor is generated for each overlay segment in the task or shared region. The segment descriptor contains information on the size, virtual address, and location of the segment within the task image file. In addition, it contains a set of link words that point to other segments. The overlay structure determines the link word contents.

The window descriptor contains information required to issue the mapping directives. One window descriptor is allocated for each memory-resident overlay in the structure.

The region descriptor contains information required to attach a resident library or common block. It is allocated within each task that refers to a shared region containing memory-resident overlays.

The following sections describe each data base component in greater detail.

B.4.1 Autoload Vectors

The autoload vector table consists of one entry per autoload entry point in the form shown in Figure B-10.

JSR	PC,sub
$AUTO	
Segment descriptor address	
Entry point address	

Figure B-10 Autoload Vector Entry

The autoload vector contains a JSR instruction to the autoload processor, $AUTO, followed by a pointer to the descriptor for the segment to be loaded and the real address of the entry point.

B.4.2 Segment Descriptor

The segment descriptor consists of a fixed part and two optional parts. The fixed part is six words long. If the manual-load feature is used ($LOAD), two words are added containing the segment name. When a memory-resident overlay structure is included, a ninth word is appended that points to the window descriptor.

Figure B-11 illustrates the contents of the segment descriptor.

Word	15 12	11 0	
0	Status	Relative disk address	Fixed Part
1	Load address		
2	Length in bytes		
3	Link up		
4	Link down		
5	Link next		
6	Segment		
7	name		
8	Window descriptor address		

Figure B-11 Segment Descriptor

Word 0 contains the relative disk address in bits 0 through 11 and the segment status in bits 12 through 15. Each segment in the task image file begins on a disk block boundary. The relative disk address is the block number of the segment relative to the start of the root segment.

The segment flags are defined as follows:

Bit	Setting
15	Always set to 1
14	0 = Segment has disk allocation 1 = Segment does not have disk allocation
13	0 = Segment is not loaded from disk 1 = Segment loaded from disk
12	0 = Segment is loaded and mapped 1 = Segment is either not loaded or not mapped

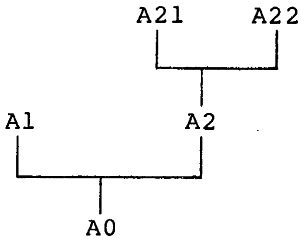
Word 1 contains the load address of the segment. This address is the first virtual address of the area where the segment will be loaded.

Word 2 specifies the length of the segment in bytes.

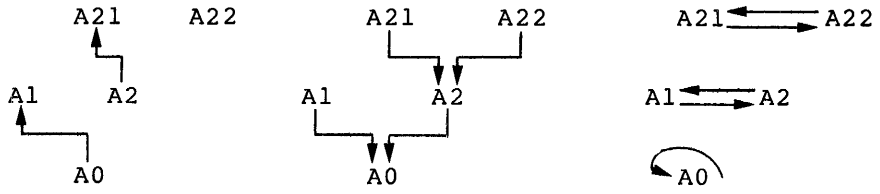
Words 3, 4, and 5 point to the following segment descriptors:

- Link up -- the next segment away from the root (0=none)
- Link down -- the next segment toward the root (0=none)
- Link next -- the adjoining segment; the link next pointers are linked in circular fashion

When the system loads a segment, the overlay run-time system follows the links to determine which segments are being overlaid and should therefore be marked out of memory. For example:



The segment descriptors are linked as follows:



link up link down link next

If there is a co-tree, the link next for the root segment descriptor points to the co-tree root segment descriptor.

Words 6 and 7 contain the segment name in Radix-50 format.

Word 8 points to the window descriptor used to map the segment (0=none).

B.4.3 Window Descriptor

The Task Builder allocates window descriptors only if you define a structure containing memory-resident overlays. Figure B-12 illustrates the format of a window descriptor.

Word	15 8	7 0
0	Base Active Page Register	Window ID
1	Virtual base address	
2	Window size in 64-byte blocks	
3	Region ID	
4	Offset in partition	
5	Length to map	
6	Status word	
7	Send/receive buffer address (always 0)	
8	Flags word	
9	Address of region descriptor	

Figure B-12 Window Descriptor

Words 0 through 7 constitute a window descriptor in the format required by the mapping directives. If the memory-resident overlay is part of the task, the region ID is zero. If the memory-resident overlay is part of a shared region, the ID is filled in at run time by the overlay loading routine.

Words 8 and 9 contain additional data that is referred to by the overlay routines. Bit 15 of the flags word, if set, indicates that the window is currently mapped into the task's address space.

Word 9 contains the address of the associated region descriptor. If the memory-resident overlay is part of the task, and no region descriptor is allocated, this value is zero.

B.4.4 Region Descriptor

The region descriptor is allocated only when the memory-resident overlay structure is part of a shared region. Figure B-13 illustrates the format of a region descriptor.

Word	
0	Region ID
1	Size of region
2	Region
3	name
4	Region
5	partition
6	Region status
7	Protection codes (always 0)
8	Flags

Figure B-13 Region Descriptor

Words 0 through 7 constitute a region descriptor in the format required by the mapping directives. The flags word is referred to by the overlay load routine. Bit 15 of the flags word, if set, indicates that a valid region identification is in word 0. If this bit is clear, the overlay load routine issues an Attach Region directive (with protection code set to zero) to obtain the identification.

