RSX-11M-PLUS and Micro/RSX Task Builder Manual

Order No. AA-JS08A-TC

RSX-11M-PLUS Version 4.0
Micro/RSX Version 4.0

Digital Equipment Corporation Maynard, Massachusetts

First Printing, December 1983
Revision, September 1987

DEC	EduSystem	UNIBUS
DEC/CMS	IAS	VAX
DEC/MMS	MASSBUS	VAXcluster
DECnet	MicroPDP–11	VMS
DECsystem–10	Micro/RSX	VT
DECSYSTEM–20	PDP	
DECUS	PDT	
DECwriter	RSTS	digital
DIBOL	RSX	

ZK3086

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*	CANADA	INTERNATIONAL
Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario K2K 2A6 Attn: Direct Order Desk	Digital Equipment Corporation PSG Business Manager c/o Digital's local subsidiary or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a trademark of the American Mathematical Society.

Contents

Chapter 2 Task Builder Functions

Chapter 3 Overlay Capability

Chapter 5 Shared Region Concepts and Examples

Chapter 8 Supervisor-Mode Libraries

Chapter 9 Multiuser Tasks

Chapter 10 TKB Switches

Examples

Figures

Preface

Manual Objectives

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

Working examples are used throughout this manual to introduce and describe features of the Task Builder. Because RSX 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 (Micro/RSX or RSX–11M–PLUS) and with the operating procedures described in the *RSX–11M–PLUS MCR Operations Manual*, the *RSX–11M–PLUS Command Language Manual*, and the *Micro/RSX User's Guide*. In addition, you should be familiar with the programming concepts described in the *RSX–11M–PLUS Guide to Program Development*.

Structure of This Document

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 and the resolution of global symbols. It also contains an introduction to supervisor-mode libraries, privileged tasks, and multiuser tasks.

Chapter 3 describes the Task Builder's overlay capability and the language you use to define an overlay structure.

Chapter 4 describes the two methods available to you to load overlay segments.

Chapter 5 describes some typical Task Builder features, including tasks that access shared regions and device commons, tasks that create dynamic regions, and virtual program sections.

Chapter 6 defines privileged tasks, describes their mapping, and shows how to build a privileged task to examine Unit Control Blocks.

Chapter 7 describes user-mode I- and D-space, the mapping of these spaces, and the advantages of using I- and D-space in user mode.

Chapter 8 describes supervisor-mode libraries. The chapter defines and shows how to build and use supervisor-mode libraries.

Chapter 9 describes and shows how to build multiuser tasks.

Chapter 10 lists and describes the Task Builder switches. The switches are listed in alphabetical order.

Chapter 11 lists and describes the qualifiers for the DCL command LINK. The qualifiers are listed in alphabetical order.

Chapter 12 lists and describes the Task Builder options. The options are listed in alphabetical order.

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

Appendix D describes two memory dumps: postmortem and snapshot.

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

Appendix F contains information on improving Task Builder performance.

Appendix G describes the Fast Task Builder.

Appendix H contains the Task Builder error messages.

A Task Builder glossary follows the appendixes.

Associated Documents

Other manuals related to this document are described in the *RSX–11M–PLUS Information Directory and Master Index.* 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

The following conventions are used in this manual:

Convention	Meaning
>	A right angle bracket is the default prompt for the Monitor Console Routine (MCR), which is one of the command interfaces used on RSX-11M-PLUS systems. All systems include MCR.
$	A dollar sign followed by a space is the default prompt of the DIGITAL Command Language (DCL), which is one of the command interfaces used on RSX-11M-PLUS and Micro/RSX systems. Many systems include DCL.
MCR>	This is the explicit prompt of the Monitor Console Routine (MCR).
DCL>	This is the explicit prompt of the DIGITAL Command Language (DCL).
xxx>	Three characters followed by a right angle bracket indicate the explicit prompt for a task, utility, or program on the system.
UPPERCASE	Uppercase letters in a command line indicate letters that must be entered as they are shown. For example, utility switches must always be entered as they are shown in format specifications.
command abbreviations	Where short forms of commands are allowed, the shortest form acceptable is represented by uppercase letters. The following example shows the minimum abbreviation allowed for the DCL command DIRECTORY: `$ DIR`
lowercase	Any command in lowercase must be substituted for. Usually the lowercase word identifies the kind of substitution expected, such as a filespec, which indicates that you should fill in a file specification. For example: `filename.filetype;version` This command indicates the values that comprise a file specification; values are substituted for each of these variables as appropriate.
/keyword, /qualifier, or /switch	A command element preceded by a slash (/) is an MCR keyword; a DCL qualifier; or a task, utility, or program switch. Keywords, qualifiers, and switches alter the action of the command they follow.
parameter	Required command fields are generally called parameters. The most common parameters are file specifications.

Convention	Meaning
[option]	Square brackets indicate optional entries in a command line or a file specification. If the brackets include syntactical elements, such as periods (.) or slashes (/), those elements are required for the field. If the field appears in lowercase, you are to substitute a valid command element if you include the field. Note that when an option is entered, the brackets are not included in the command line.
[,...]	Square brackets around a comma and an ellipsis mark indicate that you can use a series of optional elements separated by commas. For example, (argument[,...]) means that you can specify a series of optional arguments by enclosing the arguments in parentheses and by separating them with commas.
{ }	Braces indicate a choice of required options. You are to choose from one of the options listed.
:argument	Some parameters and qualifiers can be altered by the inclusion of arguments preceded by a colon. An argument can be either numerical (COPIES:3) or alphabetical (NAME:QIX). In DCL, the equal sign (=) can be substituted for the colon to introduce arguments. COPIES=3 and COPIES:3 are the same.
( )	Parentheses are used to enclose more than one argument in a command line. For example: `SET PROT = (S:RWED,O:RWED)`
,	Commas are used as separators for command line parameters and to indicate positional entries on a command line. Positional entries are those elements that must be in a certain place in the command line. Although you might omit elements that come before the desired element, the commas that separate them must still be included.
[g,m] [directory]	The convention [g,m] signifies a User Identification Code (UIC). The g is a group number and the m is a member number. The UIC identifies a user and is used mainly for controlling access to files and privileged system functions. This may also signify a User File Directory (UFD), commonly called a directory. A directory is the location of files. Other notations for directories are: [ggg,mmm], [gggmmm], [ufd], [name], and [directory]. The convention [directory] signifies a directory. Most directories have 1- to 9-character names, but some are in the same [g,m] form as the UIC. Where a UIC, UFD, or directory is required, only one set of brackets is shown (for example, [g,m]). Where the UIC, UFD, or directory is optional, two sets of brackets are shown (for example, [[g,m]]).

Convention	Meaning
filespec	A full file specification includes device, directory, file name, file type, and version number, as shown in the following example: `DL2:[46,63]INDIRECT.TXT;3` Full file specifications are rarely needed. If you do not provide a version number, the highest numbered version is used. If you do not provide a directory, the default directory is used. Some system functions default to particular file types. Many commands accept a wildcard character (*) in place of the file name, file type, or version number. Some commands accept a filespec with a DECnet node name.
.	A period in a file specification separates the file name and file type. When the file type is not specified, the period may be omitted from the file specification.
;	A semicolon in a file specification separates the file type from the file version. If the version is not specified, the semicolon may be omitted from the file specification.
@	The at sign invokes an indirect command file. The at sign immediately precedes the file specification for the indirect command file, as follows: **`@filename`**`[.`**`filetype`**`;`**`version`**`]`
...	A horizontal ellipsis indicates the following: • Additional, optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis shows where elements of command input or statements in an example or figure have been omitted because they are irrelevant to the point being discussed.
KEYNAME	This typeface denotes one of the keys on the terminal keyboard; for example, the RETURN key.
"print" and "type"	The term "print" refers to any output sent to a terminal by the system. The term "type" refers to any user input from a terminal.
black ink	In examples, what the system prints or displays is printed in black.

Convention	Meaning
red ink	In interactive examples, what the user types is printed in red. System responses appear in black.
$\boxed{\text{xxx}}$	A symbol with a 1- to 3-character abbreviation, such as $\boxed{\text{x}}$ or $\boxed{\text{RET}}$, indicates that you press a key on the terminal. For example, $\boxed{\text{RET}}$ indicates the RETURN key, $\boxed{\text{LF}}$ indicates the LINE FEED key, and $\boxed{\text{DEL}}$ indicates the DELETE key.
$\boxed{\text{CTRL/a}}$	The symbol $\boxed{\text{CTRL/a}}$ means that you are to press the key marked CTRL while pressing another key. Thus, $\boxed{\text{CTRL/Z}}$ indicates that you are to press the CTRL key and the Z key together in this fashion. $\boxed{\text{CTRL/Z}}$ is echoed on some terminals as ^Z. However, not all control characters echo.

Summary of Technical Changes

The following sections list features, switches, qualifiers, and error messages that are new to the Task Builder (TKB) or have been modified for the RSX-11M-PLUS and Micro/RSX Version 4.0 operating systems. These new or modified features are documented in this revision of the *RSX-11M-PLUS and Micro/RSX Task Builder Manual.*

Also, major changes to the organization of the manual are included at the end of this summary.

New Feature

RNDSEG option

The RNDSEG option causes TKB to round the size of a named segment up to the nearest Active Page Register (APR) boundary while building a resident library.

New Switches and Qualifiers

TKB has the following new switches and qualifiers:

New Switches:

- /CL
- /FM
- /FO
- /SB

/CL

The /CL switch tells TKB that the task is a command line interpreter.

/FM

The /FM switch tells TKB to allocate space in memory between the task and the external header for use by the fast-mapping feature of the Executive.

/FO

The /FO switch causes the task to use overlay run-time system Fast Map module.

/SB

The /SB switch causes the task to be built with the slow mode of the Task Builder.

New Qualifiers:

- /CODE:CLI
- /CODE:FAST_MAP
- /CODE:OTS_FAST
- /SLOW

/CODE:CLI

The /CODE:CLI qualifier specifies that the task is a command line interpreter.

/CODE:FAST_MAP

The /CODE:FAST_MAP qualifier specifies that space must be allocated in memory between the task and external header for use by the fast-mapping feature of the Executive.

/CODE:OTS_FAST

The /CODE:OTS_FAST qualifier specifies that the overlay run-time system (OTS) fast mapping module FSTMAP be included in the task.

/SLOW

The /SLOW qualifier invokes the slow mode of the Task Builder.

New Error Messages

TKB produces the following new error messages:

- Cluster library element, element-name, is not resident overlaid
- Incompatible OTS module

Cluster library element, element-name, is not resident overlaid

This message occurs when the listed cluster element has been built without memory-resident overlays. This kind of element cannot be used as a cluster library element. Cluster libraries 2 through 6 must be memory-resident and overlaid.

Incompatible OTS module

This message occurs when the OTS (overlay run-time system) module requested by TKB has not been found. The OTS modules are part of the system library. This error occurs if you are using an incompatible version of the system library (SYSLIB.OLB).

Changes to the Document

The following changes in organization are included in this revision of the *RSX-11M-PLUS and Micro/RSX Task Builder Manual*:

- References to unmapped systems have been removed since this version of the manual concerns only the RSX-11M-PLUS and Micro/RSX operating systems.
- The discussion of the use and size of overlay run-time routines in Chapter 4 has been expanded.
- A new section in Chapter 5 discusses using FCSRES and FCSFSL.
- Appendix F describes how to create a Task Builder that has the slow mode as its default.

Chapter 1
Introduction and Command Specifications

The basic steps in developing a program are as follows:

1. You write one or more routines in an RSX–11M–PLUS or Micro/RSX supported source language and enter each routine as an ASCII text file. You accomplish this by using an editor such as EDT.
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 (TKB), 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 an RSX–11M–PLUS or Micro/RSX system. The task is the fundamental executable unit in both systems.

If your program consists of one object module, using the Task Builder is 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 one object module. In this case, you name each of the object module files as input. TKB 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.

TKB 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.

TKB also produces (upon request) a memory allocation (or 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. This list is called a global cross-reference.

Note that the examples in this manual use both MCR and DCL as the command line interpreters (CLIs).

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

```
      MCR                       DCL

>MAC PROG,=PROG              $ MACRO PROG
>TKB PROG,,=PROG             $ LINK PROG
>INS PROG                    $ INS PROG
>RUN PROG                    $ RUN PROG
```

The first command, MAC or MACRO, 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 or LINK, causes TKB to process the file PROG.OBJ and to produce the task image file PROG.TSK. The third command, INS, causes the INSTALL task to add PROG.TSK to the Executive's directory of executable tasks (the System Task Directory). The fourth command, RUN, causes the task to execute.

The previous example includes the following command:

```
     TKB (MCR)                LINK (DCL)

>TKB PROG,,=PROG      or     $ LINK PROG
```

This command illustrates the simplest use of TKB. A single file is the input and a single file is the output.

The following sections describe basic Task Builder command forms and sequences.

1.1 Task Builder Command Line

The Task Builder command lines for both MCR and DCL are discussed in the following sections.

1.1.1 The MCR Command Line for the Task Builder

The task command line used in MCR 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. TKB allows a command line to be a maximum of 132 characters in length.

The task command line has the following MCR form:

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

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 TKB. You specify this file when you are building a resident library or common. (Resident libraries and commons are described in Chapter 3.) TKB combines the input files to create a single task image that can be installed and executed.

1.1.1.1 Printing the Map File

If you create a map file by specifying one in the TKB command line, there are a number of ways that you can print the file. The following examples show you how you may print the map file.

1. With the following two command lines, you can create a map file and then print it later. The TKB command line tells TKB to create a task file, a map file without printing it (by use of the switch /-SP), and a symbol definition file. The PRINT command line tells the system to print the map file.

   ```
   >TKB INV.TSK,INV.MAP/-SP,INV.STB=INV.OBJ RET
   >PRINT INV.MAP RET
   ```

2. With the next command line, you can print the map file directly as it is created. In this case, TKB tells the system to print the file by use of the switch /SP. However, the system task QMGPRT.TSK must be installed as PRT... for this method to work.

   ```
   >TKB INV.TSK,INV.MAP/SP,INV.STB=INV.OBJ RET
   ```

3. With the next command line, you can print the map file on a line printer that you specify. (Because it involves transparent spooling, see your system manager for specific details about using this command line.)

   ```
   >TKB INV.TSK,LPn:,SY:INV.STB=INV.OBJ RET
   ```

1.1.1.2 Omitting Specific Output Files

You can omit any output file by replacing the file specification with the delimiting comma that would normally follow it. The following examples illustrate the ways in which TKB interprets the output file names.

Examples

```
>TKB IMG1,IMG1,IMG1=IN1 RET
```

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 RET
```

The task image file is IMG1.TSK.

```
>TKB ,IMG1=IN1 RET
```

The map file is IMG1.MAP.

```
>TKB  ,,IMG1=IN1 RET
```

The symbol definition file is IMG1.STB.

```
>TKB IMG1,,IMG1=IN1 RET
```

The task image file is IMG1.TSK and the symbol definition file is IMG1.STB.

```
>TKB =IN1 RET
```

This is a diagnostic run with no output files.

1.1.2 The DCL LINK Command Line for the Task Builder

The LINK command for TKB has the following DCL form:

```
LINK/[qual]/[NO]TASK[:fspec]/MAP[:fspec]/SYMBOL_TABLE:[fspec] [,fspec[,s]]
```

This is the standard form of the LINK command for the Task Builder used in this manual. Any DCL command line, including the LINK command, has variations in the way it may be used. For possible variations, see the *RSX-11M-PLUS Command Language Manual,* both the Introduction and the section on the LINK command.

The LINK command has many qualifiers and defaults. The qualifiers, which will be discussed as they appear in the manual, correspond to the TKB switches and options listed in Chapters 10 and 12. The LINK qualifiers are listed in Chapter 11.

TKB can produce three different kinds of output files either at separate times or at the same time. These files are the task file (TSK), the map file (MAP), and the symbol definition file (STB). The input files for the LINK command are discussed in the next section. The output files—task, map, and symbol definition—are discussed after the input files.

TKB allows a command line to be a maximum of 132 characters in length.

1.1.2.1 The LINK Command Input File

You may specify only the input file when you build your task with the LINK command. The LINK command then creates, by default, an output file with the same name as the input file. This way you need only specify the input file name, which must be an object file. The default file type for the input file is OBJ. You separate the input file name from the rest of the qualifiers, if any, by a space. For example:

```
$ LINK BUN RET      and      $ LINK BUN.OBJ RET
```

each produce an output task file with the default name BUN and the file type TSK (BUN.TSK). The LINK command expects the input file to have an OBJ file type by default. Therefore, you need not specify .OBJ in the input filespec.

You may specify more than one input file in the LINK command as follows:

```
$ LINK ROLL1,ROLL2,ROLL3 RET
```

This command produces one output task file, which is a combination of the three input files. The output file has the default name ROLL1 and the file type TSK (ROLL1.TSK). LINK uses the first input file name that it encounters as the default output file name.

However, other files, such as library files, will have a different file type that must be specified. To specify a library file as an input file, you can use the following command line:

```
$ LINK COOKIE1,COOKIE2,COOKIE3,MIX4/LIBRARY RET
```

Here, MIX4 is a library file, and three OBJ input files are combined with the library file to produce one task file. A library file has the file type OLB, but this file type need not be specified in the LINK command line. However, the library file must be indicated with the /LIBRARY qualifier. The library file should be specified last in the input file string. If you use a library file, you must use it together with the object file or files that you have coded and want to build with the library. The separate input object files are named here COOKIE1, COOKIE2, and COOKIE3. This example produces the output task file with the default name of COOKIE1 and the file type TSK (COOKIE1.TSK).

Another way to specify a library file, but only use specific routines contained in the library file, is to use the /INCLUDE qualifier. A command line using this qualifier would appear as follows:

```
$ LINK COOKIE1,COOKIE2,COOKIE3,MIX4/INCLUDE:BATCH1,BATCH2 RET
```

This command line would include routines named BATCH1 and BATCH2 from the library named MIX4. When you use /INCLUDE with an input file name, you need not use the /LIBRARY qualifier.

More information about the /LIBRARY and /INCLUDE qualifiers is included in the description of qualifiers in Chapter 11.

1.1.2.2 The LINK Command Task File

The output file of the LINK command is the task file. This file has the file type TSK. The default name of the task file is the same name as that of the input file. For example:

```
$ LINK BUN RET
```

This command line produces an output file called BUN.TSK. By the same process, LINK produces one output file with a TSK file type from multiple input files and uses the name of the first input file encountered in the command line as the name of the output file. For example:

```
$ LINK ROLL,BUN,CROISSANT RET
```

This command line produces an output file called ROLL.TSK.

To give the output file any name you want, you must use the /TASK qualifier on the LINK command. For example:

```
$ LINK/TASK:BREAD ROLL,BUN,CROISSANT RET
```

This command line produces an output file named BREAD with the file type TSK from the three input files ROLL, BUN, and CROISSANT.

You may or may not want a TSK file as output. An example of not wanting a task file would occur when you wanted to see only a MAP file for a task (MAP file output is discussed in the next section), or you wanted to see if TKB would actually build without errors the files that you had specified. You can notify LINK that you do not want a TSK file by using the /NOTASK qualifier specified as follows:

```
$ LINK/NOTASK ROLL,BUN,CROISSANT RET
```

Here, TKB goes through the building process but does not produce any output.

1.1.2.3 The LINK Command Map File

In addition to the task file, you can use the LINK command to produce a map file for the task. The map file has a MAP file type. The map file contains the addresses and symbols used by your task and it describes their relationship. The LINK command will produce this file only if you specify that it do so. For example:

```
$ LINK/MAP CHIP,OAT,FLOUR RET
```

This command line produces a task file with the default name of CHIP and a map file with the default name of CHIP. CHIP is the name of the first input file.

However, you may name specifically the task file and let the map file default to the name of the first input file, as before. You can do this with the following two variations of the LINK command:

```
$ LINK/TASK:COOKIE/MAP CHIP,OAT,FLOUR RET
$ LINK/MAP/TASK:COOKIE CHIP,OAT,FLOUR RET
```

To name specifically the map file, you must use a file name after the /MAP qualifier. You can do this by entering either of the following two variations of the LINK command:

```
$ LINK/TASK:COOKIE/MAP:COOKIE CHIP,OAT,FLOUR RET
$ LINK/MAP:COOKIE/TASK:COOKIE CHIP,OAT,FLOUR RET
```

These latter two variations produce a task file called COOKIE.TSK and a map file called COOKIE.MAP.

There are other qualifiers that produce a MAP file. These qualifiers are /[NO]SYSTEM_LIBRARY_DISPLAY, /[NO]CROSS_REFERENCE, /[NO]WIDE, and /LONG. Chapter 11 explains the operation of these qualifiers.

1.1.2.4 The LINK Command Symbol Definition File

Another file can be produced by the LINK command. This file is called the symbol definition file and it has the file type STB. This file contains the symbols used or referenced by the input files. TKB uses this file when you use libraries, commons, and overlays as part of your task. Libraries and commons are discussed in Chapter 5, and overlays are discussed in Chapters 3 and 4.

To create a symbol definition file for your task, you must specifically notify the LINK command that you want to do so. For example:

```
$ LINK/TASK:COOKT/MAP:COOKM/SYMBOL_TABLE:COOKS CHIP,OAT,FLOUR RET
```

This command line produces three files: the task file COOKT.TSK, the map file COOKM.MAP, and the symbol definition file COOKS.STB.

By default, the LINK command uses the name of the first input file to create the name of the symbol definition file. For example:

```
$ LINK/TASK:COOKT/MAP:COOKM/SYMBOL_TABLE CHIP,OAT,FLOUR RET
```

This command line produces a symbol definition file called CHIP.STB.

1.1.2.5 Printing the MAP File When Using the LINK Command

Automatic printing of your MAP file may occur if your system has the system task QMGPRT.TSK installed with the PRT... name. Otherwise, the MAP file is created in your directory or the directory you specified in the LINK command line and is not immediately printed. From there, you may print it later by methods specific to your own system type or configuration.

If you use /MAP as a command qualifier, without a filespec argument, TKB puts the map in your directory with the file name of the first input file encountered. For example:

```
$ LINK/MAP CHIP,OAT,FLOUR RET
```

In this example, the name of the map file is CHIP.MAP. This file is printed if the PRT... task is installed.

If you use /MAP with a filespec argument, either on an input file or as the LINK command qualifier, TKB puts the map in your directory with the name you have specified in the filespec argument. For example:

```
$ LINK/MAP:COOKIE/TASK:COOKIE  CHIP,OAT,FLOUR RET
$ LINK/TASK:COOKIE CHIP/MAP:COOKIE,OAT,FLOUR RET
```

In these two examples, the map files are named COOKIE.MAP.

If you use /MAP as an input filespec qualifier, but do not give a specific name for the map file, TKB places the map file in your directory with the name of the file to which /MAP is attached. For example:

```
$ LINK/TASK:COOKIE CHIP/MAP,OAT,FLOUR RET
```

In this example, the map file is named CHIP.MAP.

TKB always tries to spool the map file to the printer. TKB will succeed in doing this if the system task QMGPRT.TSK is installed with the PRT... name. To prevent spooling, use the /NOPRINTER qualifier with the /MAP qualifier.

1.2 Multiline 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).

1.2.1 Multiline Input Using the TKB Command

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

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

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

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

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 (//), TKB stops accepting input, builds the task, and returns control to MCR.

1.2.2 Multiline Input Using the LINK Command

The LINK command can get very long when you use many qualifiers. One way to shorten the command line is to use the hyphen (which is the continuation character) at a logical point in the command, thus terminating the individual line at that point. For example:

```
$ LINK- RET
->/TASK:COOKIE/MAP:CRUNCH- RET
->/SYMBOL_TABLE:CRUMB CHIP,RAISIN,OAT,FLOUR RET
$
```

Or, you can do it as follows:

```
$ LINK- RET
->/TASK:COOKIE/MAP:CRUNCH/SYMBOL_TABLE:CRUMB - RET
->CHIP,RAISIN,NUT,SUGAR,OAT,FLOUR,SALT,SODA RET
$
```

Notice the space after CRUMB and before the hyphen. This space is the separation between the qualifiers and the input file specifications. It must be present whether or not you use the hyphen.

1.2.2.1 Abbreviated Qualifiers in LINK

To shorten the length of a command line, you can use an abbreviated qualifier, such as SYM for SYMBOL_TABLE. The previous command sequence could look like the following one if you use the hyphen and abbreviated qualifiers:

```
$ LINK- RET
->TAS:COOKIE/MA:CRUNCH/SYM:CRUMB CHIP,RAISIN,OAT,FLOUR RET
$
```

All the qualifiers for the LINK command can be abbreviated to some extent. The following is a sample list of abbreviations for frequently used qualifiers:

LONG FORM	SHORT FORM
/ANCILLARY_PROCESSOR	/ANC
/NOCHECKPOINT:arg	/NOCH:arg
/CHECKPOINT:arg	/CHEC:arg
/CODE:arg	/COD:arg
/NOHEADER	/NOHE
/HEADER	/HEAD or/HEA
/INCLUDE:modulename,...,	/INC:modulename,...,
/LIBRARY	/LIB
/MAP:filespec	/MA:filespec
/OPTIONS:option	/OPT:option

/OVERLAY_DESCRIPTION	/OVER
/SHAREABLE:arg	/SHARE:arg
/SYMBOL_TABLE:filespec	/SYM:filespec
/NOTASK:filespec	/NOT:filespec
/TASK:filespec	/TAS:filespec

However, be careful that you use abbreviations that DCL can recognize as unique. For example, the two qualifiers /SEQUENTIAL and /SEGREGATE can be abbreviated to /SEQ and /SEG, but not to /SE and /SE.

1.3 Task Builder Options

The Task Builder uses many options to control the way in which a task is built. Section 1.3.1 discusses entering these options in TKB if your system uses MCR as the command line interpreter. Section 1.3.2 discusses entering these options in LINK if your system uses DCL as the command line interpreter. Section 1.3.3 discusses specific methods that you may use or circumstances that you may encounter when entering these options.

1.3.1 Entering Task Builder Options in TKB

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, TKB requests option information by displaying "Enter Options:" and prompting for input. For example:

```
>TKB RET
TKB>IMG1,IMG1=IN1 RET
TKB>IN2,IN3 RET
TKB/ RET
Enter Options:
TKB>PRI=100 RET
TKB>COMMON=JRNAL:RO RET
TKB>// RET
>
```

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 (/). See the discussion of the /MP switch in Chapter 10.

1.3.2 Entering Task Builder Options in LINK

If you want to use Task Builder options, you must use the LINK command qualifier /OPTIONS in the LINK command line. After DCL reads the command line, it prompts you for the option or options. Enter each option after the prompt, and then press the RETURN key after each option. To end option input, you press the RETURN key after the option prompt. For example:

```
$ LINK/TASK:COOKIE/MAP:COOKIEM/OPTIONS CHIP,OAT,SUGAR,FLOUR  RET
Option? PRI=100  RET
Option? COMMON=JRNAL:RO  RET
Option?  RET
$
```

In this command sequence there are two options, PRI and COMMON. The RETURN key is pressed after the third option prompt. You may use the hyphen in the LINK command line to provide line continuation. The hyphen does not interfere with option input.

Alternatively, you can use a filespec on the /OPTION qualifier to designate a file that contains the options that you want to use. For example:

```
$ LINK/TASK:COOKIE/MAP:COOKIEM/OPTIONS:filespec CHIP,OAT,SUGAR,FLOUR  RET
$
```

The file named in filespec can have any name you want but must have the file type CMD. It must contain the options in a list, with each option on a single line. This file cannot contain any slash characters (/). The file would look like the following:

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

1.3.3 Entering the Option Line

The Task Builder provides numerous options, which are described in Chapter 12. The general form of an option is the name of the option followed by an equal sign (=) and an argument list. The arguments in the list are separated from one another by a colon (:). In the examples in Sections 1.3.1 and 1.3.2, 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, the following TKB command line:

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

is equivalent to the following two command lines:

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

In a similar way, the following LINK command line:

```
Option? PRI=100!COMMON=JRNAL:RO  RET
```

is equivalent to the following two command lines:

```
Option? PRI=100  RET
Option? COMMON=JRNAL:RO  RET
```

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

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

or

```
Option? COMMON=JRNAL:RO,RFIL:RW  RET
```

In these TKB and LINK command lines, 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 command sequences for TKB are equivalent:

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

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

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

Similarly, the following three command sequences for LINK are equivalent:

```
Option? COMMON=JRNAL:RO,RFIL:RW  RET

Option? COMMON=JRNAL:RO!COMMON=RFIL:RW  RET

Option? COMMON=JRNAL:RO  RET
Option? COMMON=RFIL:RW  RET
```

1.4 Multiple Task Specifications

For MCR, if you intend to build more than one task, you can use the single slash (/) following option input. This directs TKB to stop accepting input, build the task, and request information for the next task build. For example:

```
>TKB  RET
TKB>IMG1=IN1  RET
TKB>IN2,IN3  RET
TKB>/  RET
Enter Options:
TKB>PRI=100  RET
TKB>COMMON=JRNAL:RO  RET
TKB>/  RET
TKB>IMG2=SUB1  RET
TKB>//  RET
```

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

For DCL, there is no way to enter multiple task specifications with a single LINK command.

1.5 Indirect Command Files

You can enter commands to TKB directly from the terminal, or indirectly through the indirect command file facility (Indirect). To use Indirect, you prepare a file that contains the TKB commands you want to be executed. Later, after you invoke TKB, 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 information:

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

Later, you can type one of the following command sequences:

```
   TKB                 LINK

>TKB                 $ LINK
TKB>@AFIL            File(s)?@AFIL
TKB>                 $
```

or simply:

```
   TKB                 LINK

>TKB @AFIL           $ LINK @AFIL
```

Note

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

If you use DCL, it passes the indirect command file to TKB. When TKB encounters the at sign (@), it directs its search for commands to the file named AFIL.CMD.

The preceding example is equivalent to the following TKB command sequence:

```
>TKB [RET]
TKB>IMG1,IMG1=IN1 [RET]
TKB>IN2,IN3 [RET]
TKB>/ [RET]
Enter Options:
TKB>PRI=100 [RET]
TKB>COMMON=JRNAL:RO [RET]
TKB>// [RET]
>
```

The example is also equivalent to the following LINK command sequence:

```
$ LINK/TASK:IMG1/MAP:IMG1/OPTION IN1,- [RET]
->IN2,IN3 [RET]
Option? PRI=100
Option? COMMON=JRNAL:RO [RET]
Option? [RET]
$
```

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

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

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

Later, you can type the following command lines:

```
>TKB [RET]
TKB>@AFIL [RET]
```

In this case, TKB goes to the terminal and displays the following prompts:

```
Enter Options:
TKB>
```

From this point, you input options to TKB directly from the terminal. If you then conclude option input from the terminal with double slashes (//), TKB suspends command processing, as described above, and exits to MCR following the task build. If you conclude option input with a single slash (/), TKB 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 the following information:

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

The second, AFIL1.CMD, contains the following information:

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

Then, type the following command sequence at your terminal:

```
>TKB [RET]
TKB>@AFIL [RET]
TKB>@AFIL1 [RET]
>
```

TKB 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 as follows:

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

To build these files, you would type the following command lines:

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

If BFIL.CMD contains the following options:

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

then the terminal equivalent of building these files would be as follows:

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

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 TKB would report an error.

1.6 Comments in Indirect Command Files

For TKB or LINK, you can include comments at any point in the indirect command file, 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, as follows:

```
;
; 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

TKB adheres to the standard RSX–11M–PLUS and Micro/RSX conventions for file specifications. For any file, you can specify the device, the directory, the file name, the file type, the file version number, and any number of switches.

The file specification has the following form:

```
device:[directory]filename.type;version/sw1/sw2.../swn
```

When you specify files by name only, TKB applies the default switch settings for device, directory, type, and version.

For example:

```
   TKB                     LINK

>TKB                    $ LINK
TKB>IMG1,IMG1=IN1       File(s)?/TASK:IMG1/MAP:IMG1 IN1,-
TKB>IN2,IN3             File(s)?IN2,IN3
TKB>//                  $
>
```

If the default directory of the terminal from which TKB is running is [200,200], the task image file specification of the example is assumed to be the following:

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

That is, TKB creates the task image file on the system device (SY0) in directory [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 10.

The following TKB and LINK examples show how defaults are applied:

```
>TKB [RET]

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

$ LINK/TASK:[20,23]IMG1/CHECK:SYS/DEB/MAP:IMG1/CROSS IN1,- [RET]
->IN2;3,IN3 [RET]
$
```

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

The system device is always the default device unless you specify otherwise. If you specify another device on either side of the equal sign (or the space in LINK), that device becomes the default device for the files on that side of the equal sign (or space).

For example:

```
>TKB [RET]
TKB>[20,23]IMG1,IMG1,IMG1=DB1:IMG1,IN1,IN2 [RET]

$ LINK/TASK:[20,23]IMG1/MAP:IMG1/SYM:IMG1 DB1:IMG1,IN1,IN2 [RET]
$
```

This command line produces a task image file, map file, and listing file in directory [20,23] on device SY. All the object files are in directory [20,23] on device DB1. In cases where files are scattered among several devices, the devices must be specified in the command line.

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

For TKB format in MCR, this example also uses the switches /CP, /CR, and /DA, and uses the LINK command qualifiers /CHECKPOINT:SYSTEM, /DEBUG, /CROSS_REFERENCE, and /SYMBOL_TABLE. The syntax and meaning for each switch and qualifier are given in Chapters 10 and 11.

1.8 Summary of Syntax Rules

The syntax rules for issuing commands to TKB are as follows:

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

```
   TKB                          LINK

>TKB task-command-line        $ LINK command-line
```

  The second form has additional lines for input file names:

```
   TKB                          LINK

>TKB                          $ LINK
TKB>task-command-line         File(s)? /TASK:..... -
TKB>input-line                File(s)? INFILE1,INFILE2,...
       .                                 .
       .                                 .
       .                                 .
TKB>terminating-symbol                  [RET]
>                             $
```

  The third form allows you to specify options:

```
   TKB                          LINK

>TKB                          $ LINK
TKB>task-command-l            File(s)? /TAS:.../OPT INFILE1,...
TKB>/                         Options? . . .
Enter Options:                Options? [RET]
TKB>option-line               $
       .
       .
       .
TKB>terminating-symbol
>
```

  The fourth form has both input lines and option lines:

```
   TKB                          LINK

>TKB                          $ LINK
TKB>task-command-line         File(s)? /TAS:.../MAP:.../OPT -
TKB>input-line                File(s)? INFILE1,...
       .                      Option? option-line
       .                      Option? [RET]
       .                      $
TKB>/
Enter Options:
TKB>option-line
       .
       .
       .
TKB>terminating-symbol
>
```

For TKB in MCR or in indirect command files, the terminating symbol is one of the following:

/	If you intend to build more than one task
//	If you want TKB to return control to MCR

For LINK, the normal terminating symbol in command or option input is the RETURN key. However, pressing CTRL/Z will end the command without any execution by TKB. If you have specified an indirect command file for input to LINK, the terminating symbol in the indirect command file is the end-of-file if it has no options, or the double slash (//) if it has options.

- A Task Builder command line has one of the following forms:

TKB	LINK
output-file-list=input-file,...	output/qual input/qual
=input-file,...	space input-file
@indirect-command-file	@indirect-command-file

The third form in the previous list is an indirect command file specification, as described in Section 1.5.

- A TKB output file list has one of the following 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 specifications can be omitted, so that, for example, the following form is permitted:

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

- An input line has one of two forms:

```
  TKB                          LINK

=input-file,...              space  input-file,...

@indirect-command-file       @indirect-command-line
```

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

- An option line has one of two forms:

```
  TKB                          LINK

option!...                   Option?option-line

@indirect-command-file       Option?@indirect-command-file.
```

The indirect-command-file is a file specification.

- An option has the form:

  ```
  keyword=argument-list,...
  ```

  The argument-list consists of the following:

  ```
  arg:...
  ```

  The syntax for each option is given in Chapter 12.