B.4.5 Supervisor-Mode Vectors (RSX-11M-PLUS Only)

A supervisor-mode vector consists of four words. The Task Builder replaces each call from a user-mode task to the root segment of a supervisor-mode library with one of these structures. The supervisor-mode vector is shown in Figure B-14.

JSR PC,sub
$SUPL
Completion routine address
Entry point address

Figure B-14 Supervisor-Mode Vector

The supervisor-mode vector contains a JSR instruction to the context switching routine, $SUPL. $SUPL issues the SCAL$ Executive directive to context switch the processor from user mode to supervisor mode. The supervisor-mode vector also contains the address of the completion routine within the supervisor-mode library and the entry point of the library.

APPENDIX C

RESERVED SYMBOLS

Several global symbols and program section names[1] are reserved for use by the Task Builder.[2] Special handling occurs when the Task Builder encounters a definition of one of these names in a task image.

The definition of a reserved global symbol in the root segment causes a word in the task image to be modified with a value calculated by the Task Builder. The relocated value of the symbol is taken as the modification address.

The following global symbols are reserved by the Task Builder:

Global Symbol	Modification Value
.FSRPT	Address of file storage region work area (.FSRCB)
.MOLUN	Error message output device
.NLUNS	The number of logical units used by the task, not including the message output and overlay units
.NOVLY	The overlay logical unit number
N.OVPT	Address of overlay run-time system work area (.NOVLY)
.NSTBL	The address of the segment description tables; this location is modified only when the number of segments is greater than one
.ODTL1	Logical unit number for the ODT terminal device TI:
.ODTL2	Logical unit number for the ODT line printer device CL:
$OTSV	Address of Object Time System work area ($OTSVA)
.TRLUN	The trace subroutine output logical unit number
$VEXT	Address of vector extension area ($VEXTA)

[1] In RSX-11M and RSX-11M-PLUS, absolute sections (ASECTs) and both blank and named control sections (CSECTs) are supplanted by program sections (PSECTs). The .PSECT assembler directive eliminates the need for .ASECT and .CSECT directives, except for compatibility with other systems. This manual refers to all sections as program sections, unless the specific characteristics of ASECTs or CSECTs apply.

[2] All symbols and program section names containing a period (.) or a dollar sign ($) are reserved for DIGITAL-supplied software.

RESERVED SYMBOLS

The Task Builder reserves the following program section names. In some cases, the definition of a reserved program section causes that program section to be extended if you specify the appropriate option.

Section Name   Description

$$ALVC Contains the segment autoload vectors

$$DEVT The extension length (in bytes) is calculated from the formula:

EXT = <S.FDB+52>*UNITS

The definition of S.FDB is obtained from the root segment symbol table and UNITS is the number of logical units used by the task, excluding the message output, overlay, and ODT units.

$$FSR1 The extension of this section is specified by the ACTFIL option

$$IOB1 The extension of this section is specified by the MAXBUF option

$$OBF1 FORTRAN OTS uses this area to parse array type format specifications; this section can be extended by the FMTBUF keyword

$$RGDS Contains the region descriptors for resident libraries referred to by the task

$$RTS Contains the return instruction

$$SLVC Supervisor-mode library transfer vectors (RSX-11M-PLUS only)

$$SGD0 Contains the program section adjoining the task segment descriptors

$$SGD1 Contains the task segment descriptors

$$SGD2 Contains the program section following the task segment descriptors

$$WNDS Contains the task window descriptors

APPENDIX D

IMPROVING TASK BUILDER PERFORMANCE

This appendix contains procedures to assist you in maximizing Task Builder performance. These procedures include:

- Evaluating and improving Task Builder throughput

- Modifying command switch defaults to provide a more efficient user interface

- Using the Slow Task Builder when large work file space is required

These procedures assume that the program to be linked requires features not found in the Fast Task Builder (FTB) described in Appendix F.

Use of the procedures described in this appendix may require relinking the Task Builder. You can do this only in a system that has, as a minimum, a 14K user-controlled or system-controlled partition. In some cases, you can make the modifications without relinking by using the binary patch program ZAP (see the RSX-11 Utilities Manual).

Modifications to the Task Builder build file imply one or more of the following files located under UFD [1,24] (mapped) or [1,20] (unmapped):

RSX-11M systems:

```
BIGTKBBLD.CMD
TKBBLD.CMD
SLOTKBBLD.CMD
```

RSX-11M-PLUS systems:

```
TKBBIGBLD.CMD
TKBSLOBLD.CMD
```

These files reside on the RK05 disk containing the system object modules.

D.1 EVALUATING AND IMPROVING TASK BUILDER THROUGHPUT

Task Builder throughput is determined by three factors:

1. The amount of disk latency incurred because of overlays

2. The amount of memory available for table storage

3. The amount of disk latency due to input file processing

The following sections outline methods for improving throughput in each of these three cases.

D.1.1 Overlay Latency

The Task Builder is overlaid to reduce memory requirements. Two versions built with different overlay structures are supplied with the system:

- TKB.TSK, which is heavily overlaid but runs in an 8K partition
- BIGTKB.TSK, which contains fewer overlays than TKB.TSK but requires a 14K partition

You should install the appropriate version, that is, one that saves space or time, based on the system resources. In addition, the task should reside on the highest performance disk in the system.

D.1.2 Table Storage

The principal factor governing Task Builder performance is the amount of memory available for table storage. To reduce memory requirements, a work file is used to store symbol definitions and other tables. This work file cannot exceed 65,543 words. As long as the size of these tables is within the limits of available memory, the contents of this file are kept in memory and the disk is not accessed. If the tables exceed this limit, some information must be displaced and moved to the disk, degrading performance accordingly.

You can gauge work file performance by consulting the statistics portion of the Task Builder map. The map displays the following parameters:

- Number of work file references -- total number of times that work file data was referred to.
- Work file reads -- number of work file references that resulted in disk accesses to read work file data.
- Work file writes -- number of work file references that resulted in disk accesses to write work file data.
- Size of core pool -- amount of in-core table storage in words. This value is also expressed in units of 256-word pages (information is read from and written to disk in blocks of 256 words).
- Size of work file -- amount of work file storage in words. If this value is less than the pool size, the number of work file reads and writes is zero. That is, no work file pages are removed to the disk. This value is also expressed in pages (256-word blocks).
- Elapsed time -- amount of time required to build the task image and output the map. This value excludes ODL processing, option processing, and the time required to produce the global cross-reference.

You can reduce the overhead for gaining access to the work file in one or more of the following ways:

- By increasing the amount of memory available for table storage
- By placing the work file on the fastest random access device
- By decreasing system overhead required to gain access to the file
- By reducing the number of work file references

You can increase the amount of table storage by installing the Task Builder in a larger partition or, if the Task Builder is running in a system-controlled partition, by using the INSTALL/INC keyword to allocate more space.

In a system that includes support for the Extend Task directive, the Task Builder automatically increases its size if it is checkpointable and installed in a system-controlled partition. You set the maximum limit. You can increase this maximum by issuing the MCR command SET /MAXEXT.

Increasing the proportion of resident dynamic memory reduces the amount of I/O necessary for access to the Task Builder internal data structures. As stated above, once the resident memory has been filled, the data structures overflow into a temporary work file on the device assigned to the workfile logical unit number. This logical unit number (W$KLUN) is specified in the build command file. Preferably, this unit number should be assigned to a device other than the system device; e.g., a fixed head disk.

Displacement of pages to the workfile is done on a least recently used basis. The workfile extends automatically as necessary to hold all pages displaced. The parameter W$KEXT is provided in the build command file of the Task Builder and defines the file extension properties. A negative value indicates that the extend is noncontiguous, a positive value that the extend is contiguous. If a contiguous extend fails, a noncontiguous request is attempted; if a noncontiguous extend fails, a fatal workfile I/O error is reported. As long as the workfile remains contiguous, a higher access rate can be obtained.

It is not possible to state exactly how many symbols the Task Builder can process, because there are many data structures included in virtual memory. The following is a list of the structures that are stored in the virtual memory. All the sizes given are approximate only (sizes vary with characteristics of the task being built and may vary from release to release).

Structure Name	Description	Approx. Size (words)
Segment Descriptor	Contains listhead sizes, the pointers defining the overlay tree, the segment name. Part of this structure becomes the segment descriptor in the resultant task image.	60.

Structure Name	Description	Approx. Size (words)
P-section Descriptor	Contains the name, address size, and attributes of a p-section.	10.
Symbol Descriptor	Contains symbol name, value, flags, and pointers to defining segment and program section descriptors.	8. (nonoverlaid task) 15. (overlaid task)
Element Descriptor	Contains module name, ident, filename, count of program section and some flags.	8.-18.
Control Section Mapping Table	Table of program section size and program section descriptor addresses.	2 words per program section in each module

The maximum usage of virtual memory occurs during phase three of the Task Builder, when the symbol table is built. However, phase one makes significant use of virtual memory when an overlaid task is being built. It is at this point that all the segment descriptors are allocated, as well as an element descriptor for every filename encountered during the parsing of the tree description. In addition, a p-section descriptor is produced for every .PSECT directive encountered in the overlay description.

The parsing of the overlay description also makes use of dynamic memory during the processing of each directive. This memory is released upon completion of the analysis; during the analysis, however, the whole tree description must fit into the resident portion of the storage. If sufficient storage cannot be obtained in the resident dynamic memory, the error message NO DYNAMIC STORAGE AVAILABLE is returned. The method for increasing the ratio of dynamic storage to virtual memory may be applied here possibly to allow a task with a large overlay description to be built.

The amount of memory required during analysis depends on:

1. The number of directives,
2. The length of .FCTR lines,
3. The number of operators (i.e., commas, dashes, and parentheses), and
4. The number of filenames encountered.

The amount of resident storage area available depends on the version of the Task Builder used. The smaller version (TKB) has enough storage in an 8K partition to handle the overlay description for all privileged tasks.

The larger version (BIGTKB) links all DEC-supplied tasks in a 14K partition.

IMPROVING TASK BUILDER PERFORMANCE

There are a number of ways to reduce the amount of virtual memory required during the build of a specific task. Reduction of the data structures in virtual memory also increases the speed of searching the tables and reduces the amount of paging to the workfile.