- A file specification conforms to standard RSX–11M–PLUS and Micro/RSX conventions. It has the form:

  ```
  device:[directory]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:. It can also be a logical name if extended logical name support is allowed on your system.
directory	The directory may have a name of up to nine alphabetic or numeric characters, such as [MYLETTERS] or [400578369] or [DIRECT424]. You may use fewer letters or numbers if you want. The default directory is the directory you are assigned when you log in on the system. The combination of the group number and the member number is the User File Directory (directory) that contains the file name.
filename	The name of the desired file. The file name can contain up to nine alphanumeric characters.
type	The 3-character file type identification. 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 decimal on Micro/RSX systems or in octal on RSX–11M–PLUS systems, 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.

Chapter 2
Task Builder Functions

The process of building a task involves the following distinct Task Builder (TKB) functions:

- Linking object modules
- Assigning addresses to the task image
- Building data structures into the task

First, TKB 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, TKB assigns addresses to the task image. On mapped systems, TKB assigns addresses for a task beginning at address 0. The Executive then relocates the addresses at run time.

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

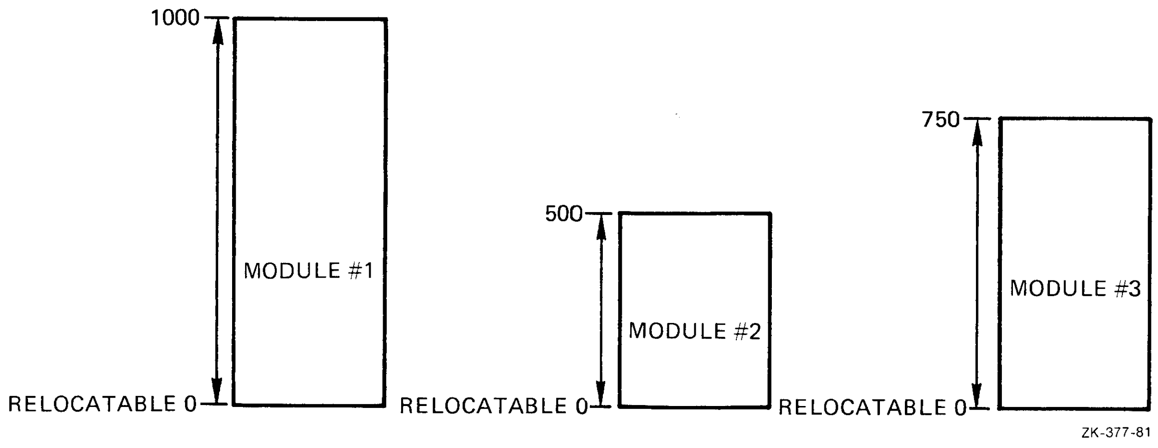
This chapter describes the three TKB functions in detail. It also describes the concepts of mapped systems. In addition, this chapter introduces regions, supervisor-mode libraries, overlays, privileged tasks, I- and D-space tasks, and many of the mapping concepts necessary for an understanding of task mapping and Task Builder functions.

2.1 Linking Object Modules

TKB links object modules within the context of program sections and resolves references to global symbols across module boundaries.

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 its 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.

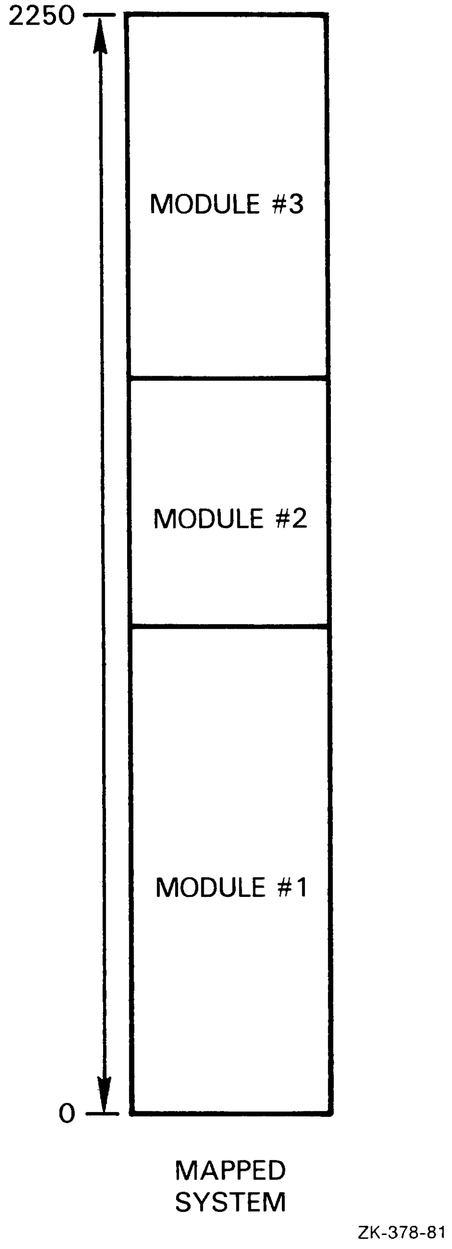
Figure 2–1: Relocatable Object Modules



If these modules represent the separate modules of a single program, TKB links them together and, for a mapped system, modifies the provisional addresses to a single sequence of addresses beginning at 0 and extending upward to the sum of the lengths of all the modules (-1 byte).

For example, Figure 2–2 shows the three modules linked for a mapped system.

Figure 2–2: Modules Linked for Mapped System



2.1.1 Allocating Program Sections

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

- A name
- A set of attributes
- A length

A program section is the basic unit used by TKB 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 TKB 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 by allowing the language translator to create them. 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 TKB reference it. When processing files, both the language translator and TKB create internal tables that contain program section names, attributes, and lengths. A named program section can be declared more than once. However, all occurrences of that named program section must have identical attributes if the section occurs more than once in the same module or if the section is a global program section. Identically named program sections within the same module and global program sections with differing attributes cause TKB to declare the program section as having multiple attributes, which is an error. However, identically named program sections with differing attributes may appear in different trees of an overlaid task if the program sections have the local (LCL) attribute.

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 TKB must reserve for it.

When a task 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 TKB 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 TKB 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: can be read from, but cannot be written into, the program section.
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.
save	SAV	The program section has the SAVE attribute, and TKB forces the program section into the root.
scope-code	GBL	Global: the program section name is recognized across overlay segment boundaries; TKB 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; TKB allocates storage for the program section from references within the defining overlay segment only.
type-code	D	Data: the program section contains data.
	I	Instruction: the program section contains either instructions, or data and instructions.

2.1.1.1 Access-Code and Allocation-Code

TKB uses a program section's access-code and allocation-code attributes to determine its placement and size in a task image. If you specify /SG (or /SEGREGATE in LINK) in the command sequence, TKB divides address space into read/write and read-only areas, and places the program sections in the appropriate area according to access code. However, the default is to order the program sections alphabetically.

TKB uses a program section's allocation code to determine its starting address and length. If a program section's allocation code indicates that TKB is to overlay it (OVR), TKB places each allocation to the program section from each module at the same address within the task image. TKB 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 TKB is to concatenate it (CON), TKB 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.

TKB 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, TKB 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, TKB 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 that each contains and their access codes and allocation codes.

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 occurrence; 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 2–2: Program Sections for Modules IN1, IN2, and IN3

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

Table 2–3: Individual Program Section Allocations

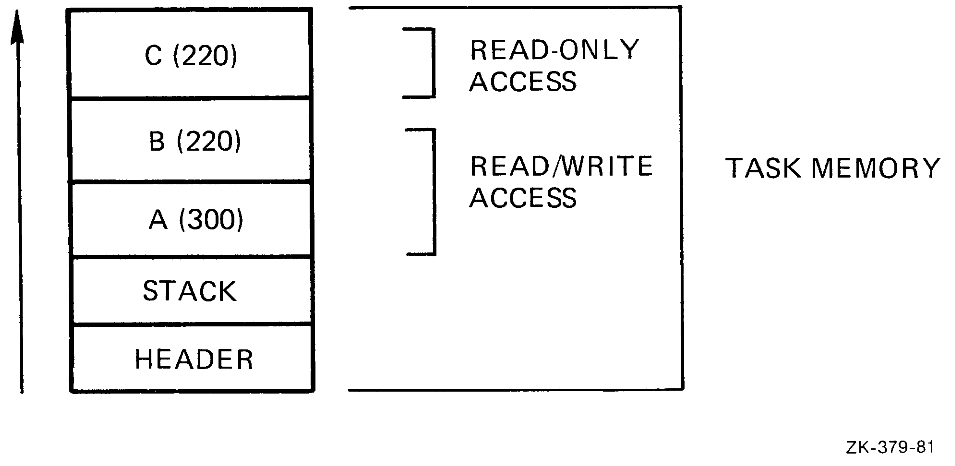
Program Section Name	Total Allocation
B	220
A	300
C	220

TKB 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 allocation of program sections if the /SG or /MU switch (or, for LINK, the /SEGREGATE or /SHAREABLE:task qualifier) is used. For more information, see the description of the /MU, /SQ, and /SG switches in Chapter 10 and the /SHAREABLE:TASK, /SEQUENTIAL, and /SEGREGATE qualifiers in Chapter 11.

Figure 2–3: Allocation of Task Memory



The save attribute (SAV) is useful in cases where the information in a program section must be kept available to all task segments. The SAV attribute of a program section causes TKB to force the program section into the root of an overlaid task. Therefore, the named common block in the FORTRAN SAVE statement or the named program section in the MACRO–11 .PSECT directive specified with the SAV attribute are in the root of the task.

When the ODL .PSECT directive or the SAV attribute is used to force a program section into a different segment, the global symbol definitions for the program section remain in the segment where the program section was defined. This means that when a program section is moved from the root to an overlay using the .PSECT directive, TKB does not generate an autoload vector for the entry point.

This enables you to separate the location of global symbol definitions from the location of code. Relocating global symbol definitions can be achieved by simply moving the program section manually to the desired segment.

To autoload a segment that contains a program section that was forced out of the root, place the segment in another segment that is autoloaded by way of an entry point not in the program section that was relocated.

2.1.1.2 Type-Code and Scope-Code

The scope-code attribute is meaningful only when you define an overlay structure for a task. (The scope code is described in Chapters 3 and 4 within the context of the descriptions of overlays.) The type-code attribute is meaningful in the context of program sections within an I- and D-space task (as described in Chapter 7).

2.1.2 Resolving Global Symbols

TKB 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 TKB links the relocatable object modules, it creates an internal table of the global symbols it encounters within each module. If, after TKB examines and links all the object modules, references remain to symbols that have not been defined, TKB 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, TKB flags the symbols as undefined. If you have not specified an output map in your TKB command sequence, TKB reports the names of the undefined symbols to you on your terminal. If you have specified an output map, TKB 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, TKB 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	B1	A
		B2	L1
	A		C1
			XXX
	C		
IN2	A	A	
	B	B1	B2
IN3	C		B1

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

When all the object files have been processed, TKB 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 TKB includes the defining modules in the task's image. Assume also that TKB cannot resolve the global symbol C1. TKB lists it as an undefined global symbol.

The relocatable global symbol B1 is defined twice. TKB lists it as a multiply defined global symbol. In this case, TKB uses the first definition for the global 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 The Task Structure

TKB builds the data structures required by other system programs and incorporates them into the task image. The Executive (which is 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. For example, TKB allocates space for the task header and initializes it.

The disk image file created by TKB 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 the following three physically contiguous parts:

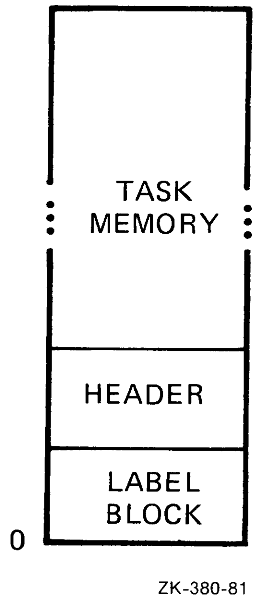
- The label block group
- The task header
- The task memory image

Figure 2–4 illustrates the basic simplified structure of this file.

The label block group contains data produced by TKB and used by INSTALL command processing. It contains information about the task, such as the task's name, the partition in which it runs, its size and priority, and the logical units assigned to it. When you install the task, INSTALL command processing (hereinafter called INSTALL) uses this information to create a Task Control Block (TCB) entry for the task in the System Task Directory (STD) 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. TKB creates and partially initializes the header; INSTALL initializes the rest of the header.

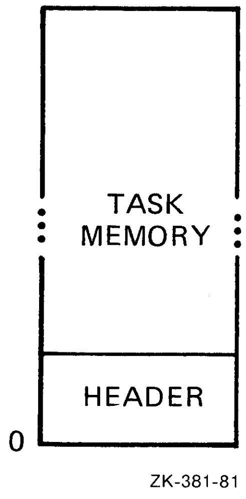
Figure 2–4: Disk Image of the Task



The task memory image 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 image are described in detail in Appendix B.

The task's memory image is the part of your task that the system reads into physical memory at run time. 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 2–5 shows the memory image.

Figure 2-5: Memory Image



2.3 Overlays

This section is an introduction to overlaid tasks. Details about overlaid tasks can be found in Chapters 3 and 4.

Using overlays can save memory space by reducing the size of the executing portion of the task or the physical memory required by the task. Parts of an overlaid task reside on disk, thereby saving memory space.

An overlaid task is a task designed to have discrete parts. The parts of a task designed this way can execute relatively independently of other parts. Parts of an overlaid task reside on disk until they are needed for their required function. The common part of the task, which stays in memory, is the root. The root calls the other parts of the task, which are referred to as segments, from disk into memory.

The RSX-11M-PLUS and Micro/RSX operating systems have two types of overlaid tasks. One type of overlaid task reads in segments from disk over other segments already in memory. A task of this type is called a disk-resident overlaid task. In this task, segments reside on disk until they are needed. The segments in disk-resident overlays that share the same memory address space of the task with other segments must be logically independent of those segments. The independence is necessary because the other segments are on disk and cannot be referenced. For example, Task A, an overlaid task root, can call either of two segments: segment B or segment C. The root of Task A initially calls segment B. Segments B and C occupy the same memory space. Segment B cannot call segment C and segment C cannot call segment B. However, if segment B returns control of the task to the root of task A, the root can then call segment C. Segment C would then be read into memory over segment B. Figure 2-6 illustrates this sequence.

Because segments of a disk-resident overlaid task can occupy the same memory space, a disk-overlaid task can occupy less memory than it would if it were not overlaid. However, more disk I/O transfers (and, therefore, more time) are needed for this type of task.

Another type of overlaid task is the memory-resident overlaid task. In this task, the segments reside on disk until they are needed. At that time, the needed segment is read into a sequentially adjacent area of memory and resides there until the task ends. For example, a memory-resident overlaid Task A has two segments: segment B and segment C. If the root of task A calls segment B, segment B is read into memory adjacent to the root. When the root regains control and then calls segment C, segment C is read into memory adjacent to segment B. Figure 2–7 illustrates this sequence.

Memory-resident overlaid tasks execute faster than disk-resident overlaid tasks. The increase in speed occurs because fewer disk I/O transfers are needed during task execution.

2.4 Addressing Concepts

The primary addressing mechanism of the PDP–11 computer is the 16-bit 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.

2.4.1 Physical, Virtual, and Logical Addresses

Physical, virtual, and logical addresses, and virtual and logical address space, are concepts that provide a basis for understanding the functions of task addressing and the use of task windows. These concepts are described as follows:

- Physical addresses—A single, physical location in memory is called the physical address.

  Memory is divided into parts called bytes. They are numbered according to their position in memory. Therefore, the lowest byte is 0 and the highest byte is whatever the upper limit of memory may be for a particular system; for example, 32K, 64K, and so forth. The assigned number is called the physical address.

  A task contains addresses (for example, 0 through 2200). TKB relocates the task's addresses in an unmapped system by a number represented by the base address of the partition in which it is installed. After installation, the task's addresses refer to physical addresses of memory, which always correspond to the same physical memory in an unmapped system.

  Therefore, the task addresses have an actual one-to-one relationship to physical memory. The same relationship exists any time the task is in memory. The memory (physical) addresses will not be from 0 through 2200. For example, after the task is installed in the partition, the task's address of 0 may become physical address 17000 because the Task Builder added in the offset, which is equal to the partition base address.

  In a mapped system, the task's addresses remain the same but the physical memory addresses may change due to Executive processes (checkpointing, swapping, and so forth). Therefore, the task addresses do not always correspond to the same physical memory. If the task uses memory management directives, the memory addressing can be changed by the task to include any part of physical memory that it is allowed to access.

Figure 2–6: Simple 2-Segment, Disk-Resident Overlay Calling Sequence

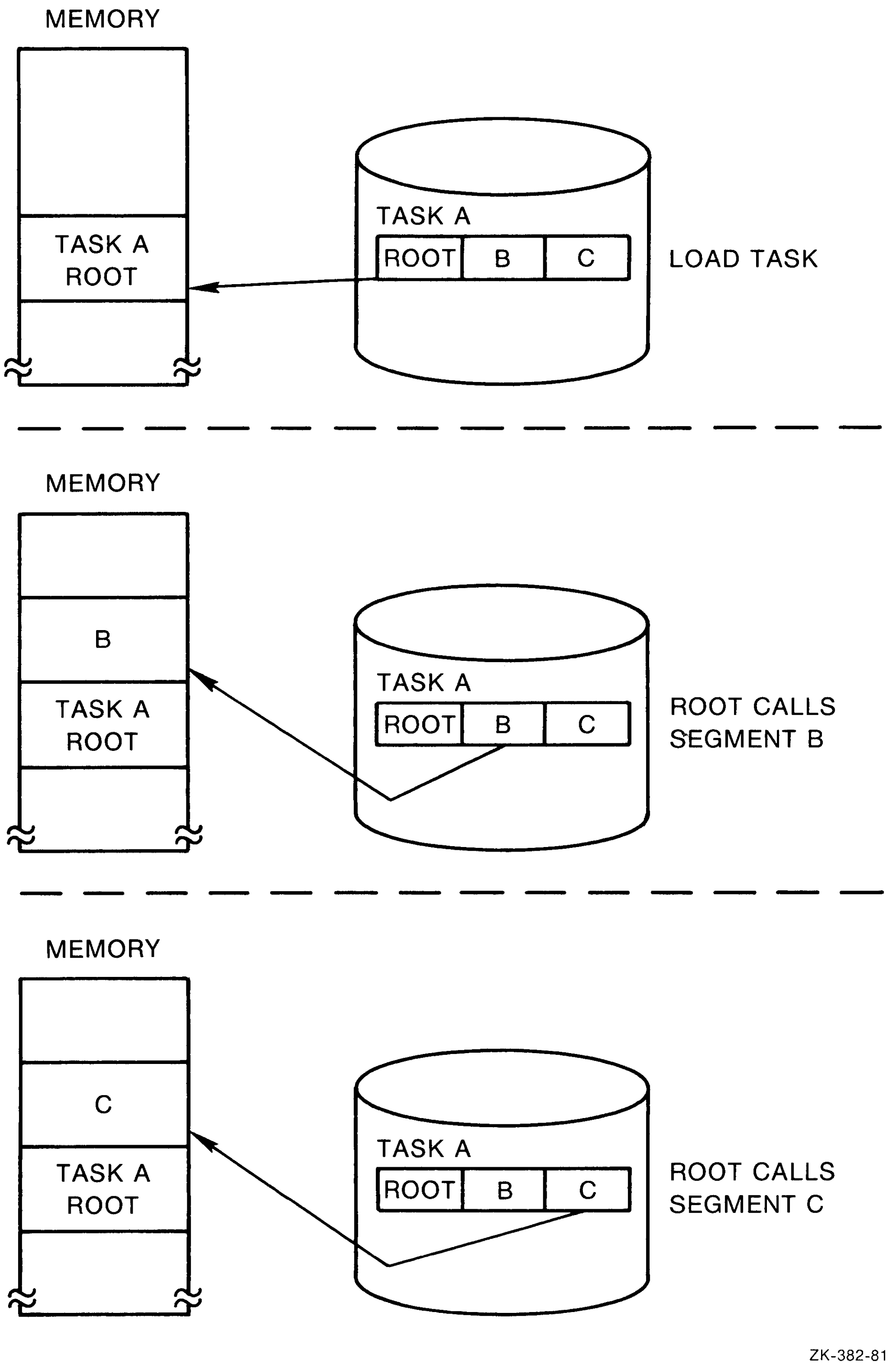
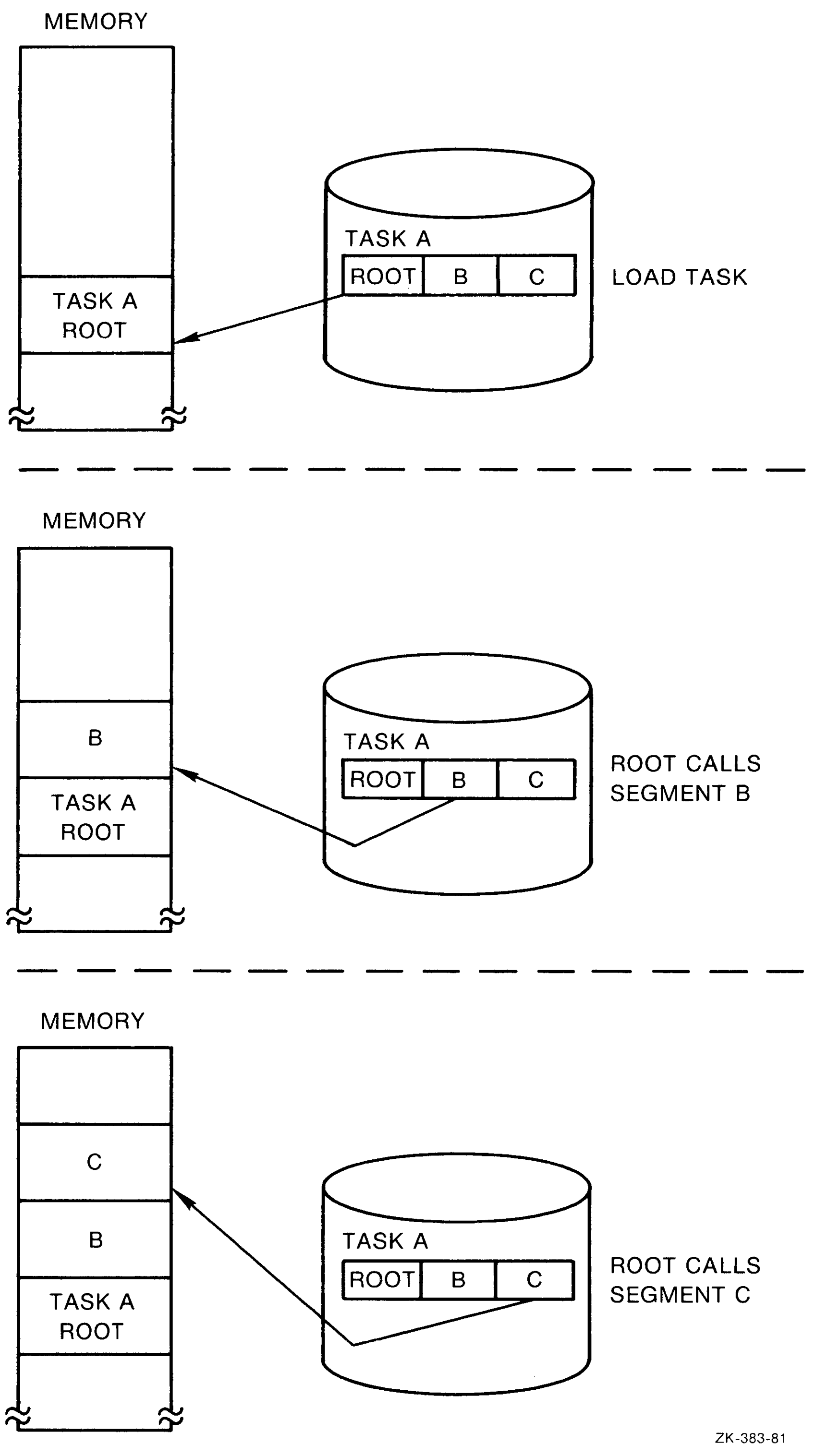


Figure 2–7: Simple 2-Segment, Memory-Resident Overlay Calling Sequence



- Virtual addresses—A task's virtual addresses are the addresses within the task.

  The PDP–11's 16-bit word length (a mapped system) imposes the address range of 32K words on the virtual addresses. Therefore, these task addresses could include addresses 0 through 177777_8 depending on the length of the task. These task addresses are not the same as the actual addresses of the memory in which the task resides.

- Virtual address space—A task's virtual address space is that space encompassed by the range of virtual addresses that the task uses.

  With the Create Address Window (CRAW$) memory management directive, a task can divide its virtual address space into segments called virtual address windows. By using address windows, you can manipulate the mapping of virtual addresses to different areas of physical memory.

- Logical addresses—A task's logical addresses are the actual physical memory addresses that the task can access.

- Logical address space—The task's logical address space is the total amount of physical memory to which the task has access rights.

  The physical memory represented by the logical addresses may or may not be continuous. The items in physical memory that logical address space includes are the task itself, and static and dynamic regions.

2.4.2 Mapped Systems

A mapped system is one in which the processor contains a KT-11 memory management unit. The processor handbook for your machine contains a complete description of the memory management unit.

Mapped processors have up to three modes of operation: kernel, supervisor, and user (the PDP–11/34 does not have supervisor mode). The information in this section is relevant to user mode only.

The primary addressing mechanism for a mapped system is still the 16-bit word, and virtual address space is still 32K words. However, due to the extent of the physical memory capacity in a mapped system, 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 22 bits. When TKB links the relocatable object modules of a task that is to run on a mapped system, it assigns 16-bit addresses 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 22-bit physical addresses. The mechanical job of task relocation is performed by the Executive and the memory management unit at task run time. Figure 2–8 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 2–9 shows how the memory management unit divides the 32K words of virtual address space.

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 2–10 illustrates this mapping.

Unless a task is privileged, the I/O page and the Executive are not normally part of a task's virtual address space, and 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 approximately 32,767 words long (32K minus 32 words needed by the loader) on a mapped system. TKB can build a task of 32K minus 1 word in size. However, overlaid tasks, and tasks that become extended, may use the entire 32K-word space.

Figure 2–8: Task Relocation in a Mapped System

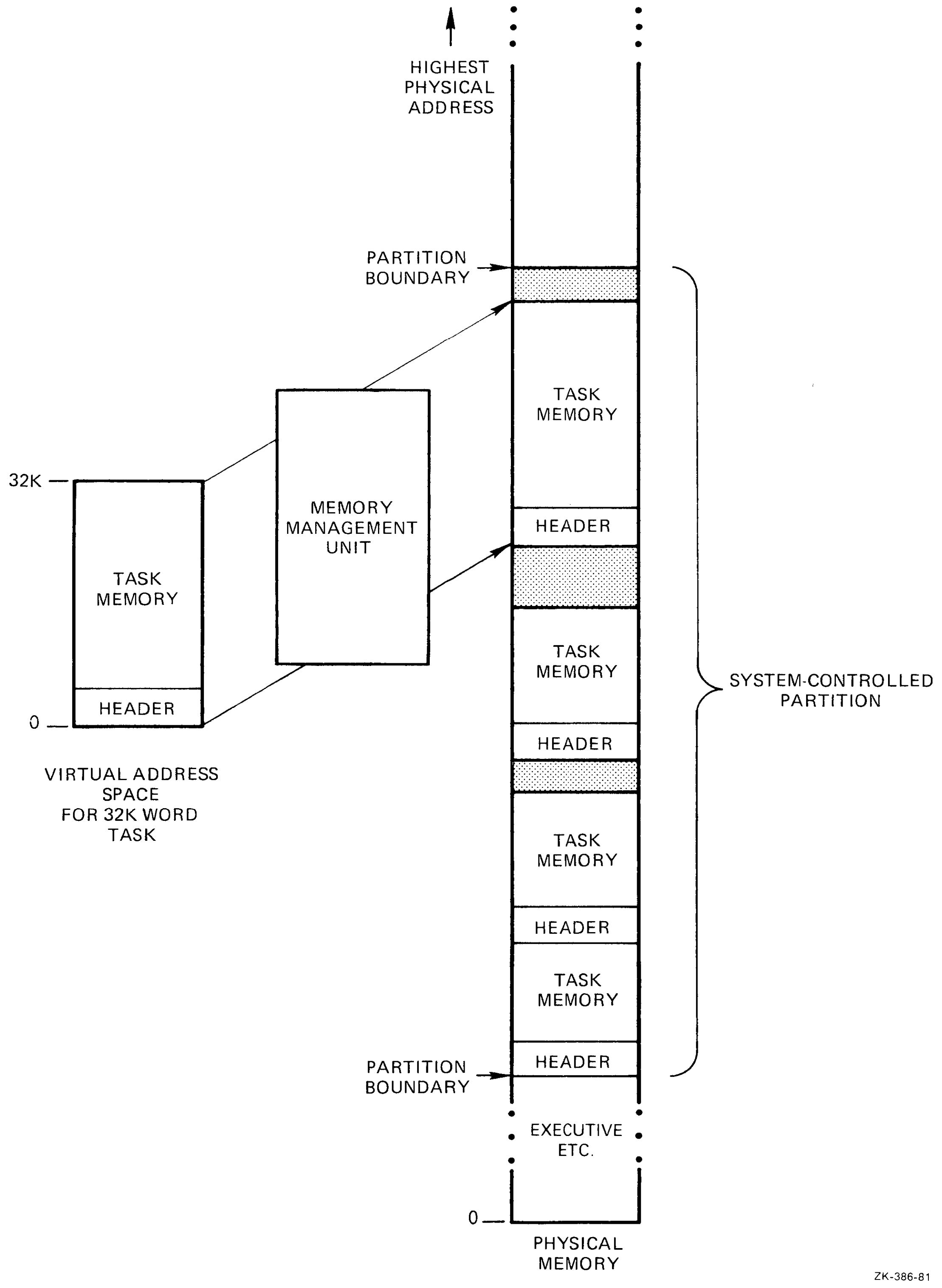
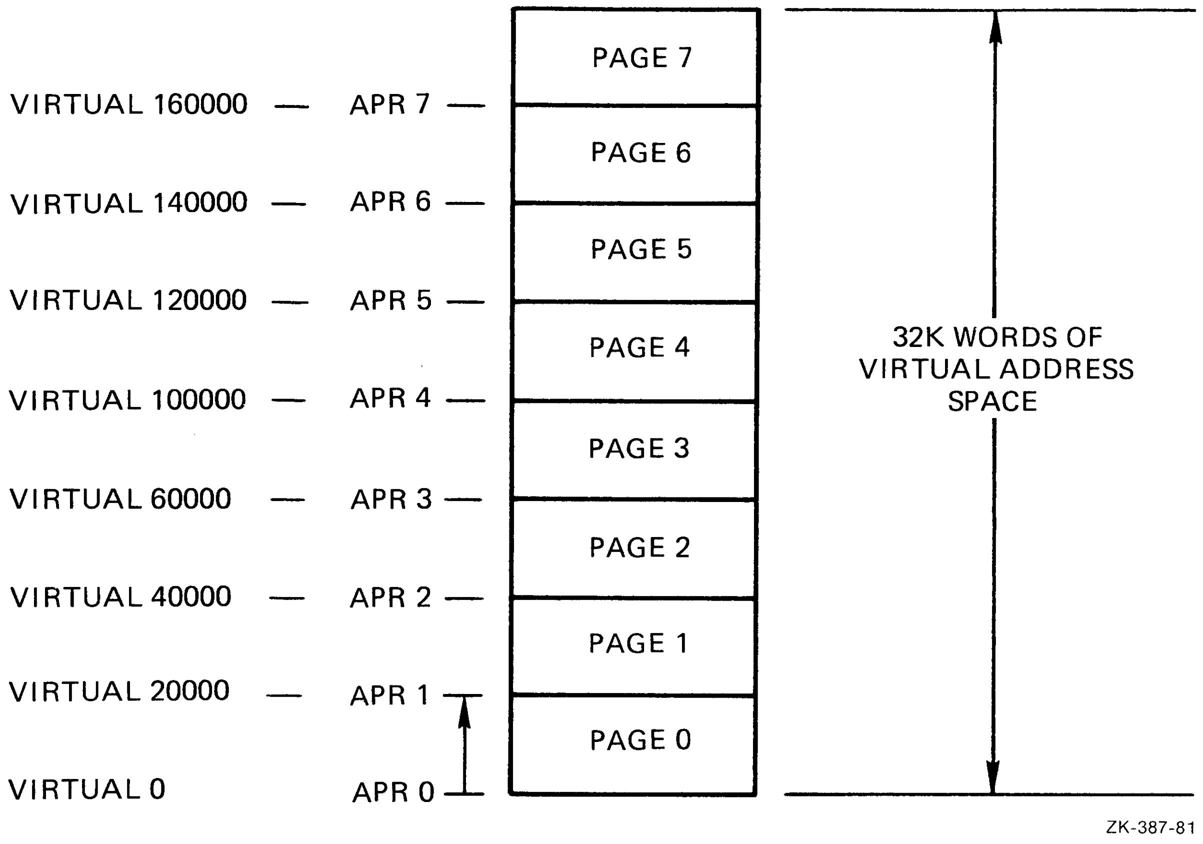


Figure 2-9: Memory Management Unit's Division of Virtual Address Space

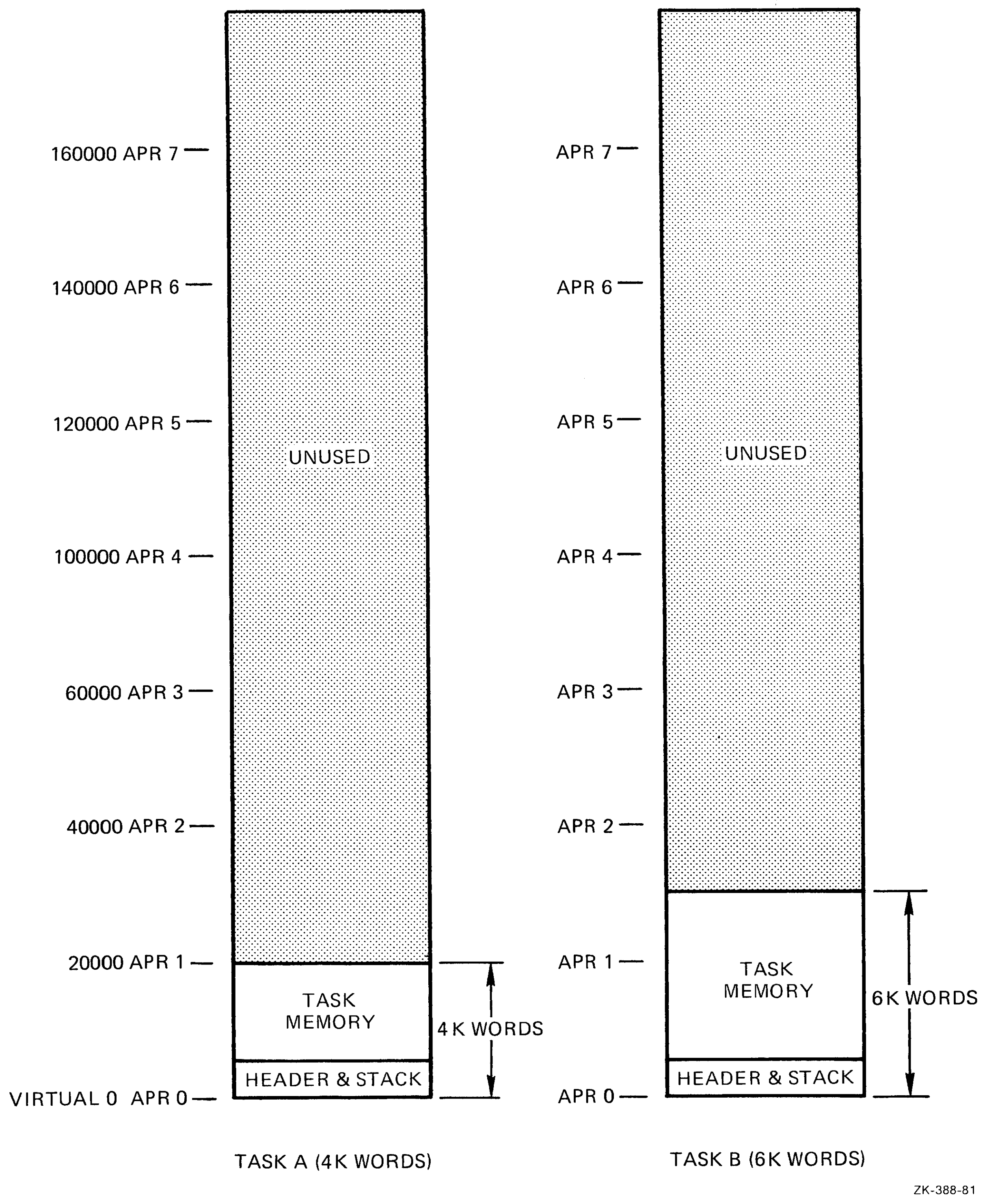


2.4.3 Regions

This section briefly describes regions and their relationship to and use by tasks. Regions and their use are more thoroughly described in Chapter 5.

A region is a defined area of memory that can contain code or data. It can also be a blank area reserved for use by one or more tasks. The region is named and built like a task except that the /HD switch (/HEADER in LINK) is negated (/-HD in TKB or /NOHEADER in LINK) because the region is not a task and does not need a task header. Tasks can also create regions dynamically as they execute. Dynamic regions are useful because they increase the task's logical address space while saving its virtual address space. Regions also allow tasks to share code and data with other tasks.

Figure 2–10: Mapping for 4K-Word and 6K-Word Tasks



Regions are named according to their use or the way in which they were built. These regions are as follows:

- Task region—A continuous block of memory in which the task runs.
- Common shared region—On unmapped systems, a shared region defined by an operator at run time or built into the system during system generation; for example, a global common area.

  Resident commons are usually called shared regions because they are used as an area in which tasks share common data. Shared regions can be absolute or position independent. Shared regions and their use are described in Chapter 5.
- Library shared region—A shared region containing common code or routines shared by tasks, and in this way saving virtual address space in the tasks.
- Dynamic region—A region created dynamically at run time by the Create Region (CRRG$) memory management directive in the task. This directive and associated directives are described in the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual.*

By convention, a shared region that contains code is a library and a shared region that contains data is a common.

Tasks must map to a region by using task windows that must be defined and numbered in the task when the task is built. Usually, a task uses one window for each region to which mapping must occur. Task windows are described in the next section, Task Mapping and Windows.

Figure 2–12 shows a sample collection of regions that could make up a task's logical address space. A task's logical address space can expand and contract dynamically as the task issues the appropriate memory management directives. The header and root segment are always part of the region. Therefore, the task header and root segment always use window 0 (UAPR 0) and region 0. Because a region occupies a continuous area of memory, each region is shown as a separate block.

2.5 Task Mapping and Windows

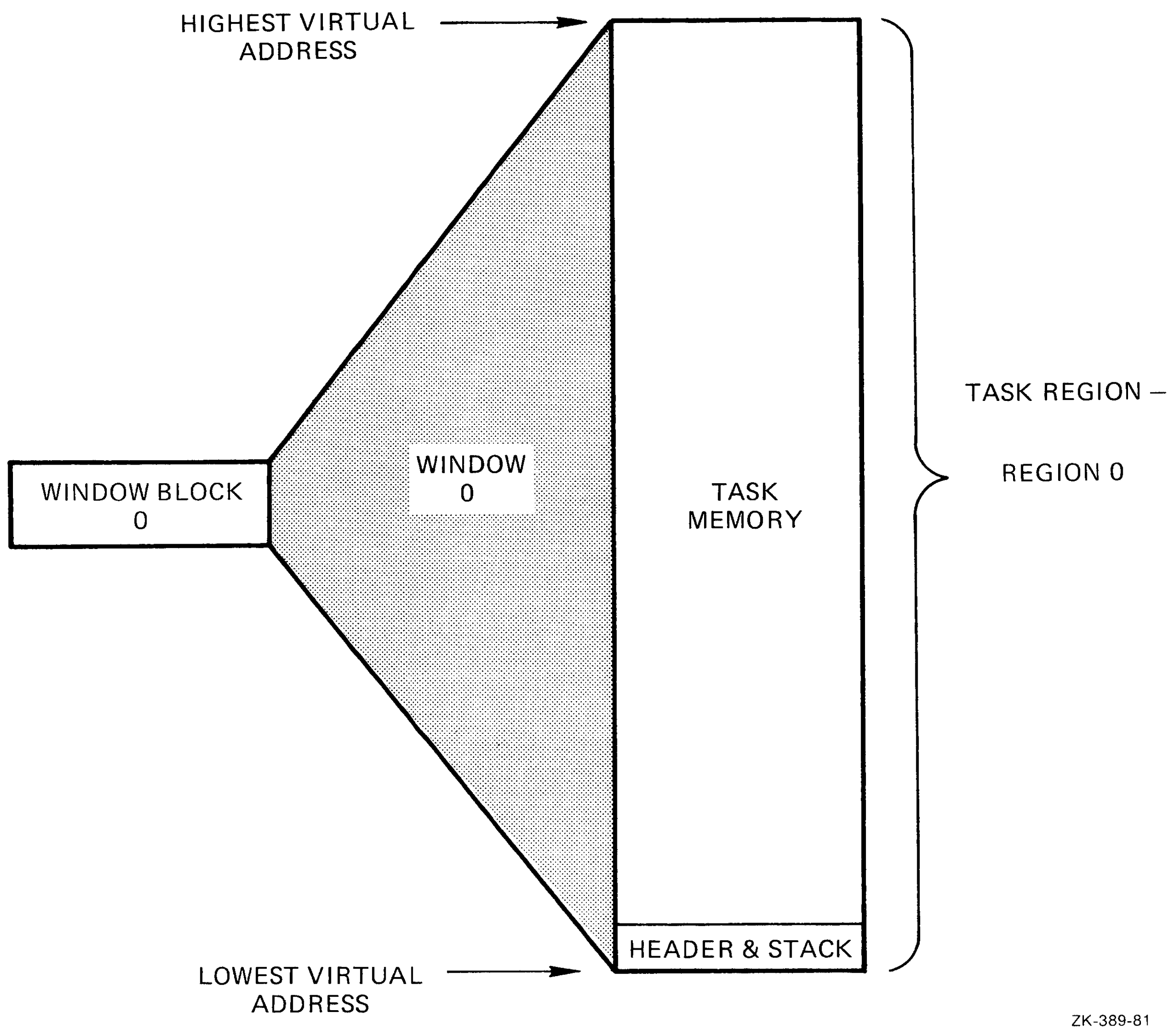
As mentioned earlier, tasks that run on mapped systems must be relocated at run time. When you build a task that is to run on a mapped system, TKB creates and places in the header of the task one or more 8-word data structures called window blocks. When you install a task, INSTALL initializes the window block or blocks. Once initialized, a window block describes a range of continuous virtual addresses called a window.

2.5.1 Task Windows

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 5), 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. Windows 0 and 1 describe the root of an I- and D-space task. Window 0 describes the I-space root and window 1 describes the D-space root and task header. Furthermore, this region is referred to as the task region and is identified as region 0. (Figure 2–11 illustrates window block 0 for a system without I- and D-space.) Windows for an I- and D-space task are described in Chapter 7.

Figure 2–11: 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 22-bit physical address range of the task.

Referring to Figure 2–12, which illustrates a mapped system without I- and D-space, you can observe that a large 32K user task contains three distinct areas of continuous space called "windows." The term "task window" is a construct that maps a continuous portion of the task's virtual address space to a continuous portion of a region in the task's logical address space. Windows must have a specified size and starting address. The window size can be from 32 words to 32K minus 32 words, and windows must start on a 4K address boundary. Figure 2–12 shows three windows that are not continuous in the task's virtual address space. However, the space within each window is continuous. In this task, the size of window 0 is 11K words, the size of window 1 is 11K words, and the size of window 2 is 8K words. The concept of windows exists for the following specific reason.

By using the concept of windows and the memory management directives, a nonprivileged task can access a larger logical memory space than that implied by the 32K-word virtual addressing range and normally accessible by the 16-bit address. A task can, in fact, only access 32K words of memory at one time. However, a nonprivileged task can change its access to logical addresses (real, physical memory). The area that your program accesses can be changed by the program during program execution. The process of accessing different logical areas of memory is called "mapping."

By referring to Figure 2–12, you can see that window 1 in the task is mapped to region 1 in physical memory. The task can change the window 1 mapping to region 0 in physical memory. In effect, then, though a task is limited to a range of 32K virtual addresses, a task can access all the physical memory available to it (determined by the way that you set up the mapping) by changing the mapping of its windows to different logical addresses. Figure 2–12 provides a visual description of the concept of mapping to different logical addresses.

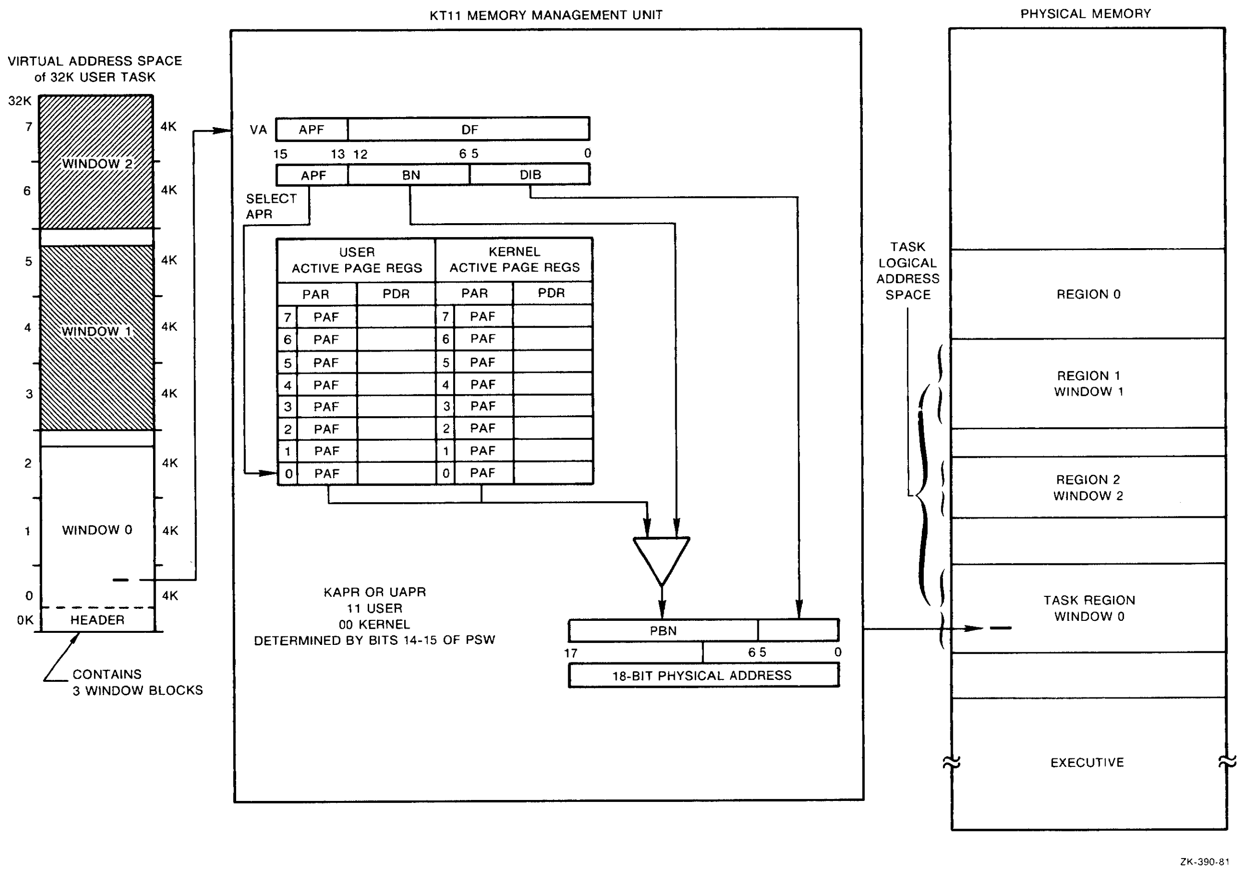
To set up the task's windows, you define task window blocks to TKB as described in the following paragraphs.

To manipulate virtual address mapping to various logical areas, you must first divide a task's 32K of virtual address space into segments. These segments are task (virtual address) windows. Each window encompasses a continuous range of virtual addresses. The first address of the window address range must be a multiple of 4K (the first address must begin on a 4K boundary) because of the way that the KT-11 memory management unit uses APRs.

Tasks that use I- and D-space or supervisor-mode libraries have a total of 16 windows. You can specify up to 14 windows in this type of task. Windows 0 and 1 are not available to nonprivileged tasks in this kind of system.

A task that includes directives that dynamically manipulate address windows must have task window blocks set up in the task header as well as Window Definition Blocks in the code for use by the Create Address Window (CRAW$) directive. The Executive uses task window blocks to identify and describe each currently existing window. When linking the task, you specify the number of extra window blocks needed by the task. The number of blocks should equal the maximum number of windows that will exist concurrently while the task is running.

Figure 2–12: Virtual to Logical Address Space Translation



In systems without I- and D-space, a window's identification is a number from 0 to 7, which is an index to the window's corresponding window block. The address window identified by 0 is the window that always maps the task's header and root segment. TKB creates window 0, which the Executive uses to map the task. No directive may specify window 0; a directive that does so is rejected.

In systems using an I- and D-space task, a window's identification is a number from 0 to 15, which is an index to the window's corresponding window block. The address windows identified by 0 and 1 are the windows that always map the task's header and root. TKB creates windows 0 and 1, which the Executive uses to map the task. No directive may specify windows 0 or 1; a directive that does so is rejected.

When a task uses memory management directives, the Executive views the relationship between the task's virtual and logical address space in terms of windows and regions. Unless a virtual address is part of an existing address window, the address does not point anywhere. This is a point to watch when setting up windows with the Create Address Window (CRAW$) directive. Similarly, a window can be mapped only to an area that is all or part of an existing region within the task's logical address space.

Once a task has defined the necessary windows and regions, the task can issue memory management directives to perform operations such as the following:

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

2.6 RSX–11M–PLUS Supervisor Mode

Three modes of operation are possible in the PDP–11: user mode, supervisor mode, and kernel mode. Each mode has associated with it 16 APRs for mapping memory: 8 I-space APRs and 8 D-space APRs. A task can use supervisor-mode libraries and thereby double the task's virtual address space to 64K words. Supervisor-mode libraries are described in Chapter 8. This section briefly describes supervisor mode and the mapping that occurs when the task uses supervisor mode.

Supervisor-mode libraries are libraries of routines that are used only in supervisor mode. The task switches to supervisor mode when it calls a routine within the supervisor-mode library. By using a supervisor-mode library, as described in Chapter 8, you make the RSX–11M–PLUS system, for large systems, use the supervisor-mode APRs.

2.6.1 Supervisor-Mode Mapping

Normally, a task has an address space of 32K words by using eight user APRs. When a conventional task links to a supervisor-mode library and calls a routine in the library, the Executive copies the user-mode I-space APRs into the supervisor-mode D-space APRs and maps the supervisor-mode library with supervisor I-space APRs. Therefore, while in supervisor mode and within the library, the task can access 32K words of its own space with D-space APRs and 32K words of library routines with I-space APRs. The amount of possible logical address space totals to 64K words.

When an I- and D-space task links to a supervisor-mode library, the Executive copies the user-mode D-space APRs into the supervisor D-space APRs. Therefore, the supervisor-mode routines can access user data space and access supervisor-mode instruction space with supervisor I-space APRs. (Figure 8-2 illustrates this mapping.) The mapping just described is the default mapping for an I- and D-space task. You can explicitly create supervisor-mode D-space mapping to override the user-mode D-space overmapping that occurs by using the Executive directive Map to Supervisor-Mode Data Space (MSDS$). Chapter 8 discusses the use of MSDS$.

The mapping of a conventional task in a system that contains a supervisor-mode library is shown in Figure 2–13.

The mapping of a conventional task in a system while using a supervisor-mode library is shown in Figure 2–14.

2.7 Privileged Tasks

RSX–11M–PLUS and Micro/RSX systems have two classes of tasks, privileged and nonprivileged. However, the term "privileged" has meaning in mapped systems only, because in mapped systems certain areas of memory are protected from nonprivileged tasks. In an unmapped system, any task has the ability to access all of physical memory if so programmed. Therefore, the distinction between these two classes of tasks is primarily one of their mapping to memory in a mapped system.

Privileged tasks in a mapped system can access system data areas and the Executive. Altering system data areas or the Executive can cause obscure and difficult problems. Therefore, privileged tasks must be programmed and used with caution.

You can specify a task as privileged by using the /PR:n switch in the TKB command line or the /PRIVILEGED:n qualifier in LINK. The /PR:0 switch or /PRIVILEGED:0 qualifier allows a task to perform certain privileged operations; but the task with a privilege of 0 cannot access the Executive or system data structures. The /PR:4 switch or the /PRIVILEGED:4 qualifier allows the task to directly map the I/O page, Executive routines, and system data structures. The /PR:4 switch or the /PRIVILEGED:4 qualifier is used for a privileged task in a system that has an Executive of 16K words or less. The /PR:5 switch or the /PRIVILEGED:5 qualifier allows a task to directly map to the I/O page, Executive routines, and system data structures. The /PR:5 switch or the /PRIVILEGED:5 qualifier is used for a privileged task in a system that has an Executive of 20K words or less.

Chapter 6 describes privileged tasks and their mapping in detail.

Figure 2-13: Mapping for a Conventional User Task and a System Containing a Supervisor-Mode Library in an RSX-11M-PLUS System

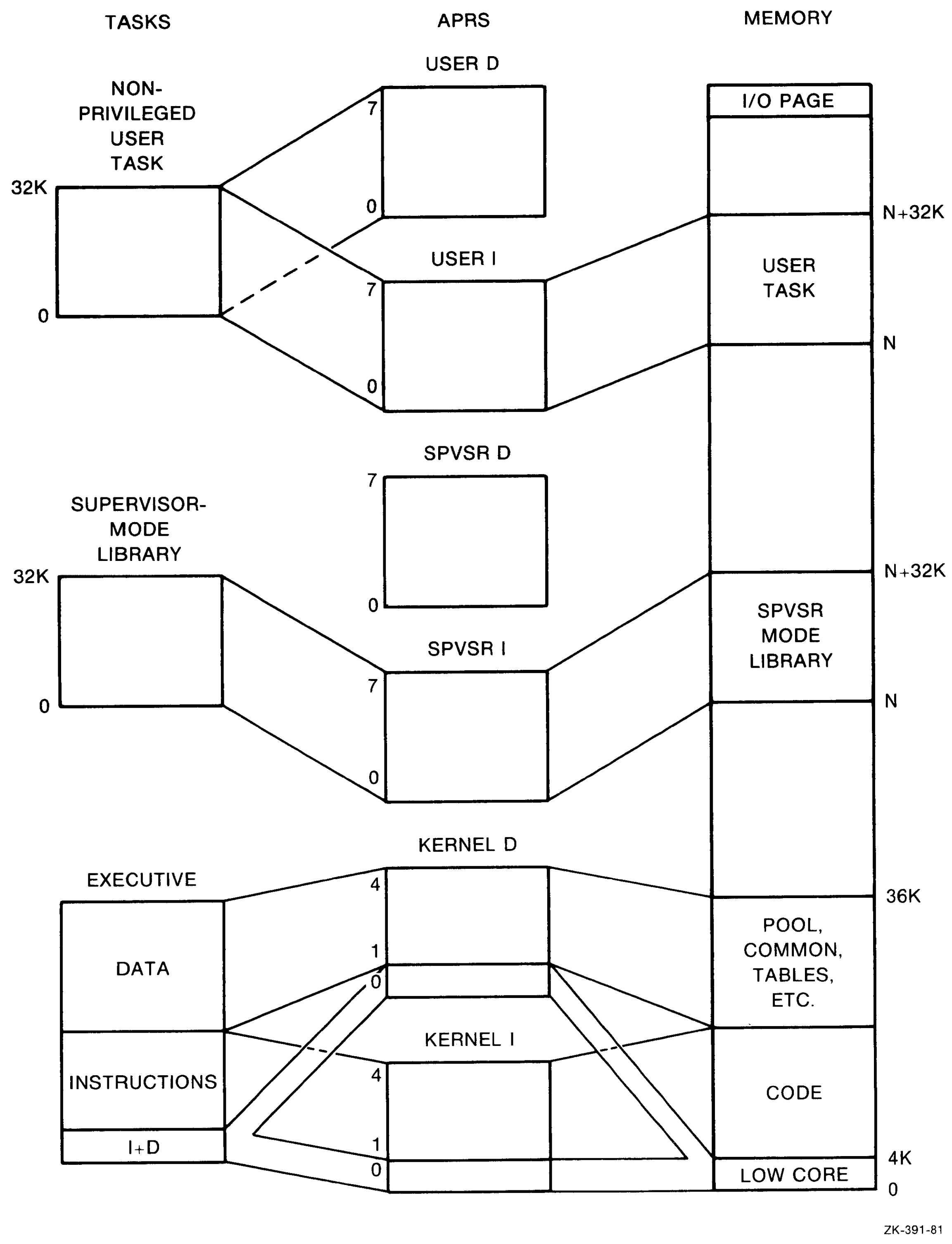
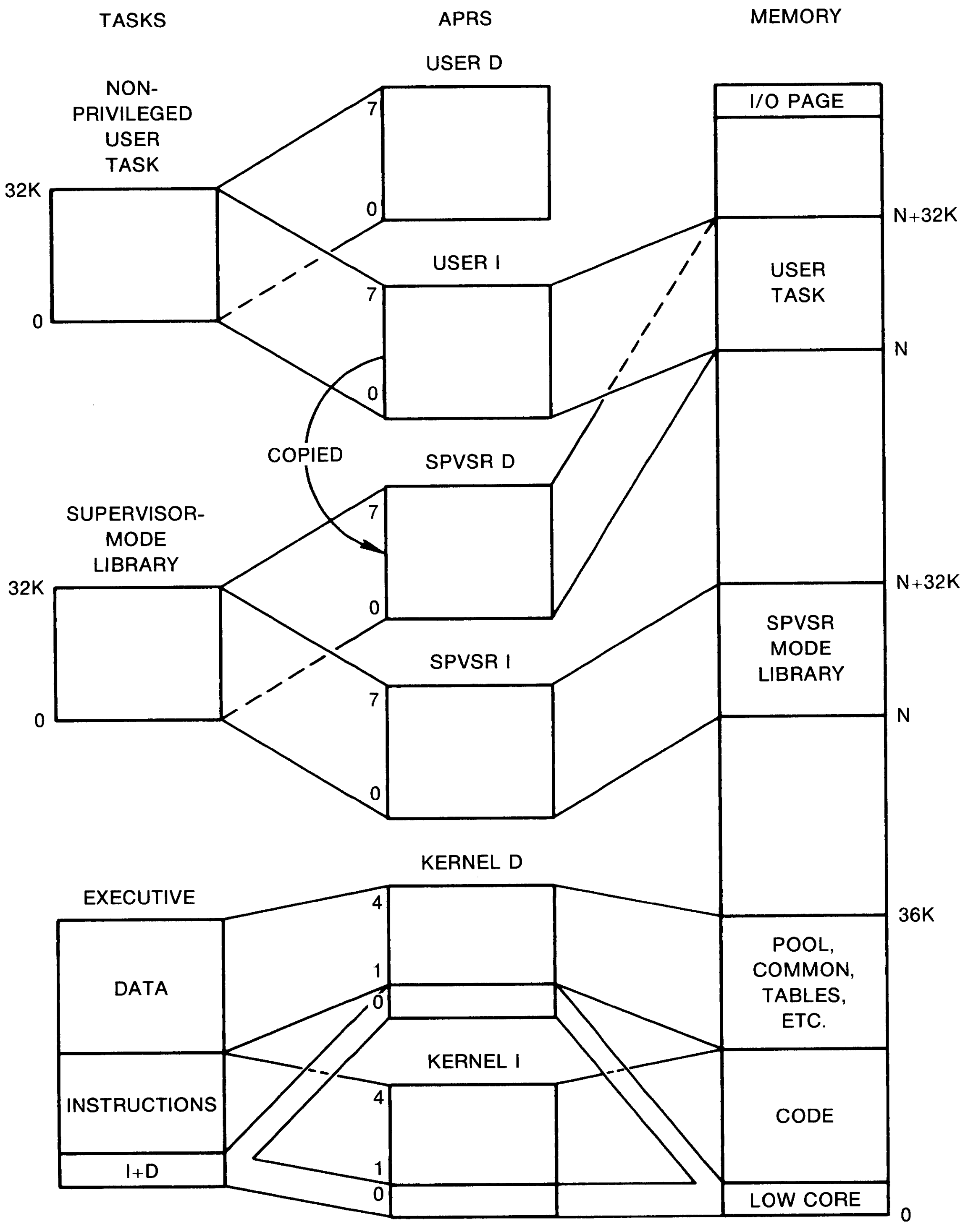


Figure 2–14: Mapping for a Conventional User Task Using a Supervisor-Mode Library



ZK-392-81

2.8 Multiuser Tasks

The following section is an introduction to multiuser tasks, which are fully described in Chapter 9.

TKB allows you to build multiuser tasks. A multiuser task is that which has one portion of its code and data designated as read-only and another portion designated as read/write. You specify the read-only portions of your task with program sections that have the read-only access code. When you then build your task with the /MU switch or /SHAREABLE:TASK qualifier, TKB places the read-only portions in a region that has a high virtual address and the read/write portion in a region that has a low virtual address. Any other requests to run the task, if the task is already running, result in a copy of the read/write portion of the task in physical memory for the other user. There is always only one copy of the read-only code regardless of the number of tasks that may be running.

The /MU switch is described in Chapter 10 and the /SHAREABLE:TASK qualifier is described in Chapter 11.

2.9 User-Mode I- and D-Space Tasks

User tasks that use both I- and D-space differ from conventional tasks because I- and D-space tasks have specifically defined locations within the task for both instructions and data. Because of this separation, the I- and D-space task image is structurally different.

Additionally, the separate instruction areas are mapped through separate APRs in the memory management unit. Hence, up to eight user-mode instruction APRs map the task's instructions, and up to eight user-mode data APRs map the task's data.

Also, overlaid I- and D-space tasks are more complex because each overlaid part (segment) of such a task may reside in both instruction space and data space.

I- and D-space tasks differ from conventional tasks in the following major ways:

- Program sections with the "I" attribute contain only instructions, and program sections with the "D" attribute contain only data.
- Two sets of APRs map the I- and D-space task: the I-space APRs and the D-space APRs.
- I- and D-space tasks can use up to 64K words of virtual space instead of 32K words because of their use of the two sets of APRs. With supervisor-mode libraries, an I- and D-space task can use up to 96K words of virtual space.

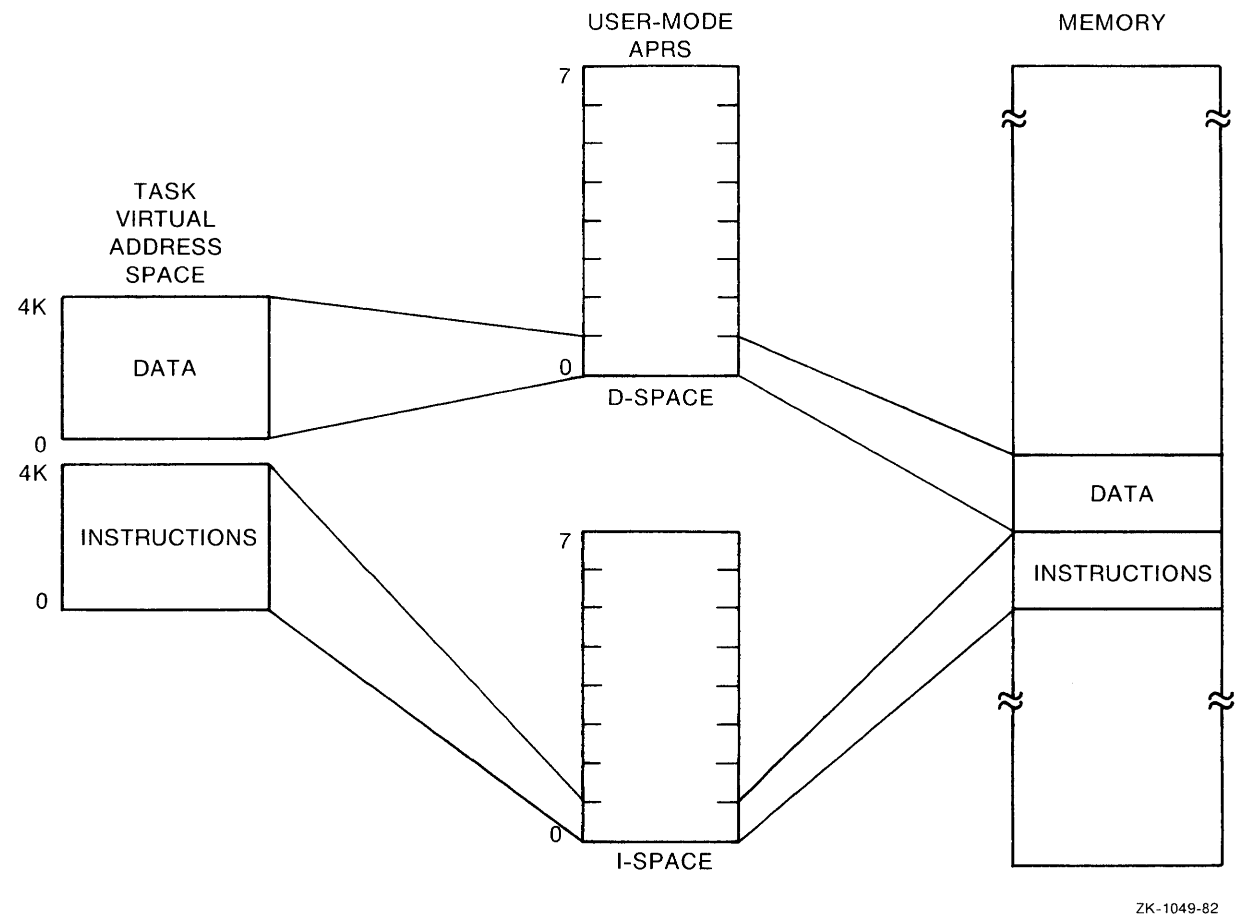
The following units have data contiguously adjacent in memory and instructions contiguously adjacent in memory:

- Nonoverlaid I- and D-space tasks
- Segments in an overlaid I- and D-space task that contain both I- and D-space

In these tasks or task segments, I-space program sections are segregated from D-space program sections in memory. They cannot be intermixed.

Figure 2–15 shows a user-mode I- and D-space task with data separated from the instructions and mapped to memory through two sets of APRs.

Figure 2–15: Simplified APR Mapping for an I- and D-Space Task



I- and D-space tasks are more fully described in Chapter 7. The task images for both conventional and I- and D-space tasks are described in Appendix B.

Chapter 3
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 divide your conventional task into pieces called segments, which are loadable with one disk access. In an I- and D-space task, an overlaid segment that contains I- and D-space program sections requires a maximum of two disk accesses to load the segment. The segments are the discrete parts of the overlay structure that form the tree. You can specify two kinds of overlay segments: those that reside on disk, and those that reside permanently in memory after being loaded from disk.

3.1 Overlay Structures

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

- A single root segment, which is always in memory
- 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 another segment with which it shares virtual address space. In addition to the logical independence of the overlay segments, you must consider the general flow of control within the task 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 are 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.

[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.

Several large classes of tasks can be handled effectively when built as overlay structures. For example, a task that moves sequentially through a set of modules is well suited to use as 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 use as an overlay structure.

Tasks that have separate I- and D-space may also use overlays where the root has instructions and data separately defined by program sections, and each individual segment of the task also has instructions and data separately defined. Chapter 7 contains more information about I- and D-space tasks.

3.1.1 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 following TKB command line:

```
>TKB TK1,,=OVRLAY.ODL/MP RET
```

or by one of the following LINK command lines:

```
$ LINK/TASK:TK1 OVRLAY.ODL/OVERLAY_DESCRIPTION RET

$ LINK/TAS:TK1 OVRLAY.ODL/OVER RET
```

The file OVRLAY.ODL contains the modules CNTRL, A, B, and C in an overlay description for the task being built. The /MP switch in TKB or the /OVERLAY_DESCRIPTION qualifier in LINK specifies that the input file is an Overlay Description Language (ODL) file.

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 is 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 allocating 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 3-1 shows the virtual address space and physical memory required 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 3-2 shows the virtual address space and physical memory required as a result of building TK1 as a multisegment task and using the overlay capability.

The multisegment task requires 50000_8 bytes.

Note

In addition to the storage required for modules A, B, and C, storage is required for overhead in handling the overlay structures. This overhead is not reflected in this example.

In using the overlay capability, the total amount of virtual address space and physical memory required for the task is determined by the sum of 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 3-3 shows the virtual address space and physical memory required for the task produced by the additional overlays defined for A and B.

Figure 3-1: TK1 Built as a Single-Segment Task

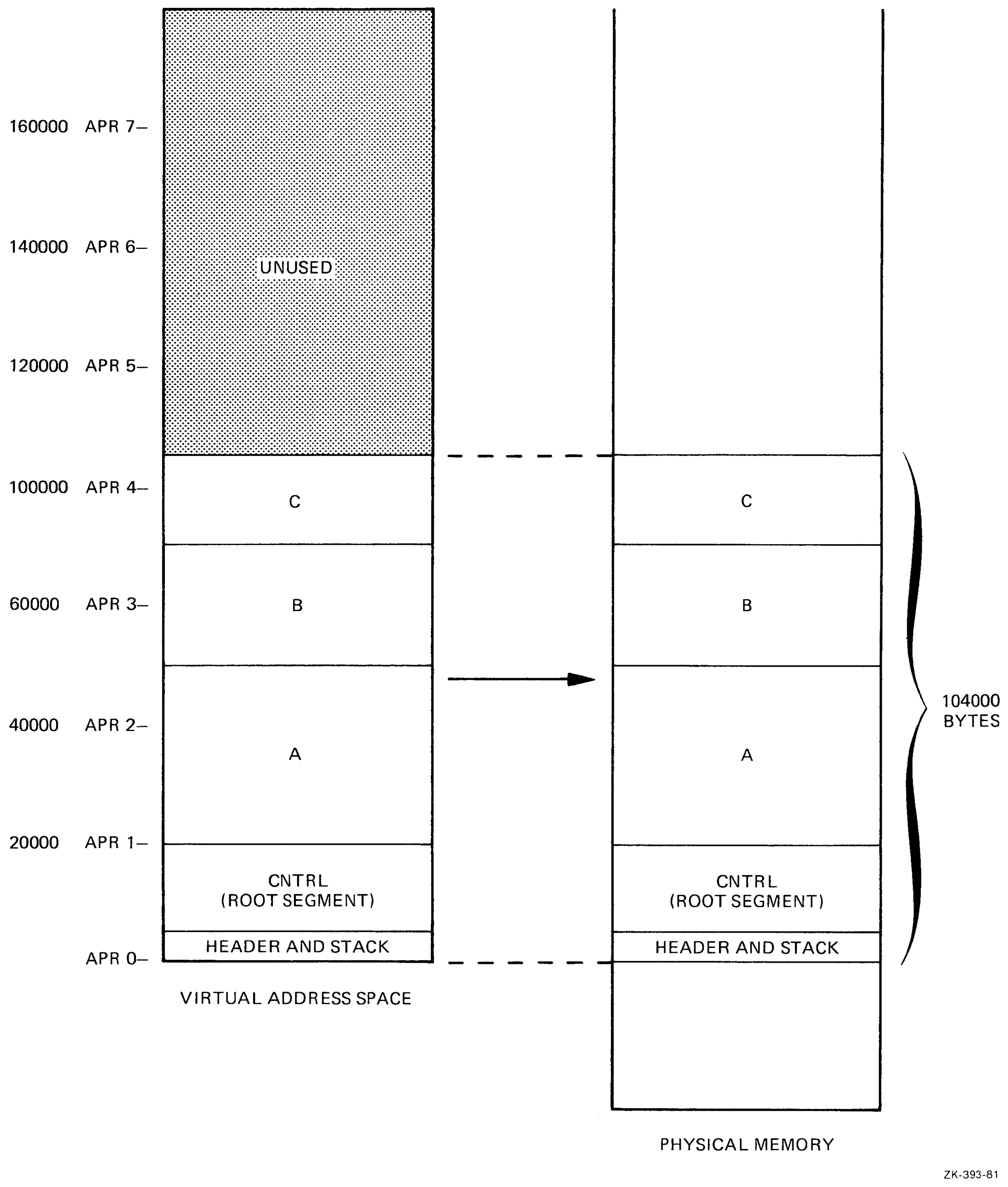
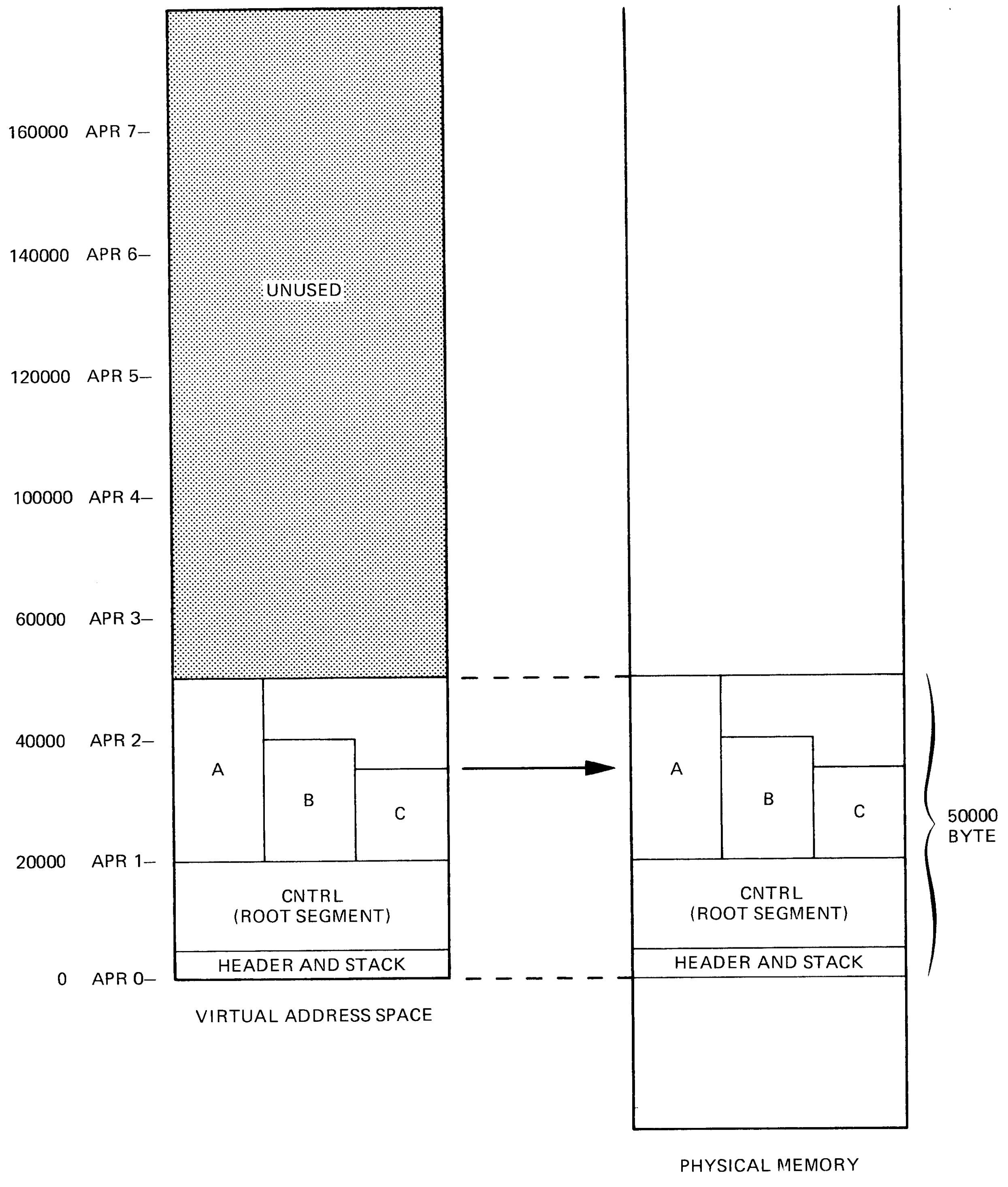
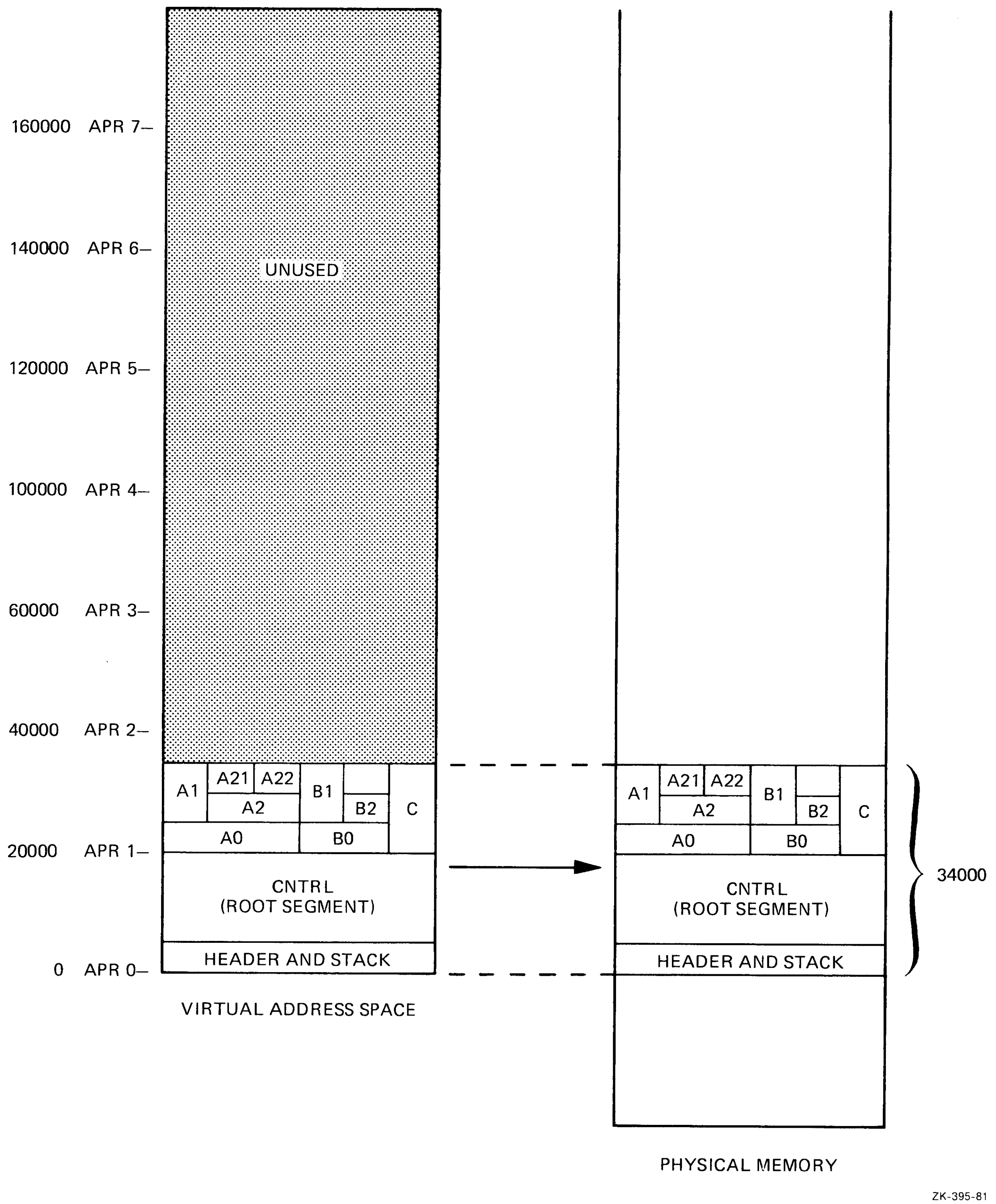


Figure 3-2: TK1 Built as a Multisegment Task



ZK-394-81

Figure 3-3: TK1 Built with Additional Overlay Defined



As a single-segment task, TK1 requires 104000_8 bytes of virtual address space and physical memory. The first overlay structure reduces the requirement by 34000_8 bytes. The second overlay structure further reduces the requirement by 14000_8 bytes.

The vertical lines in the diagrams of Figures 3-2 and 3-3 represent the state of virtual address space and physical memory at various times during the calling sequence of TK1. For example, in Figure 3-3, 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 3-2 and 3-3 indicate segments that share virtual address space and physical memory. For example, in Figure 3-3, 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.

3.1.2 Memory-Resident Overlay Structures

TKB provides for creating 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.

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 following TKB command line:

```
>TKB TK2,,=OVRLAY2.ODL/MP RET
```

or by one of the following LINK command lines:

```
$ LINK/TASK:TK2 OVRLAY2.ODL/OVERLAY_DESCRIPTION RET
```

```
$ LINK/TAS:TK2 OVRLAY2.ODL/OV RET
```

The file OVRLAY2.ODL contains the modules CNTRL, D, E, and F in an overlay description for the task being built. The /MP switch in TKB or the /OVERLAY_DESCRIPTION qualifier in LINK specifies that the input file is an Overlay Description Language (ODL) file.

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 the same virtual address space. The flow of control for the task is 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 allocating 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 3-4 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 3-5.

Figure 3-4: TK2 Built as a Single-Segment Task

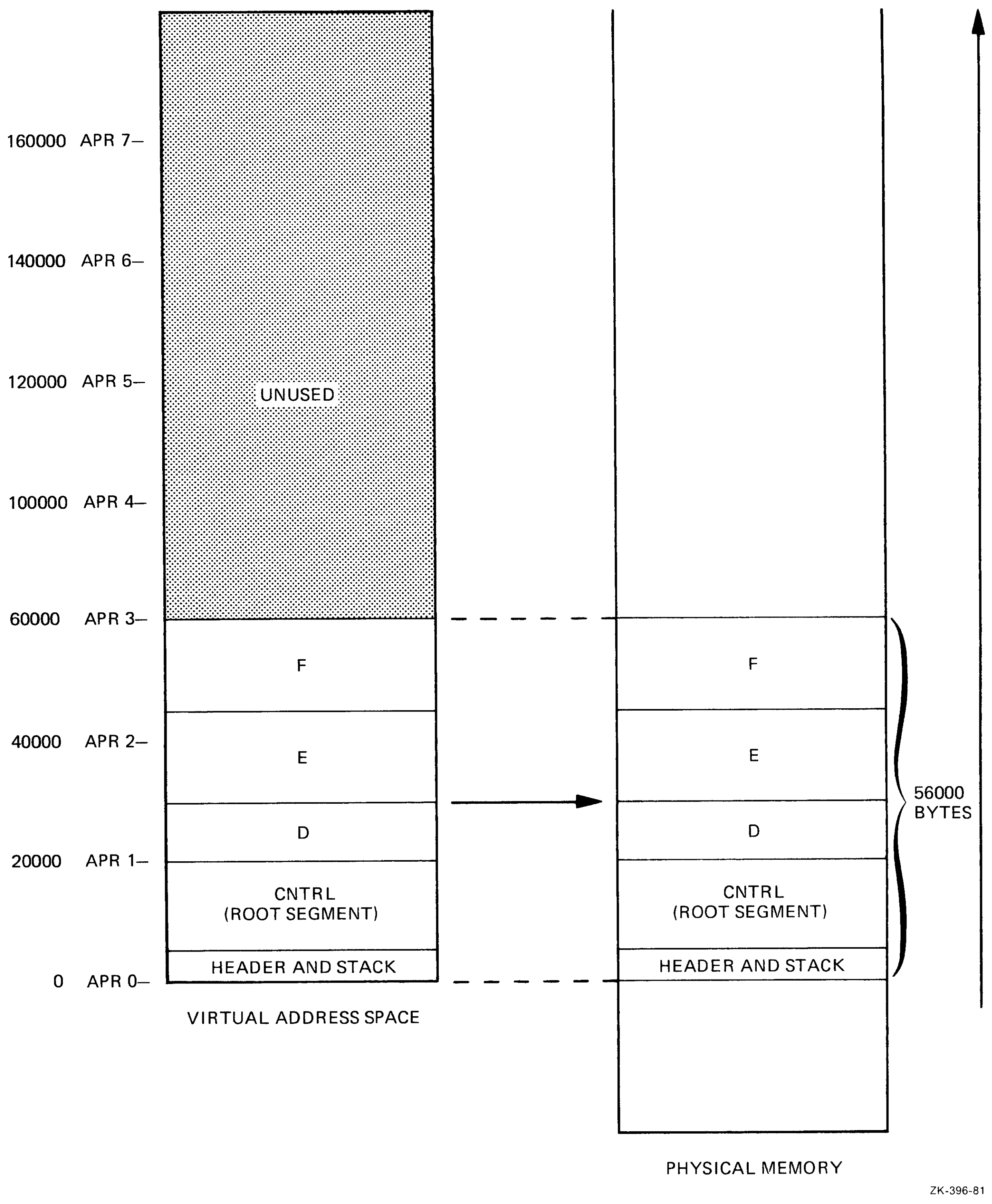
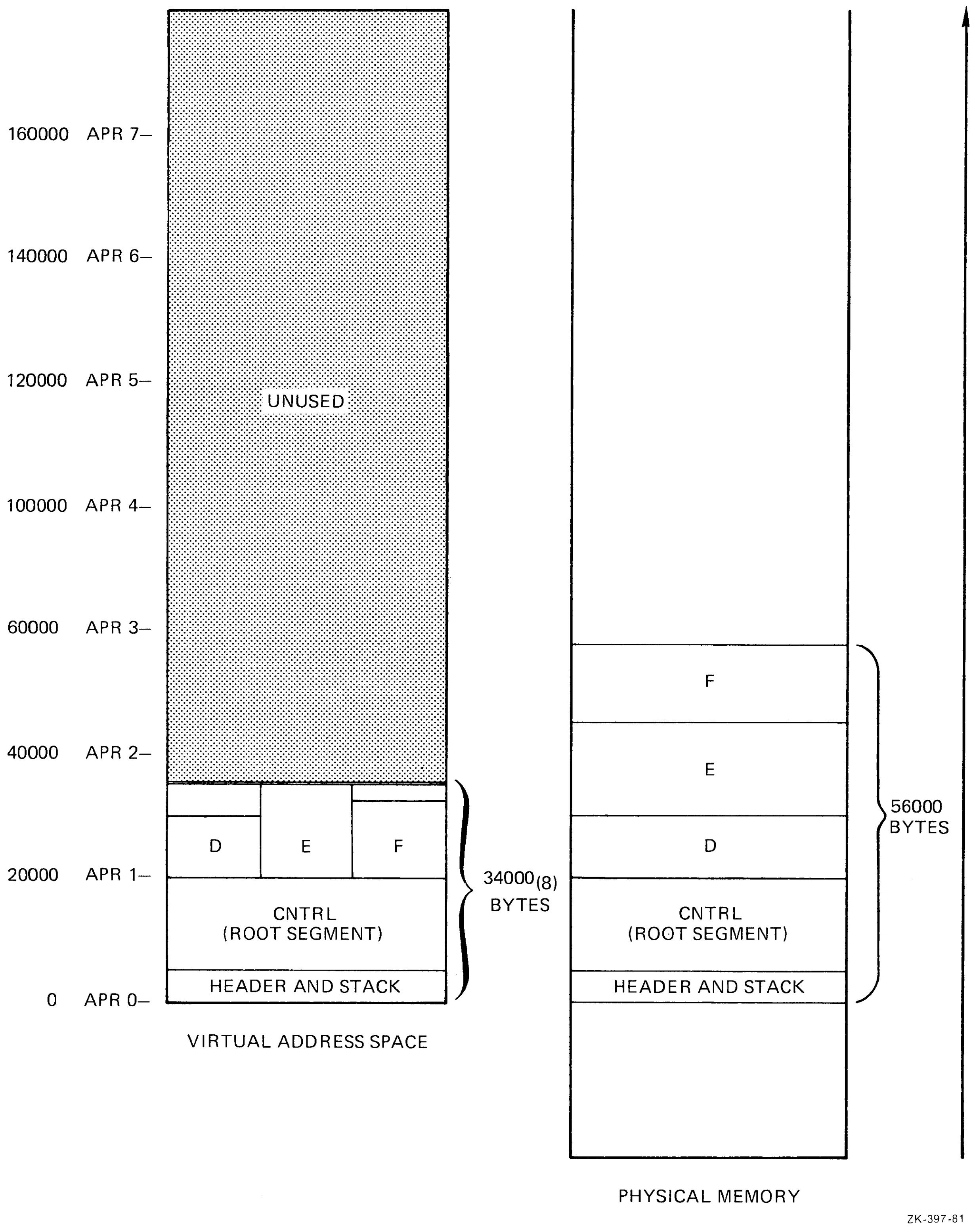


Figure 3–5: TK2 Built as a Memory-Resident Overlay



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 saving in virtual address space of 22000_8 bytes.

Note

> In addition to the storage required for modules D, E, and F, storage is required for overhead in handling the overlay structures. This overhead is not reflected in this example.

In Figure 3-5, 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. 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 3-6 and 3-7 illustrate the relationship between virtual address space and physical memory for task TK2 during the following time periods:

- TIME 1 (Figure 3-6A)—TK2 is run and the system loads the root segment (CNTRL) into physical memory and maps to it.
- TIME 2 (Figure 3-6B)—CNTRL calls segment D. The system loads segment D into physical memory and maps to it. Segment D returns to CNTRL.
- TIME 3 (Figure 3-7A)—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 3-7B)—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 3–6A: Relationship Between Virtual Address Space and Physical Memory— Time 1

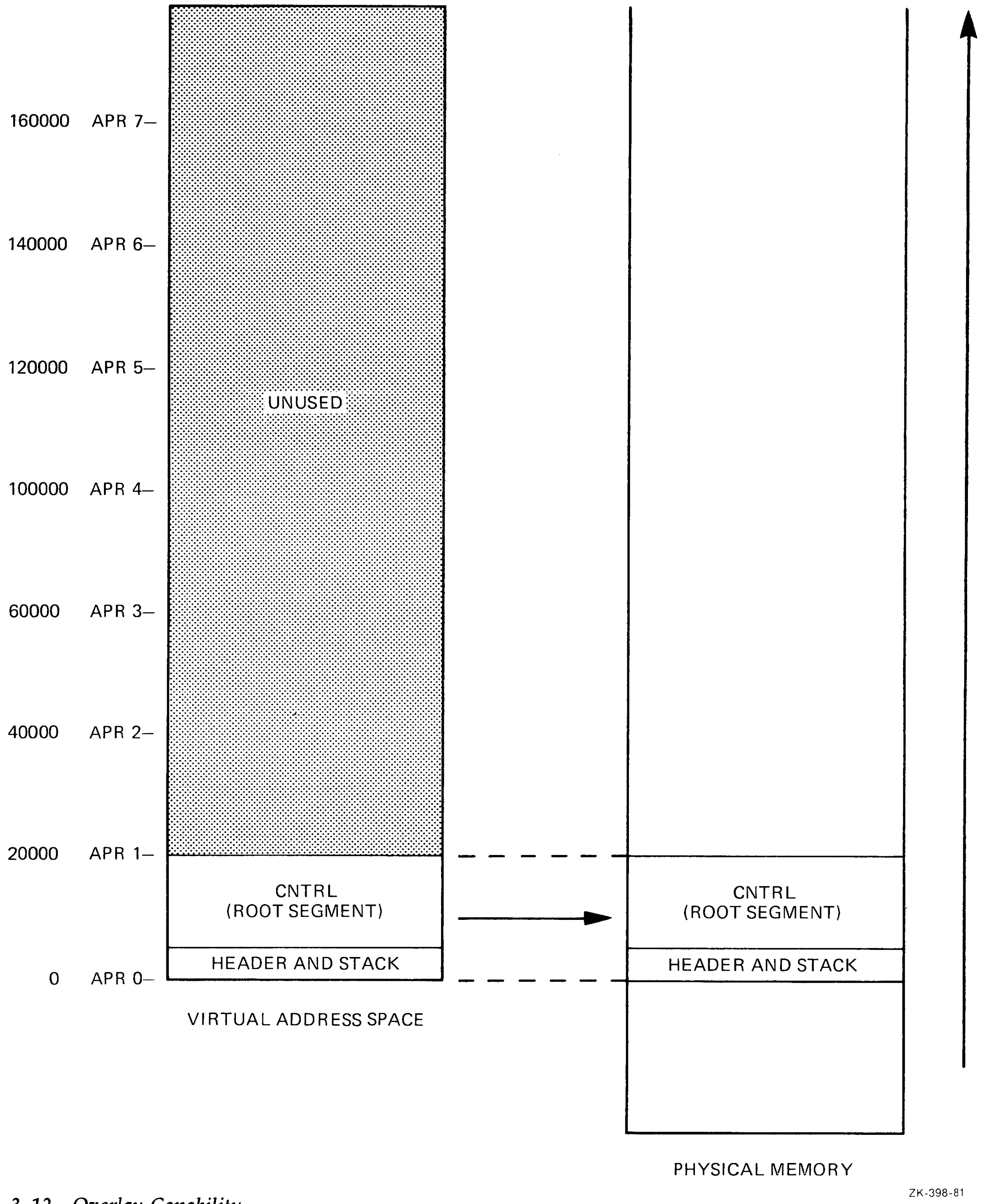


Figure 3–6B: Relationship Between Virtual Address Space and Physical Memory— Time 2

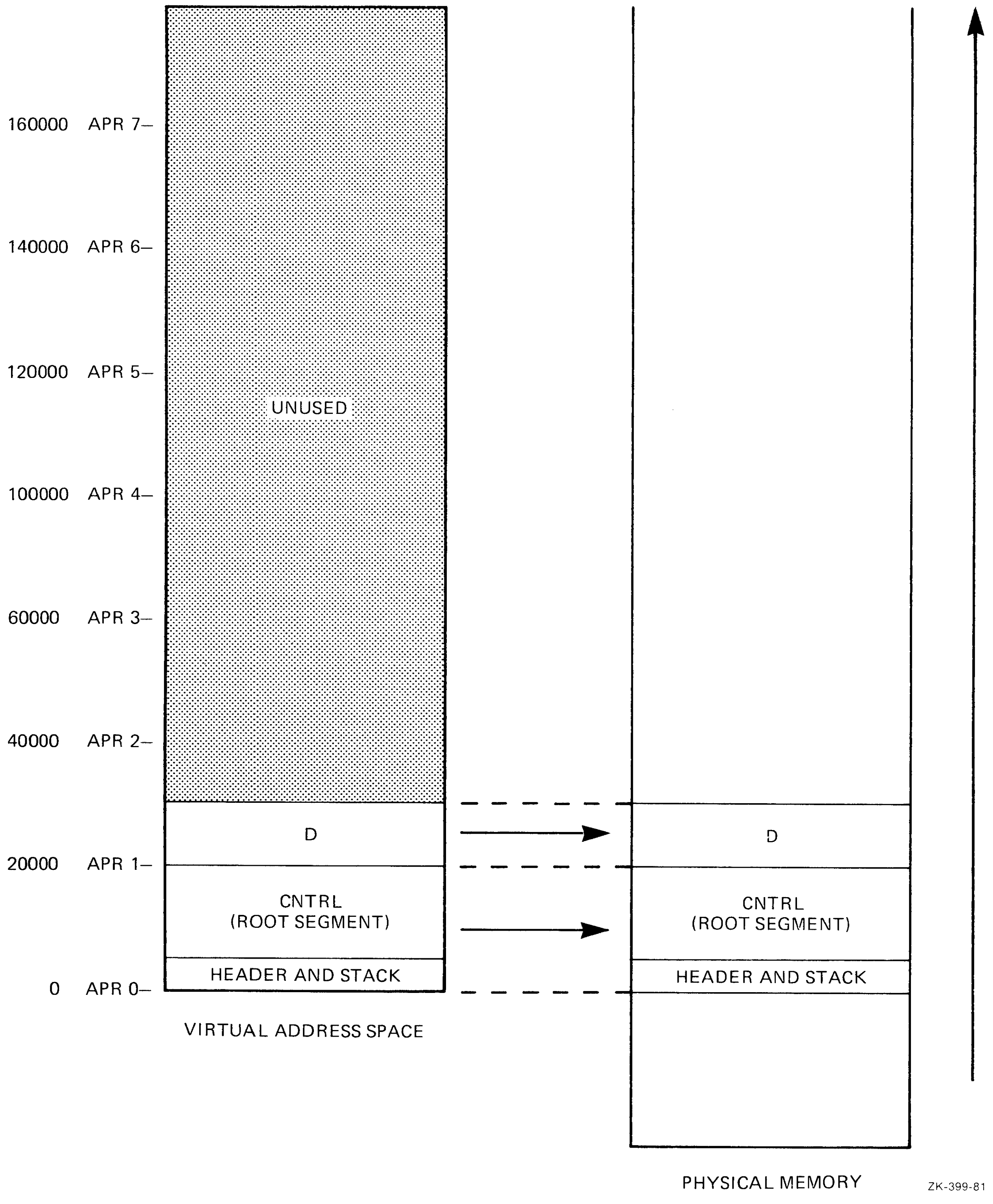


Figure 3-7A: Relationship Between Virtual Address Space and Physical Memory—Time 3

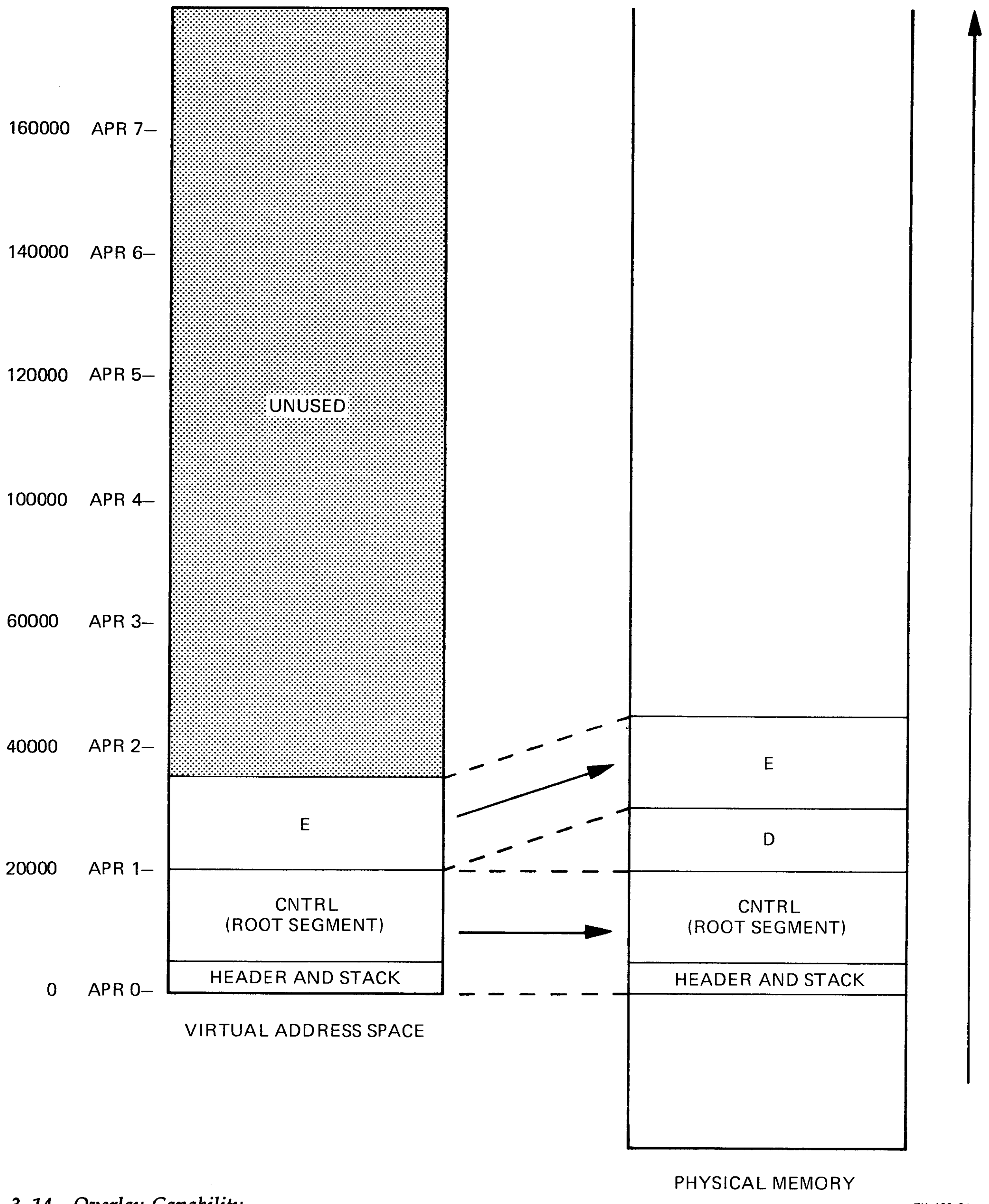
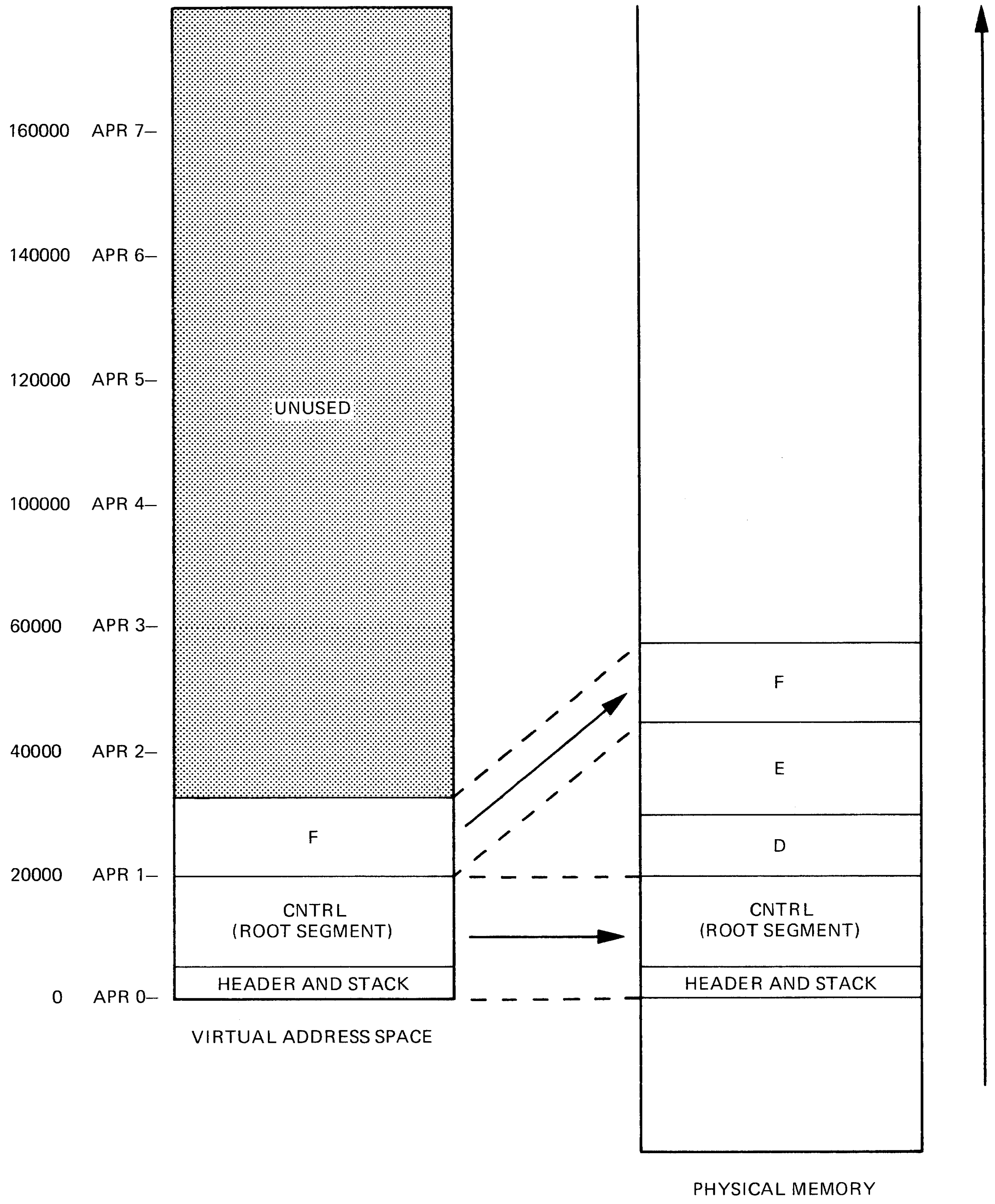


Figure 3–7B: Relationship Between Virtual Address Space and Physical Memory—Time 4



ZK-401-81

It is important to be careful in choosing whether to have memory-resident overlays in a structure. Carelessly using these segments can result in inefficient allocation of virtual address space because TKB 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, is allocated 8K words of virtual address space. All but one word of the second 4K words is unusable.)

You can also conserve physical memory by maintaining control over the contents of each segment. Including 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 down the system unacceptably) or impossible (because the segments are part of a resident library or other shared region that must reside in memory permanently).

Memory-resident overlays can help you use large systems to better advantage because of the time saving 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.

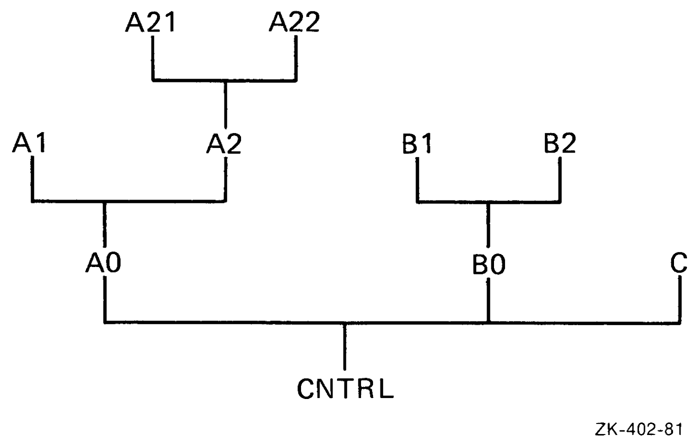
3.2 Overlay Tree

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.

TKB provides an Overlay Description Language (ODL) for representing an overlay structure consisting of one or more trees. (The ODL is described in Section 3.4.)

The single overlay tree shown in Figure 3-8 represents the allocation of virtual address space for TK1 (see Section 3.1.1).

Figure 3–8: 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.

3.2.1 Loading Mechanism

Modules can call other modules that exist on the same path. The module CNTRL (Figure 3-8) 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 4.

3.2.2 Resolution of Global Symbols in a Multisegment Task

In resolving global symbols for a multisegment task, TKB 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 3-9, 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.

TKB's procedure for resolving global symbols is summarized as follows:

1. TKB selects an overlay segment for processing.
2. TKB scans each module in the segment for global definitions and references.
3. If the symbol is a definition, TKB 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, TKB performs the tree search as described in step 3, looking for an existing definition.
5. If the symbol is new, TKB enters it in a list of global symbols associated with the segment.

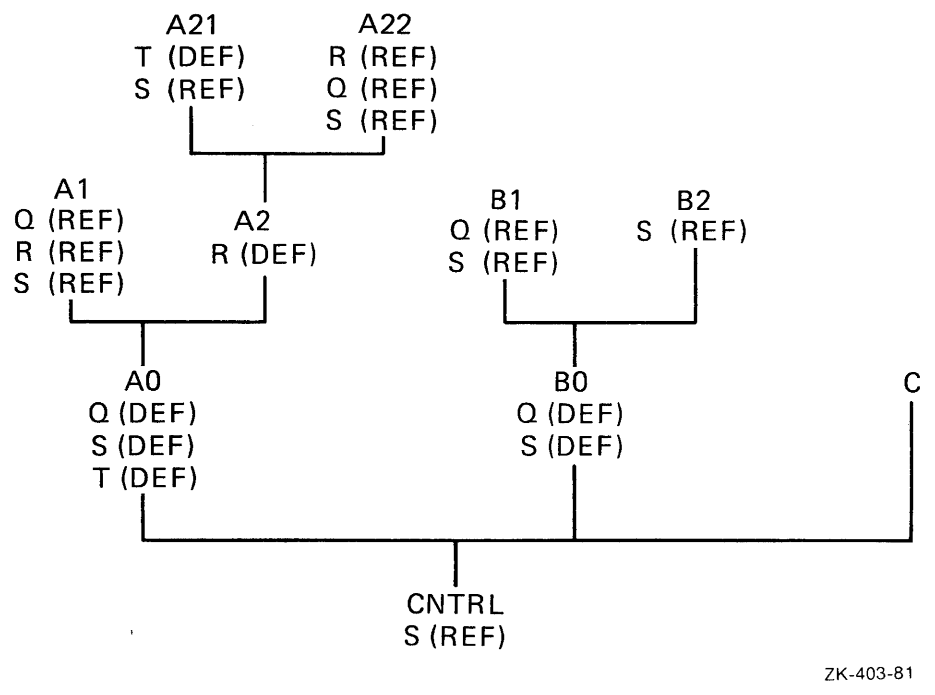
Overlay segments are selected for processing in an order corresponding to their distance from the root. That is, TKB processes the segment farthest from the root first, before processing an adjoining segment.

When TKB processes a segment, its search for global symbols proceeds as follows:

1. The segment being processed
2. All segments toward the root
3. All segments away from the root
4. All co-trees (see Section 3.5)

Figure 3-9 illustrates the resolution of global symbols in a multisegment task.

Figure 3–9: Resolution of Global Symbols in a Multisegment Task



The following notes discuss the resolution of references in Figure 3-9:

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.

3.2.3 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 TKB scans in the first pass. If any undefined symbols remain, TKB 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). TKB reports any undefined symbols remaining after its second pass.

When multiple tree structures (co-trees) are defined, as described in Section 3.5, 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 following segments:

- 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 TKB or the /FULL_SEARCH qualifier in LINK in the task image file specification. (Refer to Chapter 10 for a description of the TKB /FU switch or to Chapter 11 for a description of the LINK /FULL_SEARCH qualifier.)

3.2.4 Allocation of Program Sections in a Multisegment Task

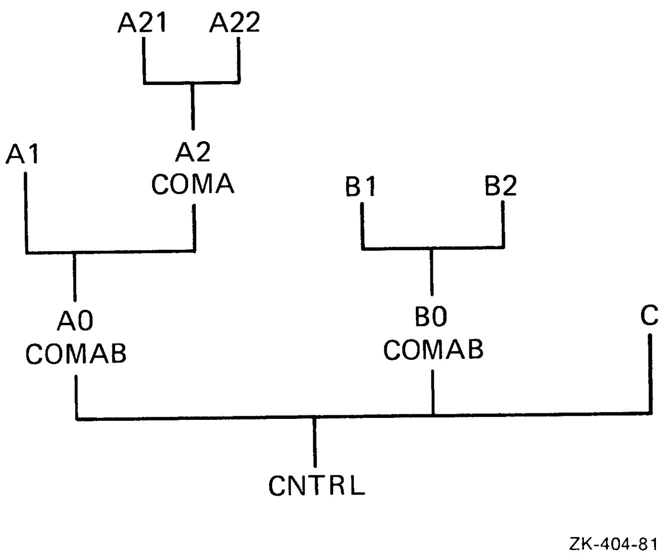
One of a program section's attributes indicates whether the program section is local (LCL) to the segment in which it is defined or is global (GBL).

Local program sections with the same name can appear in any number of segments. TKB 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 TKB.

When a global program section is defined in several overlay segments along a common path, TKB 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 3-10, the common block COMA is defined in modules A2 and A21. TKB allocates the virtual address space for COMA in A2 because that segment is closer to the root than the segment that contains A21.

Figure 3–10: Resolution of Program Sections for TK1



If the segments A0 and B0 use the common block COMAB, however, TKB 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 3.4.

3.3 Overlay Data Structures and Run-Time Routines

When TKB constructs an overlaid task, it builds additional data structures and adds them to the task image. 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. TKB also includes into the task image a number of system library routines (called overlay run-time routines). 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.

TKB 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 the following 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 4 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 sections.

3.3.1 Overlaid Conventional Task Structures

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

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

Figure 3-11 shows the typical structure of an overlay root segment.

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.

In an autoloadable, overlaid task, 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. All references to resident libraries are resolved through autoload vectors in the root.

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 (CRAW$) directive. One descriptor is allocated for each memory-resident overlay segment.

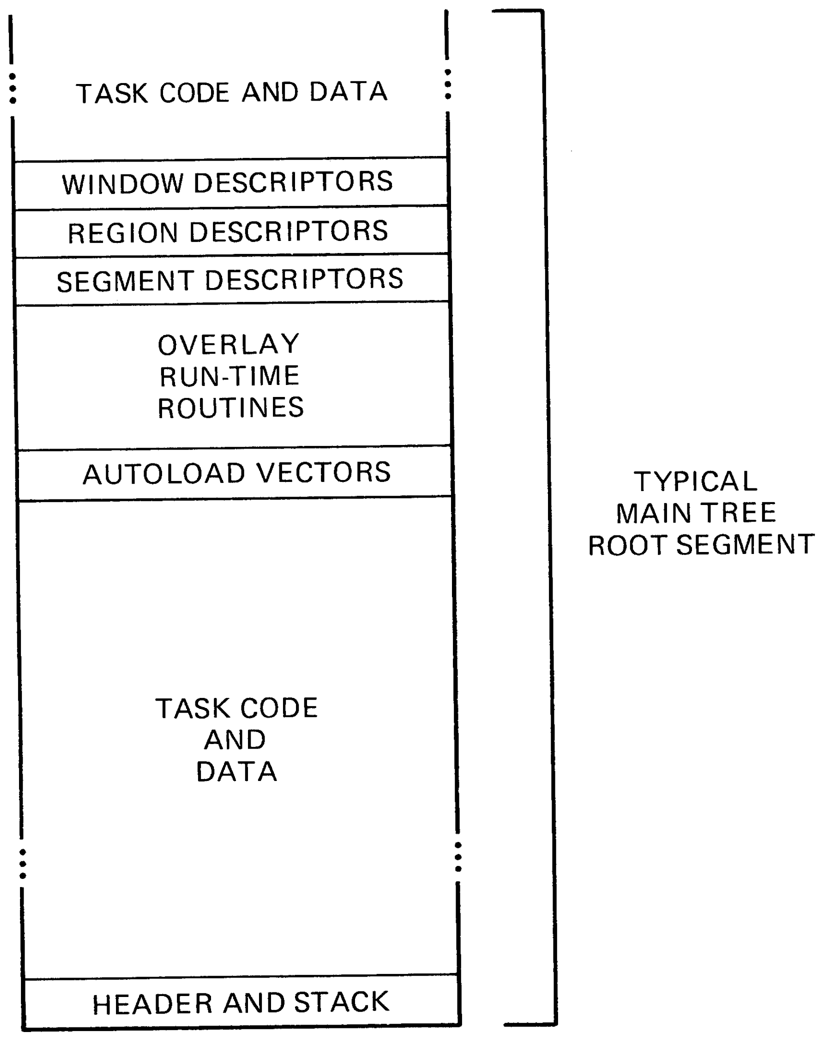
Region descriptors are allocated whenever a task is linked to a shared region containing memory-resident overlays. The descriptor contains information required by the Attach Region (ATRG$) directive.

3.3.2 Overlaid I- and D-Space Task Structures

Overlaid I- and D-space tasks contain data structures similar to those in a conventional task. These data structures differ only in their virtual address space allocation (mapping), and some structures are mapped in two different address spaces.

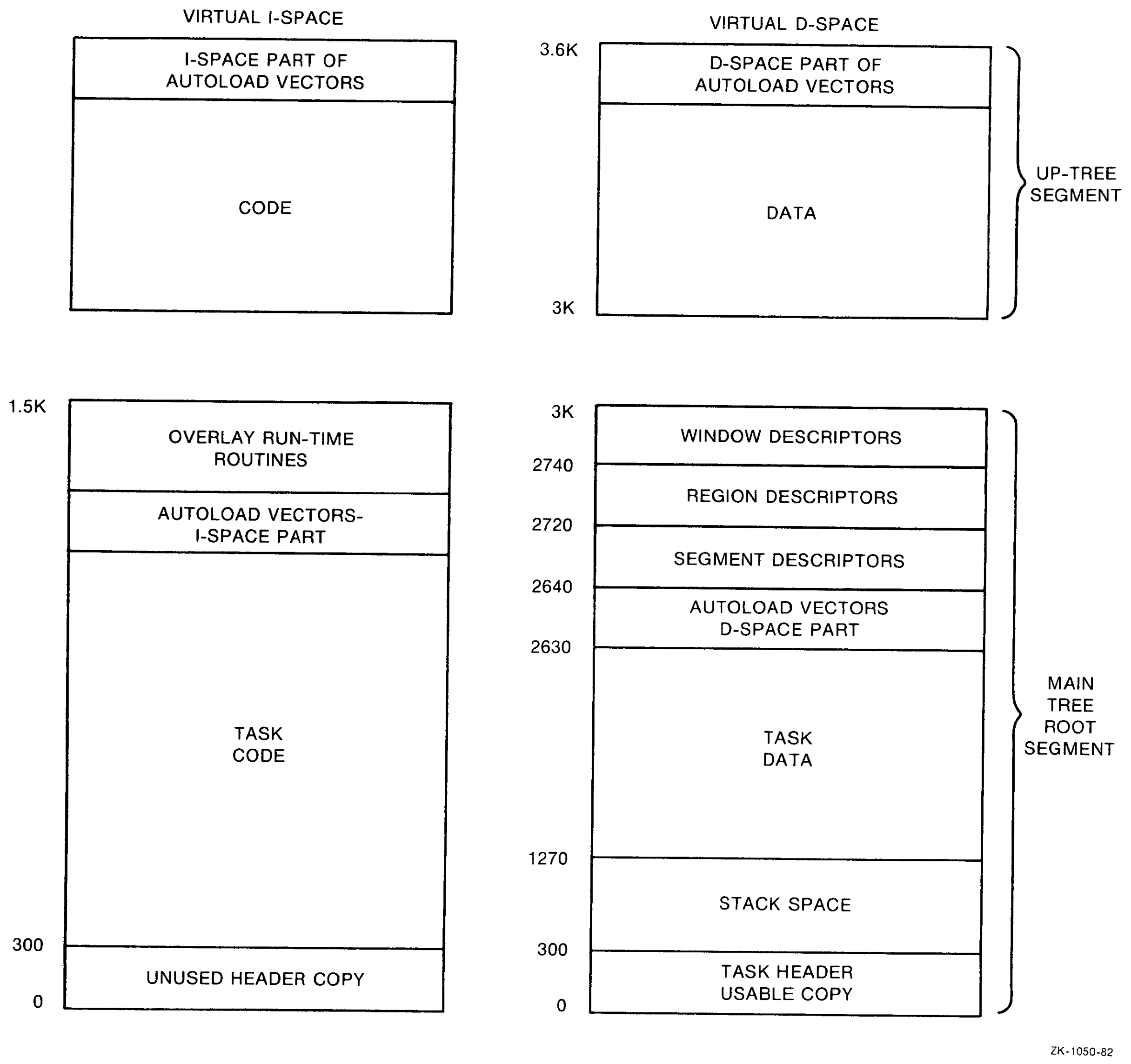
Figure 3-12 shows a typical overlaid I- and D-space task with an up-tree segment.

Figure 3–11: Typical Overlay Root Segment Structure



ZK-405-81

Figure 3–12: Typical Overlaid I- and D-Space Task with Up-Tree Segment



The following structures are located in data space:

- Segment descriptors
- Window descriptors
- Region descriptors
- Autoload vectors (the data part)

Autoload vectors contain both data and instructions. Therefore, TKB locates the instruction part of the autoload vector in I-space and the data part in D-space. Each segment of an autoloadable overlaid I- and D-space task may have an instruction part and a data part. Therefore, each I- and D-space segment in such a task would have its vectors separated into an instruction part and a data part.

The following structures are located in instruction space:

- Autoload vectors (the instruction part)
- Segment return point
- Overlay run-time system code (root segment only)

3.4 Overlay Description Language

TKB 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 on each line. Each line may have as many as 132 characters. You enter the name of this file in a TKB command line, and identify it as an ODL file by specifying the /MP switch in TKB or the /OVERLAY_DESCRIPTION qualifier in LINK to the file name.

For example, the following TKB command line specifies an ODL file:

```
>TKB TASK1,,=OVRLAY/MP RET
```

The following LINK command lines specify the same ODL file:

```
$ LINK/TASK:TASK1 OVRLAY/OVERLAY_DESCRIPTION RET
```

```
$ LINK/TAS:TASK1 OVRLAY/OVER RET
```

If you specify an ODL file to TKB, it must be the only input file you specify.

A command line in an ODL file takes the following form:

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

A label is required only for the .FCTR directive (see Section 3.4.2). Labels cannot be used with the other directives.

The ODL directives are listed below and described in Sections 3.4.1 through 3.4.6:

.ROOT and .END
.FCTR
.NAME
.PSECT

@ (at sign; specifies indirect command file)

The ODL directives can act upon the following items: named input files, overlay segments, program sections, and lines in the ODL file itself. These items follow each directive on the same line as the directive and form an argument list. Operators (such as the hyphen, exclamation point, and comma) group the argument-list items (named task elements) or attach attributes to them.

If the named task element is a file, you can enter a complete file specification. Defaults for omitted parts of the file specification are as described in Chapters 1 and 10, except that the default device is SY0, and the default directory is taken from the terminal UIC.

In addition, the following restrictions apply to argument lists:

- A period (.) is the only nonalphabetic character you can use in a file name.
- Comments cannot appear on a line ending with a file name.

3.4.1 .ROOT and .END Directives

The .ROOT directive defines the structure of the overlaid task. Because of this, .ROOT usually appears first in the overlay description. The .NAME directive may precede the .ROOT directive in certain circumstances discussed in Section 3.4.4. Each overlay description must end with one .END directive. The .ROOT directive tells TKB where to start building the tree and the .END directive tells TKB where the input ends.

The arguments of the .ROOT directive use three operators to express concatenation, memory residency, and overlaying. The operators are as follows:

- 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 module X and module Y simultaneously. TKB allocates segment X and segment Y in sequence to produce one segment.
- The exclamation point (!) operator indicates memory residency of overlays. (This operator is discussed in Section 3.4.3.)
- 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 3.5.1).

These operators can be used with the .FCTR directive also.

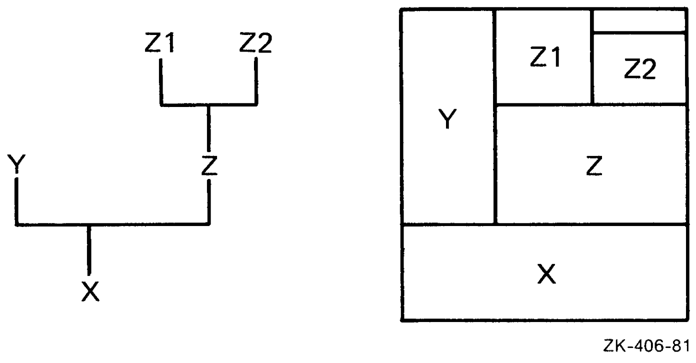
You use parentheses to delimit a group of segments that start at the same virtual address. The number of nested parenthetical groups cannot exceed 16.

For example:

```
.ROOT X-(Y,Z-(Z1,Z2))
.END
```

These directives describe the tree and its corresponding virtual address space shown in Figure 3-13.

Figure 3-13: Tree and Virtual Address Space Diagram



To create the overlay description for the task TK1 in Figure 3-3 (Section 3.1.1), you could create a file called TFIL.ODL that contains the following directives:

```
.ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0-(B1,B2),C)
.END
```

To build the task with that overlay structure, you would type one of the following command lines:

```
  TKB                          LINK

>TKB TK1,,=TFIL/MP           $ LINK/TASK:TK1 TFIL/OVERLAY_DESCRIPTION


                             $ LINK/TA:TK1 TFIL/OV
```

The /MP switch or the /OVERLAY_DESCRIPTION qualifier in the command lines above tells TKB that there is only one input file (TFIL.ODL) and that this file contains the overlay description for the task.

3.4.2 .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 other .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, enabling 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 3.4.1), 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 B0-(B1,B2) of the .FCTR directive. 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
```

3.4.3 Arguments for the .FCTR and .ROOT Directives

The arguments for the .FCTR and .ROOT directives may have different forms or syntax. The examples in this chapter use forms such as A1, B1, X, and Y for clarity, but the actual arguments that you use may have somewhat different names. This section lists the forms that the arguments may take for these directives. If you use an argument that does not fall into one of the following five categories, TKB takes the argument as that of the name of an object module file; in other words, the file name that you use must have the file type OBJ.

Note

When you use library file specifications in an ODL file, as in Sections 3.4.3.2 and 3.4.3.3, you must use the TKB /LB switch as described in those sections and in Chapter 10. There are no LINK equivalents to use within an ODL file.

3.4.3.1 Named Input File

You may use a named input file that has the object file format. For example:

```
CALC:   .FCTR [7,54]MULT.OBJ
```

The default file type is OBJ.

3.4.3.2 Specific Library Modules

You may name and therefore use specific object modules from a library file. For example:

```
BAKER:   .FCTR [300,3]COOKIE/LB:CHIP:OAT
```

COOKIE.OLB is the library file and CHIP and OAT are the modules that you want to extract from the file. The default file type is OLB and it need not be specified as part of the argument.

3.4.3.3 A Library to Resolve References Not Previously Resolved

You may specify a library as an argument in a .FCTR statement after extracting specific modules in a previous .FCTR statement. TKB uses the library to resolve symbols that may still be unresolved after extracting the modules. For example:

```
BAKER:   .FCTR [300,3]COOKIE/LB:CHIP:OAT
LIB:     .FCTR LB:[1,4]RECIPE/LB
```

3.4.3.4 A Section Name Used in a .PSECT Directive

You may use the name that you used as a program section name in the .PSECT directive as the argument in a .FCTR statement. For example:

```
        .PSECT  COM,GBL,D,RW,OVR
FSTCOM: .FCTR   COM
```

3.4.3.5 A Segment Name Used in a .NAME Directive

You may use the name that you specified as the name of a segment in the .NAME directive. For example:

```
        .NAME   SEG1,GBL,DSK
OVLY:   .FCTR   SEG1-MOD1-MOD2
```

3.4.4 Exclamation Point Operator

The exclamation point operator allows you to specify memory-resident overlay segments (see Section 3.1.2). 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 3-4 (Section 3.1.2) is as follows:

```
.ROOT CNTRL-!(D,E,F)
.END
```

In the example above, segments D, E, F are declared resident in separate areas of physical memory. The Task Builder determines the single starting virtual address for D, E, and F 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 or /COMPATIBLE qualifier is in effect) and adding this value to the total memory required by the task.

The exclamation point operator applies to that segment immediately to the right of the left parenthesis and those segments farther from the root on the same level with that segment. In other words, all parallel segments must be of the same residency type (disk resident or memory resident).

The exclamation point operator applies to segments at the same level from the root inside a pair of parentheses. Segments nested in parentheses within that level, but farther from the root, 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),!(x,y))
.END
```

In this example, B1 and (x,y) 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 valid; 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 invalid
.END
```

In this example, B1 is declared disk resident, so it is invalid to use the exclamation point to declare B2 and B3 memory resident.

3.4.5 .NAME Directive

The .NAME directive allows you to name a segment and 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. You use the .NAME directive prior to using the .ROOT or .FCTR directive. The Task Builder attaches attributes to a segment when it encounters the name in a .ROOT or .FCTR directive that defines the overlay segment. If you apply multiple names to a segment, the attributes of the last name given are in effect. This directive does the following:

- Names uniquely a segment that is loaded through the manual-load facility (see Chapter 4)
- Permits a named data-only segment to be loaded through the autoload mechanism

The format of the .NAME directive is as follows:

```
.NAME segname[,attr][,attr]
```

Parameters

segname

A 1- to 6-character name, which can consist of the Radix–50 characters A to Z, 0 to 9, and the dollar sign ($). (The period (.) cannot be used.)

attr

One of the following attributes:

GBL	The name is entered in the segment's global symbol table. The GBL attribute makes it possible to load data-only overlay segments by means of the autoload mechanism (see Chapter 4).
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), TKB 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.

3.4.5.1 Example of the Use of the .NAME Directive

In the following modified ODL file for TK1 (Figure 3-3 in Section 3.1.1), you provide names for the three main branches—A0, B0, and C—by specifying the names 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 cause TKB to enter the name BRNCH2 into the segment's global symbol table and suppress disk allocation for that segment. BRNCH2 contains uninitialized storage to be utilized at run time.

```
          .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 4.)

You can load the data overlay segment BRNCH2 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 11).

Note

In the absence of a unique .NAME specification, TKB establishes a segment name, using the first module name or library module name occurring in the segment.

3.4.6 .PSECT Directive

You can use the .PSECT directive to control 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 to Z, 0 to 9, and the dollar sign ($)) and its attributes are given in the .PSECT directive. The attributes used in the .PSECT directive must match those in the actual program section in the module. Thus, you can use the name in a .ROOT or .FCTR statement 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 shown in Figure 3-14 (the original version is shown in Figure 3-3 in Section 3.1.1).

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.

3.4.7 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 reads 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 following text:

```
B:     .FCTR B1-(B2,B3)
```

A line beginning with @BIND can replace this text at the position where the text would have appeared, as follows:

```
          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.

3.5 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.

3.5.1 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 3.4, this operator, when included within parentheses, defines a pair of segments that share storage. Including 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.

3.5.1.1 Defining Co-Trees with a Null Root by Using .NAME

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 zero in length. You can create a segment of this type, called a null segment, by means of the .NAME directive. The previous example is modified to move file Y0.OBJ to the root and include a null segment, as follows:

```
        .ROOT     X,Y
X:      .FCTR     X0-Y0-(X1,X2,X3)
        .NAME     YNUL
Y:      .FCTR     YNUL-(Y1,Y2)
        .END
```

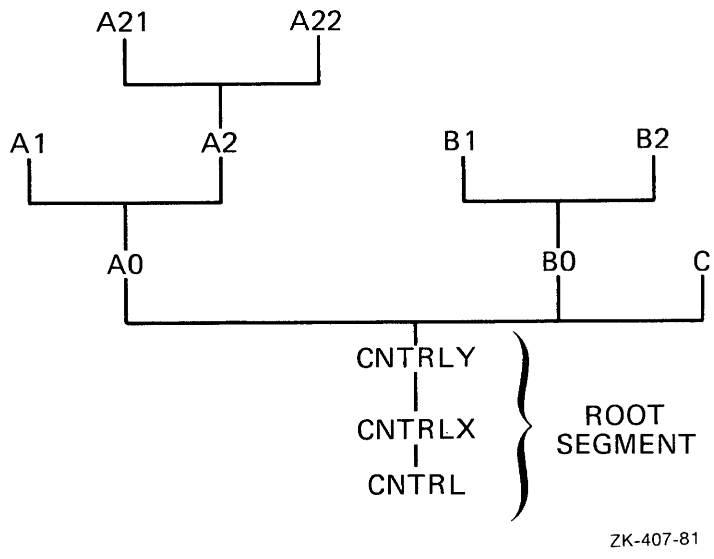
The null segment YNUL is created by using the .NAME directive, and replaces the co-tree root that formerly contained Y0.OBJ.

3.5.2 Example of a Multiple-Tree Structure

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 3.1.1) 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 3-14.