1. Extract object modules by name from relocatable object libraries (e.g., LIBRY/LB:MOD1:MOD2). This technique requires smaller element descriptors and fewer filename descriptors and is also faster because there are fewer files to open and close.

2. Use concatenated object modules for the same reasons as above.

3. Use shared regions (resident libraries and common areas) for language and overlay run time systems and file control services. Such use of shared regions allows symbols and p-sections to be defined only once, rather than on multiple branches of the tree.

4. Place modules that occur on parallel branches of the tree in a common segment (i.e., closer to root) for the same reasons as 3 above.

5. Use the /SS switch on symbol table files (.STB) that describe absolute symbol definitions so that only those symbols referenced are extracted from the module.

6. Minimize the number of segments and keep the tree balanced. For example, if one segment is very long, there is no value in putting a tree structure in parallel unless creation of one segment in parallel would be longer.

In addition to the above, a version of BIGTKB can be built which has less throughput but requires less virtual memory per element than BIGTKB. This version is built using the command file SLOTKBBLD.CMD supplied on the RK05 utility disk or the RK06 and RP system disks under UFD [1,20] (unmapped) or [1,24] (mapped).

There are four error messages associated with the virtual memory system:

1. NO DYNAMIC STORAGE AVAILABLE. This error occurs when there is insufficient resident storage for creation of some data structure. As much as possible of the data already allocated (all unlocked pages) has been paged to the workfile, but there is still not enough free memory. Such a situation might arise during the analysis of the overlay description, early in the task-build run and particularly if it is a complex tree. Reduction of the ODL and extension of the Task Builder memory allocation (see above) are the recommended recovery procedures.

2. UNABLE TO OPEN WORKFILE. The probable causes of this error are:

   - Device assigned to logical unit 8 of the Task Builder is not mounted.

   - The device is not FILES-11.

   - There is no space on the volume.

- The device is offline, not ready, write locked, or faulty.
- There is no such device.

The MCR function LUN ...TKB may be used to determine which device the Task Builder is attempting to use.

3. WORKFILE I/O ERROR. The probable causes of this error are:

   - Hardware error; e.g., parity error on the disk.
   - Device is not ready, or is write-locked.
   - An extend failure has occurred; e.g., the disk is full.

4. NO VIRTUAL MEMORY STORAGE AVAILABLE. The addressable limit of the virtual memory has been reached. There is no recovery other than to reduce the virtual memory requirements of the task being built along the lines suggested earlier.

The work file normally resides on the device from which the Task Builder was installed. You can change the device by reassigning logical unit 8 through the Monitor Console Routine or by editing the build file and relinking the Task Builder.

System overhead for work file accesses is incurred in translating a relative block number in the file to a physical disk address. To minimize this overhead, the Task Builder requests disk space in contiguous increments. The size of each increment is equal to the value of symbol W$KEXT defined in the Task Builder build file. A larger positive value causes the file to be extended in larger contiguous increments and reduces the overhead required to gain access to the file. The increment should be set to a reasonable value because the Task Builder resorts to noncontiguous allocation whenever contiguous allocation fails.

You can reduce the size of the work file by:

- Linking your task to a core-resident library containing commonly used routines (for example, FORTRAN Object Time System) whenever possible
- Including common modules, such as components of an object time system, in the root segment of an overlaid task
- Using an object library or file of concatenated object modules if many modules are to be linked

When you use either of the last two procedures, system overhead is also significantly reduced because fewer files must be opened to process the same number of modules.

You can reduce the number of work file references by eliminating unneeded output files and cross-reference processing or by obtaining the short map. In addition, you can usually exclude selected files, such as the default system object module library, from the map. In this case, you can obtain, and retain, a full map at less frequent intervals.

D.1.3 Input File Processing

The procedures for minimizing the size of the work file and number of work file accesses also drastically reduce the amount of input file processing.

A given module can be read up to four times when the task is built:

- To build the symbol table
- To produce the task image
- To produce the long map
- To produce the global cross reference

Files that are excluded from the long map are read only twice. The third and fourth passes are eliminated for all modules when you request a short map without a global cross reference.

D.1.4 Summary

In summary, you can use the following procedures to improve Task Builder throughput:

- Use the INSTALL/INC or EXTTSK keyword to allocate more table space
- Increase maximum task size by raising the system limit for dynamic task extension
- Reduce disk latency by placing the work file on the fastest random access device
- Reduce system overhead by modifying the command file to allocate work file space in larger contiguous increments
- Decrease work file size by using resident libraries, concatenated object files, and object libraries
- Decrease work file size by including common modules into the root segment of an overlaid task
- Decrease the number of work file references by eliminating the map and global cross reference, obtaining the short map, or excluding files from the map

D.2 MODIFYING COMMAND SWITCH DEFAULTS

The default switch settings and values provided by the Task Builder as released may not suit the requirements of all installations. For example, the default /-EA (no KE11 Extended Arithmetic Element) would be unsatisfactory at an installation that made frequent use of this hardware.

You can thus tailor the switch defaults by altering the contents of the words that contain initial switch states. Modifying the Task Builder in this way is a three-step process:

1. Consult Tables D-1 through D-4 to determine the switch word and bit to be altered.
2. Edit the appropriate Task Builder command file to include the switch word modification through a GBLPAT keyword referring to the global switch word name.
3. Relink the Task Builder using the modified command file.

The command files for system tasks, as provided with the released system, require the standard set of Task Builder defaults; therefore, you must retain and use an unmodified copy of the Task Builder whenever such tasks are relinked.

You use tables D-1 through D-4 to alter the defaults as follows:

- You identify the switch and the file it applies to.
- You consult the switch category entry in each table to locate the applicable switch words.
- You scan the switch settings to find the switch and associated bit.
- You OR the desired setting of the associated bit with the initial contents to obtain the new set of defaults.
- You specify the revised value and switch word as arguments in a GBLPAT keyword.
- You relink the Task Builder to produce a version containing the appropriate defaults.

For example, to change the Task Builder Extended Arithmetic Element default to /EA, perform the steps described below.

By consulting Table D-1, you determine that two switch words, $DFSWT and $DFTSK, contain task file switches. Of these, $DFTSK contains the default setting for the EA switch in bit 13. Setting this bit to 1 changes the initial switch setting to /EA. This new value is combined (ORed) with the initial contents to yield the revised setting 120002. The required keyword input is:

```
TKB>GBLPAT=TASKB:$DFTSK:120002
```

NOTE

The setting of bit positions not listed in the tables must not be altered.

The only switches that have associated values are /AC and /PR. In these cases, the value is the number of the initial APR used to map the task. The default can be altered by changing the value specified in the build file GBLDEF keyword for the symbol D$FAPR. Only values 4 or 5 can be used.

Table D-1
Task File Switch Defaults

Switch Category: Task file

Switch Word: $DFSWT

Initial Contents: 0

Switch Settings:

Bit	Condition if Set to 1	
15	AB	Abort build on error
11	SQ	Sequential PSECT allocation
4	FU	Full overlay tree search
3	-RO	Disable recognition of memory-resident overlay operator

Switch Category: Task file

Switch Word: $DFTSK

Initial Contents: 100002

Switch Settings:

Bit	Condition if Set to 1	
15	-CP, -AL	Not checkpointable[1]
14	FP	Floating-point processor
13	EA	Extended Arithmetic Element
12	-HD	No header
11	CM	Compatibility Mode
10	DA	Debugging aid
9	PI	Position independent
8	PR	Privileged
7	TR	Trace
6	PM	Postmortem Dump
5	SL	Slave task
4	-SE	No SEND to task
2	AC	Ancillary control processor
1	-AL	No checkpoint allocation[1]
0	NT	Revised network protocol

[1] The combination of not checkpointable with checkpoint allocation (100000) is illogical and should not be used.

Table D-2
Map File Switch Defaults

Switch Category: Map file

Switch Word: $DFLBS

Initial Contents: 120000

Switch Settings:

Bit	Condition if Set to 1	
15	-MA	Do not include system library and STB files in map

Switch Category: Map file

Switch Word: $DFMAP

Initial Contents: 2040

Switch Settings:

Bit	Condition if Set to 1	
10	SH	Short map
8	-SP	Do not spool
6	CR	Cref
5	WI	Wide format

Table D-3
Symbol Table File Switch Defaults

Switch Category: Symbol table file

Switch Word: $DFSTB

Initial Contents: 0

Switch Settings:

Bit	Condition if Set to 1	
12	-HD	Build task without header
9	PI	Task is position independent

Table D-4
Input File Switch Defaults

Switch Category: Input file

Switch Word: $DFINP

Initial Contents: 000100

Switch Settings:

Bit	Condition if Set to 1	
15	-MA	Do not include file contents in map
6	CC	File contains two or more concatenated object modules

D.3 THE SLOW TASK BUILDER

The TKB.TSK and BIGTKB.TSK versions of the Task Builder each use a symbol table structure that can be searched quickly, but which requires more work file space than previous versions. You may thus receive the following message in some instances:

```
NO VIRTUAL MEMORY STORAGE AVAILABLE
```

If this occurs, you should try to reduce the work file size by using the procedures described in Section D.1. If these procedures do not sufficiently reduce the work file size, you can link a third version of the Task Builder, the Slow Task Builder. This version requires less storage, but runs considerably slower than the other versions. The build file is SLOTKBBLD.CMD which resides on the same device and UFD as the other Task Builder command files.

APPENDIX E

THE FAST TASK BUILDER

The Fast Task Builder (FTB) is intended for use as a load-and-go type of linker. It contains very few options and does **not** support:

- New map format
- Overlaid programs
- Linking to resident libraries
- Production of symbol table files
- Creation of resident libraries
- Privileged tasks

The supported switches are:

/SP on map file (default = /SP)

/CP on task file (default = /CP) [1]

/MM on task file (default = /MM)

/FP on task file (default = /FP)

/DA on input or task image (default = /-DA)

The support option inputs are:

ASG (same defaults as TKB)

STACK (same default as TKB)

UNITS

EXTSCT

ACTFIL (same default as TKB)

MAXBUF (same default as TKB)

[1] No checkpoint space is allocated in the task image file.

FTB allocates symbol table space from the end of its image to the end of the partition. It does not have a virtual symbol table. An Extend Task or equivalent of 8K is recommended. FTB does not dynamically extend itself at run time.

FTB runs approximately four times faster than TKB on an 11/70 with RP04s when TKB is running with a totally resident symbol table. In smaller systems with slower disks, the ratio should be much higher.