Figure 3–14: Overlay Tree for Modified TK1

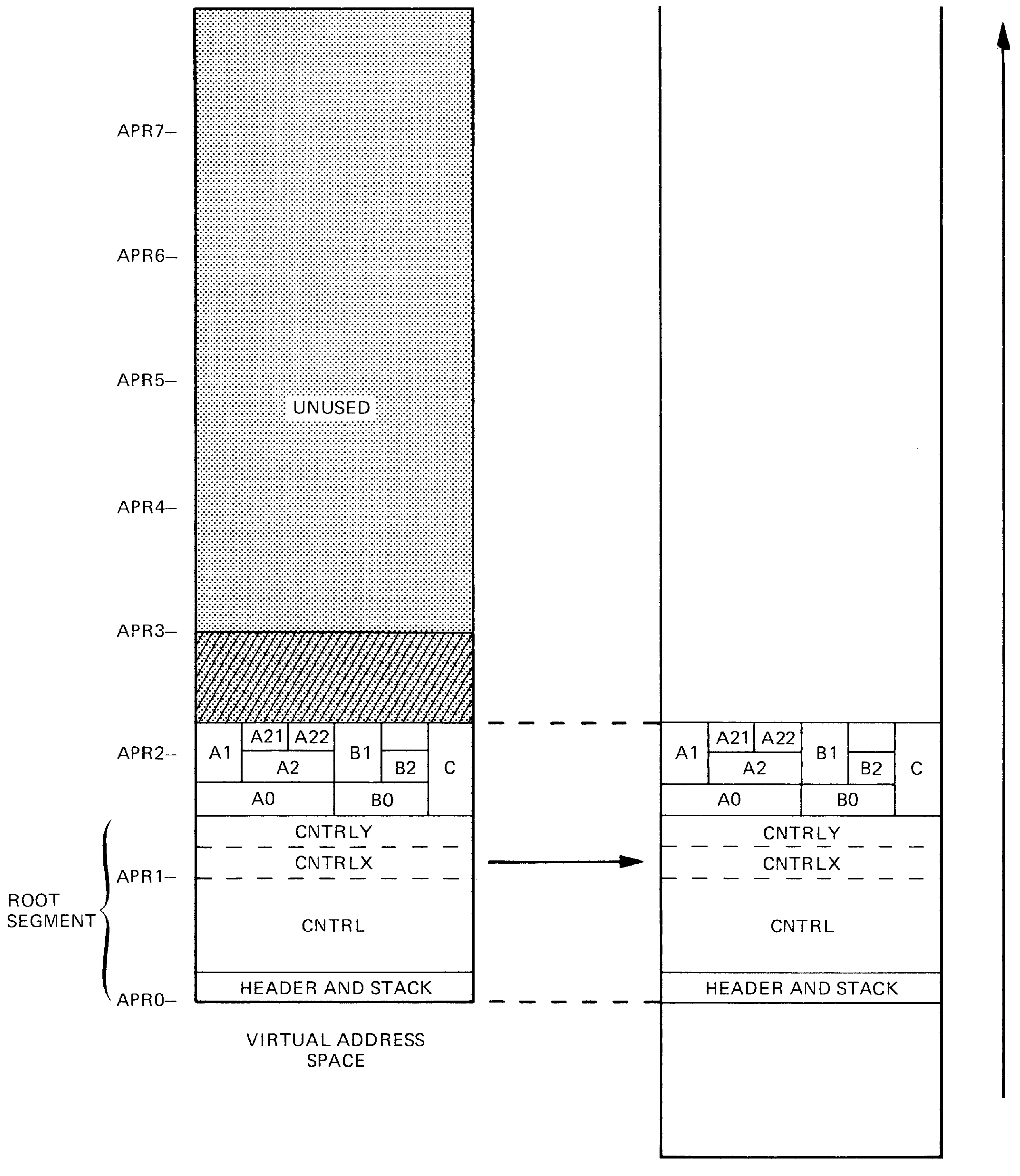


One possible overlay description for this structure is as follows:

```
          .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 3-15.

Figure 3-15: Virtual Address Space and Physical Memory for Modified TK1



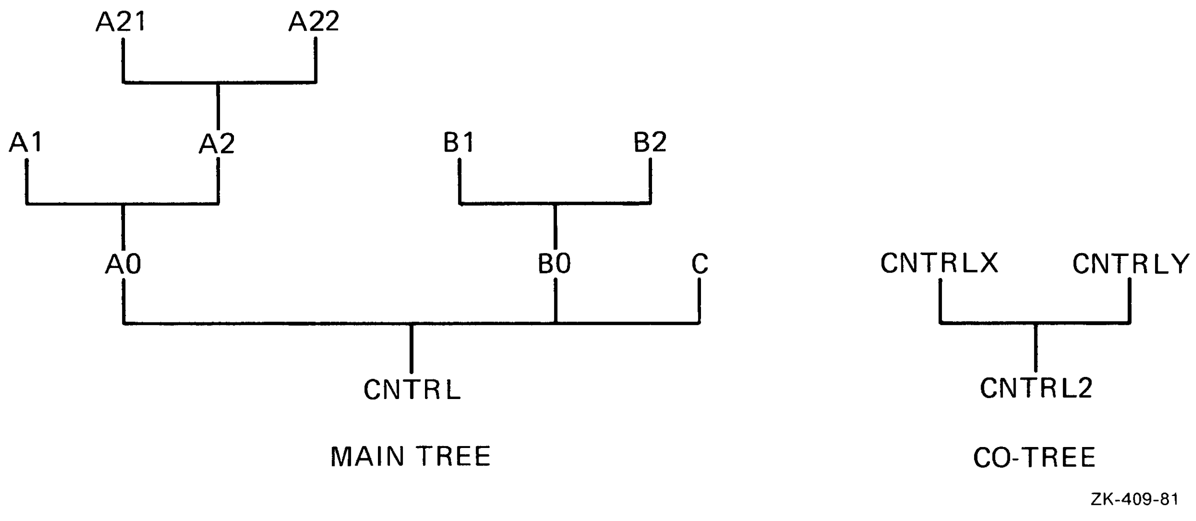
ZK-408-81

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 3-15 with the diagram in Figure 3-3.) The shaded portion of the unused virtual address space in Figure 3-15 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 5) or that contains Executive directives to extend its task region (see the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual*) requires at least 4K words of virtual address space to map each region. In such a task, using 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 3-16 shows the schematic representation of TK1 as a co-tree structure.

Figure 3-16: 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 zero. As noted earlier, the Task Builder requires the root segment in order to establish linkage with the overlay segments. One possible overlay description for building TK1 as a 2-tree structure is as follows:

```
         .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
```

You define the co-tree in the .ROOT directive by placing the comma operator outside all parentheses (immediately before CNTRL2). The .NAME directive creates the null root segment. Figure 3-16 shows the new relationship between virtual address space and physical memory.

The diagrams in Figure 3-17 illustrate the saving (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-word boundary of APR 2. TK1 has gained 4K words of potentially usable virtual address space.

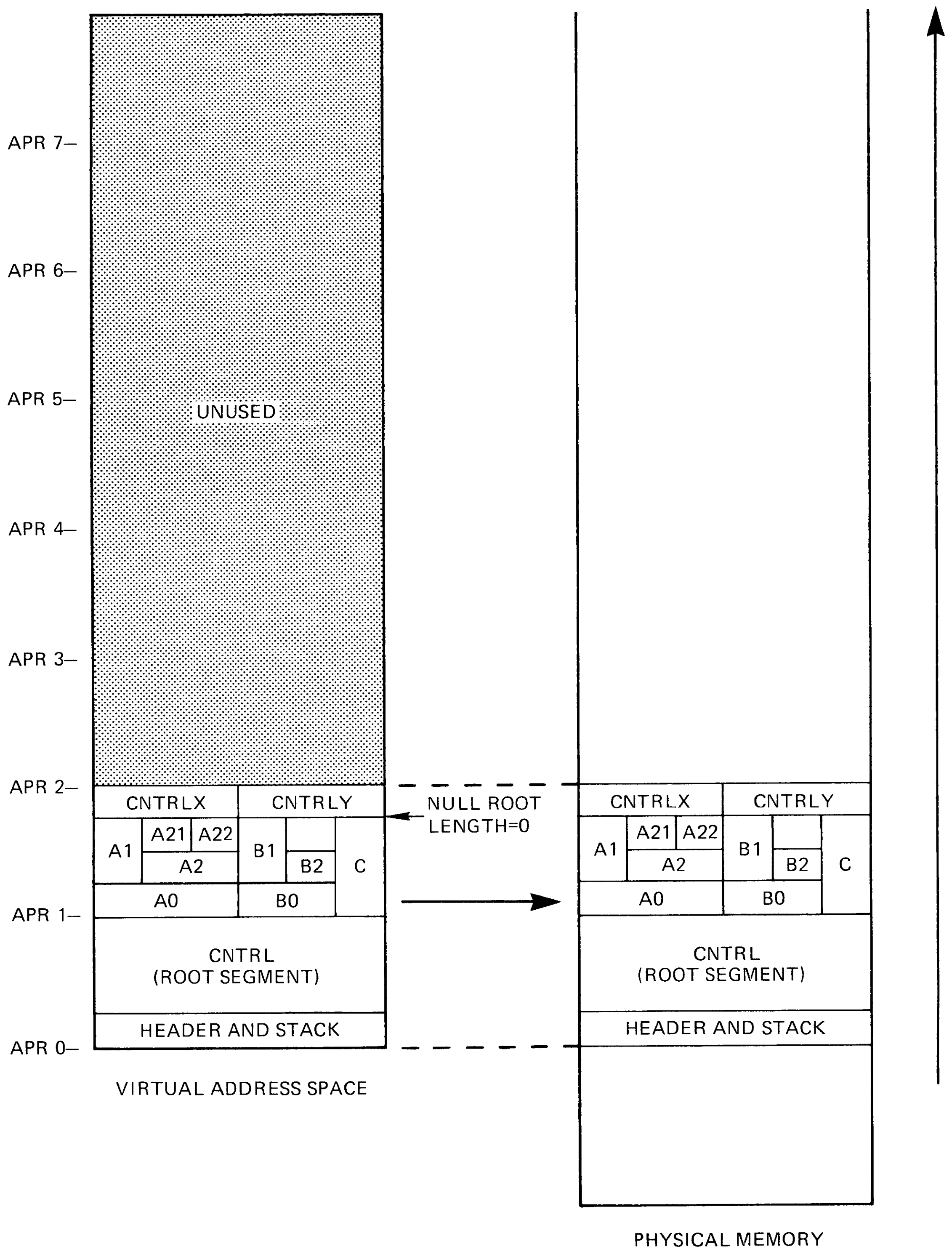
Note

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 zero bytes long, it does not require any virtual address space or physical memory and, therefore, does not appear in the diagrams in Figure 3-17.

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 3–17: Virtual Address Space and Physical Memory for TK1 as a Co-Tree

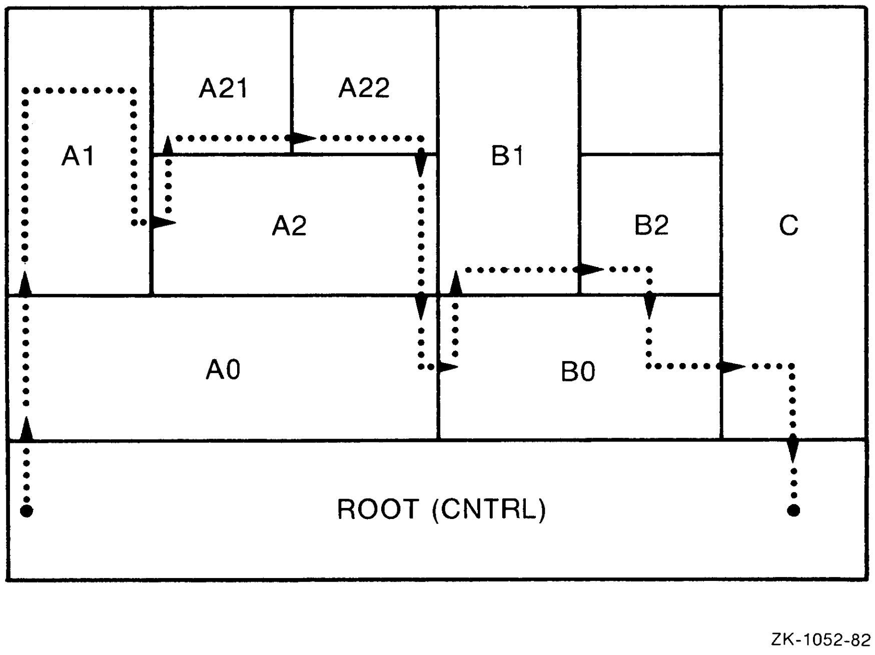


3.6 Creating an ODL File from a Virtual Address Space Allocation Diagram

You can use a graphic method as an aid to converting a virtual address space allocation diagram into the correct Task Builder ODL file.

First, create a virtual address space allocation diagram of your overlaid task, similar to that shown in Figure 3-18, without the dotted-line path shown in the diagram.

Figure 3–18: Virtual Address Space Allocation Diagram



The dotted-line path will be the basis for writing the ODL statements that you need. To determine the path through your virtual address space allocation diagram, follow these steps:

1. Start in the lower left corner of the root segment.
2. Draw a dotted line upward as far as you can go without passing through the top or into "empty" virtual space, crossing into new segments as needed.
3. When you reach the top segment, proceed to the right until you reach a vertical line.
4. If the end of your dotted line is now opposite the vertical line of the lowest segment, cross the vertical line and continue again from step 2; otherwise, proceed to step 5.

5. Because the end of your dotted line is not opposite the vertical line of the lowest segment, proceed downward until you reach the lowest segment.
6. If you are not in the root, cross the vertical line to the right and continue from step 2; otherwise, proceed to step 7.
7. If your dotted line is in the lower right corner of the root, you have finished the dotted-line walk.

Once you have drawn the dotted line, you should go back over it to verify that you followed all the steps. While doing this, draw arrowheads at each point where a line was crossed to indicate the direction of the line.

3.6.1 Creating a .ROOT Statement by Using a Virtual Address Space Allocation Diagram

Now you are ready to write the .ROOT statement. Follow these steps:

1. Write .ROOT followed by the name of the root statement (in this example, .ROOT CNTRL).
2. Follow the dotted-line path.
3. Add each successive ODL element to your root statement, using the following syntax, based on the direction of your dotted line:
   a. At an upward crossing: -("name of new segment"
   b. At a horizontal crossing: ,"name of new segment"
   c. At a downward crossing:)
4. When you have returned to the root, your root statement is complete.

Using the dotted-line path in Figure 3-18 and the above associated steps, you can write a .ROOT statement by using the following steps:

1. Step 1 : Write .ROOT CNTRL
2. Step 3A: Write .ROOT CNTRL-(A0
3. Step 3A: Write .ROOT CNTRL-(A0-(A1
4. Step 3B: Write .ROOT CNTRL-(A0-(A1,A2
5. Step 3A: Write .ROOT CNTRL-(A0-(A1,A2-(A21
6. Step 3B: Write .ROOT CNTRL-(A0-(A1,A2-(A21,A22
7. Step 3C: Write .ROOT CNTRL-(A0-(A1,A2-(A21,A22)
8. Step 3C: Write .ROOT CNTRL-(A0-(A1,A2-(A21,A22))
9. Step 3B: Write .ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0
10. Step 3A: Write .ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0-(B1
11. Step 3B: Write .ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0-(B1,B2

12. Step 3C: Write .ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0-(B1,B2)
13. Step 3B: Write .ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0-(B1,B2),C
14. Step 3C: Write .ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0-(B1,B2),C)

The steps for writing .FCTR statements and co-tree statements follow next.

3.6.2 Creating a .FCTR Statement by Using a Virtual Address Space Allocation Diagram

By using the steps for creating a .ROOT statement from a virtual address space allocation diagram, you created the following .ROOT statement:

```
.ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0-(B1,B2),C)
```

It may be desirable to simplify your specific .ROOT statement into one or more .FCTR statements. A technique similar to the one used to create the .ROOT statement may be used to create the .FCTR statement.

In this example, segments A0, A1, A2, A21, and A22 are selected to be in the .FCTR statement. Having selected these segments (normally related as a "stack" of segments), you are now ready to write down the .FCTR statement.

First, draw a virtual address space allocation diagram of the segments that you have selected. (You can use Figure 3-18 for this explanation.) Then, follow these next steps to draw a dotted-line path through the diagram:

1. Start in the lower left corner of the lowest or "base" segment (A0) in your diagram.
2. Draw a dotted line upward as far as you can go without passing through the top or into empty virtual space, crossing into new segments as needed.
3. When you reach the top segment, proceed to the right until you reach a vertical line.
4. If the end of your dotted line is now opposite the vertical line of the lowest segment, cross the vertical line and continue again from step 2; otherwise, proceed to step 5.
5. Because the end of your dotted line is not opposite the vertical line of the lowest segment, proceed downward until you reach the lowest segment.
6. If you are not in the base segment (A0), cross the vertical line to the right and continue from step 2; otherwise, proceed to step 7.
7. If your dotted line is in the lower right corner of the base segment, you have finished the dotted-line walk.

Once you have drawn the dotted line, you should go back over it to verify that you followed all the steps. While doing this, draw arrowheads at each point where a line was crossed to indicate the direction of the line.

Now you are ready to write the .FCTR statement. Follow these next steps:

1. Write a label followed by .FCTR, which is in turn followed by the name of the first segment (A0) (in this example, AFCTR .FCTR A0).
2. Follow the dotted-line path.

3. Add each successive ODL element to your root statement, using the following syntax, based on the direction of your dotted line:
    a. At an upward crossing: ("name of new segment"
    b. At a horizontal crossing: ,"name of new segment"
    c. At a downward crossing:)
4. When you have returned to the base segment, your .FCTR statement is complete.

Using the dotted-line path and the above associated steps, you can write a .FCTR statement by using the following steps:

1. Step 1 : Write AFCTR .FCTR A0
2. Step 3A: Write AFCTR .FCTR A0-(A1
3. Step 3B: Write AFCTR .FCTR A0-(A1,A2
4. Step 3A: Write AFCTR .FCTR A0-(A1,A2-(A21
5. Step 3B: Write AFCTR .FCTR A0-(A1,A2-(A21,A22
6. Step 3C: Write AFCTR .FCTR A0-(A1,A2-(A21,A22)
7. Step 3C: Write AFCTR .FCTR A0-(A1,A2-(A21,A22))

You have now reached the base segment and have written the two ODL statements:

```
          .ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0-(B1,B2),C)
AFCTR:    .FCTR A0-(A1,A2-(A21,A22))
```

The last step requires that you substitute the label AFCTR into the .ROOT statement, which results in the following:

```
         .ROOT CNTRL-(AFCTR,B0-(B1,B2),C)
AFCTR:   .FCTR A0-(A1,A2-(A21,A22))
```

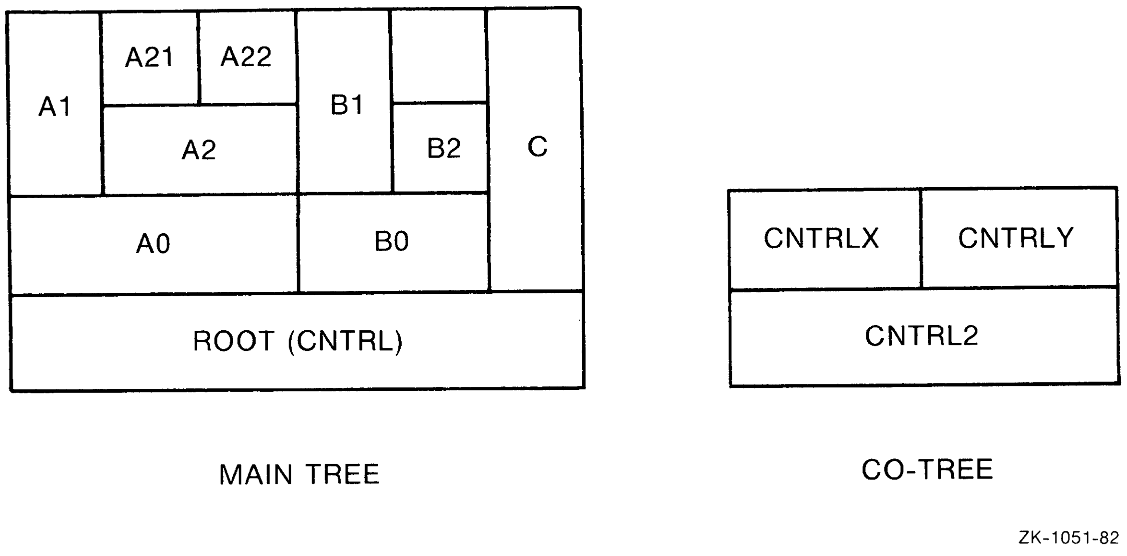
Additional .FCTR statements would be determined and written in the same way. For example, you could write a .FCTR statement labeled BFCTR for the segments B0, B1, and B2.

The following section shows how to write an ODL statement for a co-tree by using the same methods.

3.6.3 Creating an ODL Statement for a Co-Tree by Using a Virtual Address Space Diagram

Assuming that you want to write an ODL statement for a co-tree like the one in Figure 3-19, you would have two virtual address space allocation diagrams, one for the main tree and one for the co-tree. These two diagrams are shown in Figure 3-19.

Figure 3–19: Virtual Address Space Allocation for a Main Tree and Its Co-Tree



From Figure 3-19, you see that the co-tree is a stack of segments also. Therefore, it is possible to write the statement for the co-tree in the same fashion and with the same rules as that described in Section 3.6. However, the following facts must be kept in mind:

- The co-tree has a null root.
- A .NAME statement must be used to name the null root.
- A comma must be placed outside of the parentheses and at the end of that part of the .ROOT statement that defines the main tree.

Therefore, the ODL statement that you obtain before writing the co-tree part is as follows:

```
        .NAME CNTRL2
        .ROOT CNTRL-(AFCTR,BO-(B1,B2),C)
AFCTR:  .FCTR AO-(A1,A2-(A21,A22))
```

By following the rules in Section 3.6 and by using the diagram in Figure 3-19, you can then create the following ODL statement:

```
        .NAME CNTRL2
        .ROOT CNTRL-(AFCTR,BO-(B1,B2),C),CNTRL2-(CNTRLX,CNTRLY)
AFCTR:  .FCTR AO-(A1,A2-(A21,A22))
```

3.7 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:

- TKB 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 run time.
- 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:

- You can increase TKB throughput by linking commonly used library routines into the main root segment.
- You can eliminate ambiguous definitions, multiple definitions, and cross-tree references by using the /NOFU switch (the TKB default) to restrict the scope of the default library search. However, restricting the scope of the default library search may also cause problems.

If sufficient memory is available, you can effectively place the Object Time System 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.

3.8 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 refers to it, MAIN.TSK (from Example 5-3, Chapter 5), 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 (ODL). It does not reflect the most efficient use of it.

The following alterations were made to each of the routines for this example:

- A .TITLE and a .END assembler directive were added to each routine to establish it as a unique module.

- The following assembler directive was added to each arithmetic routine to increase its allocation:

  ```
  .BLKW 1024.*3
  ```

  This was done to make TKB allocation of address space more obvious for documentation purposes.

The operation of the overlaid task is identical to that of Example 5-3 in Chapter 5. 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 as follows:

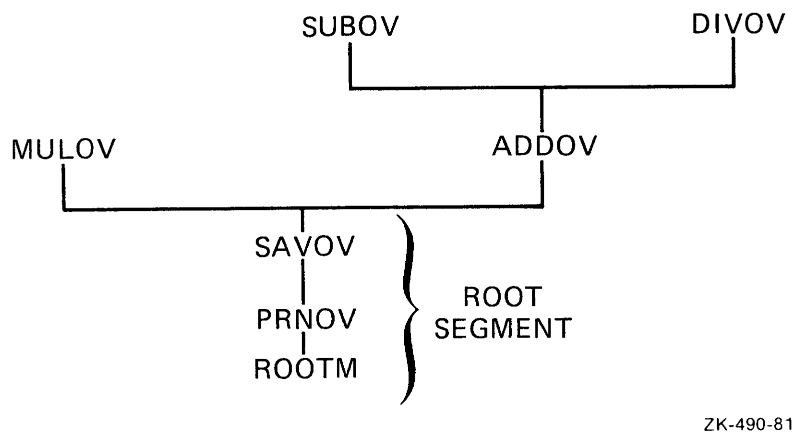
Module	**Length (in Octal)**
ADDOV	14024 bytes
SUBOV	14024 bytes
MULOV	14024 bytes
DIVOV	14026 bytes
SAVOV	4042 bytes
PRNOV	4260 bytes
ROOTM	4104 bytes

The flow of control for OVR.TSK is as follows:

1. ROOTM calls ADDOV and ADDOV returns to ROOTM.
2. ROOTM calls PRNOV to print the result and PRNOV returns to ROOTM.
3. ROOTM calls SUBOV and SUBOV returns to ROOTM.
4. ROOTM calls PRNOV to print the result and PRNOV returns to ROOTM.
5. ROOTM calls DIVOV and DIVOV returns to ROOTM.
6. ROOTM calls PRNOV to print the result and PRNOV returns to ROOTM.
7. ROOTM calls MULOV and MULOV returns to ROOTM.
8. 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, you can place PRNOV 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 3-20.

Figure 3-20: Overlay Tree of Virtual Address Space for OVR.TSK



To build this overlay, first create an ODL file (OVERTREE.ODL) that contains the description for the overlay:

```
.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 TKB command line:

```
>TKB OVR,OVR/-SP,=OVRTREE/MP RET
```

or with the following LINK command line:

```
$ LINK/TAS:OVR/MAP:OVR/NOPRINT OVRTREE/OVER RET
```

This command line instructs TKB to build a task image, OVR.TSK, and to create a map file, OVR.MAP, in the directory that corresponds to the terminal UIC. The negated spool switch /-SP, or /NOPRINT as a LINK qualifier, inhibits TKB from spooling the map file to the line printer.

The overlay switch /MP attached to the input file, or /OVER as a file qualifier, tells TKB that the input file is an ODL file. Therefore, this file will be the only input file specified. Refer to Chapter 10 for a description of the switches and Chapter 11 for the qualifiers used in this example.

A portion of the map that results from this task build is shown in Example 3-1.

Example 3-1: Map File for OVR.TSK

```
OVR.TSK    Memory allocation map  TKB M43.00          Page 1
                      01-JAN-87  10:06


Partition name : GEN
Identification : 01
Task  UIC      : [7,62]

Stack    limits: 000260 001257 001000 00512.
PRG xfr address: 001264
                                      ❷
Total address windows: 1.
Task  image  size  : 7488. words

Task address limits: 000000 035107
R-W disk blk limits: 000002 000073 000072 00058.


OVR.TSK Overlay description:

Base ❶ Top  ❸     Length
----    ---        ------
000000  005033  005034  02588.    ROOTM
005034  021057  014024  06164.         MULOV
005034  021057  014024  06164.         ADDOV
021060  035103  014024  06164.              SUBOV
021060  035107  014030  06168.              DIVOV
                           .
        ❺      ❹          .
                           .
*** Root segment:  ROOTM


R/W mem  limits: 000000 005033 005034 02588.
Disk blk limits: 000002 000007 000006 00006.


Memory allocation synopsis:

Section                                          Title  Ident  File
-------                                          -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001260 002514 01356.
                          001260 000102 00066.   ROOTM  01     ROOTM.OBJ;1
                          001362 000260 00176.   PRNOV  01     PRNOV.OBJ;1
                          001642 000042 00034.   SAVOV  01     SAVOV.OBJ;1
ANS    :(RW,D,GBL,REL,OVR) 003774 000002 00002.
                          003774 000002 00002.   ROOTM  01     ROOTM.OBJ;1
                          003774 000002 00002.   PRNOV  01     PRNOV.OBJ;1
                           .
                           .
                           .
```

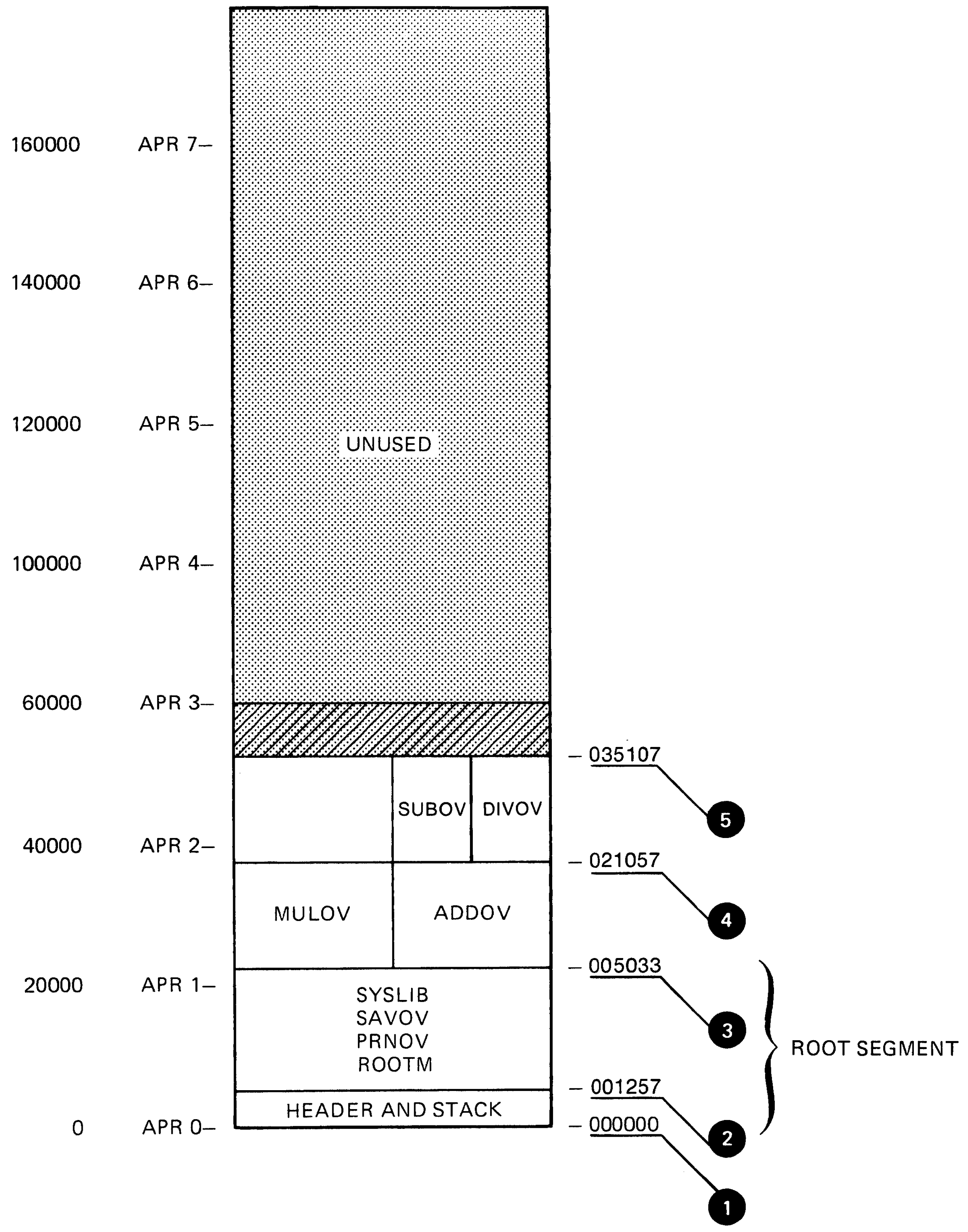
(Continued on next page)

Example 3-1 (Cont.): Map File for OVR.TSK

```
Global symbols:

AADD    004032-R  DIVV    004052-R  PRINT  001550-R  SUBB    004042-R
MULL    004022-R  SAVAL   001642-R
                            .
                            .
                            .

*** Task builder statistics:

    Total work file references: 6863.
    Work  file  reads: 0.
    Work  file writes: 0.

    Size of core pool: 7086. words (27. pages)
    Size of work file: 3072. words (12. pages)

Elapsed time:00:00:14
```

Figure 3-21 shows the allocation of virtual address space for OVR.TSK. The circled numbers in Example 3-1 correspond to those in Figure 3-21.

Figure 3-21: 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 as follows:

```
4104 + 4260 + 4042 = 14,426 bytes
```

After task building, the root segment is 20,677$_8$ bytes long. TKB 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 farther up the tree, the Task Builder would have added data structures to the calling segments as well. (Refer to Chapter 4 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 follows:

```
.ROOT   ROOTM-PRNOV-SAVOV-*!(MULOV,ADDOV-!(SUBOV,DIVOV))
.END
```

For this example, the name of the ODL file and the task image file has been changed to RESOVR.ODL to distinguish it from the disk-resident version.

You can build RESOVR with the following TKB command line:

```
>TKB RESOVR,RESOVR/-SP,=RESOVR/MP [RET]
```

or with the following LINK command line:

```
$ LINK/TASK:RESOVR/MAP:RESOVR/NOPRINT RESOVR/OVER [RET]
```

These commands direct TKB to build a task named RESOVR.TSK and to create a map file named RESOVR.MAP. The negated spooling switch /-SP, or /NOPRINT as a LINK qualifier, inhibits spooling of the map file.

The /MP switch on the input file, or /OVER as a file qualifier, tells TKB that the file is an ODL file and that it will be the only input file for this task build. Refer to Chapter 10 for a description of the switches and Chapter 11 for the qualifiers used in this example.

A portion of the map that results from this task build is shown in Example 3-2.

Example 3-2: Map File for RESOVR.TSK

```
Partition name : GEN
Identification : 01
Task  UIC      : [7,62]

Stack    limits: 000320 001317 001000 00512.
PRG xfr address: 001324
                                          ❷
Total address windows: 3.
Task  image  size  : 13920. words
Task address limits: 000000 057777
R-W disk blk limits: 000003 000074 000072 00058.


RESOVR.TSK Overlay description:
Base ❶ Top  ❸    Length
----    ---      ------
000000  005677  005700  03008.     ROOTM
020000  034077  014100  06208.          MULOV
020000  034077  014100  06208.          ADDOV
040000  054077  014100  06208.               SUBOV
040000  054077  014100  06208.               DIVOV
   ❻  ❹    ❼  ❺
                        .
                        .
                        .
*** Root segment: ROOTM

R/W mem  limits: 000000 005677 005700 03008.
Disk blk limits: 000003 000010 000006 00006.

Memory allocation synopsis:

Section                                   Title  Ident  File
-------                                   -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001320 002514 01356.
                          001320 000102 00066. ROOTM  01     ROOTM.OBJ;1
                          001422 000260 00176. PRNOV  01     PRNOV.OBJ;1
                          001702 000042 00034. SAVOV  01     SAVOV.OBJ;1

ANS   :(RW,D,GBL,REL,OVR) 004034 000002 00002.
                          004034 000002 00002. ROOTM  01     ROOTM.OBJ;1
                          004034 000002 00002. PRNOV  01     PRNOV.OBJ;1
                        .
                        .
                        .
```

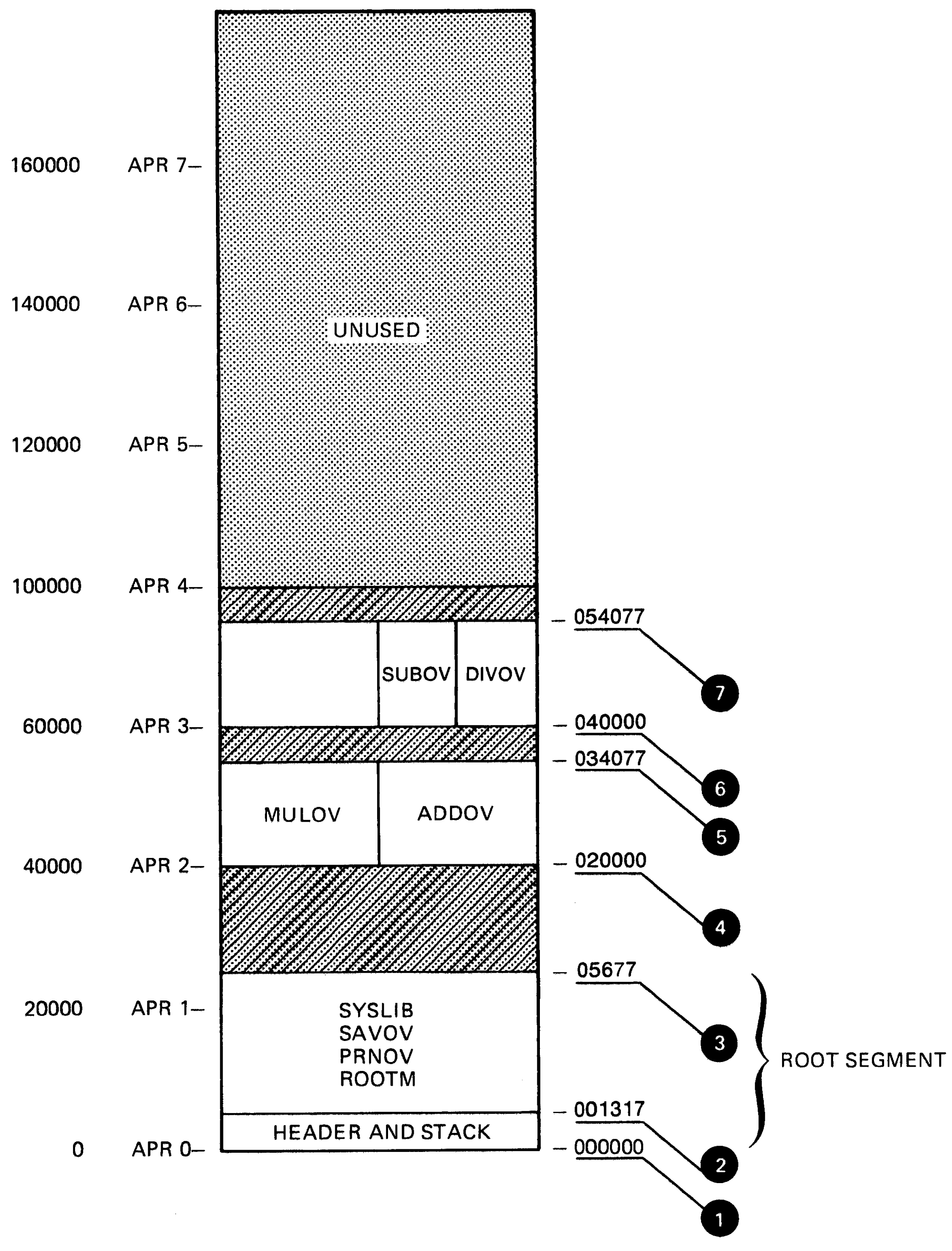
(Continued on next page)

Example 3–2 (Cont.): Map File for RESOVR.TSK

```
Global symbols:

AADD    004072-R  DIVV   004112-R  PRINT  001610-R  SUBB    004102-R
MULL    004062-R  SAVAL  001702-R
                         .
                         .
                         .

*** Task builder statistics:

    Total work file references: 6938.
    Work file reads  : 0.
    Work file writes : 0.

Size of core pool: 4178. words (16. pages)
Size of work file: 3072. words (12. pages)

Elapsed time:00:00:21
```

Figure 3-22 shows the allocation of virtual address space for RESOVR.TSK. The circled numbers in Example 3-2 correspond to those in Figure 3-22.

Figure 3–22: Allocation of Virtual Address Space for RESOVR.TSK



ZK-412-81

Note that TKB 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.

3.9 Window Blocks in Overlays

Finally, note in Figure 3-22 that TKB 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 accesses a resident library or common because of the extra window blocks required to use these features.

3.10 Summary of the Overlay Description Language

- 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.

- An ODL file is a text file consisting of a series of overlay description directives, one directive on each line. You enter this file in the TKB or LINK command line, and identify it as an ODL file by attaching the /MP switch for TKB or the /OVERLAY_DESCRIPTION qualifier for LINK to the file name. If you enter an ODL file in the TKB or LINK command line, it must be the only input file you specify.

- The Overlay Description Language provides the following directives for specifying the tree representation of the overlay structure:

  - .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.

  - .PSECT, .FCTR, and .NAME—These can be used in any order in the ODL file.

- You define the tree structure using the hyphen (-), comma (,), and exclamation point (!) operators, and by using parentheses, as follows:

  - The hyphen operator (-) indicates that its arguments are to be concatenated and thus are to coexist in memory.

  - The comma operator (,) within parentheses indicates that its arguments are to overlay each other either physically, if disk resident, or virtually, if memory resident.

  - The comma operator not within parentheses delimits overlay trees.

  - 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.

- 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.

- The .ROOT directive defines the beginning overlay structure. The arguments of the .ROOT directive are one or more of the following:
  - File specifications as described in Chapter 1
  - Factor labels
  - Segment names
  - Program section names
- The .END directive terminates input.
- The .FCTR directive provides a means for replacing text by a symbolic reference (the factor label). This replacement is useful for the following reasons:
  - The .FCTR directive extends the text of the .ROOT directive to more than one line and thus allows complex trees to be represented.
  - The .FCTR directive allows you to write the overlay description 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, you can write this overlay description 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.

- 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 to Z, 0 to 9, the dollar sign ($), or period (.)) 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, therefore, you need only specify the attributes that do not correspond to the defaults: .PSECT ALPHA,GBL.

- 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 4 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
  ```

  ### Parameters

  segname

  The designated name (composed of the characters A to Z, 0 to 9, and the dollar sign ($)).

  attr

  An optional attribute taken from the following list: GBL, NOGBL, DSK, NODSK.

  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.

- 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 4
Overlay Loading Methods

RSX-11M-PLUS and Micro/RSX systems provide the following 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, as follows:

- 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.

With 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 the main tree, as well as the root segments and the overlay segments of the co-trees. Once loaded, the root segment of a co-tree remains in memory.

4.1 Autoload

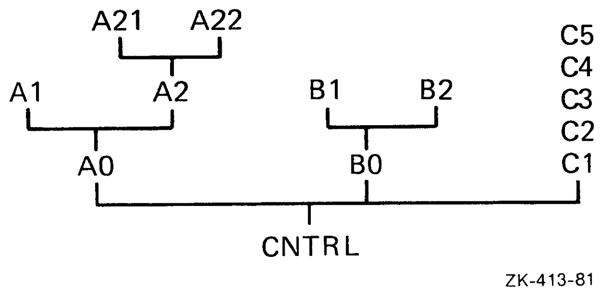
To specify the autoload method, you use the autoload indicator, an asterisk (*). You place this indicator in the ODL description of the task at the points where loading must occur. The execution of a transfer-of-control instruction to an autoloadable segment up-tree (farther away from the root) initiates the autoload process.

4.1.1 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.

In the TK1 example of Chapter 3, Section 3.1.1, if segment C consisted of a set of modules C1, C2, C3, C4, and C5, the tree diagram would be as shown in Figure 4-1.

Figure 4-1: 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 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
```

When you use autoload, the root of a co-tree is loaded by path loading if one of the branches of the co-tree is called before the root. However, if the root of the co-tree is called before the branch is called, the root must have an autoload indicator.

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 loaded properly. For example, if the co-tree root (CNTRL2) of the multiple-tree example, Section 3.5.2, 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) ❶
AFCTR:   .FCTR A0-*ASUB1-ASUB2-*(A1,A2-(A21,A22)) ❷ ❸
BFCTR:   .FCTR (B0-(B1,B2))
CFCTR:   .FCTR CNAM-C1-C2-C3-C4-C5
         .NAME CNAM,GBL ❶
         .PSECT ASUB1,I,GBL,OVR ❷
         .PSECT ASUB2,I,GBL,OVR
         .END
```

The following notes are keyed to the example above:

❶ The autoload indicator is applied to each factor name; therefore:

a. *AFCTR=*A0

b. *BFCTR=*(B0-(B1,B2))

c. *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.

❷ The autoload indicator is applied to the name of a program section with the instruction (I) attribute (*ASUB1), so that program section ASUB1 is made autoloadable.

❸ 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.

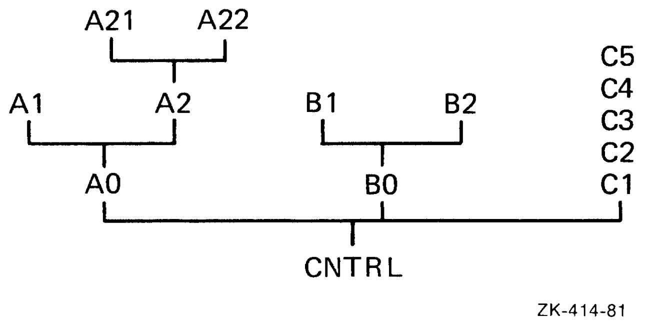
4.1.2 Path Loading

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.

In Figure 4-2, if CNTRL calls A22, all the modules between the CNTRL and A2 are loaded. In this case, modules A0 and A2 are loaded.

Figure 4–2: Path-Loading Example



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.

4.1.3 Autoload Vectors

To resolve a reference up-tree to a global symbol in an autoloadable segment, TKB 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 for conventional tasks is shown in Figure 4-3 and the format for I- and D-space tasks is shown in Figure 4-4.

Figure 4–3: Autoload Vector Format for Conventional Tasks

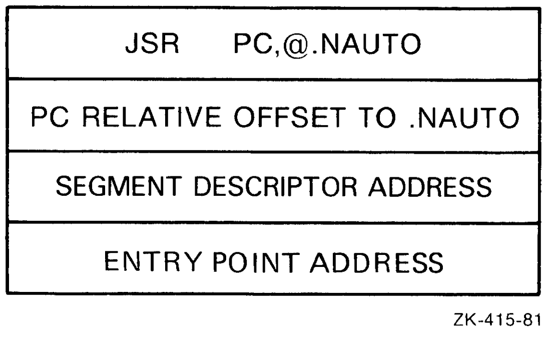
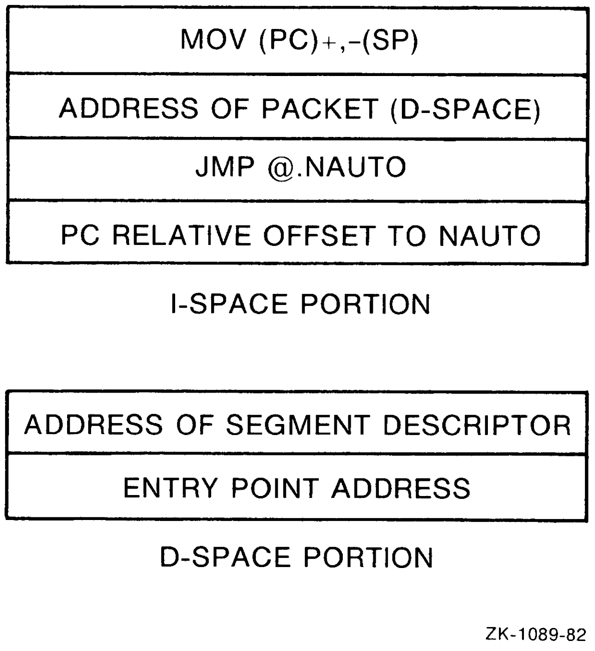


Figure 4–4: Autoload Vector Format for I- and D-Space Tasks



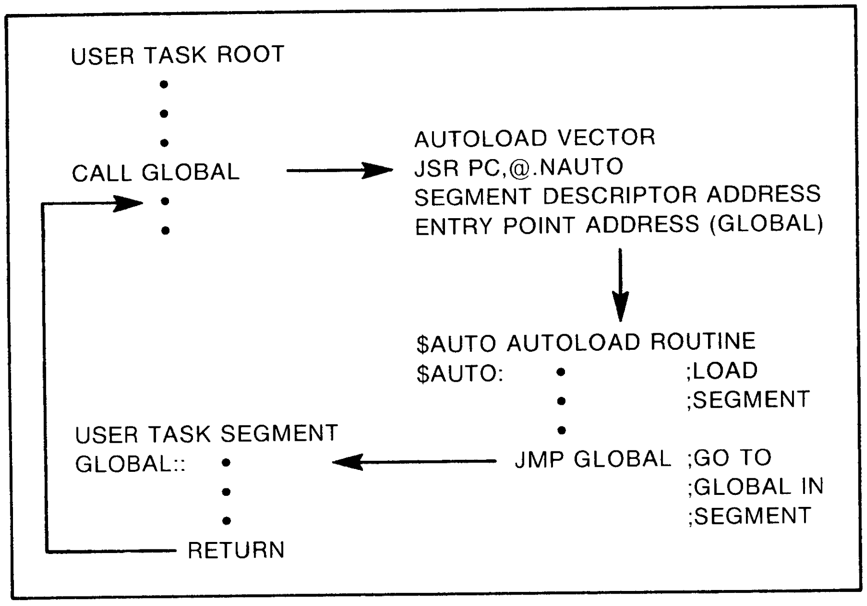
For I- and D-space tasks, TKB generates the autoload vector in a format that differs from the vector in a conventional task. The I- and D-space autoload vector is six words long and consists of two parts: one part residing in I-space and the other part residing in D-space. The I-space part consists of two 2-word instructions and the D-space part consists of two words of data. The data in the vector are the segment descriptor address and the target entry point address. The I- and D-space vector is shown in Figure 4-4.

The task root and the overlay segments may contain autoload vectors; the I-space part of the root or segment contains the I-space part of the vectors and the D-space part of the root or segment contains the D-space part of the vectors.

The MOV instruction in the I-space part of the vector places the address of the D-space part of the vector on the stack. The second instruction in the vector executes an indirect JMP to $AUTO through the location .NAUTO.

In Figures 4-3 and 4-4, a transfer-of-control instruction to the up-tree global symbol generates an autoload vector in the shown format. An example of the code sequence used in a call to a global symbol in an autoloadable segment is shown in Figure 4-5.

Figure 4-5: Example Autoload Code Sequence for a Conventional Task



ZK-416-81

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 TKB 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 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, because the call to C1 loads the segment that contains C2, C3, C4, and C5, 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, TKB generates seven autoload vectors—for A1, A21, A22, B0, B1, B2, and C1.

Note

Autoload vectors supplied by the symbol table (STB) files of the resident libraries are selectively included in your task image. Only referenced symbols result in autoload vectors that are present in the task image.

4.1.4 Autoloadable Data Segments

You can make overlay segments that contain no executable code autoloadable. To do so, you must include a .NAME directive and specify the GBL attribute, as described in Section 3.4.4. 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. Because 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.

4.2 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. For memory-resident overlays, the segment is mapped, but read in only if it was not previously read in.

A MACRO–11 programmer calls the $LOAD routine directly. A FORTRAN programmer calls $LOAD using the FORTRAN subroutine MNLOAD.

4.2.1 MACRO–11 Manual-Load Calling Sequence

A MACRO–11 programmer calls the $LOAD routine as follows:

```
MOV     #BLK,RO
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] or 0
            .WORD  [ast-trp] or 0
```

Parameters

length
The length of the parameter block (three to five 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.

4.2.2 MACRO–11 Manual-Load Calling Sequence for I- and D-Space Tasks

A MACRO–11 programmer calls the $LOAD routine as follows:

```
MOV     #PBLK,R0
CALL    $LOAD
```

PBLK is the address of a parameter block with the following format in an I- and D-space task:

```
PBLK:    BYTE 3,0
         .RAD50 /seg-name/
```

Parameters

length

The length of the parameter block (three words).

event-flag

Specify this as 0. Only synchronous load requests are possible when loading I- and D-space segments.

seg-name

The name of the segment to be loaded: a 1- to 6-character Radix–50 name, occupying two words.

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, which is the only one possible for I- and D-space segments, the return of the condition code C=0 means that the desired segment is loaded and is ready to be executed. Your program must ensure that loading has been successful by checking for the condition code rather than assuming that the segment has been loaded.

4.2.3 FORTRAN Manual-Load Calling Sequence

To use the manual-load method in a FORTRAN program, your program must refer to the $LOAD routine by means of the MNLOAD subroutine. The subroutine call has the following form:

```
CALL MNLOAD(seg-name,[event-flag],[i/o-status],[ast-trp],[ld-ind])
```

Parameters

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 *RSX-11M-PLUS and Micro/RSX Executive Reference Manual.*

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 valid:

- CALL MNLOAD (SEGA1)

  Loads segment named in SEGA1 synchronously.

- CALL MNLOAD (SEGA1,0,,,LDIND)

  Loads segment named in SEGA1 synchronously and returns success indicator to LDIND.

- CALL MNLOAD (SEGA1,1,IOSTAT,ASTSUB,LDIND)

  Loads segment named in SEGA1 asynchronously, transferring control to ASTSUB upon completion of the load request. Stores the I/O status doubleword in IOSTAT and the success indicator in LDIND.

The following example uses the program CNTRL, previously discussed in Section 4.1. 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.

4.2.4 FORTRAN Manual-Load Calling Sequence for I- and D-Space Tasks

To use the manual-load method in a FORTRAN program, your program must refer to the $LOAD routine by means of the MNLOAD subroutine. The subroutine call has the following form:

```
CALL MNLOAD(seg-name,,,,[ld-ind])
```

Parameters

seg-name

A 2-word real variable containing the segment name in Radix–50 format.

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 valid:

- CALL MNLOAD (SEGB1)

  Loads segment named in SEGB1 synchronously.

- CALL MNLOAD (SEGB1,,,,LDIND)

  Loads segment named in SEGB1 synchronously and returns success indicator to LDIND.

Only synchronous loading is possible when manually loading I- and D-space task segments.

4.3 Error Handling

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 (WTSE$) 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.

If you select the manual-load method, you must provide error-handling routines that diagnose load errors and provide appropriate recovery. A more comprehensive user-written error recovery subroutine can be substituted for the system-provided routine if the following conventions are observed:

- The error recovery routine must have the entry point name $ALERR.
- The contents of all registers must be saved and restored.

On entry to $ALERR, register 2 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 *RSX-11M-PLUS and Micro/RSX I/O Operations Reference Manual.*)

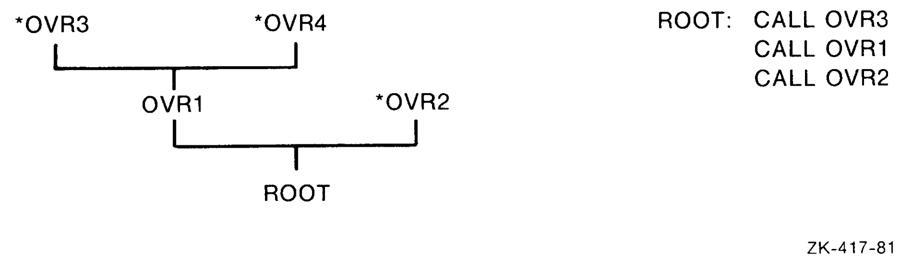
4.4 Global Cross-Reference of an Overlaid Task

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 overlay segments composed of modules OVR1, OVR2, OVR3, and OVR4. The overlay description of the file is as follows:

```
            .ROOT  ROOT-(OVR,*OVR2)
OVR:        .FCTR  OVR1-*(OVR3,OVR4)
```

Only segments OVR2, OVR3, and OVR4 are autoloadable. Figure 4-6 shows the resulting overlay tree.

Figure 4–6: Autoload Overlay Tree Example



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 because OVR1 is not autoloadable, it depends on a call to OVR3 or OVR4 to get loaded by path loading. The asterisk indicates that the module contains an autoloadable definition. The modules shown with the asterisk define the symbol.

The asterisks preceding the modules OVR2, OVR3, and OVR4 indicate that the global symbols OVR2, OVR3, and OVR4 are autoload symbols and are referenced from the module ROOT through an autoload vector, as shown by the at sign (@).

The asterisk and at sign are shown in the cross-reference listing in Example 4-1.

Down-tree references to the global symbol ROOT are made from modules OVR1, OVR2, OVR3, and OVR4. These references are resolved directly.

The segment cross-reference shows the segment name and modules in each overlay.

Example 4-1: Cross-Reference Listing of Overlaid Task

```
OVRTST     CREATED BY   TKB     ON 27-JUL-87 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

OVR3    002014-R   * OVR3    @ ROOT
OVR4    002014-R   * OVR4    @ ROOT

ROOT    001176-R   # 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
```

(Continued on next page)

Example 4-1 (Cont.): Cross-Reference Listing of Overlaid Task

```
OVRTST     CREATED BY   TKB    ON 27-JUL-87 AT 12:04      PAGE 2

SEGMENT CROSS REFERENCE                                   CREF      V01

SEGMENT NAME     RESIDENT MODULES

OVR1             OVR1
OVR2             OVR2

OVR3             OVR3
OVR4             OVR4
ROOT             ALERR   AUTO    OVCTL    CVDAT    OVRES    ROOT    SAVRG
                 VCTDF
```

4.5 Use and Size of Overlay Run-Time Routines

TKB, when constructing an overlaid task, incorporates certain modules from the system library to perform the actual overlay operations. An overlay run-time routine in the task loads overlays from disk or maps resident overlays by issuing QIO$, CRAW$, or fast-mapping directives.

Note

When building an overlaid task, you should use both the current TKB and the system library supplied for your system to ensure that the correct overlay run-time system (OTS) modules are incorporated into your task. If you use TKB with an incompatible version of the system library to build an overlaid task, you will receive the following error message:

```
TKB -- *FATAL* -- Incompatible OTS module
```

The modules and routines described below implement the TKB autoload mechanism as described in Section 4.1.

There are three major components and one optional component (FSTM) to the autoload service, as follows:

AUTO This module controls the overlay process, and the autoload vectors indirectly call AUTO through .NAUTO. AUTO determines whether the referenced overlay segment is already in memory or mapped. It then jumps to the required entry point if the entry point is available.

The AUTO module is supplied in two variations. These variations are separately named and described as follows:

AUTO — Selected by TKB by default for all overlaid tasks. It manages disk-only, memory management, and cluster library overlay structures.

AUTOT — Manually selected by you by using an explicit reference in the ODL file, as shown below. This module disables the AST traps while manipulating the overlay data structures. This is required where user task AST traps might cause modification of the overlay database. To incorporate this module in your task image, you must include the following element in the .ROOT factor of the task's ODL file:

```
-LB:[1,1]SYSLIB/LB:AUTOT-
```

In addition to including AUTOT in the .ROOT factor, the following code must be included in your task as initialization prior to the AST handling routines in your task:

```
MOV @#.NOVPT,RO
BISB #200,N.FAST(RO)
```

FSTM — This module is only included in tasks built with the Fast OTS (/FO) switch and is intended for tasks that use autoloaded memory-resident overlays. FSTM uses the fast-mapping facility to map the windows for the memory-resident overlays.

MRKS — This routine traverses the overlay descriptor data structure to mark any overlay segment that will be displaced by a new overlay as "out of memory" and consequently not available. MRKS is also used for manually loaded tasks.

RDSG — The AUTO module calls the RDSG routine repeatedly to read or map each segment along the overlay tree path into the task's virtual address space. This is referred to as "path loading." When path loading is completed, AUTO calls the required entry point. RDSG is also used for manually loaded tasks.

Several versions of MRKS and RDSG exist, reflecting the various sizes as appropriate for tasks using manual or autoload service, or having disk-only overlays, memory-resident overlays, or cluster libraries. TKB incorporates the smallest support routines appropriate for the overlay structure of your task.

Depending on whether your task uses manual or autoload service, or has disk-only overlays, resident overlays, or cluster libraries, TKB includes one of the following modules in the root of your task:

OVFCTL OVFIDL OVCTL OVIDL	Contain the MRKS and RDSG routines for autoloaded disk overlays only. No support is included for memory-resident or cluster library overlays. OVFCTL is the module included for conventional autoloaded tasks and OVFIDL is the module included for autoloaded I- and D-space tasks. OVCTL is the module included for conventional manually-loaded tasks, and OVIDL is the module included for manually-loaded I- and D-space tasks.
OVFCTR OVFIDR OVCTR OVIDR	Contain MRKS and RDSG routines for disk and memory-resident overlays. TKB selects either of these modules if the task overlay structure includes memory-resident overlays or maps a resident library. OVFCTR is the module selected for conventional autoloaded tasks. OVFIDR is the module included for autoloaded overlaid I- and D-space tasks. OVCTR is the module included for conventional manually-loaded tasks, and OVIDR is the module included for manually-loaded I- and D-space tasks.
OVFCTC OVFIDC OVCTC OVIDC	Contain the MRKS, RDSG, and cluster library support routines for disk and memory-resident overlays. TKB includes OVCTC or OVIDC if cluster libraries are included in your task. OVFCTC is the module included for conventional autoloaded tasks. OVFIDC is the module selected for autoloaded overlaid I- and D-space tasks. OVCTC is the module included for conventional manually-loaded tasks, and OVIDC is the module included for manually-loaded I- and D-space tasks.

The following modules are also incorporated into your task's image:

OVDAT A small, impure data area used by the AUTO, MRKS, RDSG, and FSTM routines. TKB includes OVDAT in all overlaid tasks, and its size is independent of the overlay structure of that task.

ALERR An error service module that AUTO invokes under one of the following circumstances:

- If an I/O error occurs while attempting to read a disk overlay into memory
- If a directive error occurs while attempting to attach or map a region containing memory-resident overlays

Table 4-1 compares the sizes of the overlay run-time support modules. You can use it to determine when it is appropriate to force certain variants into your task image.

Table 4-1: Comparison of Overlay Run-Time Module Sizes

Module	Program Section	Number of Bytes Octal/Decimal	Specific Use
One of the following modules is included in any overlaid task that uses autoload.			
AUTO	$$AUTO	126/86	All tasks that use autoload
AUTOT	$$AUTO	136/94	All tasks with ASTs disabled during autoload
	$$RTQ	32/26	
	$$RTR	34/28	

Table 4-1 (Cont.): Comparison of Overlay Run-Time Module Sizes

Module	Program Section	Number of Bytes Octal/Decimal	Specific Use
One of the following modules is included in any overlaid conventional task.			
OVFCTL	$$MRKS	102/66	Optimized version of OVCTL for autoloaded disk overlays only
	$$RDSG	154/108	
	$$PDLS	2/2	
OVCTL	$$MRKS	76/62	Disk overlays only, manual load
	$$RDSG	160/112	
	$$PDLS	2/2	
OVFCTR	$$MRKS	150/104	Optimized version of OVCTR for disk and memory management overlays with no cluster libraries, autoload
	$$RDSG	332/218	
	$$PDLS	12/10	
OVCTR	$$MRKS	234/156	Disk and memory management overlays with no cluster libraries, manual load
	$$RDSG	332/218	
	$$PDLS	12/10	
OVFCTC	$$MRKS	170/120	Optimized version of OVCTC for disk and memory management overlays with cluster libraries, autoload
	$$RDSG	352/234	
	$$PDLS	120/80	
OVCTC	$$MRKS	254/172	Disk and memory management overlays with cluster libraries, manual load
	$$RDSG	354/236	
	$$PDLS	120/80	
One of the following modules is included in any overlaid I- and D-space task.			
OVFIDL	$$MRKS	106/70	Optimized version of OVIDL for disk overlays only, autoload
	$$RDSG	224/148	
	$$PDLS	2/2	
OVIDL	$$MRKS	76/62	Disk overlays only, manual load
	$$RDSG	224/148	
	$$PDLS	2/2	
OVFIDR	$$MRKS	226/150	Optimized version of OVIDR for disk and memory management overlays with no cluster libraries, autoload
	$$RDSG	502/322	
	$$PDLS	12/10	
OVIDR	$$MRKS	304/196	Disk and memory management overlays with no cluster libraries, manual load
	$$RDSG	502/322	
	$$PDLS	12/10	
OVFIDC	$$MRKS	246/166	Optimized version of OVIDC for disk and memory management overlays with cluster libraries, autoload
	$$RDSG	522/338	
	$$PDLS	120/80	

Table 4–1 (Cont.): Comparison of Overlay Run-Time Module Sizes

Module	Program Section	Number of Bytes Octal/Decimal	Specific Use
OVIDC	$$MRKS	324/212	Disk and memory management overlays with cluster libraries, manual load
	$$RDSG	522/338	
	$$PDLS	120/80	
The overlay data vector OVDAT is included in any overlaid task and in any task that links to a memory management overlaid resident library.			
OVDAT	$$OVDT	26/22	Included in all tasks that perform overlay operations
	$$SGD0	0/0	
	$$SGD2	2/2	
	$$RTQ	0/0	
	$$RTR	0/0	
	$$RTS	2/2	
The FSTMAP module is included only when the /FO switch (/CODE:OTS_FAST qualifier) is used.			
FSTMAP	$$FSTM	212/138	OTS FSTMAP routine
	$DPBS	6/4	OTS FSTMAP routine Directive Parameter Block (DPB)
	$RTS	2/2	
The overlay error-service routine ALERR is included whenever OVDAT is included.			
ALERR	$$ALER	40/32	Overlay error
Manual overlay control (LOAD) is used in place of any AUTO routine. (See Section 4.2, Manual Load.)			
LOAD	$$LOAD	252/170	Manual overlay control
	$$AUTO	14/12	

4.5.1 The OTS Fast Map Routine (FSTM)

The overlay run-time system (OTS) Fast Map routine enables you to use fast mapping for autoloaded memory-resident overlays, thereby increasing the speed of overlay mapping by approximately 10 times. The routine saves time because it uses fast mapping whenever possible rather than using the Executive mapping directives Create Address Window (CRAW$) and Eliminate Address Window (ELAW$) after the initial loading and mapping of an overlay.

The overlay run-time system routine MARKS, located in program section $$MRKS, controls the unloading of overlays as the task executes. The optimized versions of MARKS used for autoloaded memory-resident overlays can call the OTS Fast Map routine, which uses the fast-mapping facility to map windows for autoloaded memory-resident overlays.

When you specify the Fast OTS (/FO) switch, TKB resolves the entry point $FSTIN to the OTS Fast Map routine in program section $$FSTM. If you do not specify /FO, $FSTIN is resolved to a dummy return instruction and the Fast Map routine is not accessed.

The OTS Fast Map routine determines whether fast mapping was included in the operating system during system generation. If fast mapping is available on the current system, it will be used. If fast mapping is not available, the standard memory management directives CRAW$ and ELAW$ are used for mapping. This allows a task built with /FO to run on any RSX-11M-PLUS or Micro/RSX system, even if fast mapping has not been included during system generation for that system.

When the /FO switch is set, TKB requires the task to have an extended external header and include space between the header and the task for the fast-mapping extension area. This is accomplished by building the task with the Fast Mapping (/FM) and External Header (/XH) switches (the /XH switch is the default).

The following restrictions apply to the OTS Fast Map routine:

- The OTS Fast Map routine uses the fast-mapping facility, which means that the task must not use the IOT instruction for any purpose except fast mapping. (For more information on the fast-mapping facility, see the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual.*)
- The OTS Fast Map routine uses fast mapping with the memory-resident overlays only when the memory-resident overlay being unloaded and the memory-resident overlay being loaded start at the same Active Page Register (APR), use the same number of APRs, and map to the same region. If these conditions are not met, the standard CRAW$ directive is used to do the mapping and unmapping.

Chapter 5

Shared Region Concepts and Examples

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

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

5.1 Shared Regions Defined

A shared region is a block of data or code that resides in memory and can be used by any number of tasks. A shared region can contain data for use by several tasks or it may be an area where one task writes data for use by another task. Also, a shared region can contain routines for use by several tasks.

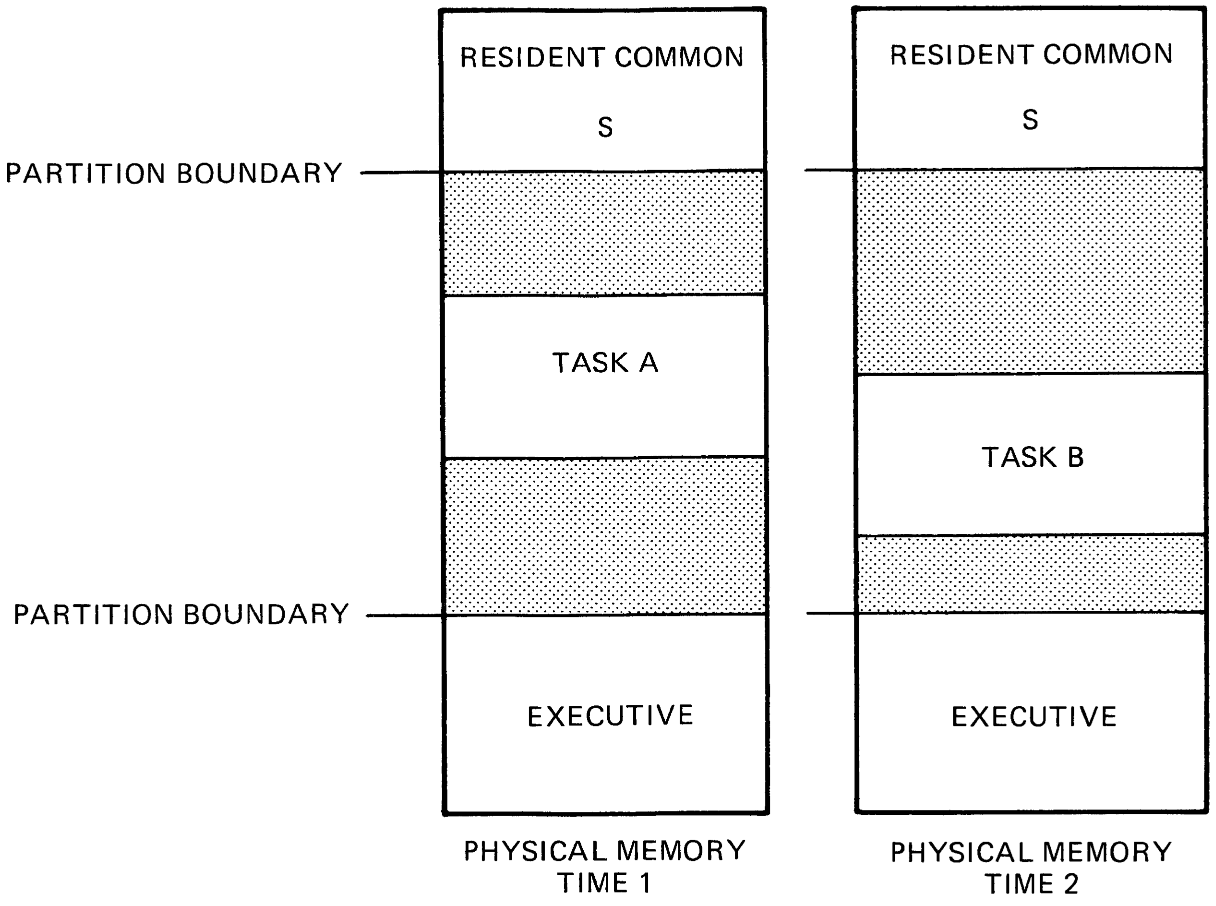
Shared regions are useful because they make more efficient use of physical memory. The two kinds of shared regions are:

- A resident common that provides a way for two or more tasks to share their data
- A resident library that provides a way for two or more tasks to share a single copy of commonly used subroutines

The term "resident" denotes a shared region that is built and installed into the system separately from the task that links to it. In other words, you use TKB to build a shared region much as you would use it to build a task. However, the region does not have a header or a stack. Also, you can use switches to designate the kind of shared region (a library or a common) to be built.

Figure 5-1 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.

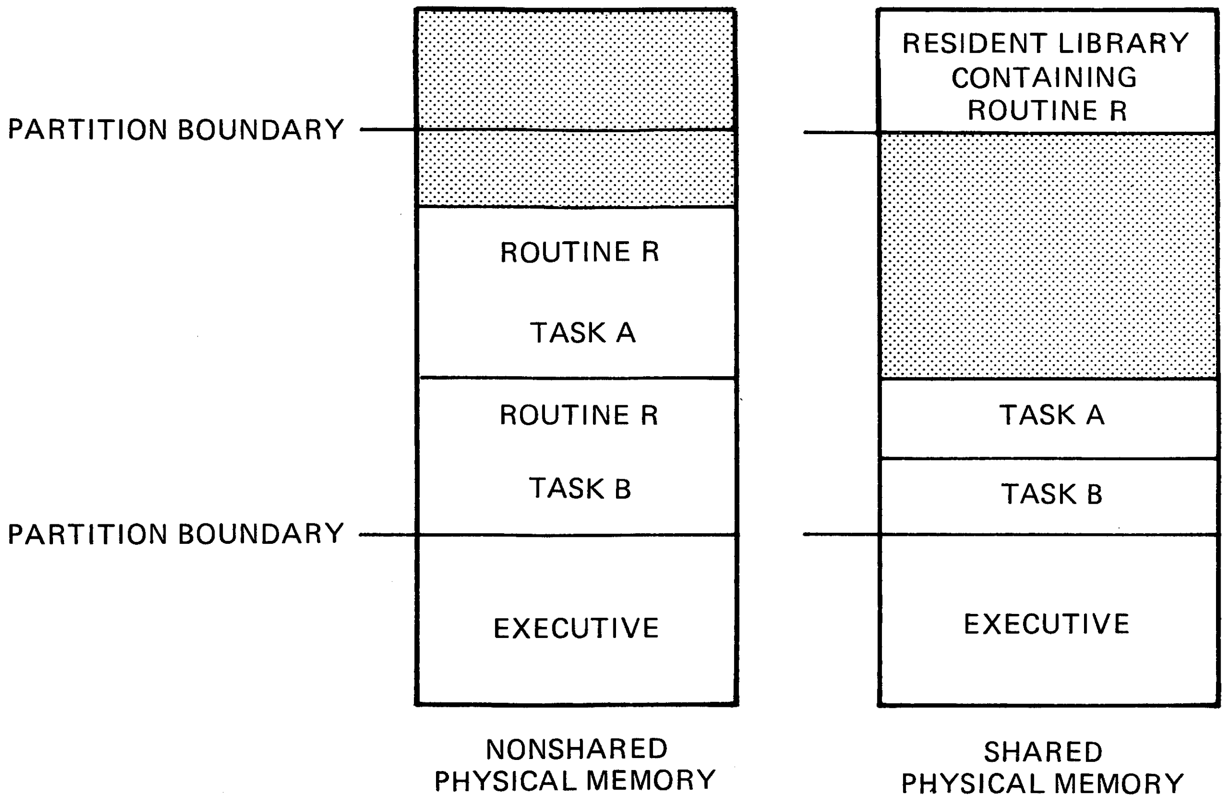
Figure 5–1: Typical Resident Common



ZK-418-81

Figure 5-2 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 5–2: Typical Resident Library



ZK-419-81

When you build a shared region, you must specify an output image file name for the region in the TKB command sequence. But, because a shared region is not an executable unit, it is not a task, and does not require a header or a stack area. Therefore, when you build a shared region, you always attach the negated header (/-HD) switch, or /NOHEADER as a LINK qualifier, to the region's image file specification. This switch or qualifier tells TKB to suppress the header within the image. To suppress the stack area in the Task Builder or LINK command sequence during option input, you specify STACK=0. (Refer to Chapters 10, 11, and 12 for a complete description of the /HD switch, the /NOHEADER qualifier, the STACK option, and other switches, qualifiers, and options.)

When you build a shared region, you use the PAR option to name the partition in which the region is to reside. You specify the partition name in the TKB command sequence during option input. (Refer to Chapter 12 for a description of the PAR option.) The partition named in the PAR option need not previously exist, but the actual partition defaults to GEN. The name used in the PAR option must be the same name as that of the region.

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. It follows then that a TKB command sequence or build file for a memory-resident overlaid library must contain the statement PAR=xxx, where xxx is the same name as that of the region being built. Then, when you attempt to install the shared region in a partition that does not exist, the INSTALL task installs it in the GEN partition and displays the following message on your terminal:

```
INS -- Partition parname not in system, defaulting to GEN
```

Also, you should consider three switches when you build the region. The /PI switch in TKB or the /CODE:PIC qualifier in LINK determines whether the region is relocatable. You can use the /CO switch in the TKB command sequence, or the /SHAREABLE:COMMON qualifier in LINK, to declare a region as a shared common. The /CO switch or /SHAREABLE:COMMON qualifier specifies the use of the region as a shared common rather than as a shared library. Alternatively, you can use the /LI switch in TKB, or the /SHAREABLE:LIBRARY qualifier in LINK, when you build the region to declare the region as a shared library. Using these three switches affects the contents of the symbol definition file, which is described in Chapter 10 under the /CO, /LI, and /PI switches or in Chapter 11 under the /SHAREABLE:COMMON, /SHAREABLE:LIBRARY, and /CODE:PIC qualifiers. See also Figure 5-3, Interaction of the /LI, /CO, and /PI Switches, and Figure 5-4, Interaction of the /SHAREABLE:LIBRARY, /SHAREABLE:COMMON, and /CODE:PIC Qualifiers. The contents of the symbol definition file are described in the following sections.

Figure 5-3: Interaction of the /LI, /CO, and /PI Switches

SWITCH SPECIFIED WITH /-HD	SHARED REGION		REGION PSECT NAME	.STB FILE PSECT	.STB FILE SYMBOLS
	ABSOLUTE	RELOCATABLE			
/PI/LI		YES	SAME AS LIBRARY ROOT	ONE PSECT RELOCATABLE	ALL SYMBOLS. RELATIVE TO START OF THE PSECT
/PI/CO		YES	ALL DECLARED PSECT NAMES INCLUDED	ALL DECLARED PSECTS RELOCATABLE	ALL PSECTS AND SYMBOLS
/-PI/LI*	YES		SAME AS LIBRARY ROOT	ONE PSECT ABSOLUTE	ALL SYMBOLS ABSOLUTE
/-PI/CO*	YES		ALL DECLARED PSECT NAMES INCLUDED	ALL DECLARED PSECTS ABSOLUTE	ALL SYMBOLS ABSOLUTE
/PI		YES	SAME AS /PI/CO		
/-PI*	YES		SAME AS /-PI/LI		
NONE	YES		SAME AS /-PI/LI		

*/-PI is the default of not using /PI

ZK-420-81

Figure 5–4: Interaction of the /SHAREABLE:LIBRARY, /SHAREABLE:COMMON, and /CODE:PIC Qualifiers

QUALIFIER SPECIFIED WITH /NOHEADER	SHARED REGION		REGION PSECT NAME	.STB FILE PSECT	.STB FILE SYMBOLS
	ABSOLUTE	RELOCATABLE			
/CODE:PIC/SHAREABLE:LIBRARY		YES	SAME AS LIBRARY ROOT	ONE PSECT RELOCATABLE	ALL SYMBOLS. RELATIVE TO START OF THE PSECT
/CODE:PIC/SHAREABLE:COMMON		YES	ALL DECLARED PSECT NAMES INCLUDED	ALL DECLARED PSECTS RELOCATABLE	ALL PSECTS AND SYMBOLS
/SHAREABLE:LIBRARY	YES		SAME AS LIBRARY ROOT	ONE PSECT ABSOLUTE	ALL SYMBOLS ABSOLUTE
/SHAREABLE:COMMON	YES		ALL DECLARED PSECT NAMES INCLUDED	ALL DECLARED PSECTS ABSOLUTE	ALL SYMBOLS ABSOLUTE
/CODE:PIC		YES	SAME AS /CODE:PIC/SHAREABLE:COMMON		
NONE	YES		SAME AS /SHAREABLE:LIBRARY		

ZK-1370-83

5.1.1 The Symbol Definition File

When you build a shared region, you must specify a symbol definition (STB) file in the TKB command sequence. This file contains linkage information about the region. (The format of an STB file as input to TKB is the same as that of an OBJ file. See Appendix A.) Later, when you build a task that links to the region, TKB uses this STB file to resolve calls from within the referencing task to locations within the region.

The STB file contains two forms of symbol definition. To maintain backward compatibility, all autoloadable symbols are entries in the global symbol directory, and the vector itself is defined in associated text records. Additionally, the STB file contains an internal symbol directory of autoloadable symbols for conventional tasks, as well as the information needed to generate autoload vectors for I- and D-space tasks.

The following equivalencies exist among the shared region switches in TKB and qualifiers in LINK:

TKB Switches	LINK Qualifiers
/LI	/SHAREABLE:LIBRARY
/CO	/SHAREABLE:COMMON
/PI	/CODE:PIC
/PI	/CODE:POSITION_INDEPENDENT
/-LI	(none)
/-CO	(none)
/-PI	Absence of both /CODE:PIC and /CODE:POSITION_INDEPENDENT

If you use TKB with MCR, the /PI switch declares a shared region to be relocatable. Conversely, the /-PI switch declares a shared region to be absolute. If you specify the /PI switch without the /CO or /LI switches to indicate a relocatable region, TKB defaults to building a relocatable (position-independent) shared region (a common) with all program sections declared in the STB file. The contents of the STB file when these three switches are used are described in Chapter 10 under the /CO, /LI, and /PI switches. See also Figure 5-3, Interaction of the /LI, /CO, and /PI Switches.

If you use the LINK command, the /CODE:PIC qualifier declares a shared region to be relocatable. Conversely, the absence of the /CODE:PIC qualifier declares the shared region to be absolute. If you specify the /CODE:PIC qualifier without the /SHAREABLE:COMMON or /SHAREABLE:LIBRARY qualifiers to indicate a relocatable region, TKB defaults to building a relocatable (position-independent) shared region (a common) with all program sections declared in the STB file. The contents of the STB file when these three qualifiers are used are described in Chapter 11 under the /CODE:PIC, /SHAREABLE:COMMON, and /SHAREABLE:LIBRARY qualifiers. See also Figure 5-4, Interaction of the /SHAREABLE:LIBRARY, /SHAREABLE:COMMON, and /CODE:PIC Qualifiers.

If you do not use either /CO or /LI, or for LINK either /SHAREABLE:COMMON or /SHAREABLE:LIBRARY, the contents of an STB file for a shared region depend on the use of the /PI switch or the /CODE:PIC qualifier, which determines whether the region is absolute or relocatable. The effects of declaring a shared region relocatable or absolute and the resulting contents of the STB file are described in the following sections.

Some STB files include an entry in the STB file for each program section in the region. Each entry declares the program section's name, attributes, and length. In addition, TKB includes in the STB file every symbol in the shared region and its value relative to the beginning of the section in which it resides.

5.1.2 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.

5.1.2.1 Position-Independent Shared Region Mapping

In the example of using the memory management Active Page Registers (APRs), shown in Figure 5-5, 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. Thus, you can specify APR 2 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, you can specify APR 5 when task B is built.

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

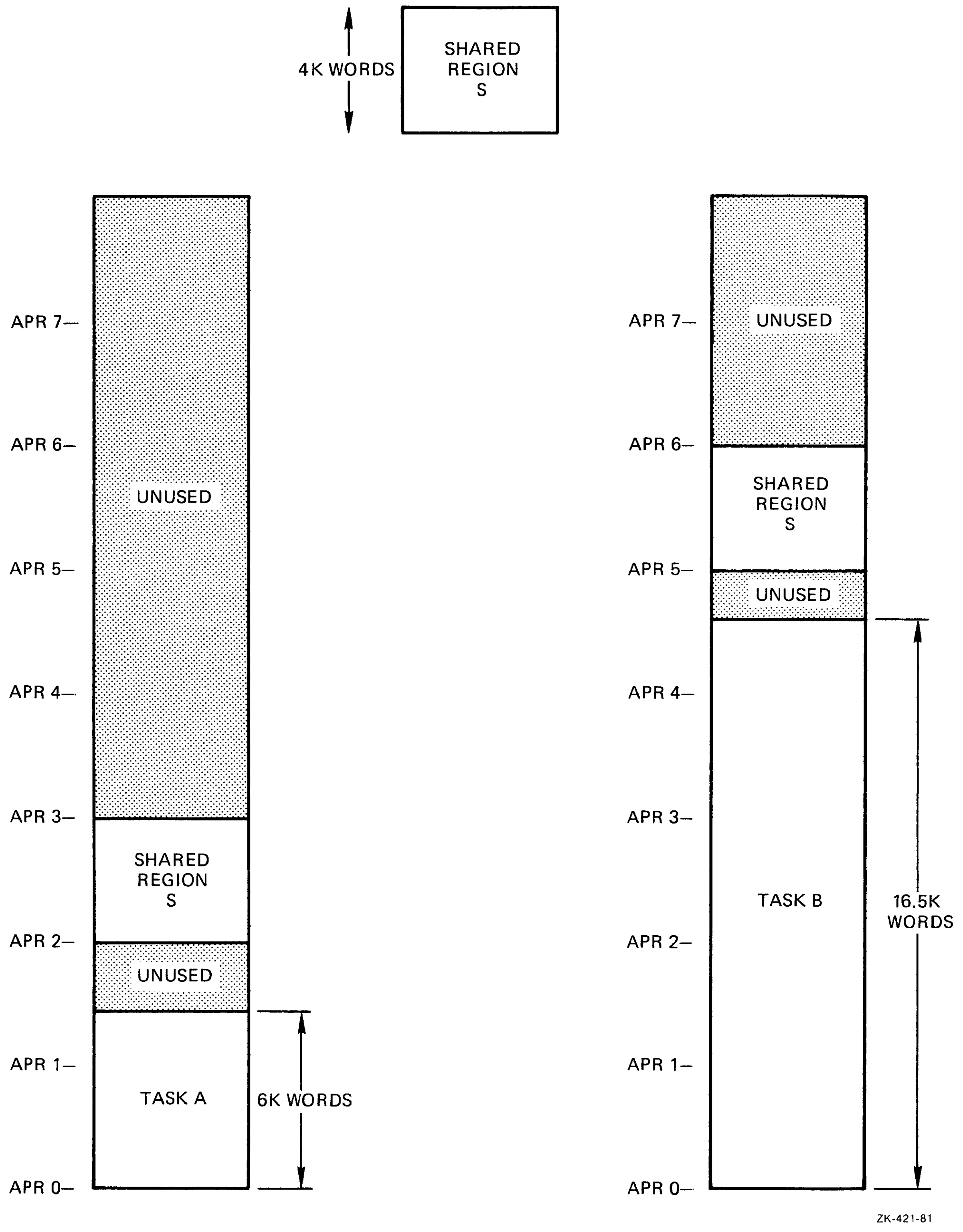
5.1.2.2 Specifying a Position-Independent Region

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. If you use the LINK command, you specify a position-independent region by using the /CODE:PIC qualifier attached to the LINK command or to the input file specification. (Refer to Chapter 10 for a description of the /PI switch or to Chapter 11 for a description of the /CODE:PIC qualifier.)

You should declare a region position independent if any one of the following conditions exists:

- 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.

Figure 5–5: Specifying APRs for a Position-Independent Shared Region



Program section names are preserved in some shared regions. All the following switch combinations produce shared regions in which program section names are preserved:

TKB	LINK
/PI/CO	/CODE:PIC/SHAREABLE:COMMON
/-PI/CO	/SHAREABLE:COMMON
/PI	/CODE:PIC

Therefore, you should observe the following precautions when building and referring to these regions:

- 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 and /SG switches in Chapter 10 or to the /SEQUENTIAL and /[NO]SEGREGATE qualifiers in Chapter 11 for an explanation of sequential ordering of program sections.)

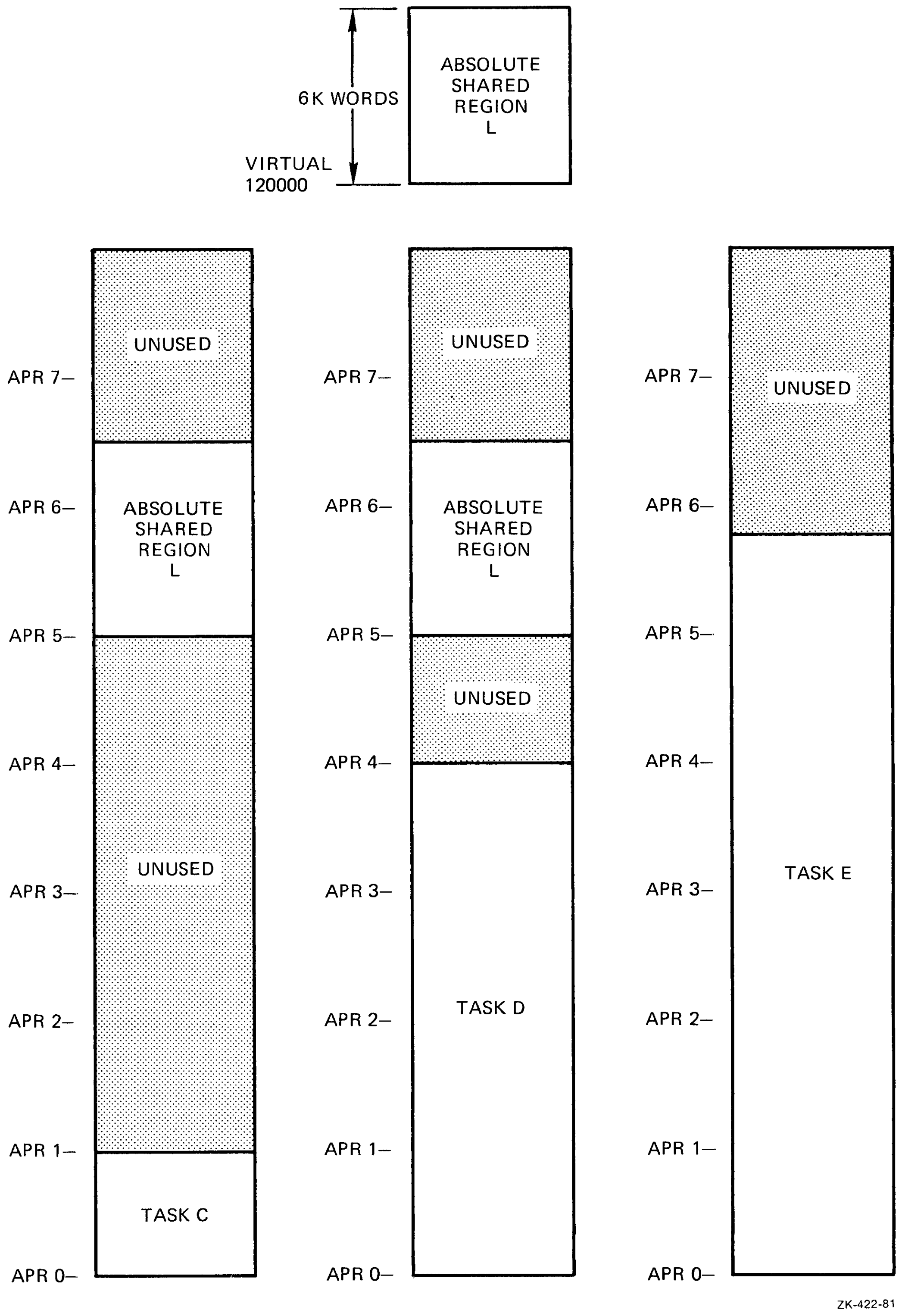
5.1.3 Absolute Shared Regions

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.

5.1.3.1 Absolute Shared Region Mapping

Figure 5-6 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.

Figure 5–6: Mapping for an Absolute Shared Region



5.1.3.2 Specifying an Absolute Shared Region

You specify that a shared region is absolute when you build it by using the /-PI switch or omitting the /PI switch or /CODE:PIC qualifier 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 an absolute shared region if any one of the following conditions exists:

- 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.

5.1.3.3 Absolute Shared Region STB File

For TKB commands, when a shared region is created with the /-PI/LI or /-PI switches, or with just the /-HD switch, the only program section name that appears in the STB file for the region is the absolute program section name (. ABS.). Similarly for LINK commands, the . ABS program section name is the only one that appears when you create the shared region with the /SHAREABLE:LIBRARY qualifier and the /NOHEADER qualifier or with only the /NOHEADER qualifier. TKB includes in the STB file for the region each symbol in the region and its value. But, because TKB 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 names. Also, because the program section names are not in the STB file, TKB places no restrictions on the way the program sections are ordered in either the absolute shared region or the tasks that reference it. You can order program sections the way you want by using the TKB /SQ and /SG switches or the LINK /SEQUENTIAL and /[NO]SEGREGATE qualifiers (described in Chapters 10 and 11).

5.1.4 Shared Regions with Memory-Resident Overlays

Shared regions with memory-resident overlays are a primary resource for conserving memory. If the shared region is larger than the available virtual address space in a task that must reference the region, you can build the region—both position-independent and absolute—with memory-resident overlays. All segments of the overlay structure are included in the shared region, but each task referencing the shared region can include only part of the shared region—that is, an overlay segment or series of segments in an overlay path—in its virtual address space. Therefore, the task need only have enough virtual address space for the largest shared region overlay segment, or series of segments in an overlay path, it is likely to access. Hence, the virtual address space of the task can be considerably smaller than the size of the shared region.

5.1.4.1 Considerations About Building an Overlaid Shared Region

In general, overlays can be disk-resident or memory-resident, but those in shared regions must, by their very nature, be memory-resident. TKB marks each overlay segment in the shared region with the NODSK attribute to suppress overlay load requests. When you build a shared region with memory-resident overlays, you must define the overlay structure through a conventional ODL file. (See Chapters 3 and 4 of this manual for information on overlays and the Overlay Description Language.) TKB does not include the overlay database (segment descriptors, autoload vectors, and so forth) or the overlay run-time routines within the region image. Instead, this database becomes a part of the STB file that is linked to the referencing task. When this task is built, its root segment automatically includes both the database and global references to overlay support routines residing in the system object module library.

The procedure for creating a shared region with memory-resident overlays can be summarized as follows:

- Define an overlay structure containing only memory-resident overlays.
- Include the GLBREF option, or provide in the root segment a module containing the appropriate global references for defining entry points within those overlay segments. TKB generates autoload vectors and global definitions for the overlay segments.

5.1.4.2 Example of Building a Memory-Resident Overlaid Shared Region

The procedure for creating a shared region is illustrated in the following example. The shared region to be constructed consists of reentrant code that resides within the overlay structure defined below:

```
.ROOT A-!(*B,C-*D)
.NAME A
.END
```

Root segment A contains no code or data and has a length of zero. All executable code exists within memory-resident overlay segments composed of the object modules B.OBJ, C.OBJ, and D.OBJ, which contain the global entry points B, C, and D, respectively.

You generate the TSK, MAP, and STB files by using the following TKB command sequence:

```
TKB>A/-HD/MM,LP:,SY:A=A/MP
Enter Options:
TKB>GBLREF=B,C,D
TKB>PAR=A:160000:20000
TKB>STACK=0
TKB>/
>
```

Or, use the following LINK command sequence:

```
$ LINK/TAS:A/NOH/MEM/MAP:LP:/SYM:SY:A/OPT A/OVER
Option? GBLREF=B,C,D
Option? PAR=A:160000:20000
Option? STACK=0
Option? RET
$
```

Note

When building a shared region, you must use the same name for the partition and the TSK and STB files.

See the descriptions of the PAR, RESLIB, LIBR, RESCOM, and COMMON options in Chapter 11.

TKB inserts references to entry points B, C, and D in the root segment of the library. The references subsequently appear in the STB file as definitions.

TKB resolves the definitions for symbol C directly to the actual entry point. TKB resolves the definitions for symbols B and D to autoload vectors that it includes in each referencing task.

5.1.4.3 Options for Use in Overlaid Shared Regions

Certain options may prove useful to you when building and linking shared regions to a task. These options are as follows:

- GBLDEF

  You can declare the definition of a symbol by means of the GBLDEF option. The option has the following syntax:

  ```
  GBLDEF=symbol-name:symbol-value
  ```

  symbol-name is a 1- to 6-character Radix–50 name of the defined symbol and symbol-value is an octal number in the range of 0 to 177777 assigned to the symbol. This option is frequently used in the TKB build file for a task or shared region to allow you to alter the value of a global symbol that resides in a module. This saves you the trouble of reassembling the source code for a module if changes are necessary.

- GBLINC

  By means of this option, you force TKB to include the specified symbols in the STB file being created by the linking process in which this option appears. The option has the following syntax:

  ```
  GBLINC=symbol-name,symbol-name,...,symbol-name
  ```

  symbol-name is the symbol or symbols to be included. Use this option when you want to force particular modules to be linked to the task that references this library. The global symbol references specified by this option must be satisfied by some module or GBLDEF specification when you build the task.

- GBLREF

  You can force the inclusion of a global reference in the root segment of the shared region by means of the GBLREF option. In this way, the necessary autoload vectors and definitions can be generated without explicitly including such references in an object module. The option has the following syntax:

  ```
  GBLREF=[,name[,name...]]
  ```

  The name consists of from one to six Radix–50 characters. If the definition resides within an autoloadable segment, TKB constructs an autoload vector and includes it in the symbol definition file. If the definition is not autoloadable, TKB obtains the real value and defines it in the root segment. No global symbol appears in the STB file unless the symbol is either defined in the root segment or is referenced in the root segment and defined elsewhere in the overlay structure.

- GBLXCL

  You can exclude a symbol or symbols from the symbol definition file of a shared region by means of the GBLXCL option. The option has the following syntax:

  ```
  GBLXCL=symbol-name,symbol-name,...,symbol-name
  ```

  symbol-name is the symbol or symbols to be excluded. You can use this option when you do not want the task to be aware of specific symbols within the library. This option is particularly useful when you cluster overlaid libraries together. See the CLSTR option in Chapter 12 and the Cluster Libraries section (Section 5.2) in this chapter.

5.1.4.4 Autoload Vectors and STB Files for Overlaid Shared Regions

When TKB builds a task image file containing memory-resident overlays, TKB allocates autoload vectors in the task image. If the task links to a shared region, autoload vectors for the shared region are also allocated in the task image. TKB allocates the autoload vectors in the task's root segment, but not in the shared region. Therefore, the shared region cannot reference unloaded (unmapped) segments of its overlay structure.

When the task executes, the shared region is effectively part of the task. In fact, when the task loads overlay segments, it makes no distinction between overlay segments of the task and those of the shared region. They are loaded as needed in a procedure that is transparent insofar as the execution of the task is concerned.

For the Fast Task Builder (FTB) and older versions of TKB that do not support overlaid I- and D-space tasks, each autoload vector in the shared region's STB file is allocated in the root of the task being linked to the region, whether or not the entry point is referenced by the task.

However, if you use a version of TKB that supports overlaid I- and D-space tasks and the library was built with one of these versions, TKB allocates autoload vectors in the root of the task only for those autoloadable entry points in the library that the task references. The STB file contains ISD records that allow TKB to dynamically create autoload vectors when linking the task to the library. TKB ignores the autoload vectors in the STB file if the ISD records are present. Therefore, tasks that link to overlaid shared regions and are built with newer versions of TKB tend to be smaller and use less virtual address space than those that are built by FTB or older versions of TKB.

Note

Libraries created with older versions of TKB do not have the ISD records in the STB file that newer versions of TKB use to include autoload vectors in the task from the STB file. Therefore, TKB must create autoload vectors for every entry point in the library.

If you are using one of these older libraries and you are linking an I- and D-space task to it, TKB will give you the following fatal error message:

```
Module module-name contains incompatible autoload vectors.
```

This message occurs because the STB file contains conventional autoload vectors that are not usable by an I- and D-space task.

Only those global symbols defined or referenced in the root segment of the shared region appear in the STB file. The STB file also contains the database required by the overlay run-time system in relocatable object module format. This database includes the following information:

- All autoload vectors
- Segment tables (linked as described in Appendix B)
- Window descriptors
- A single region descriptor

The overlay structure, as reflected in the segment table linkage, is preserved and conveyed to the referencing task by the STB file. Thus, path loading for the shared region can occur exactly as it does within a task. Aside from address space restrictions, there are no limitations on the overlay structures that can be defined for a shared region.

5.1.5 Run-Time Support for Overlaid Shared Regions

Memory-resident overlays within a shared region require little additional support from the overlay run-time system. The shared region overlay database that is linked within the image of the referencing task has a structure that is identical to the equivalent data created for an overlaid task. Therefore, memory-resident overlays within the shared region are indistinguishable from memory-resident overlays that form a part of the task image. The only additional processing is that required to attach the shared region and obtain its identification for use by the mapping directives.

Once this initialization is complete, all further processing is identical to memory-resident overlay processing performed on task overlays.

For shared regions existing as memory-resident overlays, the following restrictions apply:

- A shared region cannot use the Autoload facility to reference memory-resident overlays within itself or any other region. If each segment is uniquely named, overlays can be mapped through the manual-load facility.
- Named program sections in a shared region overlay segment cannot be referenced by the task. If reference to the storage is required, such sections must be included in the root segment of the region (with resultant loss of virtual address space).
- For FTB, and libraries built with versions of TKB that do not support I- and D-space overlaid tasks, the number of autoload vectors is independent of the entry points actually referenced. The maximum number of vectors will be allocated within each referencing task. In some cases the size of the allocation will be large.
- There is an overhead of six instructions for each autoload call, even when the segment is mapped. The overhead is seven instructions for an overlaid I- and D-space task.

As implied by the previous items, great care must be exercised if an efficient memory-resident overlay structure for library routines such as the FORTRAN IV OTS is to be implemented.

5.1.6 Linking to a Shared Region

When you build a task that links to a shared region, you must indicate to TKB 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 TKB is to allocate for mapping the region into the task's virtual address space. The following options are available for this action:

- RESLIB (resident library)
- RESCOM (resident common)
- LIBR (system-owned resident library)
- COMMON (system-owned resident common)

RESLIB and RESCOM accept a complete file specification as one of their arguments. Thus, you can specify a device and directory indicating to TKB 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 in directory [1,1] on device LB0.

The RESLIB and RESCOM options require that all users of the shared region know the directory in 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, TKB expects to find the shared region's image and STB files on device LB: in directory [1,1].

When you are building a resident library, you can use the RNDSEG option to cause the size of a specified segment to be rounded up to the nearest APR boundary.

When you install a resident library, INSTALL makes an entry for the resident library in the Common Block Directory (CBD). The system loads the resident library when a task that uses it runs.

Using RNDSEG enables you to install a new library without relinking the task to it as long as the new library has the same common block length recorded in the CBD when the previous library was installed. See Chapter 12 for more information on the RNDSEG option.

All four options accept the following additional arguments:

- The type of access that the task requires (RO or RW).
- The first APR that TKB 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, TKB expects to find a symbol definition file of the same name as that of the shared region, but with a file type of STB, on the same device and in the same directory as those of the shared region's image file.

The syntax of these options is given in Chapter 11.

When TKB builds a task, it processes first any options that appear in the TKB command sequence. When TKB 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, and the name of the partition for which it was built). TKB 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.

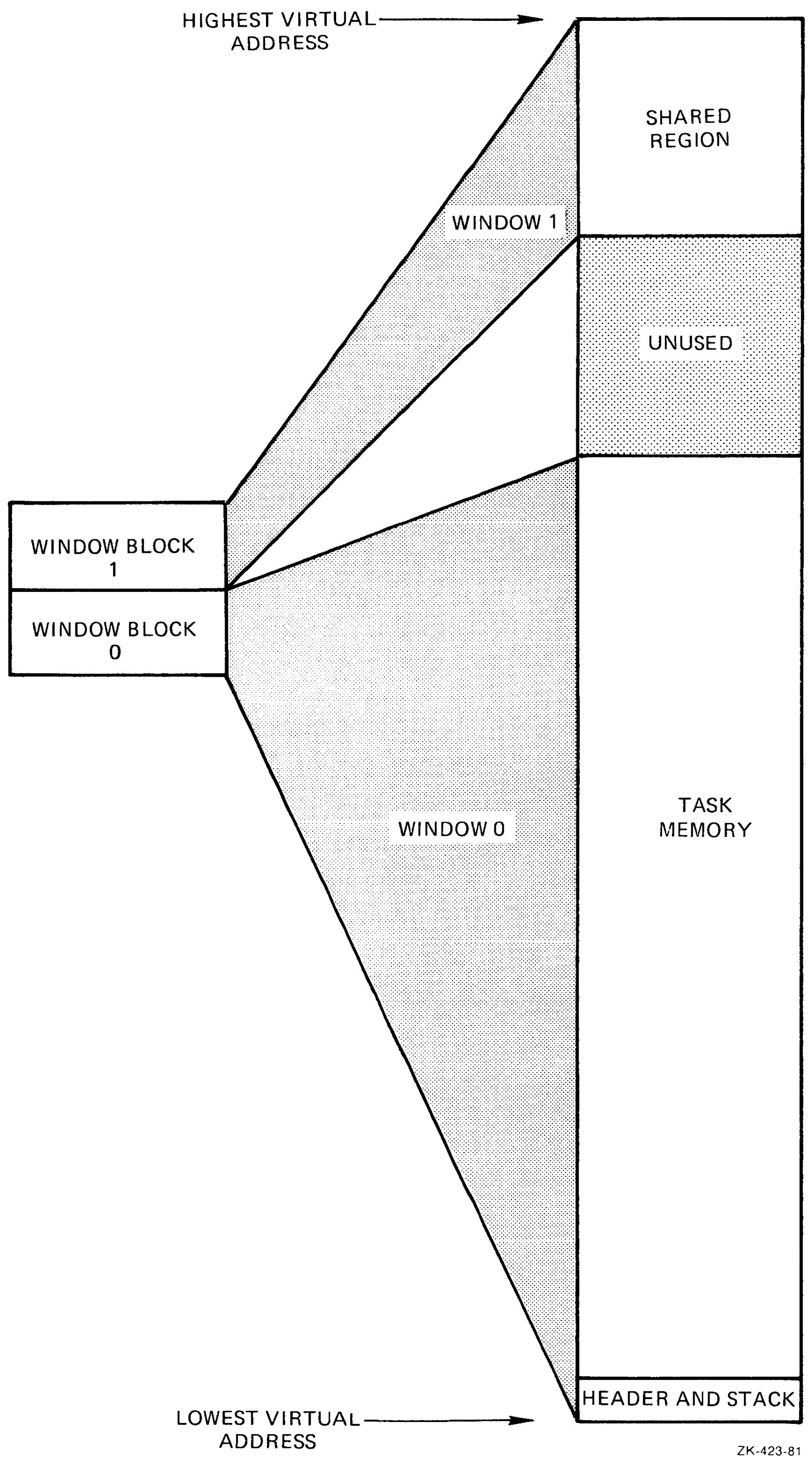
The STB file associated with the shared region is an object module file. TKB processes it as an input file. If the shared region is position independent, its STB file contains program section names, attributes, and lengths. However, the program section names are flagged within the file as "library" program sections and TKB 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 allocates two window blocks in the header of the task. (Overlays are described in Chapter 3.) When the task is installed, the INSTALL task 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 5-7 shows the window-to-region relationship of such a task.

Figure 5–7: 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 TKB 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.

Unless you use the /LI switch or the /SHAREABLE:LIBRARY qualifier, there is a restriction on the way TKB processes tasks that link to relocatable shared regions. TKB places all program section names into its internal control section table. The program section names include those from the STB file of the shared region as well as those from the other input modules. A conflict can arise when building a task that contains program sections of the same name as those in the shared region to which the task links. The conflict arises because TKB tries to add the program section allocation in the task to the already existing allocation for the program section of the same name in the 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, to avoid this conflict, the program section names within a task that links to a relocatable shared region must normally be unique with respect to program section names within the shared region.

TKB displays an error message under the following conditions:

- A program section in the task and a program section in the shared region have the same name.
- The program section in the task contains data.
- TKB tries to initialize the program section in the task.

The error message occurs when TKB tries to store data in an image outside the address limits of the task it is building. If this conflict occurs, TKB prints the following message:

```
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 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. TKB 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, TKB assigns 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 5-1 and 5-2 in the following sections illustrate how to establish these offsets.

5.1.7 Number and Size of Shared Regions

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.

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: 7 in user mode and 8 in supervisor mode. (Supervisor-mode libraries are described in Chapter 8.)

5.1.8 Example 5-1: 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 provides 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, or the /NOHEADER qualifier for LINK, and declare the common with /CO, or /SHAREABLE:COMMON for LINK. You specify the common to be position independent with the /PI switch, or the /CODE:PIC qualifier for LINK.

   Then, you specify the following options:

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

   The parname in this PAR option is the name of the partition in which the common is to reside.

   The TKB switches are described in Chapter 10. The LINK qualifiers are described in Chapter 11. The STACK and PAR options are described along with the other options in Chapter 12.

   The common can reside within any partition large enough to hold it.
4. You install the common.

Example 5-1, Part 1 shows a MACRO–11 source file that, when assembled and built, creates 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 in length.

Example 5-1: Part 1, Common Area Source File in MACRO-11

```
                .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
```

Once this common has been assembled, the following TKB command sequence shown below can be used to build it:

```
>TKB [RET]
TKB>MACCOM/PI/-HD/CO,MACCOM/-SP,MACCOM=MACCOM [RET]
TKB>/  [RET]
Enter Options:
TKB>STACK=0 [RET]
TKB>PAR=MACCOM:0:4000 [RET]
TKB>// [RET]
>
```

Or, with the LINK command, you may enter the following command sequence:

```
$ LINK/TAS:MACCOM/NOH/CODE:PIC/SHARE:COMMON/MAP:MACCOM/NOPRINT/SYM/OPT
->MACCOM [RET]
Option? STACK=0 [RET]
Option? PAR=MACCOM:0:4000 [RET]
Option? [RET]
$
```

This command sequence directs TKB to build a position-independent, headerless 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. TKB creates all three files—MACCOM.TSK, MACCOM.MAP, and MACCOM.STB—on device SY in the directory that corresponds to the terminal UIC. TKB 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 that of the common, MACCOM. In addition, the parameters in the PAR option specify a base of 0 and a length of 4000_8 bytes for the common. (Refer to Chapters 10, 11, and 12 for descriptions of the switches, qualifiers, and options used in this example.)

Example 5-1, Part 2 shows the map resulting from this command sequence.

Example 5-1: Part 2, Task Builder Map for MACCOM.TSK

```
MACCOM.TSK;1     Memory allocation map  TKB M43.00       Page 1
                      17-NOV-87   16:05

Partition name : MACCOM
Identification :
Task  UIC      : [7,62]

Task attributes: -HD,PI
Total address windows: 1.
Task  image  size  : 1024. WORDS

Task address limits: 000000 003777
R-W disk blk limits: 000002 000005 000004 00004.
                         ❶                ❸
                               ❷


*** Root segment: MACCOM


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. .MAIN.        MACCOM.OBJ;1

COM2  :(RW,D,GBL,REL,OVR) 002000 002000 01024.
                          002000 002000 01024. .MAIN.        MACCOM.OBJ;1
                          ❻             ❺
                                 ❹


*** Task builder statistics:

    Total work file references: 183.
    Work  file  reads: 0.
    Work  file writes: 0.

    Size of core pool: 7086. WORDS (27. PAGES)
    Size of work file: 768. WORDS (3. PAGES)

    Elapsed time:00:00:05
```

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 [7,62], 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 2 (❶). The last block in the file is file-relative disk block 5 (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.

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;1. 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 TKB that contained a program section named COM1, TKB 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 created automatically by the language translators. The attributes shown are the default attributes. The allocation for . BLK. is 0 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.

Figure 5-8 is a diagram that represents the disk image file for MACCOM. The circled numbers in Figure 5-8 correspond to the circled numbers in Example 5-1, Part 2.

Once you have built MACCOM, you can install it.

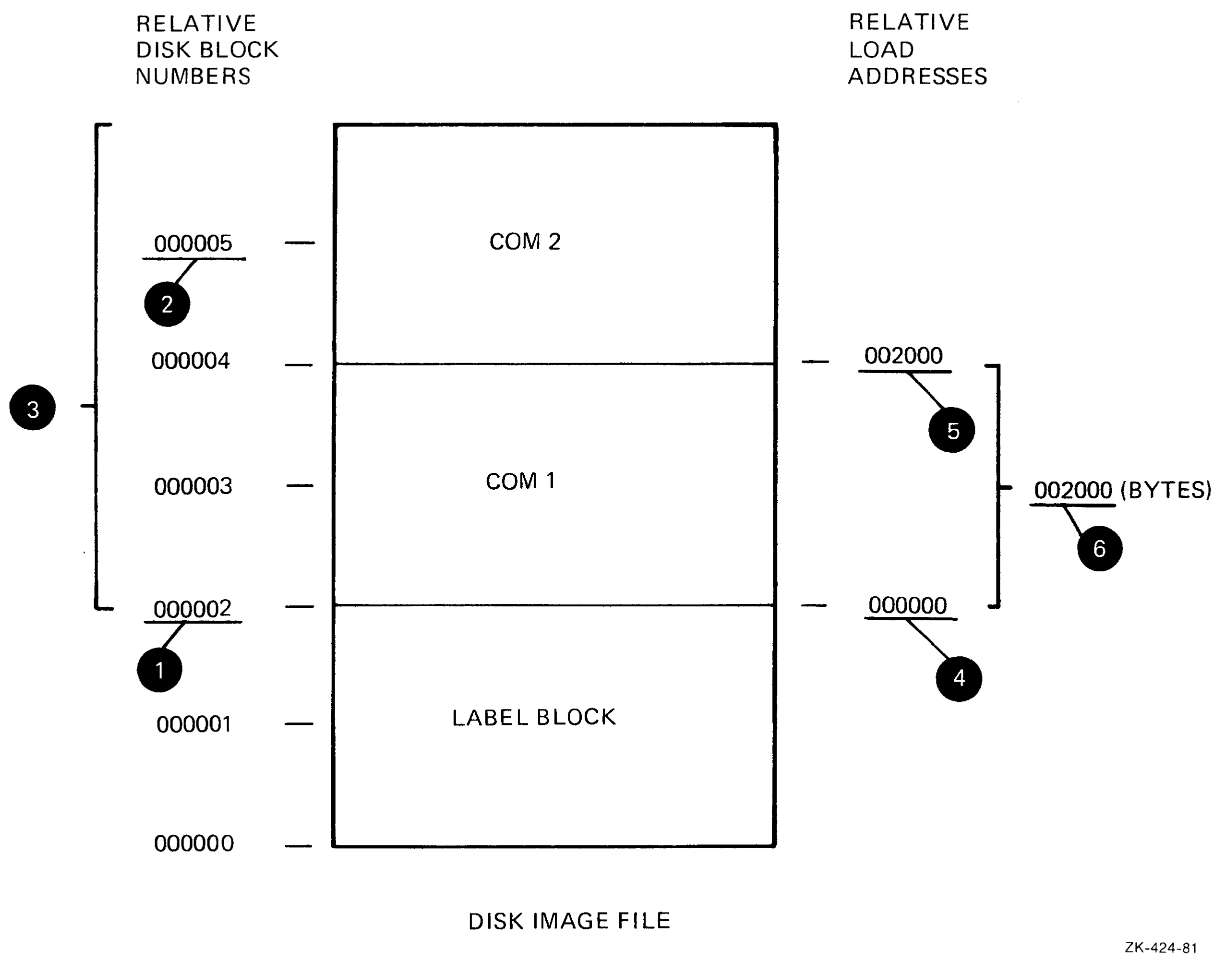
The common remains in memory until you explicitly remove it with the MCR or DCL command REMOVE. The common will not be loaded until either one of the following actions occurs:

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

Because of the checkpointable common feature, changes made in the memory image of the common are preserved when the common is removed. Using the REMOVE command causes the common, with its changes, to be written into the common's task image file. Reinstalling the common then produces in memory the common that was saved with its corresponding changed values.

Example 5-1, Parts 3 and 4 show two programs: MCOM1 and MCOM2, respectively. Both of these programs reference the common area MACCOM created above. MCOM1 in Example 5-1, Part 3 accesses the COM1 portion of MACCOM. It inserts into the first 10 words of COM1 the numbers 1 to 10 in ascending order. It then issues an Executive directive request for the task MCOM2 and suspends itself.

Figure 5–8: Allocation Diagram for MACCOM.TSK



Example 5–1: Part 3, MACRO–11 Source Listing for MCOM1

```
        .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(10) WORDS OF THE
;       COMMON.

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

;       PSECT - COM2 IS USED TO ACCESS THE SECOND 512(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     #1,R1                     ; START WITH A 1
        MOV     #INT,R3                   ; PLACE VALUES IN FIRST 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
```

Example 5-1: Part 4, MACRO-11 Source Listing for MCOM2

```
        .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(10) WORDS OF THE
;       COMMON.

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

;       PSECT - COM2 IS USED TO ACCESS THE SECOND 10(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 FIRST 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
```

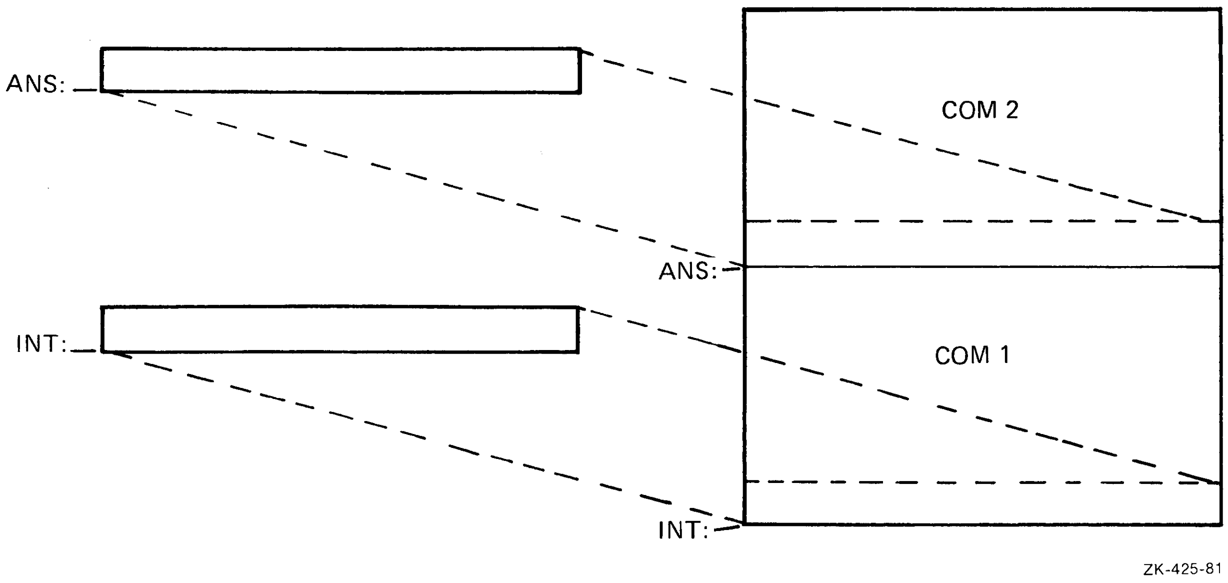
When MCOM2 runs, it adds together the integers left in COM1 by MCOM1 and leaves the sum in the first word of COM2. It then issues a Resume (RSUM$C) directive for MCOM1 and exits.

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 and Micro/RSX Executive Reference Manual*. The system library routine $EDMSG is documented in the *RSX-11M-PLUS and Micro/RSX System Library Routines Reference Manual.*

Note that both MCOM1 and MCOM2 contain .PSECT declarations establishing 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 invalid. 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, TKB assigns to the symbol INT: the base address of program section COM1, and to the symbol ANS: the base address of program section COM2. Figure 5-9 illustrates this assignment.

Figure 5-9: Assigning Symbolic References Within a Common



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

```
        TKB                              LINK

>TKB                            $ LINK/TAS/MAP:MCOM1/NOPRINT/OPT MCOM1
TKB>MCOM1,MCOM1/-SP=MCOM1       Option? RESCOM=MACCOM/RW
TKB>/                           Option? RET
Enter Options:                  $
TKB>RESCOM=MACCOM/RW
TKB>//
>

>TKB                            $ LINK/TAS/MAP:MCOM2/NOPRINT/OPT MCOM2
TKB>MCOM2,MCOM2/-SP=MCOM2       Option? RESCOM=MACCOM/RW
TKB>/                           Option? RET
Enter Options:                  $
TKB>RESCOM=MACCOM/RW
TKB>//
>
```

Under options in both of these command sequences, the RESCOM option tells TKB 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, TKB expects to find the image file and the symbol definition file for the common on device SY in the directory that corresponds to the terminal UIC. In addition, because the optional APR specification was omitted from the RESCOM option, TKB allocates virtual address space for the common starting with APR 7 in both tasks (the highest APR available in both tasks).

The TKB map for MCOM1 is shown in Example 5-1, Part 5. The map for MCOM2 is not essentially different from that of MCOM1 and is therefore not included here.

Example 5–1: Part 5, Task Builder Map for MCOM1.TSK

```
MCOM1.TSK;1      Memory allocation map TKB M43.00       Page 1
                       11-DEC-87   16:12


Partition name : GEN
Identification : 01
Task  UIC      : [7,62]
Stack    limits: 000274 001273 001000 00512.
PRG xfr address: 001650
Total address windows: 2.
Task  image  size  : 1184. words
Task address limits: 000000 004407
R-W disk blk limits: 000002 000006 000005 00005.

*** Root segment: MCOM1

R/W mem  limits: 000000 004407 004410 02312.
Disk blk limits: 000002 000006 000005 00005.

Memory allocation synopsis:

Section                                        Title  Ident  File
-------                                        -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001274 002664 01460.
                          001274 000574 00380. MCOM   01     MCOM1.OBJ;1
COM1  :(RW,D,GBL,REL,OVR) 160000 002000 01024.
                          160000 000024 00020. MCOM   01     MCOM1.OBJ;1
COM2  :(RW,D,GBL,REL,OVR) 162000 002000 01024.
                          162000 000002 00002. MCOM   01     MCOM1.OBJ;1
$DPB$$:(RW,I,LCL,REL,CON) 004160 000016 00014.
                          004160 000016 00014. MCOM   01     MCOM1.OBJ;1
$$RESL:(RO,I,LCL,REL,CON) 004176 000212 00138.


*** Task builder statistics:

    Total work file references: 1924.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 7086. words (27. pages)
    Size of work file: 1024. words (4. pages)

    Elapsed time:00:00:04
```

Note that TKB has placed two window blocks in MCOM1's header. When MCOM1 is installed, the INSTALL task 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.