APPENDIX F

ERROR MESSAGES

The Task Builder produces diagnostic and fatal error messages. Error messages are printed in the following forms:

```
TKB -- *DIAG*-error-message

              or

TKB -- *FATAL*-error-message
```

Some errors are correctable when command input is from a terminal. In such a case, a diagnostic error message can be printed, the error corrected, and the task building sequence continued. However, if the same error is detected in an indirect command file, a correction cannot be made, and the Task Builder aborts.

Some diagnostic error messages merely advise you of an unusual condition. If you consider the condition normal for your task, you can install and run the task image.

NOTE

> The Task Builder exits with 2 statuses: it returns an ERROR status when it encounters a diagnostic error and a SEVER ERROR when it encounters a fatal error. (For more information about the Exit-With-Status directive, see the RSX-11M/M-PLUS Executive Reference Manual.)

This appendix tabulates the error messages produced by the Task Builder. Most of the messages are self-explanatory. In some cases, the line in which the error occurred is printed.

A Software Performance Report (SPR) should be submitted to DIGITAL in cases where the explanation accompanying a message refers to a system error.

ALLOCATION FAILURE ON FILE file-name

The Task Builder could not acquire sufficient disk space to store the task image file, or did not have write-access to the UFD or volume that was to contain the file.

BLANK P-SECTION NAME IS ILLEGAL
overlay-description-line

The overlay-description-line printed contains a .PSECT directive that does not have a p-section name.

COMMAND I/O ERROR

I/O error on command input device. (Device may not be online, or possible hardware error.)

COMMAND SYNTAX ERROR
command-line

The command-line printed has incorrect syntax.

COMPLEX RELOCATION ERROR - DIVIDE BY ZERO: MODULE
module-name

A divisor having the value zero was detected in a complex expression. The result of the divide was set to zero. (Probable cause - division by a global symbol whose value is undefined.)

FILE file-name ATTEMPTED TO STORE DATA IN VIRTUAL SECTION

The file contains a module that has attempted to initialize a virtual section with data.

FILE file-name HAS ILLEGAL FORMAT

The file file-name contains an object module whose format is not valid.

ILLEGAL APR RESERVATION

An APR specified in a COMMON, LIBR, RESCOM, or RESLIB keyword is outside the range 0-7.

ILLEGAL DEFAULT PRIORITY SPECIFIED
option-line

The option-line printed contains a priority greater than 250.

ILLEGAL ERROR-SEVERITY CODE octal-list

System error (no recovery). An SPR should be submitted with a copy of the message containing the octal-list as printed.

ILLEGAL FILENAME
invalid-line

The invalid-line printed contains a wild card (*) in a file specification. The use of wild cards is prohibited.

ILLEGAL GET COMMAND LINE ERROR CODE

System error (no recovery).

ILLEGAL LOGICAL UNIT NUMBER
invalid-line

The invalid-line printed contains a device assignment to a unit number larger than the number of logical units specified by the UNITS keyword or assumed by default if the UNITS keyword is not used.

ILLEGAL MULTIPLE PARAMETER SETS
invalid-line

The invalid-line printed contains multiple sets of parameters for a keyword that allows only a single parameter set.

ILLEGAL NUMBER OF LOGICAL UNITS
invalid-line

The invalid-line printed contains a logical unit number greater than 250.

ILLEGAL ODT OR TASK VECTOR SIZE

ODT or SST vector size specified greater than 32 words.

ILLEGAL OVERLAY DESCRIPTION OPERATOR
invalid-line

The invalid-line printed contains an unrecognizable operator in an overlay description. This error occurs if the first character in a p-section or segment name is a dot (.).

ILLEGAL OVERLAY DIRECTIVE
invalid-line

The invalid-line printed contains an unrecognizable overlay directive.

ILLEGAL PARTITION/COMMON BLOCK SPECIFIED
invalid-line

User defined base or length not on 32-word boundary.

ILLEGAL P-SECTION/SEGMENT ATTRIBUTE
invalid-line

The invalid-line printed contains a program section or segment attribute that is not recognized.

ILLEGAL REFERENCE TO LIBRARY P-SECTION p-sect-name

A task has attempted to reference a p-sect-name existing in a shared region but has not named the shared region in a keyword. This error will occur when you explicitly specify an STB file as an input file but you have not specified the library to which the STB file belongs in an option.

ILLEGAL SWITCH
file-specification

The file-specification printed contains an illegal switch or switch value.

INCOMPATIBLE REFERENCE TO LIBRARY P-SECTION p-sect-name

A task has attempted to reference more storage in a shared region than exists in the shared region definition.

INCORRECT LIBRARY MODULE SPECIFICATION
invalid-line

The invalid-line contains a module name with a non-Radix-50 character.

INDIRECT COMMAND SYNTAX ERROR
invalid-line

The invalid-line printed contains a syntactically incorrect indirect file specification.

INDIRECT FILE OPEN FAILURE
invalid-line

The invalid-line contains a reference to a command input file which could not be located.

INSUFFICIENT PARAMETERS
invalid-line

The invalid-line contains a keyword with an insufficient number of parameters to complete its meaning.

INVALID APR RESERVATION
invalid-line

APR specified on a keyword for an absolute library.

INVALID KEYWORD IDENTIFIER
invalid-line

The invalid-line printed contains an unrecognizable keyword.

INVALID PARTITION/COMMON BLOCK SPECIFIED
invalid-line

Partition is invalid for one of the following reasons:

1. The Task Builder cannot find the partition name in the host system in order to get the base and length.

2. The system is mapped, but the base address of the partition is not on a 4K boundary for a non-runnable task or is not 0 for a runnable task.

3. The memory bounds for the partition overlap a shared region.

4. The partition name is identical to the name of a previously defined COMMON or LIBR shared region.

5. The top address of the partition for a runnable task exceeds 32K minus 32 words for a mapped system, or exceeds 28K minus 1 for an unmapped system.

6. A system-controlled partition was specified for an unmapped system.

INVALID REFERENCE TO MAPPED ARRAY BY MODULE module-name

The module has attempted to initialize the mapped array with data. An SPR should be submitted if this problem is caused by DIGITAL-supplied software.

INVALID WINDOW BLOCK SPECIFICATION
invalid-line

The number of extra address windows specified exceeds the number permitted. On an RSX-11M system, you can specify as many as 7 extra window blocks; on an RSX-11M-PLUS system, you can specify as many as fifteen extra window blocks.

If you build a task on an RSX-11M system and specify more window blocks, you get this error message, but the task will build. However, it cannot be installed and run on an RSX-11M system.

I/O ERROR LIBRARY IMAGE FILE

An I/O error has occurred during an attempt to open or read the Task Image File of a shared region.

I/O ERROR ON INPUT FILE file-name

This error occurs when the Task Builder cannot read an input file specification (for example, when the command line is greater than eighty characters).

I/O ERROR ON OUTPUT FILE file-name

LABEL OR NAME IS MULTIPLY DEFINED
invalid-line

The invalid-line printed defines a name that has already appeared as a .FCTR, .NAME, or .PSECT directive.

LIBRARY FILE filename HAS INCORRECT FORMAT

A module has been requested from a library file that has an empty module name table.

LIBRARY REFERENCES OVERLAID LIBRARY
invalid-line

An attempt was made to link the resident library being built to a shared region that has memory-resident overlays.

LOAD ADDR OUT OF RANGE IN MODULE module-name

An attempt has been made to store data in the task image outside the address limits of the segment. This problem is usually caused by one of the following:

1. an attempt to initialize a p-section contained in a shared region

2. an attempt to initialize an absolute location outside the limits of the segment or in the task header

3. a patch outside the limits of the segment it applies to

4. an attempt to initialize a segment having the NODSK attribute

LOOKUP FAILURE ON FILE filename
invalid-line

The invalid-line printed contains a filename that cannot be located in the directory.

LOOKUP FAILURE ON SYSTEM LIBRARY FILE

The Task Builder cannot find the system Library (SY0:[1,1]SYSLIB.OLB) file to resolve undefined symbols.

LOOKUP FAILURE RESIDENT LIBRARY FILE
invalid-line

No symbol table or task image file can be found for the shared region.

MAXIMUM INDIRECT FILE DEPTH EXCEEDED
invalid-line

The invalid-line printed gives the file reference that exceeded the permissible indirect file depth (2).

MODULE module-name AMBIGUOUSLY DEFINES P-SECTION p-sect-name

The p-section p-sect-name has been defined in two modules not on a common path, and referenced from a segment that is common to both paths.

MODULE module-name AMBIGUOUSLY DEFINES SYMBOL sym-name

Module module-name references or defines a symbol sym-name whose definition exists on two different paths, but is referenced from a segment that is common to both paths.

MODULE module-name ILLEGALLY DEFINES XFR ADDRESS p-sect-name addr

1. The start address printed is odd.

2. The module module-name is in an overlay segment and has a start address. The start address must be in the root segment of the main tree.

3. The address is in a p-section that has not yet been defined. An SPR should be submitted if this is caused by DIGITAL-supplied software.

MODULE module-name MULTIPLY DEFINES P-SECTION p-sect-name

1. The p-section p-sect-name has been defined more than once in the same segment with different attributes.

2. A global p-section has been defined more than once with different attributes in more than one segment along a common path.

MODULE module-name MULTIPLY DEFINES SYMBOL sym-name

Two definitions for the relocatable symbol sym-name have occurred on a common path. Or two definitions for an absolute symbol with the same name but different values have occurred.

MODULE module-name MULTIPLY DEFINES XFR ADDR IN SEG
segment-name

This error occurs when more than one module making up the root has a start address.

MODULE module-name NOT IN LIBRARY

The Task Builder could not find the module named on the LB switch in the library.

NO DYNAMIC STORAGE AVAILABLE

The Task Builder needs additional symbol table storage and cannot obtain it. (If possible, install the Task Builder in a larger partition.)

NO MEMORY AVAILABLE FOR LIBRARY library-name

The Task Builder could not find enough free virtual memory to map the specified shared region.

NO ROOT SEGMENT SPECIFIED

The overlay description did not contain a .ROOT directive.

NO VIRTUAL MEMORY STORAGE AVAILABLE

Maximum permissible size of the work file exceeded. The user should consult Appendix D for suggestions on reducing the size of the work file.