5.1.9 Linking Shared Regions Together

Shared regions can link to other shared regions. You may find it convenient to have code in a shared library and have access to routines in another shared library to which it links.

The following text describes, as an example for a mapped system, the TKB command sequence for building a resident library named FILEB. That text is followed by TKB and LINK command sequences that show an example of building another resident library named FORCOM that links to FILEB. Following that, TKB and LINK command sequences show the building of a task that links to FORCOM. In the TKB and LINK command sequences to follow, it is assumed that you know the contents of the libraries and the task. The examples show the linkage only.

The first shared region to be built is called FILEB. The library FILEB is a position-dependent library. You use the /-PI switch or the /NOCODE:PIC qualifier to signify that the library is absolute. You build the library with the /-HD switch or the /NOHEADER qualifier to indicate that the library has no header. The /LI switch or the /SHAREABLE:LIBRARY qualifier indicates that FILEB is to be a shared library. The program section name of the library is . ABS, which is the only one in the library. FILEB is to be loaded into a user-controlled partition on a mapped system. The name of the partition in which FILEB resides has the same name, FILEB, that you specify in the PAR option. The PAR option also specifies the base address and the length of the partition. Because FILEB is absolute, a base address must be specified; here, the base address is 160000. The length in this example is 4K bytes. If neither the base nor the length is specified, TKB tries to determine the length.

For TKB, use the following command sequence:

```
>TKB RET
TKB>FILEB/-PI/-HD/LI,FILEB/-SP,FILEB=FILEB.OBJ RET
TKB>/ RET
Enter Options:
TKB>STACK=0 RET
TKB>PAR=FILEB:160000:40000 RET
TKB>// RET
```

For the LINK command, use the following command sequence:

```
$ LINK/TAS/SHARE:LIBRARY/NOHEAD/MAP:FILEB/NOPRINT/SYM/OPT FILES RET
Option? STACK=0 RET
Option? PAR=FILEB:160000:40000 RET
Option? RET
$
```

The next TKB command sequence specifies a shared library called FORCOM. FORCOM links to the read-only library called FILEB. You build FORCOM with the /LI switch or /SHAREABLE:LIBRARY qualifier to specify a library to the Task Builder. FORCOM is relocatable. You specify in the RESLIB option that the resident library to which FORCOM links is called FILEB. The access required is read-only, which /RO specifies in the RESLIB option line.

For TKB, use the following command sequence:

```
>TKB [RET]
TKB>FORCOM/-HD/LI/PI,FORCOM/-SP,FORCOM=FORCOM.OBJ [RET]
TKB>/ [RET]
Enter Options:
TKB>STACK=0 [RET]
TKB>PAR=FORCOM:0:4000 [RET]
TKB>RESLIB=FILEB/RO [RET]
TKB>// [RET]
>
```

For LINK, use the following command sequence:

```
$ LINK/TAS:FORCOM/NOHEAD/CODE:PIC/SHARE:LIB/MAP:FORCOM/NOPRINT/SYM/OPT- [RET]
->FORCOM [RET]
Option? STACK=0 [RET]
Option? PAR=FORCOM:0:4000 [RET]
Option? RESLIB=FILEB/RO [RET]
Option? [RET]
$
```

The next command sequences build the task and specify that the task links to the library called FORCOM. The RESLIB option line specifies the link to the resident library called FORCOM.

For TKB, use the following command sequence:

```
>TKB [RET]
TKB>FOTASK,FOTASK/-SP,FOTASK=FOTASK.OBJ [RET]
TKB>/ [RET]
Enter Options:
TKB>RESLIB=FORCOM/RW [RET]
TKB>// [RET]
>
```

For LINK, use the following command sequence:

```
$ LINK/TAS:FOTASK/MAP:FOTASK/NOPRINT/SYM/OPT FOTASK [RET]
Option? RESLIB=FORCOM/RW [RET]
Option? [RET]
$
```

Build the libraries before you build the task, and install the libraries before you run or install the task. See Chapter 10 for a description of the /PI, /HD, /CO, and /LI switches; and see Chapter 11 for a description of the /CODE:PIC, /[NO]HEADER, /SHAREABLE:COMMON, and /SHAREABLE:LIBRARY qualifiers. See Chapter 12 for a description of the PAR, RESCOM, and RESLIB options.

5.1.10 Example 5-2: 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. Example 5-2, Part 1 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.

Example 5-2: Part 1, Assembly Listing for TTCOM

```
        .TITLE TTCOM
        .PSECT TTCOM,GBL,D,RW,OVR
        .=.+60
$RCSR:: .BLKW   1
$RBUF:: .BLKW   1
$XCSR:: .BLKW   1
$XBUF:: .BLKW   1
        .END
```

The *PDP-11 Peripherals Handbook* defines the control and data register addresses for the console terminal. In Example 5-2, Part 1, 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 Example 5-2, Part 1 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 TKB command sequence:

```
>TKB RET
TKB>LB:[1,1]TTCOM/-HD/PI,LB:[1,1]TTCOM/-WI/SP,LB:[1,1]TTCOM=TTCOM RET
TKB>/ RET
Enter Options:
TKB>STACK=0 RET
TKB>PAR=TTCOM:0:100 RET
TKB>// RET
>
```

Or, by using the following LINK command sequence:

```
$ LINK/TAS:LB:[1,1]TTCOM/NOH/COD:PIC/MAP/NOWIDE/PRINT/SYM/OPT TTCOM RET
Option? STACK=0 RET
Option? PAR=TTCOM:0:100 RET
Option? RET
$
```

This command sequence directs TKB to create a common image named TTCOM.TSK and a symbol definition file named TTCOM.STB. TKB places both files on device LB in directory [1,1]. The command sequence also specifies that TKB is to queue a map listing to the line printer.

In TKB, the /-WI switch specifies an 80-column line printer listing format. In the LINK command, /NOWIDE specifies an 80-column format. The /PRINT qualifier need not be present because printing of the map file is the default operation.

Note

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

The STACK=0 option suppresses the stack area in the common's image file. The PAR option also specifies that the base of the common is 0 and that it is 100_8 bytes long.

The TKB map for TTCOM that results from the command sequence above is shown in Example 5-2, Part 2. 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 in directory [1,1].

The map in Example 5-2, Part 2 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.

Example 5–2: Part 2, Task Builder Map for TTCOM

```
TTCOM.TSK;1       Memory allocation map  TKB M43.00       Page 1
                         1-DEC-87   17:02


Partition name : TTCOM
Identification :
Task  UIC      : [7,62]                                    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: 000002 000002 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,D,GBL,REL,OVR) 000000 000070 00056.
                          000000 000070 00056. .MAIN.        TTCOM.OBJ;1


Global symbols:

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


*** Task builder statistics:

    Total work file references: 214.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 6666. WORDS (26. PAGES)
    Size of work file: 768. WORDS (3. PAGES)

    Elapsed time:00:00:02
```

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.

On an RSX–11M–PLUS or Micro/RSX system, use the following command line:

```
>SET /PAR=TTCOM:177775:1:DEV [RET]
```

These previous SET command lines create a main partition named TTCOM that begins at physical address 777500 in 18-bit systems and physical address 1777750 in 22-bit systems. The partition is one 64-byte block in length (100_8 bytes). The argument DEV identifies the partition type. You can establish the partition for a device common at any time. 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.

Example 5-2, Part 3 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.

Example 5-2: Part 3, Assembly Listing for TEST

```
        .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
```

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

```
>TKB RET
TKB>TEST,TEST/-WI/MA=TEST RET
TKB>/ RET
Enter Options:
TKB>COMMON=TTCOM:RW:1 RET
TKB>// RET
>
```

For the LINK command, you can build TEST with the following command sequence:

```
$ LINK/TAS/MAP/SYS/NOWIDE/OPT TEST RET
Option? COMMON=TTCOM:RW:1 RET
Option? RET
$
```

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

The TKB map that results from the command sequence above is shown in Example 5-2, Part 4.

Example 5-2: Part 4, Memory Allocation Map for TEST

```
TEST.TSK;1      Memory allocation map  TKB M43.00        Page 1
                       1-DEC-87   17:03


Partition name : GEN
Identification : 01
Task  UIC      : [7,62]
Stack    limits: 000274 001273 001000 00512.
PRG xfr address: 001274
Total address windows: 2.
Task  image  size  : 384. WORDS
Task address limits: 000000 001377
R-W disk blk limits: 000002 000003 000002 00002.

*** Root segment: TEST

R/W mem  limits: 000000 001377 001400 00768.
Disk blk limits: 000002 000003 000002 00002.


Memory allocation synopsis:

Section                                        Title  Ident  File
-------                                        -----  -----  ----
  BLK.:(RW,I,LCL,REL,CON) 001274 000100 00068.
                          001274 000100 00068. .MAIN.        TEST.OBJ;1

TTCOM :(RW,D,GBL,REL,OVR) 200000 000070 00056.
                          200000 000070 00056. TTCOM         TTCOM.STB;1

Global symbols:

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


*** Task builder statistics:

Total work file references: 243.

    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 6666. WORDS (26. pages)
    Size of work file: 768. WORDS (3. pages)

    Elapsed time:00:00:03
```

This map contains a global symbols section. TKB 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 TKB 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.

5.1.11 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.

Example 5-3, Part 1 shows five FORTRAN-callable subroutines, as follows:

- 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 constitute an example of a resident library. FORTRAN-callable routines were used in this example so that the routines can be accessed by either FORTRAN or MACRO–11 programs.

Example 5-3: Part 1, Source Listing for Resident Library LIB.MAC

```
        .TITLE  LIB
        .IDENT  /01/


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

;** FORTRAN CALLABLE SUBROUTINE TO ADD TWO INTEGERS

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

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

;** FORTRAN CALLABLE SUBROUTINE TO MULTIPLY TWO INTEGERS

MULL::  CALL    $SAVAL              ; SAVE RO-R5
        MOV     @2(R5),RO           ; FIRST OPERAND
        MOV     @4(R5),R1           ; SECOND OPERAND
        MUL     RO,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 RO-R5
        MOV     @2(R5),R3           ; FIRST OPERAND
        MOV     @4(R5),R1           ; SECOND OPERAND
```

(Continued on next page)

Example 5-3 (Cont.): Part 1, Source Listing for Resident Library LIB.MAC

```
        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
```

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

```
TKB>LIB/PI/-HD/LI,LIB/-WI,LIB=LIB RET
TKB>/ RET
Enter Options:
TKB>STACK=0 RET
TKB>PAR=LIB:0:200 RET
TKB>// RET
>
```

Or, for LINK, you can use the following command sequence:

```
$ LINK/TAS/CODE:PIC/NOHEAD/SHARE:LIB/MAP/NOWIDE/SYM/OPT LIB RET
Option? STACK=0 RET
Option? PAR=LIB:0:200 RET
Option? RET
$
```

The TKB command sequence just shown instructs TKB to build a position-independent, headerless library image named LIB.TSK. It instructs TKB to create a map file, LIB.MAP, and to output an 80-column listing to the line printer. It also specifies that TKB is to create a symbol definition file, LIB.STB. TKB creates all three files—LIB.TSK, LIB.MAP, and LIB.STB—on device SY in the directory that corresponds to the terminal UIC. The /LI and /PI switches used together cause TKB to name the program section LIB, which is the root segment of the library. LIB becomes the only named program section in the library.

The LINK command sequence takes the name of the input file (LIB) as the default name for the task file, the map file, and the symbol definition file. The qualifiers in the LINK command have the following functions: the /CODE:PIC qualifier specifies a relocatable library; the /NOHEAD qualifier is required for building a library or common; the /SHARE:LIB qualifier specifies that a library be built; the /MAP qualifier requests a map, uses the input file name for the default name, and outputs the map file to the line printer by default; the /NOWIDE qualifier requests an 80-column listing; the /SYM qualifier requests a symbol definition file; and the /OPT qualifier requests a prompt for options.

If you used the command sequence above without the /LI switch or /SHAREABLE:LIBRARY qualifier, TKB would create a common by default.

The STACK=0 option suppresses the stack area within the resident library's image. The PAR option tells TKB that the resident library will reside within a partition of the same name as that of the library. In addition, the PAR option specifies that the base of the library is 0 and that it is 200_8 bytes in length. (For more information on the switches, qualifiers, and options used in this example, refer to Chapters 10, 11, and 12, respectively.)

Example 5-3, Part 2 shows the TKB map that results from the command sequence shown previously.

Note in the global symbols section of the map in Example 5-3, Part 2 that TKB has assigned offsets to the symbols for each library function. When the task that links to this library is built, TKB will assign virtual addresses to these symbols.

The program MAIN in Example 5-3, Part 3 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.

Example 5–3: Part 2, Task Builder Map for LIB.TSK

```
LIB.TSK;1      Memory allocation map  TKB M43.00      Page 1
                     11-DEC-87   13:50


Partition name : LIB
Identification : 01
Task  UIC      : [7,62]
Task Attributes: -HD,PI
Total address windows: 1.
Task  image  size  : 64. words
Task address limits: 000000 000163
R-W disk blk limits: 000002 000002 000001 00001.

*** 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

*** Task builder statistics:

    Total work file references: 368.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 7086. words (27. pages)
    Size of work file: 768. words (3. pages)

    Elapsed time:00:00:03
```

Example 5-3: Part 3, Source Listing for MAIN.MAC

```
        .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
```

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

```
TKB>MAIN,MAIN/MA/-WI/-SP=MAIN  RET
TKB>/  RET
Enter Options:
TKB>RESLIB=LIB/RO:3  RET
TKB>//  RET
>
```

Or, you can use the following LINK command sequence to build it:

```
$ LINK/TAS/MAP:MAIN/SYS/NOWIDE/NOPRINT/OPT MAIN [RET]
Option? RESLIB=LIB/RO:3 [RET]
Option? [RET]
$
```

These command sequences instruct TKB to build a task file named MAIN.TSK on device SY in the directory that corresponds to the terminal UIC. It also specifies that TKB is to create a map file MAIN.MAP. The /MA switch or /SYS qualifier requests an extended map format. In the TKB example, /MA was applied to the device specification so that TKB would include in the map for the task the symbols within the library LIB. In DCL, the /SYS qualifier includes the symbols within the library into the map. The negated form of the wide listing switch (/-WI) was appended to the map specification to obtain an 80-column map format. In DCL, the /NOWIDE qualifier specified an 80-column map format. In this example, /-SP and /NOPRINT prevent TKB from spooling a map listing to the line printer.

The RESLIB option specifies that the task MAIN is to access the library LIB and that it requires read-only access to LIB. TKB uses APR 3 to map the library.

The TKB map that results from this command sequence is shown in Example 5-3, Part 4.

Example 5–3: Part 4, Task Builder Map for MAIN.TSK

```
MAIN.TSK;1      Memory allocation map  TKB M43.00       Page 1
                      11-DEC-87   13:51

Partition name : GEN
Identification : 01
Task  UIC      : [7,62]
Stack     limits: 000274 001273 001000 00512.
PRG xfr address: 001634
Total address windows: 2.
Task  image  size  : 1152. WORDS
Task address limits: 000000 004327
R-W disk blk limits: 000002 000006 000005 00005.

*** Root segment: MAIN

R/W mem  limits: 000000 004327 004330 02264.
Disk blk limits: 000002 000006 000005 00005.


Memory allocation synopsis:

Section                                    Title  Ident  File
-------                                    -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001274 002620 01424.

                          001274 000530 00344. MAIN   01     MAIN.OBJ;1
                          002024 001050 00552. EDTMG  15     SYSLIB.OLB;1034
                          003074 000216 00142. CBTA   04.3   SYSLIB.OLB;1034
                          003312 000074 00060. CATB   03     SYSLIB.OLB;1034
                          003406 000250 00168. EDDAT  03     SYSLIB.OLB;1034
                          003656 000126 00086. CDDMG  00     SYSLIB.OLB;1034
                          004004 000110 00072. C5TA   02     SYSLIB.OLB;1034
LIB   :(RO,I,GBL,REL,CON) 060000 000166 00118.
                          060000 000166 00118. LIB    01     LIB.STB;17
$$RESL:(RO,I,LCL,REL,CON) 004114 000212 00138.
                          004114 000024 00020. SAVRG  04     SYSLIB.OLB;1034
                          004140 000066 00054. ARITH  03.04  SYSLIB.OLB;1034
                          004226 000100 00064. DARITH 0007   SYSLIB.OLB;1034

Global symbols:

AADD    060000-R  $CBDSG 003110-R  $CDTB   003312-R  $EDMSG 002122-R
DIVV    060024-R  $CBOMG 003116-R  $COTB   003320-R  $MUL    004140-R
IO.WVB 011000     $CBOSG 003124-R  $C5TA   004004-R  $SAVRG 004114-R
MULL    060052-R  $CBTA  003154-R  $DAT    003452-R  $TIM    003532-R
SUBB    060140-R  $CBTMG 003132-R  $DDIV   004264-R
$CBDAT 003074-R   $CBVER 003116-R  $DIV    004170-R
$CBDMG 003102-R   $CDDMG 003656-R  $DMUL   004226-R
```

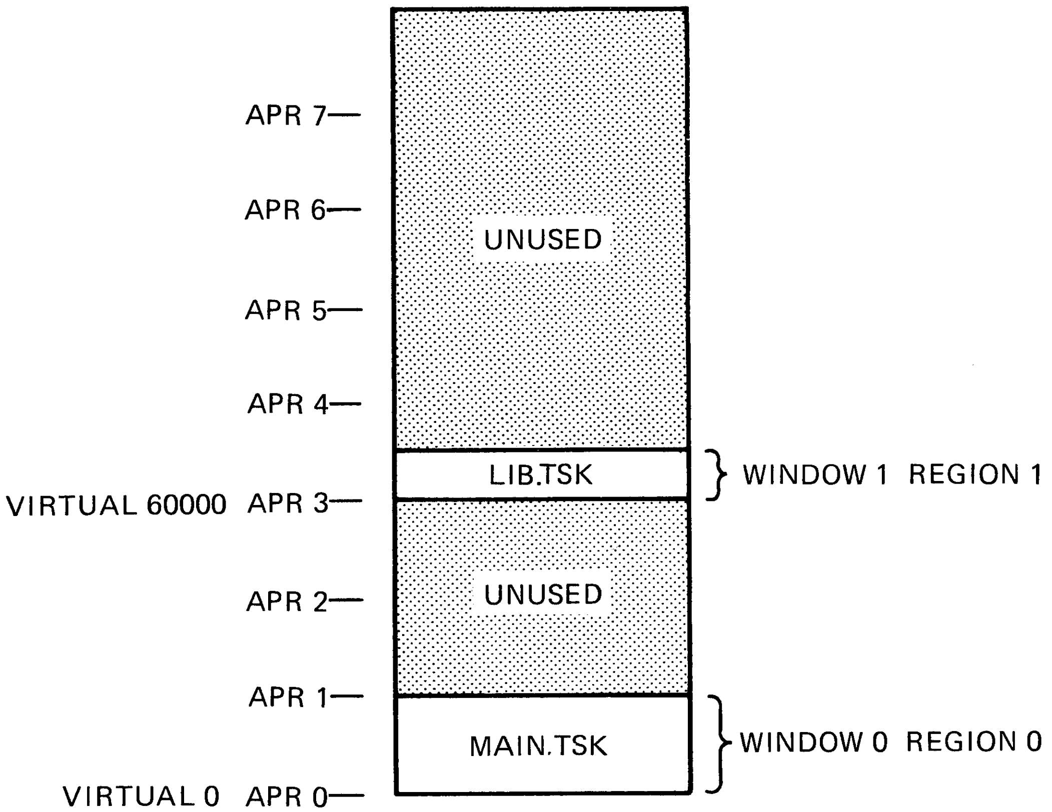
(Continued on next page)

Example 5-3 (Cont.): Part 4, Task Builder Map for MAIN.TSK

```
*** Task builder statistics:

    Total work file references: 2218.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 2066. words (8. pages)
    Size of work file: 1024. words (4. pages)

    Elapsed time:00:00:19
```

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, the virtual base address of APR 3. The Task Builder's allocation of virtual address space for MAIN.TSK is shown in Figure 5-10.

Figure 5-10: Allocation of Virtual Address Space for MAIN.TSK



ZK-426-81

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 5-1 (Section 5.1.7). If the optional APR parameter in the RESLIB option above had been left blank, TKB would have allocated the highest available APR to map the library.

5.1.11.1 Resolving Program Section Names in a Shared Region

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 TKB resolves a call from within a module that 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 5.1.6 for additional information on the sequence in which TKB 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 5-3. When this example was first created, only the arithmetic routines were included in the source file of the resident library (LIB.MAC in Example 5-3, Part 1). The system library coroutine ($SAVAL) was resolved from SYSLIB. Because the first instruction of each arithmetic routine called $SAVAL, TKB 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 in 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 as absolute. In this case, TKB would not have included program section names from the resident library into the STB file for the resident library.

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, TKB 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, TKB resolves the reference to the routine in the resident library instead of in the object module library.

5.1.12 Building a Task That Creates a Dynamic Region

In all the examples of tasks shown thus far in this chapter, TKB 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 task has been responsible for initializing the window blocks when the task was installed. In all the examples, this has been possible because both TKB and the INSTALL task 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 or INSTALL. Therefore, when TKB 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 TKB how many window blocks (in excess of those created by TKB 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 include as many as 16 window blocks.

The text in the remainder of this section and the figures associated with it illustrate the development of a task that creates dynamic regions. Example 5-4 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 to create and manipulate the region (RDBBK$, WDBBK$, DTRG$S, EXIT$S, CRRG$S, CRAW$S, QIOW$S, and QIOW$C) are described in the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual.*

Example 5–4: Part 1, Source Listing for DYNAMIC.MAC

```
        .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(10) 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-WORD BLOCKS IN REGION
```

(Continued on next page)

Example 5–4 (Cont.): Part 1, Source Listing for DYNAMIC.MAC

```
        .REPT   5               ; MULTIPLY
        ASL     R2              ; BY
        .ENDR                   ; 32(10)
        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                  ;

;
;       ERROR ROUTINES
;
1$:     MOV     #ERR1,R0        ; CREATE FAILED
        MOV     #SIZ1,R1        ; SIZE OF MESSAGE
        BR      6$              ; 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
```

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

```
TKB>DYNAMIC,DYNAMIC/-WI/-SP=DYNAMIC [RET]
TKB>/ [RET]
Enter Options:
TKB>WNDWS=1 [RET]
TKB>// [RET]
>
```

Or, you can use the following LINK command sequence:

```
$ LINK/TAS/MAP:DYNAMIC/NOWIDE/NOPRINT/OPT DYNAMIC [RET]
Option? WNDWS=1 [RET]
Option? [RET]
$
```

This command sequence directs TKB to create a task image named DYNAMIC.TSK and an 80-column (/-WI; or /NOWIDE in DCL) map file named DYNAMIC.MAP on device SY under the terminal UIC. Because /-SP, or /NOPRINT, is attached to the map file in the command line, TKB does not spool the file to the line printer.

Under options, the WNDWS option directs TKB 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 Example 5-4, Part 2.

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 (CRRG$) directive would fail.

Example 5-4: Part 2, Task Builder Map for DYNAMIC.TSK

```
DYNAMIC.TSK;1        Memory allocation map  TKB M43.00        Page 1
                           11-DEC-87  16:05

Partition name : GEN
Identification : V01
Task  UIC      : [7,62]

Stack    limits: 000274 001273 001000 00512.
PRG xfr address: 001470
Total address windows: 2.

Task  image  size  : 512. WORDS
Task address limits: 000000 001753
R-W disk blk limits: 000002 000003 000002 00002.


*** Root segment: DYNAMI

R/W mem  limits: 000000 001753 001754 01004.
Disk blk limits: 000002 000003 000002 00002.


Memory allocation synopsis:

Section                                        Title  Ident  File
-------                                        -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001274 000430 00280.
                          001274 000430 00280. DYNAMI V01    DYNAMIC.OBJ;1

$DPB$$:(RW,I,LCL,REL,CON) 001724 000030 00024.
                          001724 000030 00024. DYNAMI V01    DYNAMIC.OBJ;1


*** Task builder statistics:

    Total work file references: 549.
    Work  file  reads: 0.
    Work  file writes: 0.

    Size of core pool: 7086. words (27. pages)
    Size of work file: 768. words (3. pages)

    Elapsed time:00:00:06
```

5.2 Cluster Libraries

The term "cluster libraries" refers to both a function and a structure created by the Task Builder that allow a task to dynamically map memory-resident shared regions at run time. Cluster libraries permit a task to use, for example, an F77CLS library, an FMS–11 library, and an FCS-11 library, all mapped through the same task address window. The run-time routines put into the task by the Task Builder remap the library regions so that, instead of occupying 48K bytes of virtual address space, they share 16K bytes of virtual address space.

Note

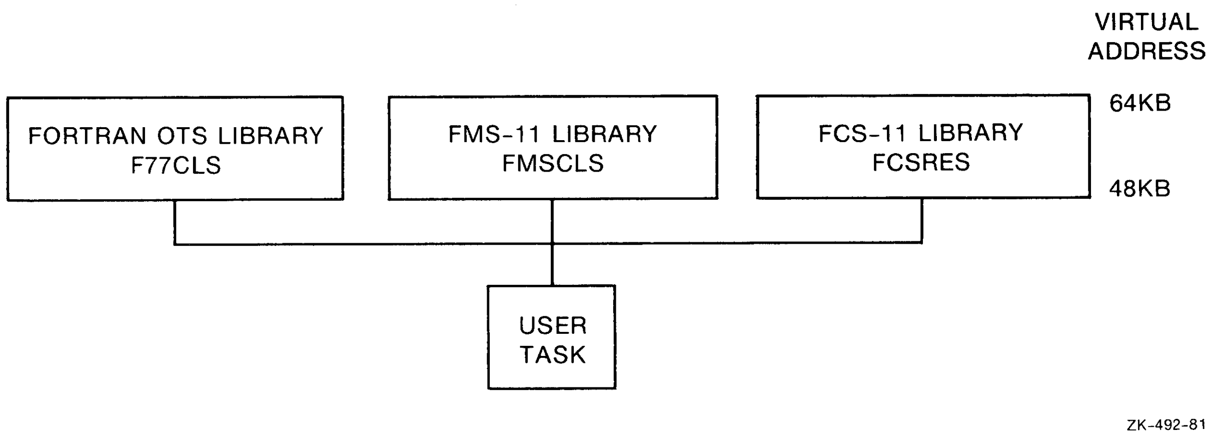
Read-only and read/write libraries cannot be clustered together.

One task address window (window 1) maps the libraries into the same span of virtual address space (48K bytes to 64K bytes). TKB maps your task from virtual 0 upward.

TKB implements the cluster library function in two parts. The first part, revectoring of interlibrary calls, is independent of the actual remapping mechanism but is required for remapping to work. The second part executes the required MAP$ directives to map the appropriate library.

The following examples use the library and task structure shown in Figure 5-11. Note that in the following examples, the FMS–11/RSX Version 2.0 and FORTRAN–77 software products are sold under separate license and are not included with the RSX–11M–PLUS and Micro/RSX systems. Cluster library support may be used with RMS–11 Version 2.0 or later versions, and operates in a fashion similar to the FCS-11 example. Also, the particular FCSRES library used below is generated by SYSGEN. It consists of two memory management overlays and a null root.

Figure 5–11: Example of Library and Task Structure



5.2.1 Building the Libraries

You must follow several rules when designing and building shareable clustered libraries. The rules are summarized as follows:

- All libraries but the first require resident overlays.
- User task vectors indirectly resolve all interlibrary references.
- Revectored entry point symbols must not appear in the "upstream" STB file.
- A called library procedure must not require parameters on the stack.

- All the non-position-independent libraries must be built for the same address.
- Trap or asynchronous entry into a library is not permitted.

The rules are discussed in detail in the following sections.

5.2.2 Rule 1: All Libraries but the First Require Resident Overlays

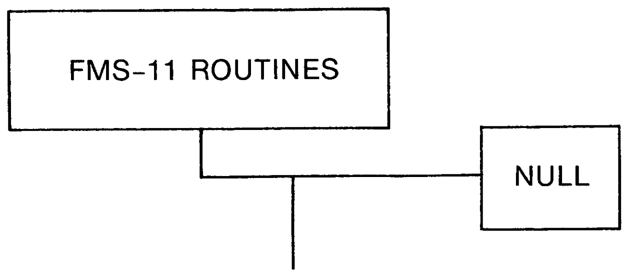
The first library is the first named library in the CLSTR option line. To obtain the required run-time overlay data structures in your task, you must define all the libraries (except possibly the first one) by using memory-resident overlays. Although the first library can be an overlaid library, it need not be and can be a single-segment structure. If the first library is overlaid with a null root, the overlay run-time system cannot distinguish between the first library and the other libraries in the cluster (those named in the CLSTR option after the first library). Therefore, if the first library called is not the first library named in the CLSTR option, severe performance degradation may be noticed because of excessive mapping and unmapping of the libraries. Therefore, to avoid performance degradation if the first library is overlaid with a null root, make certain that the first library called is the first library named in the CLSTR option.

All the libraries, except the first, must have a null root if overlaid. You can achieve this in cases where a library is not normally overlaid by creating an unbalanced overlay structure with a null module. For example, the following ODL specification for FMSCLS and a null module would suffice:

```
        .NAME FMSCLS
        .ROOT FMSCLS-*(NULL,FMSLIB)
NULL:   .FCTR LB:[1,1]SYSLIB/LB:NULL          ;NULL MODULE
FMSLIB: .FCTR SY:FMSLIB-LB:[1,1]FDVLIB/LB     ;FMS-11 ROUTINES
        .END
```

The above ODL specification creates an unbalanced tree in the form shown in Figure 5-12.

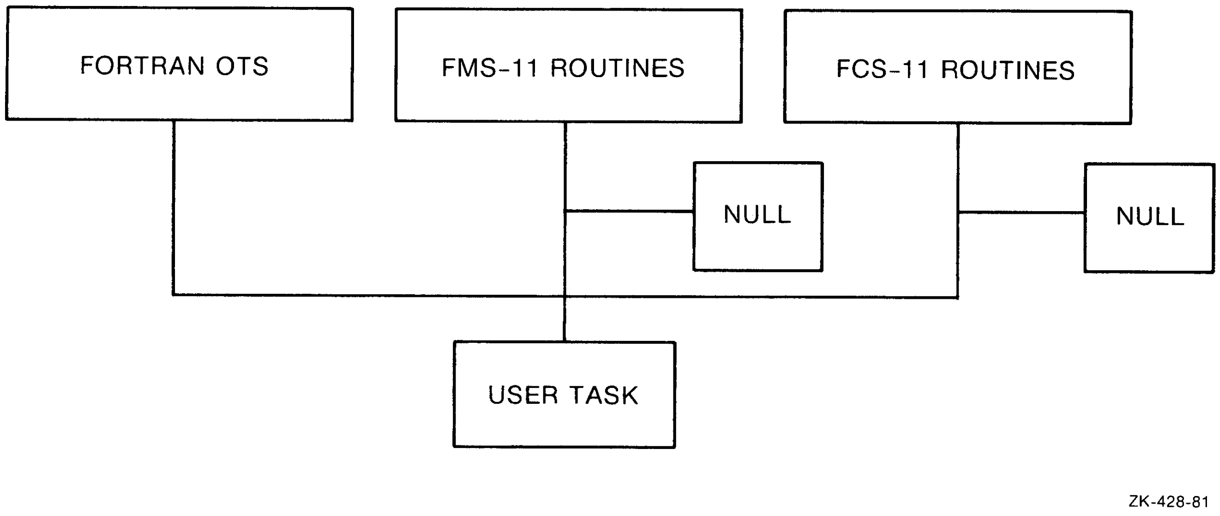
Figure 5–12: Example of an Unbalanced Tree with Null Segment



ZK-427-81

The effect, after you build your task, is an overlay structure that is represented in Figure 5-13.

Figure 5-13: Example of an Overlay Cluster Library Structure

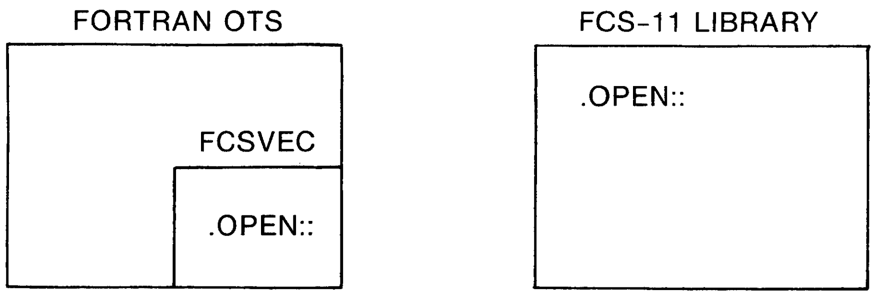


TKB provides the cross-library linkage that it creates from the overlay segment data contained in the individual STB files of each library.

5.2.3 Rule 2: User Task Vectors Indirectly Resolve All Interlibrary References

Figure 5-14 illustrates rule 2 and is a part of the example in Figure 5-13. In Figure 5-14, if the FORTRAN OTS library references the FCS-11 entry point .OPEN, the transfer of control from the FORTRAN OTS library to the FCS-11 library must be resolved by a jump vector in your task. Or, to state it in another way, the CALL instruction in the FORTRAN OTS library must not reference directly the target address (the address of .OPEN) in the FCS-11 library. The system library contains the modules that perform the indirect transfer for FCS-11 based libraries and user tasks. If you want to duplicate the indirect referencing mechanism for your own purposes, Figure 5-14 and the following text describe the control flow for FCS-11.

Figure 5–14: Example of a Vectored Call Between Libraries



Sample code from FCSVEC module:

```
.OPEN::   MOV     #30,-(SP)              ; STACK OFFSET INTO USER TASK
                                         ; JUMP TABLE
          BR      DISPAT                 ; JOIN COMMON DISPATCH
          .
          .
          .
DISPAT:   MOV     R0,-(SP)               ; SAVE REGISTER
          MOV     @#.FSRPT,R0            ; GET FCS-11 POINTER
          ADD     A.JUMP(R0),2(SP)       ; ADD VECTOR BASE TO OFFSET
          MOV     (SP)+,R0               ; RESTORE REGISTER
          MOV     @(SP)+,-(SP)           ; PICK UP ADDRESS OF TARGET
          RETURN                         ; AND TRANSFER TO TARGET
```

ZK-429-81

In this example, the module FCSVEC defines the .OPEN entry point. The code at that location stacks an offset (or "entry number") and joins common dispatch code. The dispatch code, using the low core FCS-11 impure pointer called .FSRPT, obtains the address of the FCS-11 impure data area. At offset A.JUMP in that area is the address of a vector of FCS-11 entry points. A return is executed, which transfers control to the routine whose address is now on top of the stack. If the target routine is an overlaid library, the run-time support ($AUTO) loads the appropriate overlay and relays the transfer of control.

You may use this vectoring mechanism to isolate the linkages between two libraries whether or not you use them in the cluster library scheme. You can replace either the FORTRAN OTS or the FCS-11 library in your system without relinking the other library. However, you must relink your task when you replace either of these libraries.

5.2.4 Rule 3: Revectored Entry Point Symbols Must Not Appear in the "Upstream" STB File

This rule means that the GBLXCL=symbol option must appear for each revectored symbol, as in FORTRAN OTS in this example. In the brief example given in the previous section, the following line must appear in the build file for the FORTRAN OTS library:

```
GBLXCL=.OPEN
```

5.2.5 Rule 4: A Called Library Procedure Must Not Require Parameters on the Stack

This rule applies to routines contained in libraries other than the "default" library, as represented by the FMSCLS and FCSRES libraries of the above example. In addition, the called procedures must use the JSR PC and RTS PC call and return convention. The flow of control for a call into a cluster library member other than the default library proceeds as follows.

Only your task can call and reference the FCSRES library routine .OPEN. All references from other libraries are revectored as described above. TKB resolves all such references to an appropriate task-resident autoload vector. As in the example, when the FORTRAN OTS library calls .OPEN, the code revectors the call through your task and hence to the autoload vector. At this point, the TKB run-time routine $AUTO gets control and searches the overlay segment descriptor tree, noting which segments are resident and which must be loaded or mapped to access the target routine.

Next, $AUTO notes that a member of a library cluster must be unmapped to comply with the map adjustments required to access the target routine. The reference to the unmapped library and the segment within the library is placed on the stack, the target library is mapped, and the target routine is accessed through a JSR PC instruction. That target routine must not attempt to access parameters by offsets from the stack pointer (SP) because of the presence of $AUTO saved information. Upon return from the target by an RTS PC instruction, the target library is unmapped, and the previous library is remapped using the saved segment and library data on the stack. Finally, $AUTO executes an RTS PC instruction to return to the caller.

Note that if your task contains a mix of cluster libraries and noncluster libraries, the call format rule applies only to control transfers to cluster library routines. Other noncluster libraries that you create may use any appropriate call and parameter-passing convention.

5.2.6 Rule 5: All the Non-Position-Independent Libraries Must Be Built for the Same Address

TKB must be able to place each library of the cluster at the same virtual address. To do this, the libraries must be built as position independent or be built to the exact address specified in the CLSTR command described in Section 5.2.8.

5.2.7 Rule 6: Trap or Asynchronous Entry into a Library Is Not Permitted

A routine built as part of a library that is to be used in a cluster may not be specified as the service routine for a synchronous trap, or for asynchronous entry as a result of I/O completion or Executive service. This restriction is required because at the moment of the trap or fault, the appropriate library may not be the one that is mapped. For example, if the default library contains the service routine to display an error message upon odd address trap (the odd address fault occurs within one of the other libraries of the cluster), the routine will not be available to service the trap. It will have been unmapped by the run-time routines to map the called library.

I/O completion and fault service vectors and routines must be placed in libraries or task segments that are resident at all times that the faults, traps, or I/O completions may occur.

5.2.8 Building Your Task

After building the individual libraries and placing the TSK and STB files for all the libraries into the LB:[1,1] directory, you may build your task. The TKB option line that you must use for your task has the following syntax:

```
CLSTR=library_1,library_2,...library_n:switch:apr
```

Parameters

library_n

The first specification denotes the first or the default library, which is the library to which the task maps when the task starts up and remaps after any call to another library.

The total number of libraries to which a task may map is seven. The number of the component libraries in clusters is limited to a maximum of six. A cluster must contain a minimum of two libraries. It is possible to have two clusters of three libraries each or three clusters of two libraries each; any combination of clusters and libraries must equal at least two or a maximum of six. If six libraries are used in clusters, the task may map to only one other, separate library.

:switch:apr

The switch :RW or :RO indicates whether the cluster is read-only or read/write for this particular task. The APR specification is optional and indicates which APR is to be used as the starting APR when mapping to cluster libraries. If not specified, TKB assigns the highest available APRs and as many as required to map the library.

5.2.9 Examples

The sample build files for F77CLS, FDVRES, and FCSRES, and for the FMS-11 demonstration task FMSDEM, are given here as an example of the cluster library build process.

5.2.9.1 F77CLS—Build the Default Library for the FORTRAN-77 OTS

If you use TKB syntax, enter the following command sequence:

```
>TKB RET
TKB>F77CLS/-HD,F77CLS/CR/-SP/MA,F77CLS=F77RES RET
TKB>LB:[1,1]F77OTS/LB RET

TKB>LB:[1,1]SYSLIB/LB:FCSVEC RET ; INCLUDE THE FCS JUMP VECTOR
TKB>/ RET

Enter Options:
STACK=0 RET
PAR=F77CLS:140000:40000 RET

;
; FORCE THE JUMP TABLE TO BE LOADED FROM THE SYSTEM
; LIBRARY WHEN THE USER TASK IS BUILT
;

GBLINC=.FCSJT                       ; REFERENCE SYMBOL DEFINED IN
                                    ; THE MODULE SYSLIB/LB:FCSJMP

;
; PREVENT DEFINITIONS FOR FCS-11 ENTRY POINTS FROM APPEARING
; IN THE STB FILE FOR THIS LIBRARY OR OTHER SYSTEM LIBRARY
;

GBLXCL=.ASLUN RET
GBLXCL=.CLOSE RET
GBLXCL=.CSI1 RET
GBLXCL=.CSI2 RET

GBLXCL=.DELET RET
GBLXCL=.DLFNB RET
GBLXCL=.ENTER RET
GBLXCL=.EXTND RET

GBLXCL=.FCTYP RET
GBLXCL=.FIND RET
GBLXCL=.FINIT RET
GBLXCL=.FLUSH RET

GBLXCL=.GET RET
GBLXCL=.GETSQ RET
GBLXCL=.GTDID RET
GBLXCL=.GETDIR RET

GBLXCL=.MARK RET
GBLXCL=.MRKDL RET
GBLXCL=.OPEN RET
GBLXCL=.OPFID RET
GBLXCL=.OPFNB RET

GBLXCL=.PARSE RET
GBLXCL=.POINT RET
GBLXCL=.POSIT RET
GBLXCL=.POSRC RET

GBLXCL=.PRINT RET
GBLXCL=.PRSDV RET
GBLXCL=.PRSFN RET
GBLXCL=.PUT RET
GBLXCL=.PUTSQ RET
```

```
GBLXCL=.REMOV [RET]
GBLXCL=.SAVR1 [RET]
GBLXCL=.TRNCL [RET]
GBLXCL=.READ [RET]
GBLXCL=.WAIT [RET]
GBLXCL=.WRITE [RET]
// [RET]
```

If you use DCL syntax for the command and options shown, you must do two things. First, create a command file that contains the options and name it (for example, CLUSTR.CMD). The reason you must do this is that DCL cannot contain all these options within its command buffer. This command file can contain the options in the following example sequence:

```
STACK=0!PAR=F77CLS:140000:40000
GBLINC=.FCSJT
GBLXCL=.CSI1,.CSI2,.DLFNB,.FINIT,.GET,.GETSQ,.GTDID
GBLXCL=.MRKDL,.OPFNB,.PARSE,.POINT,.POSRC,.PRINT
GBLXCL=.PUT,.PUTSQ,.SAVR1,.READ,.WAIT
```

Second, enter the following DCL command sequence:

```
$ LINK/TAS:F77CLS/NOH/MAP:F77CLS/NOPRINT/SYS/CROSS/SYM:F77CLS/OPT - [RET]
->F77RES,LB:[1,1]F77OTS/LIB,LB:[1,1]SYSLIB/INC:FCSVEC [RET]
Option? @CLUSTR.CMD [RET]
Option? [RET]
$
```

The GBLINC option, as shown in the TKB and DCL examples, forces TKB to add a global reference entry in the library STB file. This ensures that TKB links certain modules required by the library, such as impure data areas or root-only routines, without further user action. These modules should be in the system library (LB:[1,1]SYSLIB.OLB) or in a library always referenced by your task so that this forced loading mechanism is entirely invisible to you.

5.2.9.2 FDVRES—Build an FMS-11/RSX Version 2.0 Shareable Library

The following is an example command file. You name it FDVRES.CMD. If you use TKB syntax, you can use the following TKB command line:

```
>TKB @FDVRES [RET]
```

If you use DCL syntax, you can use the following LINK command line:

```
$ LINK @FDVRES [RET]
```

```
; TITLE OF THE EXAMPLE COMMAND FILE THAT BUILDS THE FORMS
; MANAGEMENT MEMORY MANAGEMENT RESIDENT LIBRARY FOR USE WITH THE
; TASK BUILDER CLSTR OPTION.

; FDVRES.CMD

;
; THE FOLLOWING CODE IS THE EXAMPLE TKB COMMAND FILE:
;

LB:[1,1]FDVRES/-HD/MM/SG,MP:[1,34]FVRES/MA/-SP,LB:[1,1]FDVRES= [RET]

SY:[1,24]FDVRESBLD/MP [RET]
```

```
STACK=0 [RET]
PAR=FDVRES:140000:40000 [RET]
TASK=FDVRES [RET]

;
; THE FOLLOWING LINE FORCES THE FCS JUMP TABLE TO BE INCLUDED IN THE
; SYMBOL TABLE FILE FOR THE FORMS MANAGEMENT LIBRARY.

;
GBLINC=.FCSJT [RET]

;
; THE FOLLOWING LINES FORCE LIBRARY ENTRY POINTS AND DEFINITIONS INTO
; THE TASK ROOT:

;
GBLREF=CB$CUR,CB$REV,CB$TST,CB$132,DV$BLD,DV$BLK,DV$DHW,DV$DWD [RET]
GBLREF=DV$GRA,DV$REV,DV$UND,D$ATT1,D$ATT2,D$CLRC,D$FID,D$FXLN [RET]
GBLREF=D$LNCL,D$PICT,D$PLEN,D$RLEN,D$VATT,D$2ATT,D1$ALN,D1$ALP [RET]
GBLREF=D1$ARY,D1$COM,D1$MIX,D1$NUM,D1$SCR,D1$SNM,D2$DEC,D2$DIS [RET]

GBLREF=D2$FUL,D2$NEC,D2$REQ,D2$RTJ,D2$SPO,D2$TAB,D2$VRT,D2$ZFL [RET]
GBLREF=FC$ALL,FC$ANY,FC$CLS,FC$CSH,FC$DAT,FC$GET,FC$GSC,FC$LST [RET]
GBLREF=FC$OPN,FC$PAL,FC$PSC,FC$PUT,FC$RAL,RC$RTN,FC$SHO,FC$SLN [RET]
GBLREF=FC$SPF,FC$SPN,FC$TRM,FE$ARG,FE$DLN,FE$DNM,FE$DSP,FE$FCD [RET]

GBLREF=FE$FCH,FE$FLB,FE$FLD,FE$FNM,FE$FRM,FE$FSP,FE$ICH,FE$IFN [RET]
GBLREF=FE$IMP,FE$INI,FE$IOL,FE$IOR,FE$LIN,FE$NOF,FE$NSC,FE$STR [RET]
GBLREF=FE$UTR,FE$INC,FS$SUC,FT$ATB,FT$KPD,FT$NTR,FT$NXT,FT$PRV [RET]
GBLREF=FT$SBK,FT$SFW,FT$SNX,FT$SPR,FT$XBK,FT$XFW,F$ASIZ,F$CHN [RET]

GBLREF=F$FNC,F$IMP,F$LEN,F$NAM,F$NUM,F$REQ,F$RSIZ,F$STS [RET]
GBLREF=F$TRM,F$VAL,IS$ALT,IS$CLR,IS$DEC,IS$DSP,IS$ERR,IS$HFM [RET]
GBLREF=IS$HLP,IS$INS,IS$LST,IS$MED,IS$NMS,IS$SCR,IS$SGN,I$ADVO [RET]
GBLREF=I$ALLC,I$BADR,I$BEND,I$BPTR,I$BSIZ,I$CFRM,I$CURC,I$CURP [RET]

GBLREF=I$DISP,I$DLN1,I$DLN2,I$FADR,I$FBLK,I$FCHN,I$FDES,I$FDST [RET]
GBLREF=I$FDS1,I$FDS2,I$FIXD,I$FMST,I$FOFF,I$FORM,I$FSIZ,I$FXD1 [RET]
GBLREF=I$FXD2,I$HLEN,I$HLPF,I$ILEN,I$IMPA,I$LCOL,I$LINE,I$LLIN [RET]
GBLREF=I$LNCL,I$LPTR,I$LVID,I$NBYT,I$NDAT,I$NFLD,I$PATN,I$PBLN [RET]

GBLREF=I$RESP,I$ROFF,I$STAT,I$STKP,I$SVST,I$VATT,L$CLSZ,L$FDES [RET]
GBLREF=L$LNCL,L$RESP,$$FDVT [RET]
GBLREF=$FDV [RET]

;
; THE FOLLOWING LINES PREVENT THE DEFINITIONS FOR FCS-11 ENTRY POINTS
; FROM APPEARING IN THE FORMS MANAGEMENT LIBRARY STB FILE:

;
GBLXCL=.ASCPP [RET]
GBLXCL=.ASLUN [RET]
GBLXCL=.CLOSE [RET]
GBLXCL=.CTRL [RET]

GBLXCL=.DELET [RET]
GBLXCL=.DLFNB [RET]
GBLXCL=.ENTER [RET]
GBLXCL=.EXTND [RET]
```

```
GBLXCL=.FATAL  RET
GBLXCL=.FCTYP  RET
GBLXCL=.FIND  RET
GBLXCL=.FINIT  RET
GBLXCL=.FLUSH  RET

GBLXCL=.GET  RET
GBLXCL=.GETSQ  RET
GBLXCL=.GTDID  RET
GBLXCL=.GTDIR  RET

GBLXCL=.MARK  RET
GBLXCL=.MBFCT  RET
GBLXCL=.MRKDL  RET

GBLXCL=.OPEN  RET
GBLXCL=.OPFID  RET
GBLXCL=.OPFNB  RET

GBLXCL=.PARSE  RET
GBLXCL=.POINT  RET
GBLXCL=.POSIT  RET
GBLXCL=.POSRC  RET

GBLXCL=.PPASC  RET
GBLXCL=.PPR50  RET
GBLXCL=.PRINT  RET
GBLXCL=.PRSDI  RET

GBLXCL=.PRSDV  RET
GBLXCL=.PRSFN  RET
GBLXCL=.PUT  RET
GBLXCL=.PUTSQ  RET

GBLXCL=.RDFDR  RET
GBLXCL=.RDFFP  RET
GBLXCL=.RDFUI  RET
GBLXCL=.REMOV  RET

GBLXCL=.SAVR1  RET
GBLXCL=.TRNCL  RET
GBLXCL=.WRITE  RET
//  RET
```

5.2.9.3 FDVRESBLD.ODL—Overlay Description for FMS-11/RSX Version 2.0 Cluster Library

The following example file is an overlay description file named FDVRESBLD.ODL. If you use TKB syntax, you enter the command line as follows:

```
>TKB outfile(s)=FDVRESBLD/MP  RET
```

If you use DCL syntax, you enter it as follows:

```
$ LINK/.../.../... FDVRESBLD/OVER  RET
```

```
;
; THE FOLLOWING LINE IS THE FILE NAME OF THE ODL FILE FOR THE
; MEMORY MANAGEMENT RESIDENT FORMS MANAGEMENT LIBRARY:
;
; FDVRESBLD.ODL
;
; THE FOLLOWING LINES OF CODE ARE CONTAINED IN THE ODL FILE FOR THE
; MEMORY MANAGEMENT RESIDENT FORMS MANAGEMENT LIBRARY:
;
          .NAME   FDVROT RET
          .ROOT   FDVROT-*!(MAIN,NULO) RET
NULO:     .FCTR   LB:[1,1]SYSLIB/LB:NULL RET

FCSV:     .FCTR   LB:[1,1]SYSLIB/LB:FCSVEC RET

MAIN:     .FCTR
LB:[1,1]FDVLIB/LB:FDV-LB:[1,1]FDVLIB/LB-FCSV RET

          .END RET
```

5.2.9.4 FCSRES Library Build

FCSRS1BLD.BLD is distributed with the RSX-11M-PLUS and Micro/RSX distribution kits. Refer to the build command and overlay description contained in the files FCSRS1BLD.CMD and FCSRS1BLD.ODL, which can be generated by SYSGEN if you want.

5.2.9.5 F77TST.CMD—File to Build the FMS-11/RSX Version 2.0 FORDEM Test Task

The following is an example build command file named F77TST.CMD. If you use TKB syntax, enter the following command line:

```
>TKB @F77TST.CMD RET
```

If you use DCL syntax, enter the following command line:

```
$ LINK @F77TST.CMD RET

;THE FOLLOWING ARE THE CONTENTS OF THE COMMAND FILE RET
FORDEM/FP,FORDEM/MA/-SP=FORDEM,HLLFOR RET

LB:[1,1]FDVLIB/LB RET
LB:[1,1]F77OTS/LB RET
/ RET

EXTSCT=$$FSR1:2000 RET
CLSTR=F77CLS,FDVRES,FCSRES:RO RET

STACK=200 RET
// RET
```

5.2.10 Overlay Run-Time Support Requirements

The Task Builder uses the STB files of the cluster libraries to obtain the information needed to create the overlay database. For each memory management overlaid cluster library, TKB places autoload vectors, segment descriptors, window descriptors, and a region descriptor in the root of the task. This information comprises the overlay run-time support for the cluster libraries. In Appendix B, Figure B-9 and the accompanying text describe this information. Table 5-1 describes the space needed for the overlay run-time system support that includes cluster libraries. For a complete description of overlay run-time routine sizes, see Section 4.5.

Using cluster libraries conserves virtual space and may require only one window.

Table 5–1: Comparison of Overlay Run-Time Module Sizes

Module	Program Section	Number of Bytes Octal/Decimal	Specific Use
One of the following modules is included in any overlaid task that uses autoload and in any task that links to a memory management overlaid resident library.			
AUTO	$$AUTO	122/82	All tasks that use autoload
AUTOT	$$AUTO	132/90	All tasks with ASTs disabled during autoload
	$$RTQ	32/26	
	$$RTR	30/24	
One of the following modules is included in any overlaid conventional task. OVCTR or OVCTC is included in any nonoverlaid task (conventional or I- and D-space) that links to a memory management overlaid resident library.			
OVCTL	$$MRKS	76/62	Disk overlays only
	$$RDSG	160/112	
	$$PDLS	2/2	
OVCTR	$$MRKS	234/156	Disk and memory management overlays with no cluster libraries
	$$RDSG	332/218	
	$$PDLS	12/10	
OVCTC	$$MRKS	254/172	Disk and memory management overlays with cluster libraries
	$$RDSG	352/234	
	$$PDLS	120/80	
One of the following modules is included in any overlaid I- and D-space task.			
OVIDL	$$MRKS	76/62	Disk overlays only
	$$RDSG	224/148	
	$$PDLS	2/2	
OVIDR	$$MRKS	304/196	Disk and memory management overlays with no cluster libraries
	$$RDSG	502/322	
	$$PDLS	12/10	

Table 5-1 (Cont.): Comparison of Overlay Run-Time Module Sizes

Module	Program Section	Number of Bytes Octal/Decimal	Specific Use
OVIDC	$$MRKS	324/212	Disk and memory management overlays with cluster libraries
	$$RDSG	522/338	
	$$PDLS	120/80	
The overlay data vector OVDAT is included in any overlaid task and in any task that links to a memory management overlaid resident library.			
OVDAT	$$OVDT	24/20	Included in all tasks that perform overlay operations
	$$SGD0	0/0	
	$$SGD2	2/2	
	$$RTQ	0/0	
	$$RTR	0/0	
	$$RTS	2/2	
The overlay error service routine ALERR is included whenever OVDAT is included.			
ALERR	$$ALER	24/20	Overlay error
Manual overlay control (LOAD) is used in place of any AUTO routine. (See Section 4.2, Manual Load.)			
LOAD	$$LOAD	252/170	Manual overlay control
	$$AUTO	14/12	

5.3 Task Building an F4PRES, FORRES, or FMSRES Library with or without FCSRES

The following section describes how and why you may want to link an application task to one or more languages, FMS, or an FCS resident library. Also, this section describes trade-offs of memory, speed, flexibility, and ease of use. For the sake of simplicity, the example of a FORTRAN IV-PLUS OTS resident library (F4PRES) is used in the rest of this text to represent FORTRAN IV-PLUS, FORTRAN IV (FORRES), and FMS (FMSRES) resident libraries

In general, the presence of a permanent resident library is justified when it is used frequently enough that it saves physical memory, compared to having the FCS or OTS code in the task images of frequently used tasks.

In all cases, the application task need only be linked to properly built resident libraries using the LIBR, COMMON, RESLIB, RESCOM, or CLSTR options.

5.3.1 FCSRES—The Types of FCS Resident Libraries

It is possible to build two kinds of FCS resident libraries. They are described in the following sections.

5.3.1.1 Building a Memory-Resident Overlaid FCSRES

SYSGEN can automatically generate a memory-resident overlaid FCS library that uses one APR of task address space. This FCSRES makes available all of FCS (except the little-used routines .CTRL, .PRSDI, and .PPR50), .CSI1, .CSI2, and many other system library routines. (See [1,20]FCSRS1BLD.BLD for a list of routines.) When FCSRES is built from memory, and when it is built from LB:[1,1]ANSLIB.OLB, it uses 6624_{10} words. FCSRES uses one APR in either case because it is composed of two memory-resident overlays and a null root segment.

SYSGEN Phase II and SYSGEN Phase III can link utility tasks to this FCSRES, frequently improving task execution speed and virtual address space and lessening task image size. This feature, plus the fact that no editing of a source file is required, makes the memory-resident FCSRES easier to use than the non-memory-resident FCSRES.

A memory-resident FCSRES requires memory management support.

To task build an application task to both the memory-resident FCSRES and F4PRES, you must take special measures when building the F4PRES. Such a resident library is said to have revectored FCS. Once the F4PRES is built, application tasks may be linked to it with or without linking to FCSRES as well. The TKB cluster library facility, using the CLSTR option, may be used to task build the application task to two or more resident libraries, which saves in virtual address space.

5.3.1.2 Building a Non-Memory-Resident FCSRES

You can manually assemble and build a non-memory-resident FCSRES from the file [200,200] FCSRES.MAC.

If you build the resident library using LB:[1,1]SYSLIB.OLB without editing FCSRES.MAC, you can produce a 3744_{10}-word FCSRES that uses one APR.

If you build the resident library using LB:[1,1]ANSLIB.OLB, you must edit FCSRES.MAC to remove enough FCS routines from FCSRES to bring it below 4K words. If you use ANSLIB.OLB without editing FCSRES.MAC, TKB builds a 4448_{10}-word FCSRES that uses two APRs.

The FCSRES library built from an unedited FCSRES.MAC contains all of FCS. Note that .CSI1 and .CSI2, used by FORTRAN IV and FORTRAN IV-PLUS ASSIGN and OPEN statements, are not present in the non-memory-resident FCSRES. The 1426_{10} bytes used by .CSI1 and .CSI2 will be present in your task's image if you use the ASSIGN or OPEN statement, or they can be included in F4PRES.

If you want to link RSX–11M–PLUS and Micro/RSX utilities to a non-memory-resident FCSRES, you must create and edit the TKB CMD and ODL files and task build the utilities manually.

No memory management support is required for a non-memory-resident FCSRES.

5.3.1.3 Using FCSRES and FCSFSL

FCSRES and FCSFSL are merged into a single vectored-entry memory image that can be used in either user- or supervisor-mode libraries. The [1,1]FCSRES.TSK image file installs under the library name FCSRES and is used to satisfy both user and supervisor library requests.

Although the build commands are LIBR=FCSRES and SUPLIB=FCSFSL, respectively, TKB forces references to FCSFSL to become references to FCSRES in order to use the same library image.

Tasks built prior to the merging of FCSRES and FCSFSL that reference FCSFSL can continue to use the previous version of FCSFSL until they are rebuilt. However, they cannot use the new image file supplied under the name [1,1]FCSFSL.TSK because this image does not contain executable code. You do not need to relink tasks built prior to the merging of FCSRES and FCSFSL if your library has vectored entry points and you install the task using INS LB:[1,1]FCSRES/task=FCSFSL.

The possible FCSRES and FCSFSL references and the results are as follows:

TKB Option	Library Specification	Result
LIBR=	FCSRES	Memory-resident overlaid, PIC, uses one APR
LIBR=	FCSFSL	User-mode nonoverlaid FCSRES, PIC, requires two APRs
SUPLIB=	FCSRES	Supervisor-mode overlaid, PIC, one supervisor-mode APR
SUPLIB=	FCSFSL	Supervisor-mode FCSRES usage, PIC, two supervisor-mode APRs

5.3.1.4 Building F4PRES

Building an optimal F4PRES depends on the specific F4P OTS routines that your task uses, their need for virtual address space, and the available physical memory on your system.

You must decide which F4P OTS routines are used frequently enough by your task to warrant their presence in F4PRES. Routines are included or excluded by editing F4PRES.MAC.

The key factor is often the number of APRs used to map to the resident libraries. For example, you may have an important privileged application task that has only one APR available. In this case, if you construct an F4PRES library that uses two APRs and is clustered with FCSRES, for a total of two APRs, it may be best to edit more routines out of F4PRES to trim it to one APR.

The key to building F4PRES usable with the memory-resident FCSRES is that no FCS code is present in F4PRES, but all subroutine calls to FCS in F4PRES are resolved when F4PRES is built. This scheme involves revectoring the FCS calls through the application task image.

5.3.1.5 Options and Trade-Offs

There are a number of ways to link application tasks with resident libraries.

The following cases assume a minimally sized F4PRES; 4096_{10} words mapped by one APR if FCS is not contained in it, and 8192_{10} words mapped by two APRs if FCS is contained in it. These numbers will vary according to the F4P OTS routines that you include in F4PRES. It may not be possible for you to construct a useful F4PRES of one or both of these sizes; yours may use two APRs without FCS or three APRs with FCS.

In the following cases, the "virtual and physical memory" descriptions are always relative to a task with no overlays or resident libraries. Your task's disk- or memory-resident overlays may add overlay run-time routines, autoload vectors, and segment and region descriptors to your task.

The cases are as follows:

- Case 1—Linking to F4PRES with revectored FCS calls and memory-resident FCSRES used as a cluster library. Link the application task with the following TKB option:

  ```
  CLSTR=F4PRES,FCSRES:RO
  ```

  This case uses a total of one APR, making available maximum virtual address space in the application task. This is most appropriate for tasks that can take advantage of the increased virtual address space. On a system with the memory-resident FCSRES, F4P application tasks that do not profit from the increased address space should be built according to Case 2, which has two LIBR= TKB options. MACRO–11 application tasks can be built with one LIBR=FCSRES:RO TKB option (see Case 3 for the memory characteristics in this case).

  If FCS routines are called from the task image, the calls are resolved to entry points in FCSRES. (FCS routines might be called by either OTS code in the task image or by your task's MACRO–11 subroutines.)

  Virtual and physical memory: Case 1 requires one APR. The application task root incurs a load of 1250_{10} bytes; 32 bytes from FCSJMP.OBJ, 650 bytes from FCSRES.STB (autoload vectors, segment and region descriptors), and 568 bytes from the overlay run-time routines.

  Execution speed: Some execution time is consumed when the overlay run-time code in the task image must change the APRs from one resident library to another.

- Case 2—Linking to F4PRES with revectored FCS calls and to a memory-resident FCSRES not used as a clustered library. Link the application task with the following TKB options:

  ```
  LIBR=FCSRES:RO
  LIBR=F4PRES:RO
  ```

  This case uses two APRs for the resident libraries, but there is less overhead than with a cluster of libraries as in Case 1. Case 2 is best for tasks that cannot profit by using the extra APR that a cluster could make available. If FCSRES is predominantly being used in this way (little use of FCSRES linked to RSX utilities or MACRO–11 application tasks, and no clustered FCSRES and F4PRES), you should also consider Case 4, where a two-APR F4PRES contains FCS (with no cost in autoload vectors), FCSJMP, or overlay run-time routines for FCSRES. Other tasks can reference the resident libraries with one or more TKB LIBR options or with CLSTR.

If FCS routines are called from the task image, the calls are resolved to entry points in FCSRES. (FCS routines might be called by either the OTS code in the task image or by your task's MACRO–11 subroutines.)

Virtual and physical memory: Case 2 requires two APRs. The application task root incurs a load of 1164_{10} bytes; 32 bytes from FCSJMP.OBJ, 650 bytes from FCSRES.STB (autoload vectors, segment and region descriptors), and 482 bytes from the overlay run-time routines.

Execution speed: Some execution time is consumed when the overlay run-time code in the task image must change the mapping of the APR for FCSRES from one overlay to another, but less time is used than with a cluster.

- Case 3—Linking to a memory-resident FCSRES and having the OTS code present in your task's image. Link the application task with the following TKB option:

  ```
  LIBR=FCSRES:RO
  ```

  This case is appropriate when FCSRES is necessary, but you cannot justify having a permanent F4PRES on your system.

  Virtual and physical memory: Case 3 requires one APR for FCSRES. The OTS code, which may be thousands of bytes, is included in the application task image. (The OTS code should be overlaid.) The task root also incurs a load of 1132 bytes: 650 bytes from FCSRES.STB (autoload vectors, segment and region descriptors) and 482 bytes from the overlay run-time routines.

  Execution speed: Some execution time is consumed when the overlay run-time code in the task image must change the mapping of the APRs of resident libraries from one library to another. More execution time is used if you overlay the OTS code in the task image.

- Case 4—Linking to F4PRES with revectored FCS so that FCS code is present in your task's image. This combination is never the best choice because F4PRES and your task will include FCSJMP and FCSVEC with no benefit. However, tasks will link and execute correctly. Link the application task with the following TKB option:

  ```
  LIBR=F4PRES:RO
  ```

- Case 5—Linking to an F4PRES that contains FCS. You can link the task with the following TKB option:

  ```
  LIBR=F4PRES:RO
  ```

  This case is appropriate when F4PRES is necessary, you do not need a permanent FCSRES on your system, and no critical application tasks would profit from the increased address space of a clustered FCSRES and F4PRES.

  The program combination for this case tends to contain more OTS code in the same number of APRs than Case 6 because only the FCS used by F4PRES is present, leaving more room for OTS code.

  If your task contains macro subroutines that use FCS, try to use the FCS routines already contained in F4PRES (for example, OFNB$, OFID$, or DELET$). Otherwise, the task will contain large amounts of FCS code.

  Virtual and physical memory: Case 5 requires two APRs. There is no overlay overhead due to the resident library.

Execution speed: There is no overlay overhead due to the resident library.

- Case 6—Linking F4PRES to a non-memory-resident FCSRES (LIBR=FCSRES:RO in the F4PRES TKB command file). Link the task to F4PRES with the following TKB option:

  ```
  LIBR=F4PRES:RO
  ```

  If you have a non-memory-resident FCSRES on your system, Case 6 may be appropriate. Note that Case 5 tends to include more OTS code in the same number of APRs.

  If FCS routines are called from the task image, a space problem can occur. (FCS routines can be called by either OTS code in the task image or by your MACRO-11 subroutines.) FCSRES entry points are available only to a task or resident library linked directly to FCSRES; they are not available to a task linked to F4PRES in this case. Thus, any FCS routine called in your task will bring a number of FCS modules into the task image.

  Virtual and physical memory: Case 6 requires two APRs. There is no overlay overhead due to the resident library.

  Execution speed: There is no overlay overhead due to the resident library.

- Case 7—Linking to a non-memory-resident FCSRES with the OTS code in the task image. Link the application task with the following TKB option:

  ```
  LIBR=FCSRES:RO
  ```

  Case 7 is appropriate when the non-memory-resident FCSRES is necessary, but you cannot justify having a permanent F4PRES on your system.

  Virtual and physical memory: Case 7 requires one APR for FCSRES. The OTS code, which could be thousands of bytes, is included in the task image. The OTS code should be overlaid.

  Execution speed: There is no overlay overhead due to the resident library.

5.4 Virtual Program Sections

A virtual program section is a special TKB 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 TKB through the VSECT option and in FORTRAN through a set of FORTRAN-callable subroutines that issue the necessary mapping directives at run time. With the TKB 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 TKB 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 run time, the physical memory that the program section will be mapped into at run time. TKB 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 run time through the Create Region (CRRG$) 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 TKB attaches the virtual attribute to a relocatable program section you have specified in the VSECT option only if the section is defined in the root segment of your task through either a FORTRAN COMMON or a MACRO-11 .PSECT statement. The relocatable program section with the virtual attribute in the root does not use address space in your task; using this procedure merely assigns an address, window size, and physical length to a region yet to be mapped at run time by your task. For example:

```
VSECT=MARRAY:160000:20000:2000 [RET]
```

In this example, virtual program section MARRAY is allocated with a window size of 4K words (20000_8 bytes) and a base virtual address of 160000. In physical memory, 32K words are reserved for mapping the section at run time.

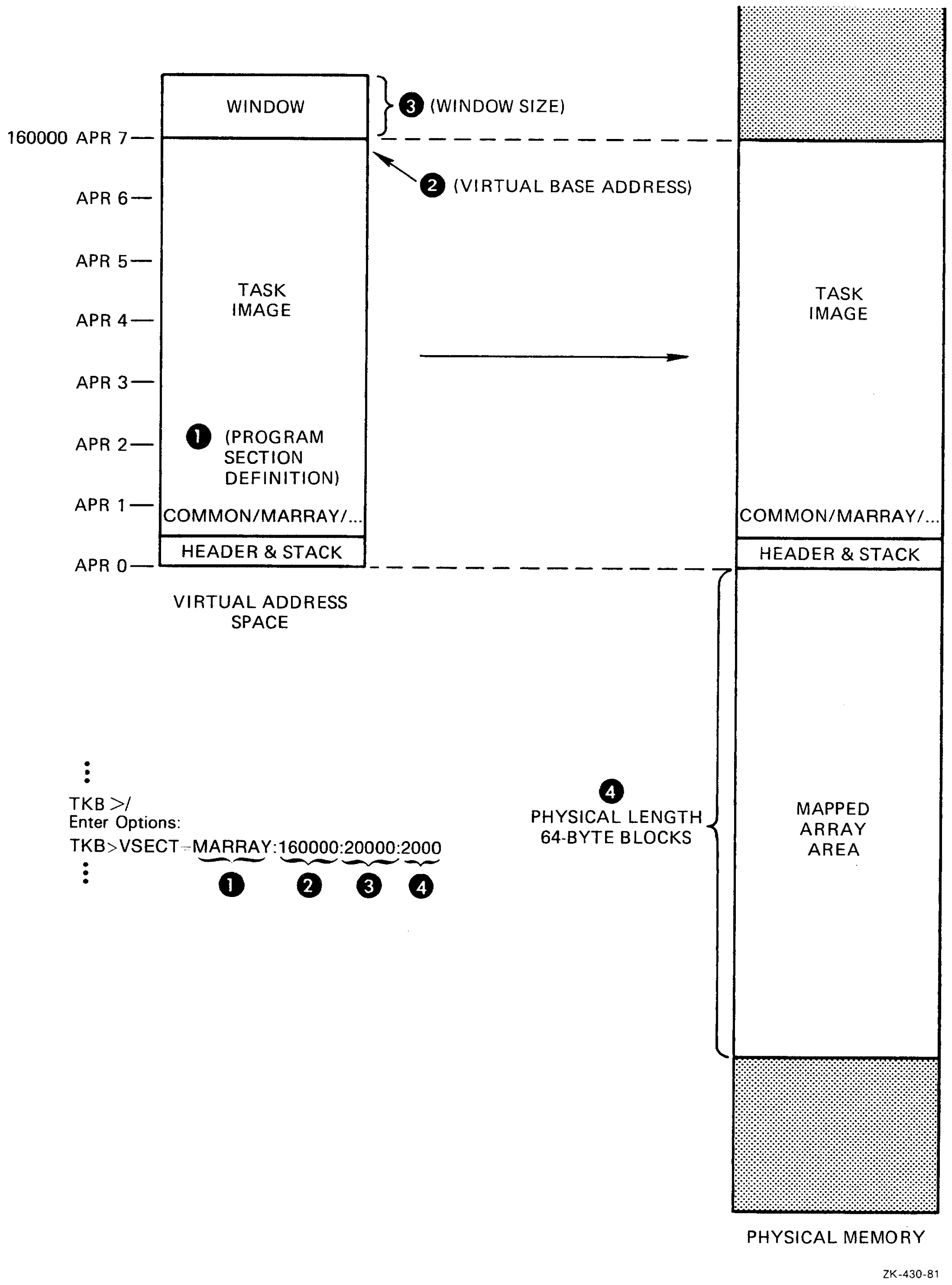
Assume that the program is written in FORTRAN and includes the following statement:

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

This statement generates a program section to which TKB attaches the virtual attribute. However, this program section is not a FORTRAN virtual array. A reference to the first element of the section, ARRAY(1), is translated by TKB to the virtual address 160000.

Figure 5-15 shows the effect of this use of the VSECT option.

Figure 5–15: VSECT Option Usage



```
   :
TKB>/
Enter Options:
TKB>VSECT=MARRAY:160000:20000:2000
   :
```

$$\text{TKB>VSECT=}\underbrace{\text{MARRAY}}_{1}:\underbrace{160000}_{2}:\underbrace{20000}_{3}:\underbrace{2000}_{4}$$

As mentioned previously, TKB 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 contains 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 wraparound 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.

5.4.1 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])
```

Parameters

ireg

A one-dimensional integer array that is nine 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 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 0 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 the region ID and size. The subroutine does not refer to elements 3 through 8, but rather the caller must set them up as required by the applicable system directives. For a detailed description of these parameters, refer to the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual.*

iwnd

A one-dimensional array that is 11 words long. The first eight 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.

You must set up iwnd(10) before calling ALSCT.

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-PLUS and Micro/RSX Executive Reference Manual* for a detailed description of these parameters.

ists

An area that 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)
```

Parameters

ireg

A one-dimensional integer array that is nine 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:

1. You allocate storage in the program for one window descriptor for each virtual program section and for a single region descriptor.
2. Your task calls the subroutine ALSCT to reserve the physical memory to which the virtual program section will be mapped.
3. 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).
4. When the space is no longer required, the task unmaps it and releases it with a call to RLSCT.

5.4.2 Building a Program That Uses a Virtual Program Section

Example 5-5, Part 1 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, as follows:

```
SUBSCRIPT?
```

When you input a subscript, it responds with ELEMENT= and the contents of the array element for the subscript you typed. VSECT continues to prompt until you press CTRL/Z. Upon receiving a CTRL/Z, VSECT exits.

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

```
TKB>VSECT,VSECT/-SP=VSECT,LB:[1,1]F770TS/LB  RET
TKB>/  RET
Enter Options:
TKB>WNDWS=1  RET
TKB>VSECT=MARRAY:160000:20000:200  RET
TKB>//  RET
>
```

Or, if you use LINK, use the following command sequence:

```
$ LINK/TAS/MAP:VSECT/NOPRINT/OPT VSECT,LB:[1,1]F770TS/LIB  RET
Option? WNDWS=1  RET
Option? VSECT=MARRAY:160000:20000:200  RET
Option?  RET
$
```

This command sequence directs TKB to create a task image file named VSECT.TSK and a short (by default) map file named VSECT.MAP. Because /-SP is appended to the map file in TKB, or /NOPRINT is specified in LINK, TKB does not spool the map to the line printer.

The library switch (/LB; /LIB in LINK) specifies that TKB 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, TKB extracts from the library and includes in the task image any modules in which references are defined.

The WNDWS option directs TKB to add a window block to the header in the task image. The Executive initializes this window block when it processes the mapping directives within the task.

The VSECT option directs TKB to establish for the program section named MARRAY 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 TKB 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, qualifiers, and options used in this example, refer to Chapters 10, 11, and 12, respectively.)

The map that results from this command sequence is shown in Example 5-5, Part 2.

Example 5–5: Part 1, Source Listing for VSECT.FTN

```
C
C       VSECT.FTN
C
        INTEGER *2 SUB,IRDB(9),IWDB(11),DSW
        INTEGER *2 IARRAY(4096)
        COMMON /MARRAY/IARRAY
        IWDB (1) = "3400                !USE APR 7 FOR WINDOW
        IWDB (3) = 128                  !WINDOW SIZE = 128*32 WORDS = 4K
        IWDB (5) = 0                    !OFFSET
        IWDB (7) = "422                 !STATUS = WS.64B!WS.WRT!WS.UDS
        IWDB (10) = 128                 !SIZE TO ALLOCATE
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
```

(Continued on next page)

Example 5-5 (Cont.): Part 1, Source Listing for VSECT.FTN

```
300     WRITE (5,301)DSW
301     FORMAT (' ERROR FROM MAPPING. ERROR = ',I7)
1000    CALL EXIT
        END
```

Example 5-5: Part 2, Task Builder Map for VSECT.TSK

```
VSECT.TSK;1       Memory allocation map  TKB M43.00        Page 1
                        11-DEC-87   16:12


Partition Name : GEN
Identification : FORV02
Task  UIC      : [303,1]
Stack    limits: 000300 001277 001000 00512.
PRG xfr address: 016270
Total address windows: 2.
Mapped array area: 4096. words
Task  image  size  : 8736. words
Task address limits: 000000 042043
R-W disk blk limits: 000002 000044 000043 00035.

*** Root segment: VSECT

R/W mem  limits: 000000 042043 042044 17444.
Disk blk limits: 000002 000044 000043 00035.

Memory allocation synopsis:

Section                                        Title  Ident  File
-------                                        -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001300 001160 00624.
MARRAY:(RW,D,GBL,REL,OVR) 160000 020000 08192.
                          160000 020000 08192. .MAIN. FORV02 VSECT.OBJ;3
OTS$F :(RW,I,GBL,REL,CON) 002460 002332 01242.
                          002460 000406 00262. $CONVI F40003 FOROTS.OLB;2
                          003066 001724 00980. $FIO   F40006 FOROTS.OLB;2
OTS$I :(RW,I,LCL,REL,CON) 005012 011220 04752.
       .
       .
       .
Global symbols:

ADI$IA 005032-R  CAL$  005140-R  ICI$   022466-R  MOI$PS 006050-R
       .
       .
       .

*** Task builder statistics:

    Total work file references: 27855.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 7086. words (27. PAGES)
    Size of work file: 4325. words (17. PAGES)

    Elapsed time:00:00:29
```

Chapter 6
Privileged Tasks

This chapter discusses privileged tasks: what they are, their possible hazards, how they are mapped, and an example of their use.

6.1 Privileged and Nonprivileged Task Distinction

RSX-11M-PLUS and Micro/RSX systems have two classes of tasks: privileged and nonprivileged. The distinction between privileged and nonprivileged tasks is primarily based upon system-access capabilities.

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, whereas a nonprivileged task cannot.

6.2 Privileged Task Hazards

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 system data structures. 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.

Make certain that your privileged task has completed its operation when you log out of the system. Logging out does not abort privileged tasks as it does nonprivileged tasks because the privileged task may be in the process of changing the system database. Therefore, the task must be allowed to complete its processing. Also, if the privileged task is in system state, no other task can execute until the privileged task completes its processing while in system state. However, when the privileged task leaves system state, you can log out of the system, leaving the privileged task still in operation.

If a processor trap occurs in a privileged task while the task is in user state, the Executive aborts the task. If the processor trap occurs in the privileged task while the task is in system state, the system fails. However, even while in user state, the privileged task that is mapped to the Executive can cause a system failure by incorrectly changing system data. Please note that a privileged task in user state should not modify system data.

6.3 Specifying a Task as Privileged

In TKB, you designate a task as privileged with the /PR (privileged) TKB switch. In DCL, you use the /PRIVILEGED:n qualifier. The /PR switch is described in Chapter 10 and the /PRIVILEGED:n qualifier is described in Chapter 11. TKB allocates address space for a privileged task based on the memory management APR that you specify as an argument to this switch or qualifier. The argument is optional; the default is 5, but you can change it by modifying the TKBBLD.CMD file and rebuilding TKB. TKB accepts three arguments: 0, 4, and 5. Choosing which of these arguments to specify is based on the considerations described below.

6.4 Privileged Task Mapping

When you specify an argument of 0 on the switch or qualifier, your task is marked as privileged but not mapped to the Executive or I/O page. 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 TKB does not reserve an APR to map the I/O page.

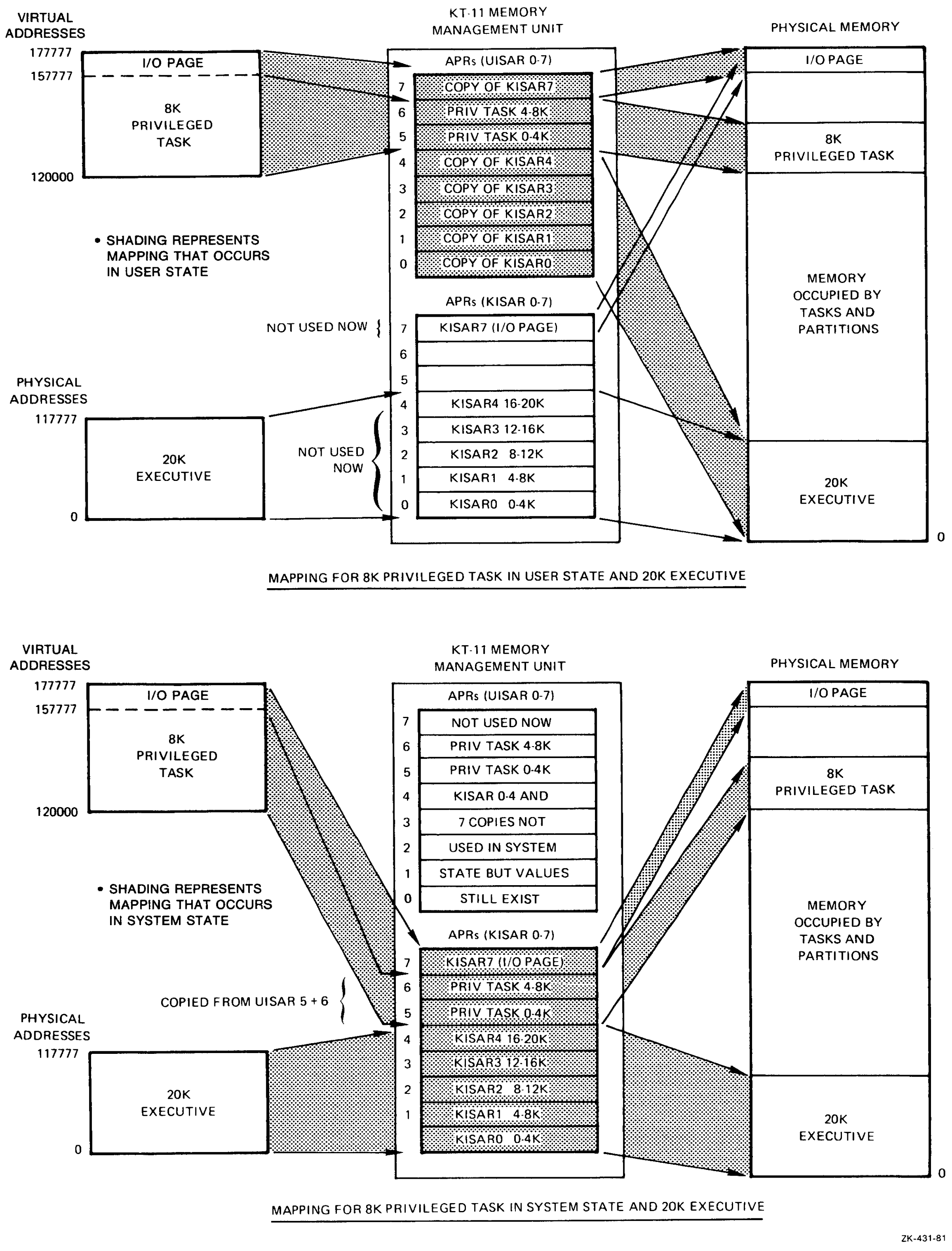
When you specify /PR:4 or /PR:5 in TKB, or /PRIV:4 or /PRIV:5 in LINK, TKB reserves APR 7 for mapping the I/O page. Moreover, TKB makes the Executive available to your task by reserving the APRs necessary to map the Executive into your task's virtual address space. Therefore, if your task requires access to the Executive, you must specify an argument of either 4 or 5. Five is the default.

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 may specify an argument of 4 or 5. The value specified depends on the task size. A privilege 4 task can be 12K words in size and map the I/O page. TKB 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. TKB applies a bias of 120000 (20K) to all addresses within your task.

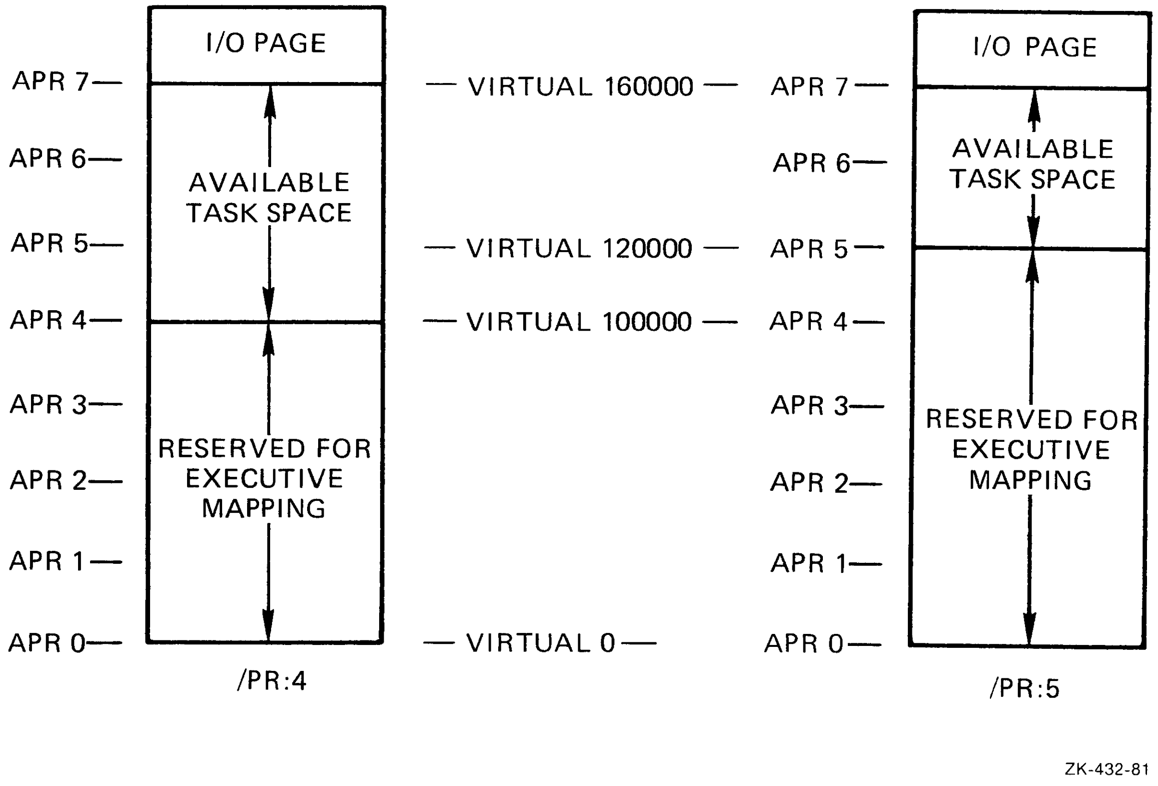
The mapping for privileged tasks is shown in Figure 6-1.

Figure 6–1: Privileged Task Mapping



The mapping for APR 4 and 5 is shown in Figure 6-2.

Figure 6-2: Mapping for /PR:4 and /PR:5



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, TKB uses 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.

When a task overlaps the I/O page, TKB does not generate an error message. Before TKB 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 INSTALL generate the following message:

```
INS--Warning--Privileged task not mapped to the I/O page
```

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

You can use the /-IP switch in TKB or the /NOIO_PAGE qualifier in LINK to inform TKB that the task is purposely over 12K and does not need to be mapped to the I/O page.

A task with a privilege of 4 or 5 can access all of the Executive, system control blocks, and I/O page. It can use Executive routines and do logical block I/O to a volume that is physically mounted on a device. Also, the task can issue a $SWSTK macro to change from user to system state. This allows the task to access the Executive or system data structures without interruption or fear of the data being changed while it is being accessed.

6.5 Privilege 0 Task

Using the /PR:0 switch in TKB or the /PRIV:0 qualifier in LINK causes TKB to build the task in the same way as any other task. Virtual address space begins at virtual address 0 and extends upwards as far as 32K minus 32 words. This task cannot access the Executive routines and system data structures or directly access the I/O page because the Task Builder has not reserved APRs for these purposes.

There are advantages to using a task with privilege 0 and having it mapped into APR 0. A task with privilege 0 can do the following:

- Bypass file protection.
- Use the Alter Priority (ALTP$) directive.
- Issue any directive that has a target task.
- Specify a device name in spawn directives.
- Write logical block I/O to a physically mounted volume, regardless of who issued the MOUNT or ALLOCATE command. For example, the VMR task is a task with privilege 0 and writes to mounted volumes during system generation. However, this advantage can be hazardous for obvious reasons.

A task with privilege 0 runs in user state and cannot switch to system state. Also, a task with privilege 0 is not mapped to the Executive. If you want to write a privileged task that does I/O processing, it is advantageous to use the /PR:0 switch in TKB or the /PRIV:0 qualifier in LINK for your task because there is less chance of corrupting the Executive or system code and data.

6.6 Privilege 4 Task

If you want your privileged task to map to the Executive and I/O page, and your Executive is 16K words or less, use the /PR:4 switch for TKB or the /PRIV:4 qualifier for LINK in the command line. If you specify privilege 4 for your task, TKB reserves APR 7 to map the I/O page and reserves APRs 0 through 3 to map the Executive as part of your task's virtual address space. The privilege 4 switch or qualifier can be used only if your Executive size is 16K words or less because the 16K-word Executive uses APRs 0 through 3 and your task is assigned mapping that starts with APR 4. Therefore, TKB applies a bias of 100000 (16K) to all virtual addresses within the task. This specific mapping of APRs 0 through 4 and 7 occurs whether the task is in user or system state.

Up to 12K words of virtual address space are possible in a privilege 4 task. The beginning of the task marks the end of the Executive code. If the task is 12K words in size, the end of the task marks the start of the I/O page. If the task is going to access the I/O page through APR 7, the task cannot exceed the 12K limit. If the task does exceed the limit, TKB is forced to assign APR 7 to the task code. When building the task, TKB does not give you an error message if your task exceeds the 12K limit. However, when you install the task, INSTALL sends you the following message:

```
INS--Warning--Privileged task not mapped to the I/O page
```

6.7 Privilege 5 Task

If you want your privileged task to map to the Executive and I/O page, and your Executive is between 16K and 20K words, use the /PR:5 switch for TKB or the /PRIV:5 qualifier for LINK in the command line. If you specify your task as privilege 5, TKB reserves APR 7 to map the I/O page and reserves APRs 0 through 4 to map the Executive as part of your task's virtual address space. The /PR:5 switch or /PRIV:5 qualifier can be used only if your Executive size is between 16K and 20K words because the 20K-word Executive uses APRs 0 through 4 and your task is assigned APR 5. (APR 5 may be used if the Executive is less than 16K words, but this wastes virtual address space.) Therefore, TKB applies a bias of 120000 (20K) to all virtual addresses within the task. This specific mapping of APRs 0 through 5 and 7 occurs whether the task is in user or system state.

Up to 8K words of virtual address space (12K if the I/O page is overmapped) are possible in a privilege 5 task. The beginning of the task marks the end of the Executive code. If the task is 8K words in size, the end of the task marks the start of the I/O page. If the task is going to access the I/O page through APR 7, the task cannot exceed the 8K-word limit. If the task does exceed the limit, TKB is forced to assign APR 7 to the task code. When building the task, TKB does not give you an error message if your task exceeds the 8K-word limit. However, when you install the task, INSTALL sends you the following message:

```
INS--Warning--Privileged task not mapped to the I/O page
```

Note

When you use a privileged task, the Executive has dedicated almost all the APRs to the necessary mapping for the Executive, the I/O page, and your task. Your task can issue memory management directives to remap any number of these APRs to regions. However, such remapping can cause obscure and difficult-to-find system problems. Also, note that when a directive unmaps a window that formerly mapped to the Executive or the I/O page, the Executive restores the former mapping.

6.8 Example 6-1: Building a Privileged Task to Examine Unit Control Blocks

The MACRO-11 source program PRIVEX.MAC in Example 6-1 illustrates one possible use of a privileged task.

Note

The nature of a privileged task is such that you must have a working knowledge of system concepts to understand its operation 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.

Example 6–1: Part 1, Source Code for PRIVEX

```
; MACRO  LIBRARY CALLS
        .TITLE PRIVEX
        .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 RADIX-50 WORD OF NAME OF ATTACHED TASK
;               FIRST RADIX-50 WORD OF NAME OF ATTACHED TASK
;
;               SECOND RADIX-50 WORD OF NAME OF ATTACHED TASK
;
;
;  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
```

(Continued on next page)

Example 6-1 (Cont.): Part 1, Source Code for PRIVEX

```
; 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     RO                  ;;FORM LOWEST UNIT NUMBER ON
        BISB    D.UNIT(R3),RO       ;; 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     RO,(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$:    INC     RO                  ;;INCREMENT UNIT NUMBER
        BR      20$                 ;;
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
FORMAT: TST     (R2)                ;ANY MORE INFORMATION IN BUFFER?
        BEQ     EXIT                ;IF EQ NO
        CMP     #-1,(R2)            ;OVERFLOWED BUFFER?
        BNE     40$                 ;IF NE NO
```

(Continued on next page)

Example 6-1 (Cont.): Part 1, Source Code for PRIVEX

```
        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
```

(Continued on next page)

Example 6–1 (Cont.): Part 1, Source Code for PRIVEX

```
;
; 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
```

If you assemble, build, and install PRIVEX into your system, it will scan the system device tables and examine the UCBs of all nonpseudo 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 *RSX-11M-PLUS and Micro/RSX System Library Routines Reference Manual.*

PRIVEX.MAC should be assembled with a command line similar to the following one in TKB:

```
MAC>PRIVEX,PRIVEX/-SP=DR0:[1,1]EXEMC/ML,[11,10]RSXMC/PA:1,DR2:[303,1]PRIVEX RET
```

If you use LINK, you may enter the following command line, which is similar to the one preceding:

```
$ MACRO/OBJ:PRIVEX/LIST:PRIVEX DR0:[1,1]EXEMC/LIB,- RET
->[11,10]RSXMC/PA:1,DR2:[303,1]PRIVEX RET
```

The file EXEMC is the Executive macro library and the file RSXMC is the Executive prefix file. The switches used in the command line are described in the *PDP-11 MACRO-11 Language Reference Manual.*

The TKB command sequence for PRIVEX is as follows:

```
>TKB RET
TKB>PRIVEX/PR:5,PRIVEX/-SP=PRIVEX RET
TKB>DR0:[3,54]RSX11M.STB,DR0:[1,1]EXELIB/LB RET
TKB>/ RET

Enter Options:
TKB>UNITS=1 RET                ;DEFINE NUMBER OF LUNS
TKB>GBLDEF=OUTLUN:1 RET        ;DEFINE LUN ON WHICH TO PRINT MESSAGES

TKB>ASG=TI0:1 RET              ;ASSIGN LUN TO DEVICE
TKB>// RET
>
```

If you use LINK, use the following command sequence to build PRIVEX:

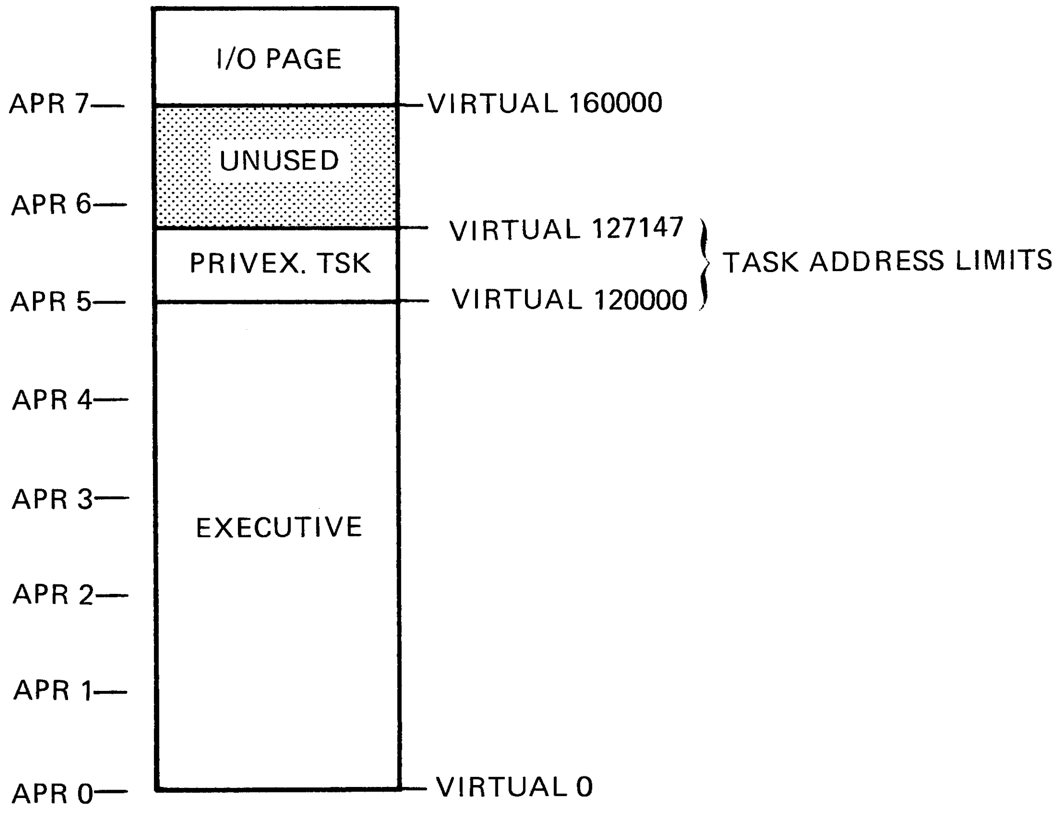
```
$ LINK/TAS/PRIV:5/MAP:PRIVEX/NOPRINT/OPT PRIVEX RET
Option? UNITS=1 RET                 ;DEFINE NUMBER OF LUNS

Option? GBLDEF=OUTLUN:1 RET         ;DEFINE LUN ON WHICH TO PRINT MESSAGES
Option? ASG=TI0:1 RET               ;ASSIGN LUN TO DEVICE
Option? RET
$
```

These command sequences direct TKB 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 in length. If the Executive in your system is 16K words or less, you can use assign privilege 4 when you build the task.

In the options sections of these command sequences, the UNITS=1 option specifies that PRIVEX will 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 TKB map for PRIVEX is shown in Example 6-1, Part 2. The "Global symbols" section has been shortened to save space. Note that the task's address limits begin at virtual address 120000. Figure 6-3 illustrates how TKB allocates virtual address space for the program.

Figure 6–3: Allocation of Virtual Address Space for PRIVEX



ZK-433-81

Example 6–2: Part 2, Task Builder Map for PRIVEX

```
PRIVEX.TSK;1   Memory allocation map TKB M43.00       Page 1
                        7-OCT-87   13:26


Partition name : GEN
Identification : 01
Task  UIC      : [303,1]
Stack limits: 120230 121227 001000 00512.
PRG xfr address: 124610
Task attributes: PR
Total address windows: 1.
Task  image  size  :  1920. words
Task address limits: 120000 127323
R-W disk blk limits: 000002 000011 000010 00008.

*** Root segment:PRIVEX

R/W mem limits:  120000 127323 007324 03796.
Disk blk limits: 000002 000011 000010 00008.

Memory allocation synopsis:

Section                                          Title    Ident    File
-------                                          -----    -----    ----
. BLK.:(RW,I,LCL,REL,CON) 121230 005746 03046.
                          121230 003656 01966.  PRIVEX 01          PRIVEX.OBJ;2
$$RESL:(RO,I,LCL,REL,CON) 127176 000124 00084.

Global symbols:

AS.DEL 000001    BT.UAB 000002    DV.SDI 000020    D.RS81 177657
D.VOUT 000004    F.NWAC 000034    IE.DAA 177770

AS.EXT 000004    B.DIR  000026    DV.SQD 000040    D.RS83 177655
D.VPWF 000006    F.SCHA 000015    IE.DNA 177771
                           .
                           .
                           .
$PDVTA 020000    $REMOV 054044    $SGFFR 020652    $TICLR 041032
$YHCTB 022674    .TT14  023770


*** Task builder statistics:
    Total work file references: 250535.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 13486. words (52. PAGES)
    Size of work file: 12032. words (47. PAGES)
    Elapsed time:00:00:51
```

6.9 Privileged Tasks in an I- and D-Space System

The following text describes the available privilege and mapping for privileged tasks in I- and D-space RSX–11M–PLUS and Micro/RSX systems.

6.9.1 Privilege Available to Privileged Tasks in an I- and D-Space System

A privileged task in an instruction- and data-space system, which is a system with an I- and D-space Executive, may have either conventional mapping or I- and D-space mapping. That is, it may map its own instructions and data with I-space APRs, or map instructions with I-space APRs and data with D-space APRs. Regardless of its mapping, either kind of privileged task in an I- and D-space system may have either privilege 0 or privilege 5. That is, a conventional privileged task or a privileged task that uses I- and D-space may be built with either the /PR:0 or /PR:5 TKB switches, or the /PRIVILEGED:0 or /PRIVILEGED:5 LINK qualifiers. A privilege 0 task in this system has the same abilities as a privilege 0 task in other RSX systems. The same is true for a privilege 5 task in this system as compared to other RSX systems.

A privilege 4 task (one built with the /PR:4 switch or /PRIVILEGED:4 qualifier) is not available in an I- and D-space system. The reason is that the Executive in this system occupies 20K of virtual instruction space and 20K of virtual data space. Therefore, this Executive uses instruction APRs 0 through 4 to map its instructions and data APRs 0 through 4 to map its data. The Executive's data APR 0 and instruction APR 0 are overmapped and map the first 4K of the Executive. Executive APRs 0 through 4 are copied into the privileged task's APRs 0 through 4 in different combinations (see Section 6.9.2, Privileged Task Mapping in an I- and D-Space System). Therefore, the privileged task can use only its own APRs 5 through 7 to map its instructions, data, and the I/O page. Hence, a privileged task in this system must be built with the /PR:5 switch or /PRIVILEGED:5 qualifier.

The following text describes the mapping available to a privileged I- and D-space task.

6.9.2 Privileged Task Mapping in an I- and D-Space System

Table 6-1 shows the APR mapping used by a conventional privileged task in an I- and D-space system.

Table 6–1: Conventional Privileged Task Mapping in an I- and D-Space System

User IAPRs	Use
0 through 4	Contain the same contents as those of the Executive's I-space APRs. The privileged task may use Executive routines, but must not do so without first issuing the SWSTK$ directive (go to system state) to linearize access to the Executive's routines and data.
5 through 6	Contain offsets that map the user task's instructions and data, combined.
7	Contains an offset to map the I/O page or the user task's instructions and data if the task is large enough.

An I- and D-space privileged task uses the APR mapping in an I- and D-space system as shown in Table 6-2.

Table 6–2: I- and D-Space Privileged Task Mapping in an I- and D-Space System

User IAPRs	Use
0 through 4	Contain the same contents as those of the Executive's D-space APRs. The privileged task may read the Executive's data space or use Executive routines, but must not do so without first issuing the SWSTK$ directive (go to system state) to linearize access to the Executive's routines and data.
5 through 6	Contain offsets to map the privileged task's instructions.
7	Contains an offset that maps the privileged task's instructions if the privileged task is large enough.

User DAPRs	Use
0 through 4	Contain the same contents as those of the Executive's D-space APRs. The privileged task may read the Executive's data space or use Executive routines, but must not do so without first issuing the SWSTK$ directive (go to system state) to linearize access to the Executive's routines and data.
5 through 6	Contain offsets that map the privileged task's data.
7	Contains an offset that maps the privileged task's data (if the task is large enough) or the I/O page data.

Chapter 7

User-Mode I- and D-Space

This chapter discusses the Task Builder's ability to divide a user task into instruction and data space (I- and D-space). A series of figures and text explains task mapping and the use of task windows in RSX-11M-PLUS and Micro/RSX systems with an I- and D-space task. In the text, comparisons are made between conventional tasks and I- and D-space tasks. A conventional task is one that does not separately map instruction space and data space.

The I- and D-space feature is an RSX-11M-PLUS system-generation option, and is also available on Micro/RSX systems where the hardware supports separate I- and D-space. The feature is available only on specific processor hardware. Conventional tasks can be run in an I- and D-space system, but an I- and D-space task cannot run in a system that does not have the option included.

7.1 User-Task Data Space Defined

User-task data space contains data. The user task accesses the data space through D-space APRs. The function of I- and D-space allows a total of 16 APRs to map your task: 8 APRs for data space and 8 APRs for instruction space. If your task uses both I- and D-space to its maximum capacity, it can contain 64K words of virtual address space. In addition to both I- and D-space, if your task links to a 32K-word supervisor-mode library, it can contain 96K words of virtual address space.

To separate the data and instructions, your task can use program sections to contain the data or instructions. Also, your task can use the Executive directives CRAW$ and CRRG$ to dynamically create and map to data-space regions. See the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual* for the use of these directives.

Conventional tasks and tasks that separate instruction space and data space differ in only a few areas of interest. The next sections discuss these areas.

7.2 I- and D-Space Task Identification

Two fields denote an I- and D-space task. In the task header, the byte that has the offset H.DMAP identifies the task D-space mapping mask. In the Task Control Block (TCB), the T4.DSP bit in the fourth task status word identifies the I- and D-space task to the system.

The system task loader or the VMR FIX command initializes these two fields at the time the task is loaded. Therefore, tasks built on a system other than an I- and D-space system may be run without rebuilding on an RSX-11M-PLUS or Micro/RSX system that supports I- and D-space.

The I- and D-space task is one in which TKB separates the data areas and instructions. In this task, data areas should be defined by the MACRO-11 .PSECT directive that has the D attribute. Similarly, the .PSECT directive with the I attribute defines instruction areas.

7.3 Comparison of Conventional Tasks and I- and D-Space Tasks

A conventional task operating in user mode can contain 32K words of virtual address space and access approximately 32K words of physical memory. However, a task using both I- and D-space APRs can contain 64K words of virtual address space and access approximately 64K words of memory.

The conventional task in an I- and D-space system uses both sets of APRs. However, the relocation addresses in both I-space and D-space APRs are identical. Also, the task windows refer to I-space APRs in a task that does not use D-space.

An I- and D-space task can use separately both I- and D-space APRs; that is, APRs used in this way are not overmapped. Because of this, the task can use eight D-space APRs to access and use data, and eight I-space APRs to access and execute instructions. Using 16 APRs allows the I- and D-space task to access a total of 64K words of physical memory at one time.

Table 7-1 contains a brief mapping summary of the combinations of I- and D-space tasks, I- and D-space systems, and the APR mapping that occurs.

Table 7–1: Mapping Comparison Summary

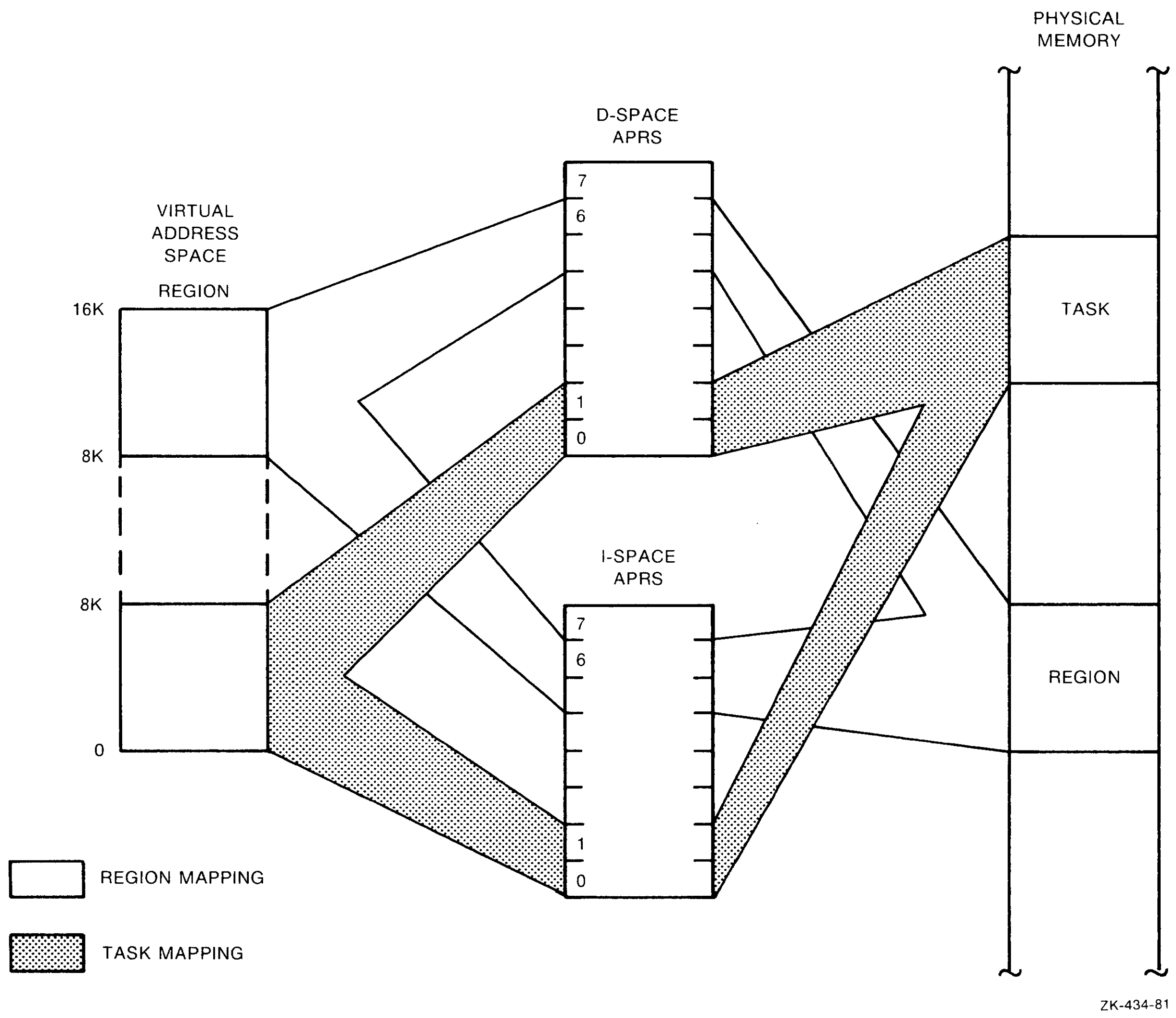
I/D Task	I/D System	Mapping Summary
No	Yes	I-space APRs and D-space APRs contain the same relocation addresses.
No	No	I-space APRs contain relocation.
Yes	Yes	I-space APRs map instruction space. D-space APRs map data space.
Yes	No	Not possible.

7.4 Conventional Task Mapping

Conventional tasks map their virtual addresses to their logical addresses through both I-space and D-space APRs. That is, TKB does not separate instruction space or data space nor does the system differentiate the spaces except by the logic inherent in the task. Therefore, the task must map to its logical address space by both sets of APRs, which are overmapped.

Figure 7-1 shows an 8K-word conventional task linked to an 8K-word region that maps to its logical address space through both D-space and I-space APRs in an I- and D-space system.

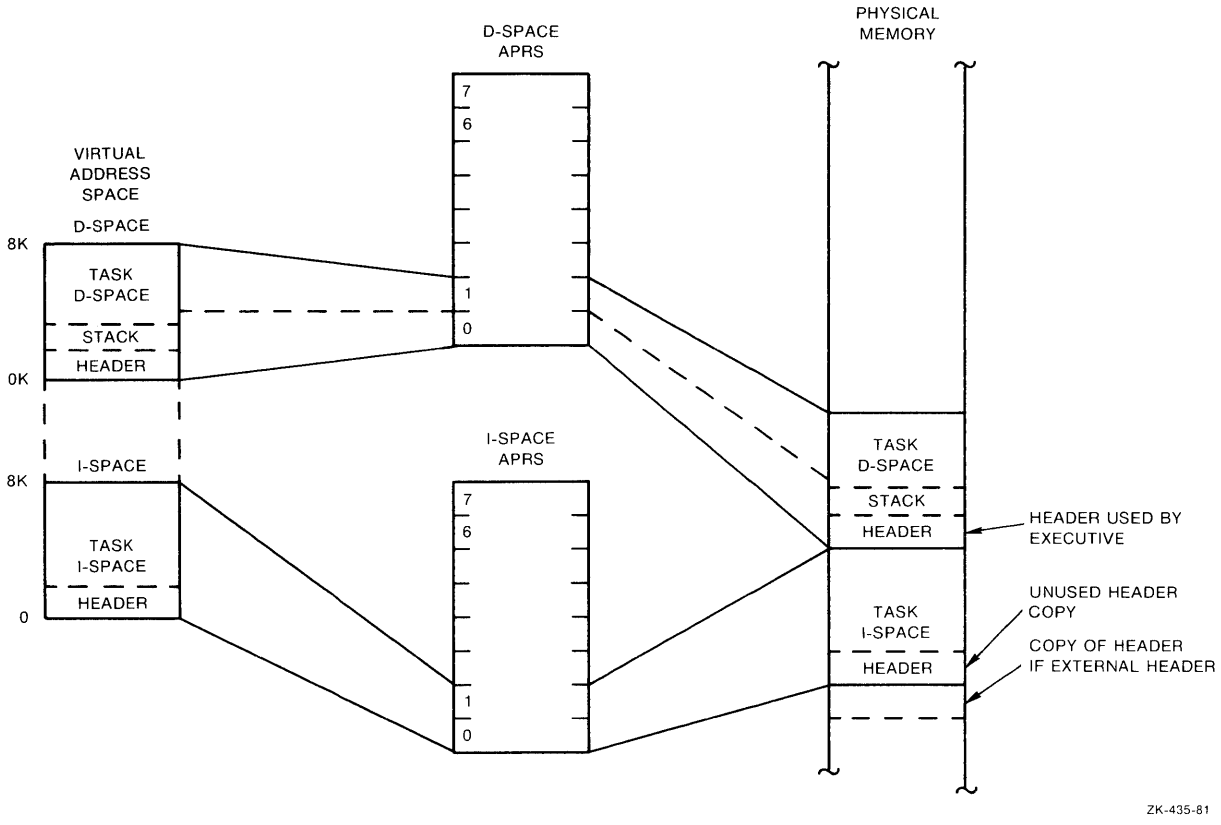
Figure 7–1: Conventional Task Linked to a Region in an I- and D-Space System



7.5 I- and D-Space Task Mapping

Figure 7-2 shows an 8K-word I- and D-space task. TKB separated the data and instructions in this task. Because of the way TKB processes task space, the task header must physically reside at the beginning of the task in I-space. TKB puts the header that the Executive uses for task control in D-space. Also, the task's stack is in D-space. If the task is to have an external header (under control of the /XH switch or /EXTERNAL qualifier), the Executive copies the header in D-space and puts it into the contiguous space immediately before the task's I-space in memory. For more details, see Figure B-4, Image on Disk of Overlaid I- and D-Space Task, in Appendix B.

Figure 7-2: I- and D-Space Task Mapping in an I- and D-Space System



The task shown uses two APRs because of its size (8K words). D-space APR 0 maps the task's header and stack and part of D-space.

7.6 Task Windows in I- and D-Space Tasks

TKB uses different windows to map various portions of an I- and D-space task. Window 0 in an I- and D-space task cannot be used because it maps the root in I-space. Similarly, you cannot use window 1 because it maps the D-space part of the root. The root of the task, which TKB divides into I- and D-space, therefore requires two windows. TKB reserves the use of these two windows. You can specify up to 14 windows for a task that uses I- and D-space.

7.7 Specifying Data Space in Your Task

You design an I- and D-space task by specifying data space separately from instruction space. Good programming practice suggests that all data areas and buffers should be located in adjacent locations. Similarly, all instructions should be located in adjacent locations. On its own, TKB will separate and cluster instruction and data space when it builds the task. For TKB to do this, however, you must use a method of informing it about which statements are data and which are instructions.