OPEN FAILURE ON FILE file-name

OPTION SYNTAX ERROR
invalid-line

The invalid-line printed contains unrecognizable syntax.

OVERLAY DIRECTIVE HAS NO OPERANDS
invalid-line

All overlay directives except .END require operands.

OVERLAY DIRECTIVE SYNTAX ERROR
invalid-line

The invalid-line printed contains a syntax error or references a line that contains an error.

PARTITION partition-name HAS ILLEGAL MEMORY LIMITS

1. The partition-name defined in the host system has a base address alignment that is not compatible with the target system.

2. The user has attempted to build a privileged task in a partition whose length exceeds the task's available address space (8K or 12K).

PASS CONTROL OVERFLOW AT SEGMENT segment-name

System error. An SPR should be submitted with a copy of the ODL file associated with the error.

PIC LIBRARIES MAY NOT REFERENCE OTHER LIBRARIES
invalid-line

The user has attempted to build a position-independent shared region that references another shared region.

P-SECTION p-sect-name HAS OVERFLOWED

A section greater than 32K has been created.

REQUIRED INPUT FILE MISSING

At least one input file is required for a task-build.

REQUIRED PARTITION NOT SPECIFIED

The PAR keyword was not used when running the Task Builder on an RSX-11D host system. The keyword must contain explicit base address and length specifications.

RESIDENT LIBRARY HAS INCORRECT ADDRESS ALIGNMENT
invalid-line

The invalid-line specifies a shared region that has one of the following problems:

1. The library references another library with invalid address bounds (i.e., not on 4K boundary in a mapped system).

2. The library has invalid address bounds.

RESIDENT LIBRARY MAPPED ARRAY ALLOCATION TOO LARGE
invalid-line

The invalid-line printed contains a reference to a shared region that has allocated too much memory in the task's mapped array area. The total allocation exceeds 2.2 million bytes.

RESIDENT LIBRARY MEMORY ALLOCATION CONFLICT
keyword-string

One of the following problems has occurred:

1. More than seven shared regions have been specified.

2. A shared region has been specified more than once.

3. Non-position-independent shared regions whose memory allocations overlap, have been specified.

ROOT SEGMENT IS MULTIPLY DEFINED
invalid-line

The invalid-line printed contains the second .ROOT directive encountered. Only one .ROOT directive is allowed.

SEGMENT seg-name HAS ADDR OVERFLOW: ALLOCATION DELETED

Within a segment, the program has attempted to allocate more than 32K. A map file is produced, but no task image file is produced.

TASK HAS ILLEGAL MEMORY LIMITS

An attempt has been made to build a task whose size exceeds the partition boundary. If a task image file was produced, it should be deleted.

TASK HAS ILLEGAL PHYSICAL MEMORY LIMITS
mapped-array   task-image   task extension

The sum of the parameters displayed -- mapped array size, task image size, and task extension -- exceeds 2.2 million bytes. The quantities are shown as octal numbers in units of 64-byte blocks. Any resulting task image file should be deleted.

TASK IMAGE FILE filename IS NON-CONTIGUOUS

Insufficient contiguous disk space was available to contain the task image. A non-contiguous file was created. After deleting unnecessary files, the /CO switch in PIP should be used to create a contiguous copy.

TASK REQUIRES TOO MANY WINDOW BLOCKS

The number of address windows required by the task and any shared regions, exceeds 8 for RSX-11M tasks and sixteen for RSX-11M-PLUS tasks.

TASK-BUILD ABORTED VIA REQUEST
option-line

The option-line contains a request from the user to abort the task-build.

TOO MANY NESTED .ROOT/.FCTR DIRECTIVES
invalid-line

The invalid-line printed contains a .FCTR directive that exceeds the maximum nesting level (16).

TOO MANY PARAMETERS
invalid-line

The invalid-line printed contains a keyword with more parameters than required.

TOO MANY PARENTHESES LEVELS
invalid-line

The invalid-line printed contains a parenthesis that exceeds the maximum nesting level (16).

TRUNCATION ERROR IN MODULE module-name

An attempt has been made to load a global value greater than +127 or less than -128 into a byte. The low-order eight bits are loaded.

UNABLE TO OPEN WORK FILE

The work file device is not mounted. (The work file is usually located on the same device as the Task Builder.)

UNBALANCED PARENTHESES
invalid-line

The invalid-line printed contains unbalanced parentheses.

n UNDEFINED SYMBOLS SEGMENT seg-name

The segment named contains n undefined symbols. If no memory allocation file is requested, the symbols are printed on the terminal.

VIRTUAL SECTION HAS ILLEGAL ADDRESS LIMITS
option-line

The option-line printed contains a VSECT keyword whose base address plus window size exceeds 177777.

WORK FILE I/O ERROR

I/O error during an attempt to reference data stored by the Task Builder in its work file.

INDEX

RSX-11M/M-PLUS
Task Builder Manual
AA-H266A-TC

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- □ Assembly language programmer
- □ Higher-level language programmer
- □ Occasional programmer (experienced)
- □ User with little programming experience
- □ Student programmer
- □ Other (please specify)____________________

Name____________________ Date__________

Organization____________________

Street____________________

City__________ State__________ Zip Code or Country__________

Please cut along this line.

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

RT/C SOFTWARE PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

Do Not Tear - Fold Here

Cut Along Dotted Line