For the MACRO–11 programmer, the way to separate data and instructions is to use the MACRO–11 .PSECT directive. You can use this directive with the instruction (I) attribute for all the instruction locations in your task's code. Also, you can use .PSECT and the data (D) attribute for all the data locations. You must define a data .PSECT in an I- and D-space task even though no actual data is contained in the task. In this case, the .PSECT can be of zero length.

Note that I- and D-space libraries have not been defined and are not a possible configuration.

7.8 Overlaid I- and D-Space Tasks

Except for the mapping of an I- and D-space task and the location of instructions and data, the I- and D-space task differs little from a conventional task. However, there are structural differences between a nonoverlaid and an overlaid I- and D-space task. By comparing the two kinds of tasks, the figures and text in the following sections describe the nonoverlaid and overlaid I- and D-space tasks. Also, you may want to refer to the description of overlaid conventional tasks in Chapter 3.

Figure 7-3 shows a simplified disk image of the nonoverlaid I- and D-space task. This task contains four I-space program sections and four D-space program sections. TKB collects all the I-space program sections together in one part of the root and all the D-space program sections in another part of the root.

Figure 7–3: Simplified Disk Image of a Nonoverlaid I- and D-Space Task

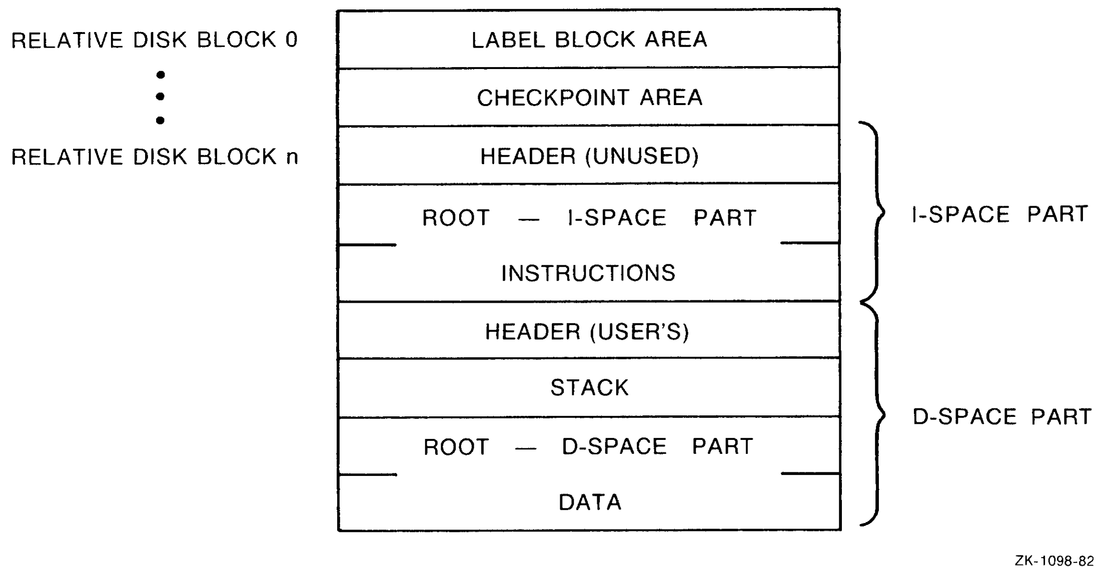
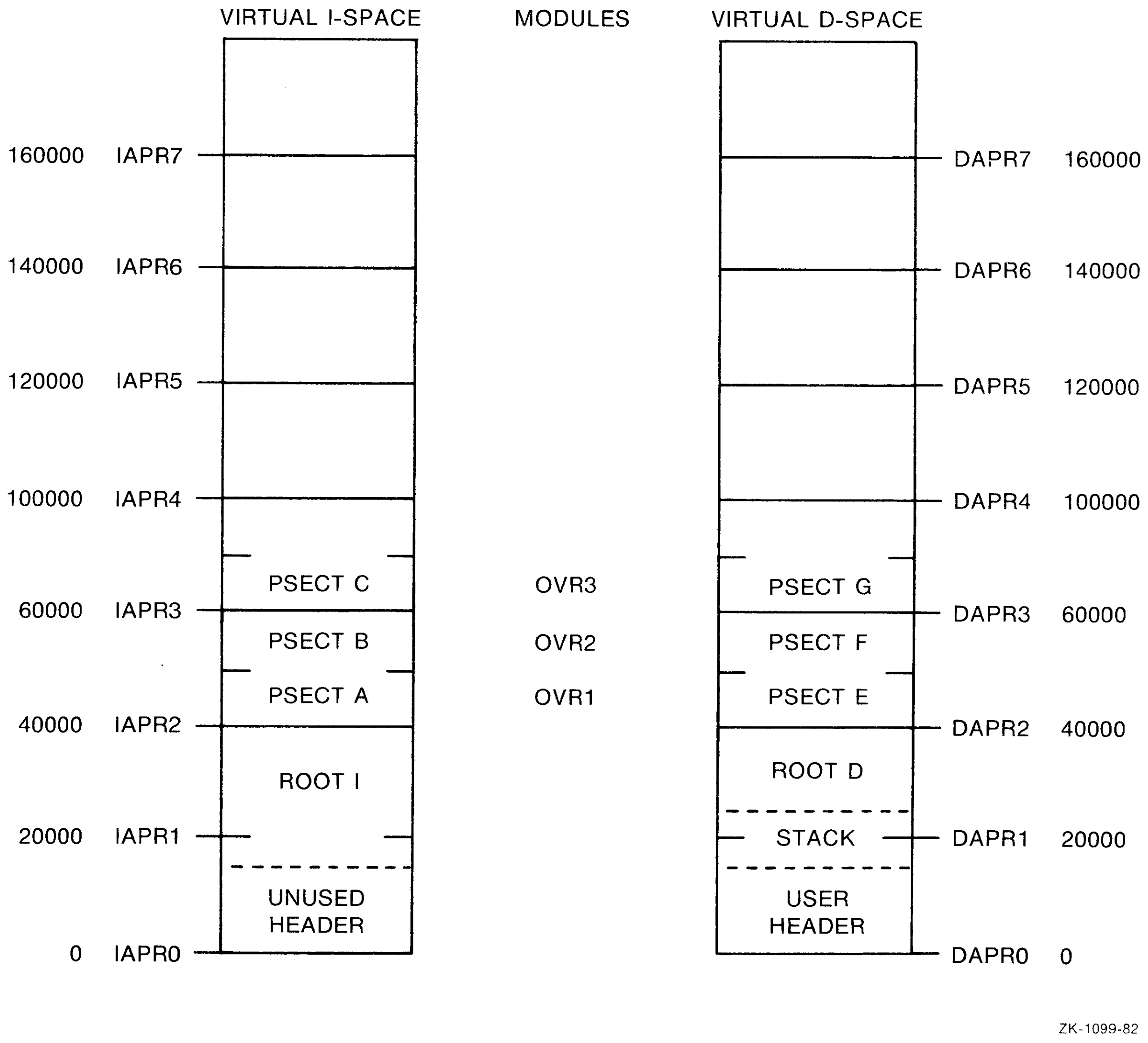


Figure 7-4 shows the virtual address space and physical memory occupied by an overlaid I- and D-space task called IAND. The task has a total physical size of 160000_8 bytes. (You may want to compare Figure 7-4, which is shown next, with Figure 3-1 in Chapter 3, which shows a conventional overlaid task.) The instructions and data occupy the same virtual address space and are of equal physical size; but because they are mapped through different APRs, they occupy different locations in physical memory. The instructions in IAND occupy four program sections that have the instruction (I) attribute, and the data in IAND occupy four program sections that have the data (D) attribute.

Figure 7–4: Overlaid I- and D-Space Task Virtual Address Space

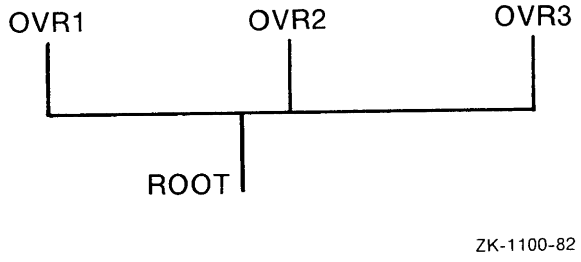


In Figure 7-4, the virtual instruction space contains program sections A, B, and C, which are those that contain instructions, and ROOT I, which is the program section in the root that contains instructions. TKB places the unused header in the I-space part of the root.

Also, in Figure 7-4, the virtual data space contains program sections D, E, and F, which are those that contain data, and ROOT D, which is the program section in the root that contains data. TKB places the task's user header in the D-space part of the root.

As an overlaid task, a possible overlay tree may look like the one shown in Figure 7-5.

Figure 7–5: Example Overlay Tree for Overlaid I- and D-Space Task IAND



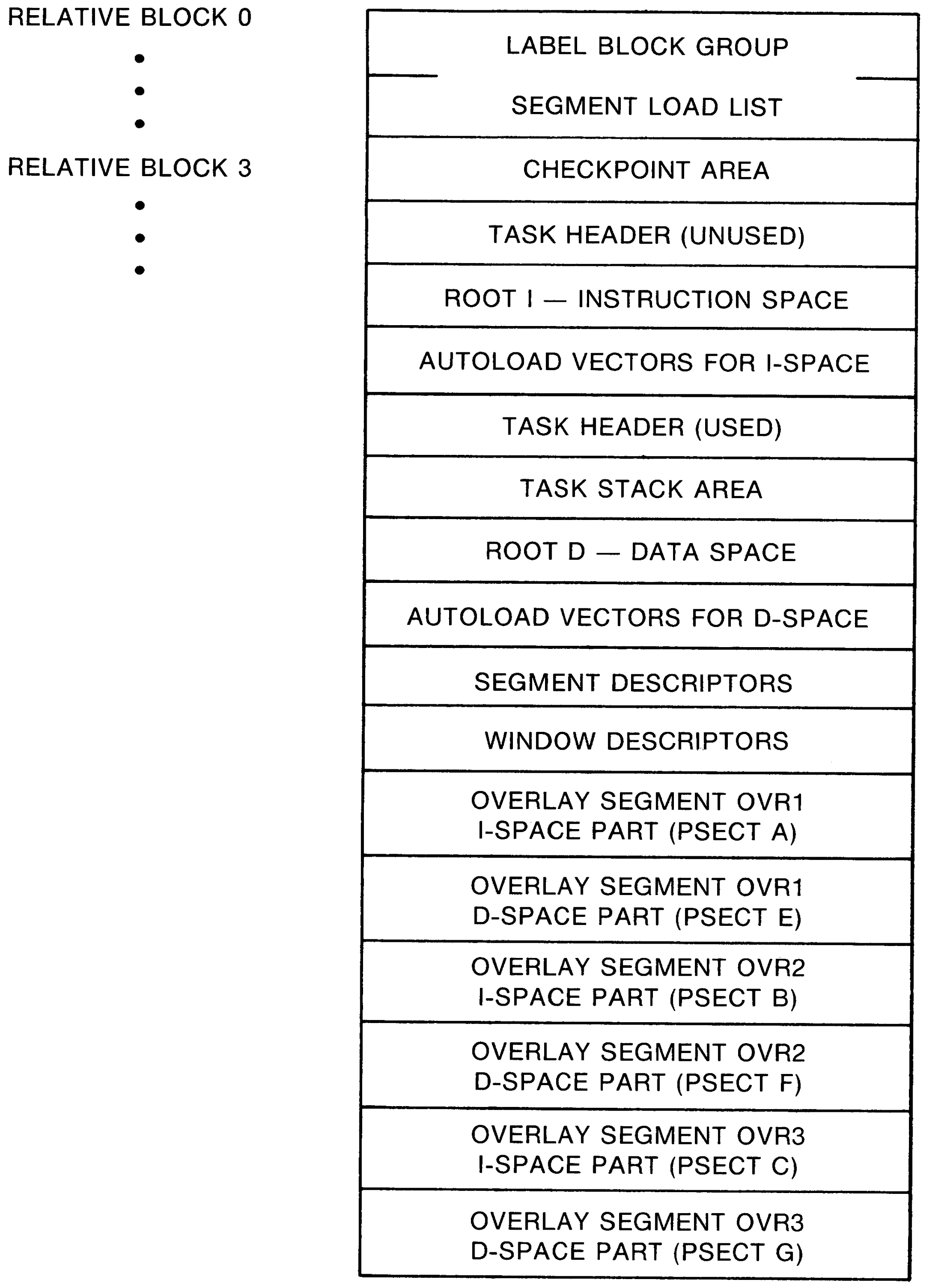
The accompanying ODL statement for this task is as follows:

```
.ROOT  ROOT-(OVR1,OVR2,OVR3)
```

Notice that this ODL statement is not different from any overlaid task with this tree structure. In this statement, the module OVR1 contains the instruction program section A and the data program section E, the module OVR2 contains the instruction program section B and the data program section F, and the module OVR3 contains the instruction program section C and the data program section G. Also, the ROOT module contains the instruction program section I and the data program section D.

The disk image of this overlaid task, shown in Figure 7-6, contains the instruction and data program sections in separate areas. Figure 7-6 also illustrates the difference between disk images of overlaid and nonoverlaid I- and D-space task disk images when you compare it with the disk image shown in Figure 7-3. Notice that TKB separates the segments of the overlaid IAND task into instruction parts and data parts. Any autoload vectors generated because of calls from these segments are also included in the segment area. The autoload vectors for I- and D-space tasks contain two parts: an I-space part and a D-space part. TKB places each part with its corresponding segment part as shown in Figure 7-6. Autoload vectors for I- and D-space tasks are discussed in detail in Chapter 4.

Figure 7–6: Simplified Disk Image of Overlaid I- and D-Space Task IAND



RELATIVE BLOCK 0
RELATIVE BLOCK 3
LABEL BLOCK GROUP
SEGMENT LOAD LIST
CHECKPOINT AREA
TASK HEADER (UNUSED)
ROOT I — INSTRUCTION SPACE
AUTOLOAD VECTORS FOR I-SPACE
TASK HEADER (USED)
TASK STACK AREA
ROOT D — DATA SPACE
AUTOLOAD VECTORS FOR D-SPACE
SEGMENT DESCRIPTORS
WINDOW DESCRIPTORS
OVERLAY SEGMENT OVR1 I-SPACE PART (PSECT A)
OVERLAY SEGMENT OVR1 D-SPACE PART (PSECT E)
OVERLAY SEGMENT OVR2 I-SPACE PART (PSECT B)
OVERLAY SEGMENT OVR2 D-SPACE PART (PSECT F)
OVERLAY SEGMENT OVR3 I-SPACE PART (PSECT C)
OVERLAY SEGMENT OVR3 D-SPACE PART (PSECT G)
ZK-1101-82

7.8.1 Autoload Vectors and STB Files

If your I- and D-space task links to an overlaid shared region, that region must have been built with a version of TKB that supports overlaid I- and D-space tasks. The reason for this is that the STB files for overlaid shared regions built by older versions of TKB do not contain the ISD records that are needed to create the type of autoload vectors that I- and D-space tasks use.

For newer versions of TKB that support overlaid I- and D-space tasks, TKB allocates autoloadable vectors in the root of the task only for those entry points in the library referenced by the task. To create the autoload vectors, TKB uses ISD records in the STB file when linking the task to the library if the ISD records are present. Therefore, tasks built with newer versions of TKB tend to be smaller because fewer autoload vectors are present.

For the Fast Task Builder (FTB) and older versions of TKB that do not support I- and D-space tasks, each autoload vector in the shared region's STB file is allocated in the root of the task being linked to the region, whether or not the entry point is referenced by the task.

For user-mode I- and D-space tasks, the autoload vectors consist of two parts:

- An I-space part consisting of four words contained in the program section $$ALVI
- A D-space part consisting of two words contained in the program section $$ALVD

Note

Libraries created with older versions of TKB do not have the ISD records in the STB file that newer versions of TKB use to include autoload vectors in the task from the STB file. Therefore, TKB must create autoload vectors for every entry point in the library.

If you are using one of these older libraries and you are linking an I- and D-space task to it, TKB will give you the following fatal error message:

```
Module module-name contains incompatible autoload vectors
```

This message occurs because the STB file contains conventional autoload vectors that are not usable by an I- and D-space task.

For more information about linking shared regions to I- and D-space tasks, see the section in Chapter 5 entitled Autoload Vectors and STB Files for Overlaid Shared Regions.

7.9 I- and D-Space Task Memory Allocation and Example of Maps

The following section discusses and shows the differences between two versions of a task that is built both as a conventional task and as an I- and D-space task. The conventional task is called MAIN.TSK, and the I- and D-space version of MAIN.TSK is called MAINID.TSK. Both of these tasks are similar to, but not the same as, the task called MAIN.TSK shown in Chapter 5. After MAIN.TSK was coded, built, and the map printed, MAIN.TSK was rebuilt as an I- and D-space task to create MAINID.TSK. To do this, the /ID switch was used in the TKB command line. (To do the same with DCL, use the /CODE:DATA qualifier in the LINK command line.) Both the conventional version and the I- and D-space version of this task are overlaid and link to a library.

Use the following TKB command sequence to build MAIN as an I- and D-space task:

```
>TKB [RET]
TKB>MAIN/ID,MAIN/MA/-SP/-WI=MAIN [RET]
TKB>/ [RET]
Enter Options:
TKB>RESLIB=LIB/RO:3 [RET]
TKB>// [RET]
>
```

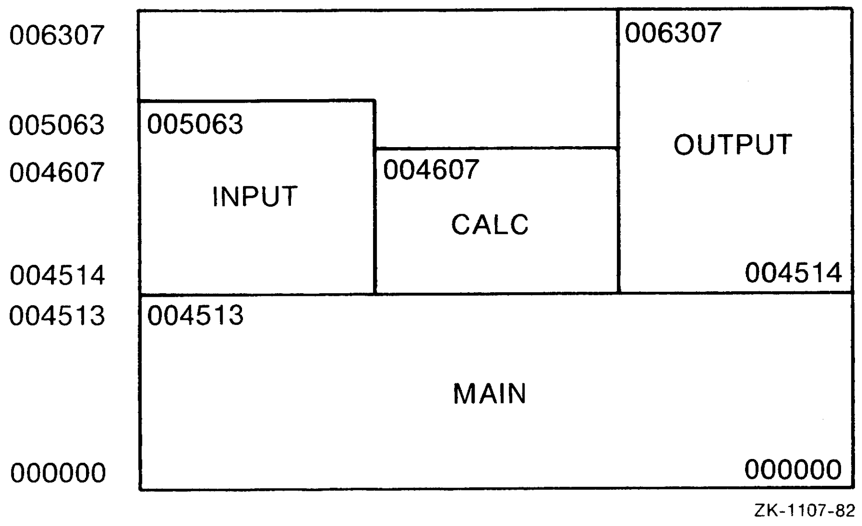
For LINK, use the following command sequence:

```
$ LINK/TAS/CODE:DATA/MAP:MAIN/NOPRINT/NOWIDE/SYS/OPT MAIN [RET]
Option? RESLIB=LIB/RO:3 [RET]
Option? [RET]
$
```

7.9.1 Virtual Memory Allocation for MAIN.TSK

MAIN.TSK has a root called MAIN and three overlay segments called INPUT, CALC, and OUTPUT. By comparing the map of this task in Example 7-1 and the memory allocation diagram in Figure 7-7, you will be able to determine the virtual memory space allocation and structure of this task. Note that the overlay segments occupy the same virtual address space, and the root and segments are mapped in both I-space and D-space.

Figure 7–7: Memory Allocation Diagram for MAIN.TSK



7.9.2 Virtual Memory Allocation for MAINID.TSK

MAINID.TSK has a root called MAIN and three overlay segments. In this way MAINID.TSK resembles MAIN.TSK. However, the instruction program sections and data program sections in MAINID.TSK are separated and they are mapped through their respective I-space or D-space APRs. Therefore, MAINID.TSK has two virtual address spaces: an I-space and a D-space. Figures 7-8 and 7-9 show the memory allocation for the I-space and D-space in MAINID.TSK.

Figure 7–8: Memory Allocation Diagram for MAINID.TSK I-Space

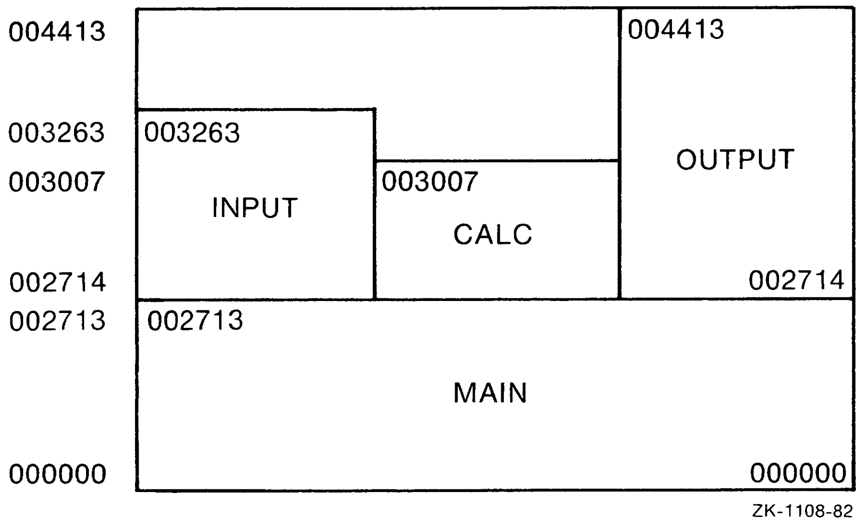
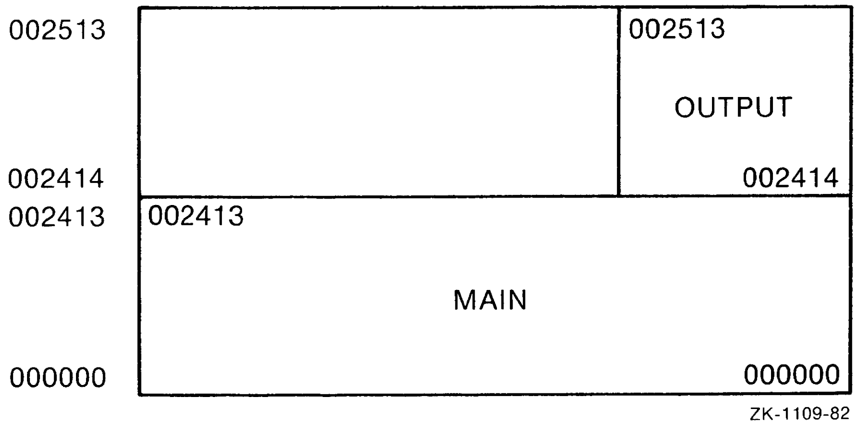


Figure 7–9: Memory Allocation Diagram for MAINID.TSK D-Space



The three segments INPUT, CALC, and OUTPUT occupy the same virtual I-space because these three segments contain instructions and, therefore, instruction program sections. However, the overlay segment OUTPUT is the only segment that occupies virtual D-space because the segments INPUT and CALC do not contain data or D-space program sections. Note that the two overlay segments INPUT and CALC have no D-space. You can see this in both Figure 7-9 and in the map in Example 7-2. The map in Example 7-2 shows the virtual address space allocation for both I-space and D-space.

An I- and D-space task uses more virtual memory space than a conventional task. The map in Example 7-2 shows that MAINID.TSK uses 1888_{10} words of space as opposed to the 1664_{10} words used by MAIN.TSK. The reasons for the increase in size of MAINID.TSK over MAIN.TSK are as follows:

- An I- and D-space task contains an unused task header.
- Autoload vectors in an I- and D-space task contain two more words than conventional autoload vectors. Program sections $$ALVD and $$ALVI in Example 7-2 contain the autoload vectors. You can see from this map that they use more space than the $$ALVC program section in MAIN.TSK, which contains conventional autoload vectors.
- The segment descriptors in an overlaid I- and D-space task contain an extension.

The size of the segment descriptor block that is internal to TKB depends on whether an I- and D-space task is being built. For a conventional task, the size of the internal segment descriptor is 230_8 bytes. For an I- and D-space task, the size is 262_8 bytes.

In addition to these reasons for the increase in size of an overlaid I- and D-space task, a memory-resident overlaid I- and D-space task would be even larger because of the need for two window descriptors for each memory-resident segment.

Example 7-1: Map of Overlaid Task MAIN.TSK

```
MAIN.TSK;1   Memory allocation map  TKB M43.00       Page 1
                     20-OCT-87   10:08


Task    name   : ...CBP
Partition name : GEN
Identification : V00.00
Task  UIC      : [240,1]
Stack    limits: 000320 001317 001000 00512.
PRG xfr address: 002350
Total address windows: 3.
Task  image  size  : 1664. words
Task address limits: 000000 006307
R-W disk blk limits: 000002 000012 000011 00009.

MAIN.TSK;1 Overlay description:

Base   Top         Length
----   ---         ------
000000  004513  004514  02380.    MAIN
004514  005063  000350  00232.      INPUT
004514  004607  000074  00060.      CALC
004514  006307  001574  00892.      OUTPUT

MAIN.TSK;1   Memory allocation map  TKB M43.00       Page 2
MAIN                 20-OCT-87   10:08

*** Root segment: MAIN

R/W mem  limits: 000000 004513 004514 02380.
Disk blk limits: 000002 000006 000005 00005.

Memory allocation synopsis:

Section                                            Title  Ident  File
-------                                            -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)     001320 000466 00310.
                              001320 000250 00168. EDDAT  03     SYSLIB.OLB;5
                              001570 000216 00142. CBTA   04.3   SYSLIB.OLB;5
COM1   :(RO,D,GBL,REL,CON)    002006 000024 00020.
                              002006 000024 00020. MAIN   V00.00 MAIN.OBJ;36
COM2   :(RW,D,GBL,REL,CON)    002032 000032 00026.
                              002032 000032 00026. MAIN   V00.00 MAIN.OBJ;36
COM3   :(RW,D,GBL,REL,CON)    002064 000010 00008.
                              002064 000010 00008. MAIN   V00.00 MAIN.OBJ;36
```

(Continued on next page)

Example 7-1 (Cont.): Map of Overlaid Task MAIN.TSK

```
COM4   :(RW,D,GBL,REL,CON)      002074 000234 00156.
                                002074 000234 00156. MAIN    V00.00 MAIN.OBJ;36
COM5   :(RW,D,GBL,REL,CON)      002330 000020 00016.
                                002330 000020 00016. MAIN    V00.00 MAIN.OBJ;36
LIBROT:(RW,I,GBL,REL,CON)       000000 000140 00096.
                                000000 000140 00096. LIBROT 03.01  LIBFSO.STB;1
MAIN   :(RO,I,LCL,REL,CON)      002350 000142 00098.
                                002350 000142 00098. MAIN    V00.00 MAIN.OBJ;36
$$ALER:(RO,I,LCL,REL,CON)       002512 000024 00020.
                                002512 000000 00000. OVCTR   15.03  SYSLIB.OLB;5
                                002512 000024 00020. ALERR   02.00  SYSLIB.OLB;5
$$ALVC:(RO,I,LCL,REL,CON)       002536 000070 00056.
                                      .
                                      .
                                      .

Global symbols:

AADD    002556-R  N.DTDS 000020    $CBDAT 001570-R  .FSRPT 000050
ARGBLK  002046-R  N.FAST 000013    $CBDMG 001576-R  .NALER 003354-R
                                      .
                                      .
                                      .

MAIN.TSK;1   Memory allocation map  TKB M43.00      Page 4
INPUT                20-OCT-87   10:08


*** Segment: INPUT

R/W mem  limits: 004514 005063 000350 00232.
Disk blk limits: 000007 000007 000001 00001.

Memory allocation synopsis:

Section                                      Title  Ident  File
-------                                      -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)       004514 000074 00060.
                                004514 000074 00060. CATB    03     SYSLIB.OLB;5
INPUT :(RO,I,LCL,REL,CON)       004610 000252 00170.
                                004610 000252 00170. INPUT   01     INPUT.OBJ;32
$$ALVC:(RO,I,LCL,REL,CON)       005062 000000 00000.
                                      .
                                      .
                                      .
```

(Continued on next page)

Example 7–1 (Cont.): Map of Overlaid Task MAIN.TSK

```
Global symbols:

INPUT  004610-R  $CDTB  004514-R  $COTB  004522-R
                                         .
                                         .
                                         .


MAIN.TSK;1   Memory allocation map  TKB M43.00      Page 5
                     20-OCT-87   10:08

*** Task builder statistics:

    Total work file references: 13626.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 5198. words (20. pages)
    Size of work file: 4096. words (16. pages)

    Elapsed time:00:00:19
```

Example 7–2: Map of Overlaid I- and D-Space Task MAINID.TSK

```
MAINID.TSK;1   Memory allocation map  TKB M43.00      Page 1
                     15-OCT-87   11:51


Task    name    : ...CBP
Partition name : GEN
Identification : V00.00
Task  UIC      : [240,1]
Stack    limits: 000256 001255 001000 00512.
PRG xfr address: 000744
Task attributes: ID
Total address windows: 4.
Task  image  size  : 1184. words, I-Space
                      704. words, D-Space
Task Address limits: 000000 004413 I-Space
                     000000 002513 D-Space
R-W disk blk limits: 000002 000014 000013 00011.


MAINID.TSK;1 Overlay description:

Base    Top       Length
----    ---       ------
000000  002713  002714  01484. I    MAIN
000000  002413  002414  01292. D

002714  003263  000350  00232. I      INPUT
002414  002413  000000  00000. D

002714  003007  000074  00060. I      CALC
002414  002413  000000  00000. D

002714  004413  001500  00832. I      OUTPUT
002414  002513  000100  00064. D

MAINID.TSK;1   Memory allocation map  TKB M43.00      Page 2
MAIN                 15-OCT-87   11:51

*** Root segment: MAIN

R/W mem  limits: 000000 002713 002714 01484. I-Space
                 000000 002413 002414 01292. D-Space
```

(Continued on next page)

Example 7–2 (Cont.): Map of Overlaid I- and D-Space Task MAINID.TSK

```
Disk blk limits: 000002 000004 000003 00003. I-Space
                 000005 000007 000003 00003. D-Space

Memory allocation synopsis:

Section                                            Title  Ident File
--------                                           -----  ----- ----
. BLK.:(RW,I,LCL,REL,CON)     000256 000466 00310.
                              000256 000250 00168. EDDAT  03    SYSLIB.OLB;10
                              000526 000216 00142. CBTA   04.3  SYSLIB.OLB;10
COM1  :(RO,D,GBL,REL,CON)     001256 000024 00020.
                              001256 000024 00020. MAIN   V00.00 MAIN.OBJ;34
COM2  :(RW,D,GBL,REL,CON)     001302 000032 00026.
                              001302 000032 00026. MAIN   V00.00 MAIN.OBJ;34
COM3  :(RW,D,GBL,REL,CON)     001334 000010 00008.
                              001334 000010 00008. MAIN   V00.00 MAIN.OBJ;34
COM4  :(RW,D,GBL,REL,CON)     001344 000234 00156.
                              001344 000234 00156. MAIN   V00.00 MAIN.OBJ;34
COM5  :(RW,D,GBL,REL,CON)     001600 000020 00016.
                              001600 000020 00016. MAIN   V00.00 MAIN.OBJ;34
LIBROT:(RW,I,GBL,REL,CON)     000000 000140 00096.
                              000000 000140 00096. LIBROT 03.01 LIBFSO.STB;1
MAIN  :(RO,I,LCL,REL,CON)     000744 000142 00098.
                              000744 000142 00098. MAIN   V00.00 MAIN.OBJ;34
$$ALER:(RO,I,LCL,REL,CON)     001106 000024 00020.
                              001106 000000 00000. OVIDR  01    SYSLIB.OLB;10
                              001106 000024 00020. ALERR  02.00 SYSLIB.OLB;10
$$ALVD:(RO,D,LCL,REL,CON)     001620 000034 00028.
$$ALVI:(RO,I,LCL,REL,CON)     001132 000070 00056.
                                      .
                                      .
                                      .

Global symbols:

AADD   001152-R  N.DTDS 000020   $CBDAT 000526-R  $TIM  000402-R
ARGBLK 001316-R  N.FAST 000013   $CBDMG 000534-R  $VEXT 000056
                                .
                                .
                                .
```

(Continued on next page)

Example 7-2 (Cont.): Map of Overlaid I- and D-Space Task MAINID.TSK

```
MAINID.TSK;1    Memory allocation map  TKB M43.00      Page 4
INPUT                  15-OCT-87    11:51

*** Segment: INPUT

R/W mem  limits: 002714 003263 000350 00232. I-Space
                 002414 002413 000000 00000. D-Space

Disk blk limits: 000010 000010 000001 00001. I-Space
                 000011 000011 000000 00000. D-Space

Memory allocation synopsis:

Section                                     Title  Ident  File
-------                                     -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)       002714 000074 00060.
                                002714 000074 00060. CATB   03     SYSLIB.OLB;10
INPUT :(RO,I,LCL,REL,CON)       003010 000252 00170.
                                003010 000252 00170. INPUT  01     INPUT.OBJ;32
$$ALVD:(RO,D,LCL,REL,CON)       002414 000000 00000.
$$ALVI:(RO,I,LCL,REL,CON)       003262 000000 00000.
$$RTS :(RO,I,GBL,REL,OVR)       002710 000002 00002.
$$SLVC:(RO,I,LCL,REL,CON)       003262 000000 00000.


Global symbols:

INPUT  003010-R  $CDTB  002714-R  $COTB  002722-R
                                  .
                                  .
                                  .
MAINID.TSK;1    Memory allocation map  TKB M43.00      Page 5
CALC                   15-OCT-87    11:51


                                  .
                                  .
                                  .

*** Task builder statistics:

    Total work file references: 13920.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 5010. words (19. pages)
    Size of work file: 4096. words (16. pages)

    Elapsed time:00:00:28
```

Chapter 8
Supervisor-Mode Libraries

A supervisor-mode library is a resident library that doubles a user task's virtual address space by mapping the instruction space of the processor's supervisor mode.

A call from within a user task to a subroutine within a supervisor-mode library causes the processor to switch from user mode to supervisor mode. The user task transfers control to a mode-switching vector that the Task Builder includes within the task. The mode-switching vector performs the mode switch and then transfers control to the called subroutine within the supervisor-mode library. The library routine executes with the processor in supervisor mode. When the library routine finishes executing, it transfers control to a completion routine within the library. The completion routine mode switches the processor back to user mode. The user task continues executing with the processor in user mode at the return address on the stack. This process recurs whenever the user task calls a subroutine in the supervisor-mode library.

8.1 Mode-Switching Vectors

In a task that links to a supervisor-mode library, TKB includes a 4-word, mode-switching vector in the user task's address space for each entry point referred to in a subroutine in the library.

The following example shows the contents of a mode-switching vector:

```
MOV #COMPLETION-ROUTINE,-(SP)
CSM #SUPERVISOR-MODE-ROUTINE ADDRESS
```

Note

When switching from user mode to supervisor mode, all registers of the referencing task are preserved. All condition codes in the PSW saved on the stack are cleared and must be restored by the completion routine.

8.2 Completion Routines

After the subroutine finishes executing, its RETURN statement transfers control to a completion routine that switches from supervisor mode to user mode. The completion routine returns program control back to the referencing task at the instruction after the call to the subroutine. The system library (SYSLIB) contains two completion routines, as follows:

- $CMPCS restores only the carry bit in the user-mode PSW.
- $CMPAL restores all the condition code bits in the user-mode PSW.

8.3 Restrictions on the Contents of Supervisor-Mode Libraries

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

- Only subroutines using the form JSR PC, x should be used within the library.
- The library must not contain subroutines that use the stack to pass parameters.
- If both the library and the referencing task link to a subroutine from the system library, then the entry point name of the subroutine must be excluded from the STB file for the library.
- Unless you include the Executive directive Map Supervisor D-space (MSDS$) within the library, the library must not contain data of any kind (even read-only) because the user supervisor D-space APRs map the user task by default. This includes user data, buffers, I/O status blocks, and Directive Parameter Blocks (only the $S directive form can be used, because the DPB for this form is pushed onto the user stack at run time).

  Using the MSDS$ directive, the library can map data within the instruction space of the supervisor-mode library by using the supervisor D-space APRs. The directive maps specific supervisor D-space APRs to supervisor instruction space by copying the supervisor I-space APRs that map the data portion of the library. To effectively contain data within a supervisor-mode library, you must know which APRs map the data portions of your task and library.

Note

You cannot use MSDS$ to map supervisor D-space APR 0. Mapping library data and the user task simultaneously should be done with extreme care. The *RSX-11M-PLUS and Micro/RSX Executive Reference Manual* discusses the MSDS$ directive in detail.

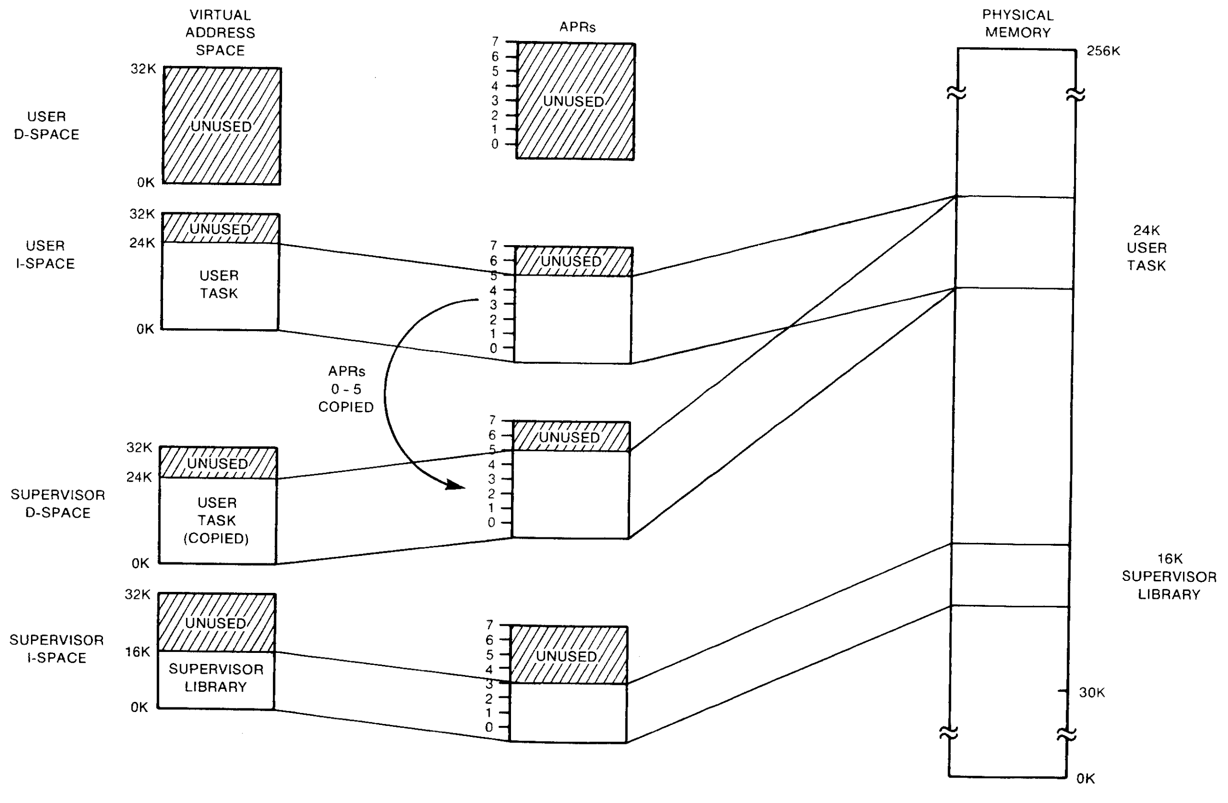
8.4 Supervisor-Mode Library Mapping

Supervisor-mode libraries are mapped with the supervisor I-space APRs. Supervisor D-space APRs can map the user task, data within the library, or both the user task and library data simultaneously. They map the user task by default.

The supervisor D-space APRs can be mapped differently according to whether the library contains data.

Supervisor D-space APRs are copies of user I-space APRs, which map the entire user task. This gives the library access to data within the user task. Figure 8-1 illustrates this mapping.

Figure 8-1: Mapping of a 24K-Word Conventional User Task That Links to a 16K-Word Supervisor-Mode Library



APR MAPPING

USER D-SPACE	UNUSED
USER I-SPACE	0-5 map entire user task
SUPERVISOR D-SPACE	0-5 map entire user task
SUPERVISOR I-SPACE	0-3 map library

ZK-439-81

8.4.1 Supervisor-Mode Library Data

Libraries that contain data require extremely complicated mapping that may overwrite the user task or cause the task to fail.

Supervisor D-space APRs are copies of user I-space APRs, which map the entire user task. For I- and D-space tasks, the supervisor D-space APRs are copies of the user D-space APRs. Including the MSDS$ directive (see Section 8.3) within the library code enables the library to map data within its own instruction space. The user task may be overmapped. The library has access to data within its instruction space and to data in the user task that is not overmapped by the MSDS$ directive.

Figure 8-2 illustrates this mapping.

8.4.2 Supervisor-Mode Libraries with I- and D-Space Tasks

I- and D-space tasks may link to supervisor-mode libraries. Instead of mapping to the entire user task, the supervisor-mode library's D-space APRs map the task's data space. Because the I- and D-space task maps its data with the D-space APRs, the task's D-space APRs are copied into the supervisor-mode library's D-space APRs. Therefore, the supervisor-mode library maps its own instructions with supervisor-mode I-space APRs and maps the task's data with supervisor-mode D-space APRs.

Figure 8-3 illustrates the mapping of an I- and D-space task linked to a supervisor-mode library.

8.5 Building and Linking to Supervisor-Mode Libraries

Building and linking to a supervisor-mode library is essentially the same as building and linking to a conventional resident library (discussed in Chapter 5). When you build a supervisor-mode library using the TKB command line, you suppress the header by attaching /-HD to the task image file. If you use LINK, you use the /NOHEAD qualifier in the LINK command line. 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.

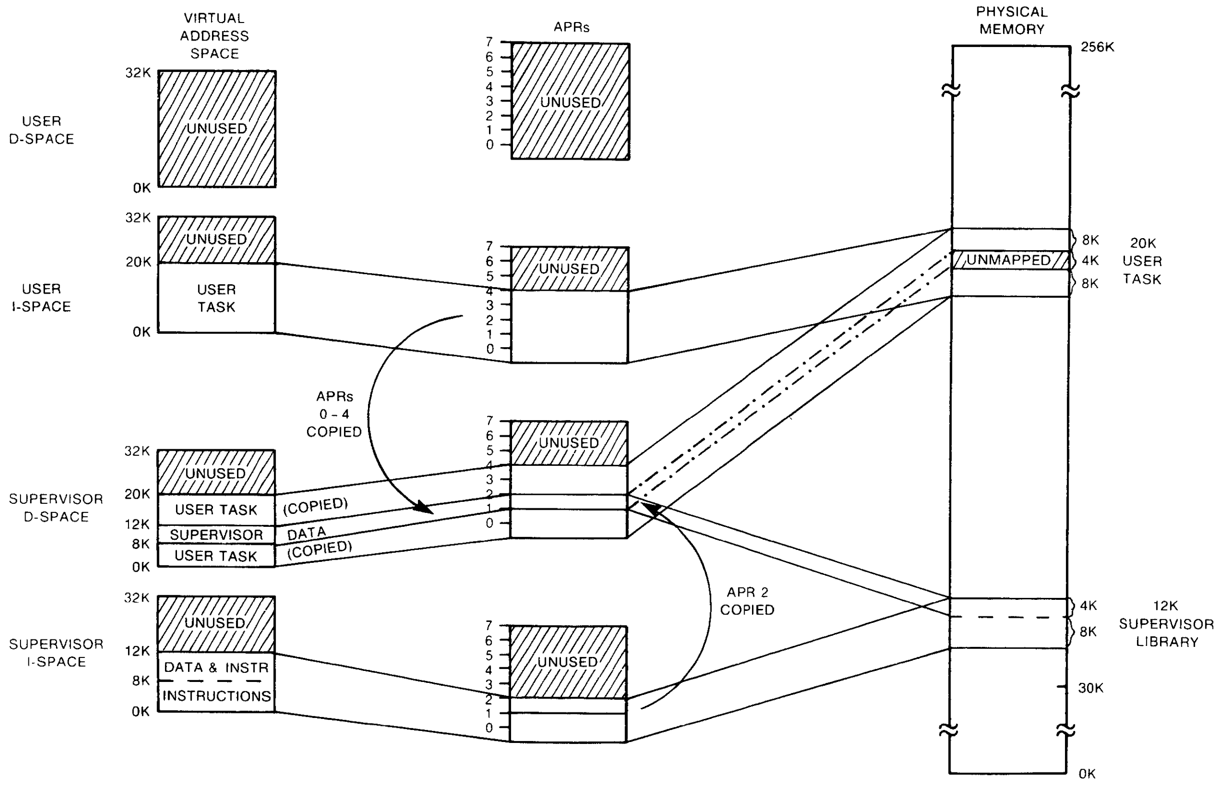
8.5.1 Relevant TKB Options

Use the following options to build and reference supervisor-mode libraries:

CMPRT	Indicates that you are building a supervisor-mode library and specifies the name of the completion routine
RESSUP (SUPLIB)	Indicates that your task references a supervisor-mode library
GBLXCL	Excludes a global symbol from the STB file of the supervisor-mode library

These options are discussed briefly in the next sections and are fully documented in Chapter 11.

Figure 8–2: Mapping of a 20K-Word Conventional User Task That Links to a 12K-Word Supervisor-Mode Library Containing 4K Words of Data

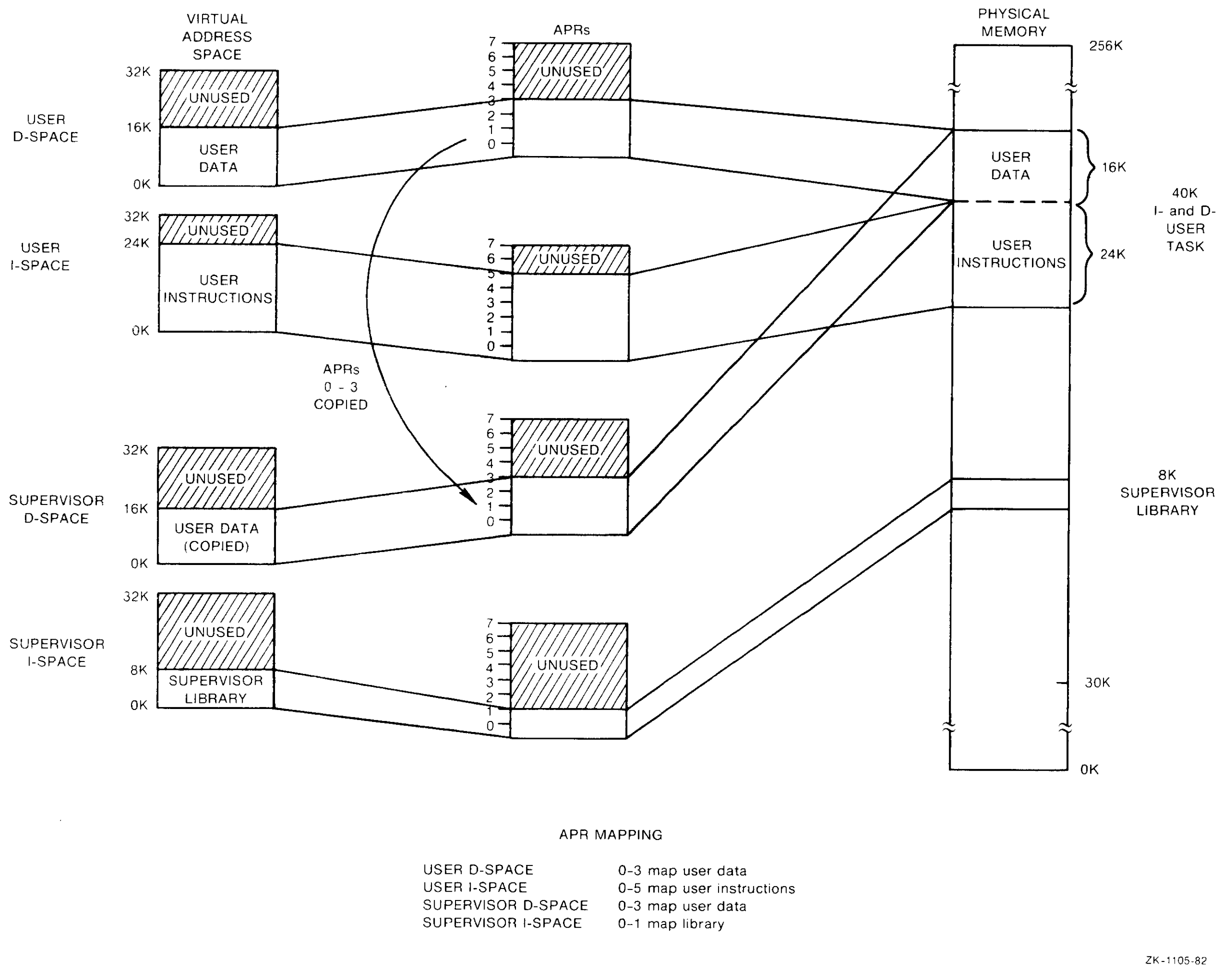


APR MAPPING

USER D-SPACE	UNUSED
USER I-SPACE	0-4 map entire user task
SUPERVISOR D-SPACE	0-1 and 3-4 map user task
	2 remapped to supervisor data using MSDS$
SUPERVISOR I-SPACE	0-2 map library

ZK-440-81

Figure 8–3: Mapping of a 40K-Word I- and D-Space Task That Links to an 8K-Word Supervisor-Mode Library



APR MAPPING

USER D-SPACE	0-3 map user data
USER I-SPACE	0-5 map user instructions
SUPERVISOR D-SPACE	0-3 map user data
SUPERVISOR I-SPACE	0-1 map library

8.5.2 Building the Library

You indicate to TKB 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 TKB processes this option, it places the completion routine entry point in the library's STB file. To exclude a global symbol from the library's STB file, you specify the name of the global symbol as the argument of the GBLXCL option. You must exclude from the STB file of a supervisor-mode library any symbol defined in the library that represents the following:

- An entry point to a subroutine that uses the stack to pass parameters
- An entry point to a subroutine mapped in user mode that the referencing user task calls

8.5.3 Building the Referencing Task

When you build a task that references a supervisor-mode library, use the RESSUP option if you are referencing a user-owned, supervisor-mode library, and SUPLIB if you are referencing a system-owned, supervisor-mode library. (As with the RESLIB and LIBR options for linking to conventional libraries, RESSUP and SUPLIB are functionally the same.) The arguments for these options are as follows:

- The file specification (RESSUP option) or name (SUPLIB option) of the library to be referenced
- A switch that tells TKB whether to use system-supplied vectors to switch from user mode to supervisor mode.
- For position-independent libraries, the first available supervisor-mode I-space APR that you want to map the library.

8.5.4 Mode-Switching Instruction

Mode switching occurs with a hardware instruction available with all processors that have the the KDJ11 chip set. Throughout the remainder of the chapter, supervisor-mode libraries are referred to as CSM (change supervisor mode) libraries.

8.6 CSM Libraries

This section discusses how you build and link to CSM libraries. It also shows an extended example of building and linking to a CSM library and explains the context-switching vectors and completion routines for CSM libraries.

8.6.1 Building a CSM Library

You indicate to the Task Builder that you are building a CSM library by specifying the name of the completion routine as the argument for the CMPRT option. This option places the name of the completion routine into the library's STB file. Link the completion routine, either $CMPAL or $CMPCS, located in LB:[1,1]SYSLIB.OLB, as the first input file. Although the completion routines are located in the system library (which is ordinarily referenced by default), you must explicitly indicate it and link it as the first input file. You must also specify in the PAR option a base of 0 for the partition in which the library will reside. These two steps locate the completion routine at virtual 0 of the library's virtual address space.

You specify the name of any global symbols that you would like to exclude from the library's STB file as the argument to the GBLXCL option. You must exclude from the STB file of a supervisor-mode library any symbol defined in the library that represents the following:

- An entry point to a subroutine that uses the stack to pass parameters
- An entry point to a subroutine mapped in user mode that the referencing user task calls

The following sample TKB command sequence builds a CSM library in directory [301,55] on device SY:

```
TKB>CSM/-HD/LI/PI,CSM/MA,CSM= RET
TKB>LB:[1,2]SYSLIB/LB:CMPAL,SY:[301,55]CSM RET
TKB>/ RET
Enter Options:
TKB>STACK=0 RET
TKB>PAR=GEN:0:2000 RET
TKB>CMPRT=$CMPCS RET
TKB>GBLXCL=$SAVAL RET
TKB>// RET
>
```

Or, you can use the following LINK command sequence to build the same library:

```
$ LINK/TAS:CSM/NOH/SHARE:LIB/CODE:PIC/MAP:CSM/SYS/SYM:CSM/OPT - RET
->LB:[1,2]SYSLIB/INCLUDE:CMPAL,SY:[301,55]CSM RET
Option? STACK=0 RET
Option? PAR=GEN:0:2000 RET
Option? CMPRT=CMPCS RET
Option? GBLXCL=$SAVAL RET
Option? RET
$
```

The library is built without a header or stack, like all shared regions. It is position independent and has only one program section named .ABS. The /LI switch in TKB (or the /CODE:PIC qualifier in LINK) accomplishes this, eliminating program section name conflicts between the library and the referencing task.

The completion routine module CMPAL is specified first in the input line. The library will run in partition GEN at 0 and is not more than 1K words. These are two aspects of building supervisor-mode libraries specific to CSM libraries: the completion routine must be linked first and must reside at virtual 0. (Why the CSM library must reside at virtual 0 is discussed in Section 8.6.2.)

The CMPRT option specifies the global symbol $CMPCS, which is the entry point of the completion routine. Note that the name for the system library module is CMPCS and its corresponding global symbol is $CMPCS.

The GBLXCL option excludes $SAVAL from the library's STB file because the user task must reference a copy of $SAVAL that is mapped with user mode APRs.

8.6.2 Linking to a CSM Library

If your task links to a user-owned CSM library, you use the RESSUP option. If your task links to a system-owned CSM library, you use the SUPLIB option. These options tell TKB that the task will link to a supervisor-mode library. The option takes up to three arguments, as follows:

- The file specification (RESSUP option) or name (SUPLIB option) of the library
- A switch that tells TKB whether to use system-supplied, mode-switching vectors
- For position-independent libraries, an APR that must be APR 0 so that the library's completion routine is mapped at virtual 0

This information enables TKB to find the STB file for the CSM library, include a 4-word mode-switching vector within the user task for each call to a subroutine within the library, and correctly map the library at virtual 0 in the library image.

The following examples of TKB and LINK command sequences build a task named REF, which references the library SUPER that you built in the previous section:

```
TKB>REF,REF=REF  RET
TKB>/  RET
Enter Options:
TKB>RESSUP=SUPER/SV:0  RET
TKB>//  RET
>

$ LINK/TAS/MAP/OPT REF  RET
Option? RESSUP=SUPER/SV:0  RET
Option?  RET
$
```

This sequence tells TKB to include in the logical address space of REF a user-owned, supervisor-mode library named SUPER. TKB includes a 4-word mode-switching vector within the user task for each call to a subroutine within the library. The CSM library is position independent and is mapped with APR 0.

8.6.3 Example of a CSM Library and Linking Task

This example shows you the code and maps and the TKB and LINK command sequences for building and linking to a CSM library that contains no data in a system not having user data space. Example 8-1 shows the code for the library SUPER and Example 8-2 shows its accompanying map. Example 8-3 shows the code for the completion routine $CMPCS that is linked into SUPER from the system library. Example 8-4 shows the code for referencing task TSUP and Example 8-5 shows its accompanying map.

Example 8-1: Code for SUPER.MAC

```
        .TITLE  SUPER
        .IDENT  /01/

SORT::
        CALL    $SAVAL          ; SAVE ALL REGISTERS
        TST     (R5)+           ; SKIP OVER NUMBER OF ARGUMENTS
        MOV     (R5)+,R0        ; GET ADDRESS OF LIST
        MOV     (R5)+,R4        ; GET ADDRESS OF LENGTH OF LIST
        MOV     (R4),R4         ; GET LENGTH OF LIST
        BEQ     40$             ; IF NO ARGUMENTS
        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$:
        DEC     R3              ; DECREMENT LOOP COUNT
        BGE     20$             ; IF NE LOOP
        DEC     R4              ; DECREMENT
        BLE     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
        BEQ     20$             ; IF NO ARGUMENTS
        MOV     (R5),R5         ; ADDRESS OF RETURNED VALUE
        MOV     R2,R3           ; COPY LENGTH
```

(Continued on next page)

Example 8-1 (Cont.): Code for SUPER.MAC

```
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
        .END
```

Example 8–2: Memory Allocation Map for SUPER

```
SUPER.TSK;3 Memory allocation map  TKB M41.00      Page 1
                     29-DEC-87   15:01

Partition name : GEN
Identification : 03.01
Task  UIC      : [7,61]
Task attributes: -HD,PI
Total address windows: 1.
Task  image  size  : 128. words
Task address limits: 000000 000343
R-W disk blk limits: 000002 000002 000001 00001.

 Root segment: CMPAL

R/W mem  limits: 000000 000341 000342 00226.
Disk blk limits: 000002 000002 000001 00001.


Memory allocation synopsis:

Section                                       Title  Ident  File
-------                                       -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)    000000 000342 00226.
                             000000 000140 00096. CMPAL  03.01  SYSLIB.OLB;1
                             000140 000140 00096. SUPER  01     SUPER.OBJ;3
                             000300 000042 00034. SAVAL  00     SYSLIB.OLB;1

Global symbols:

SEARCH 000220-R  SORT   000140-R  $CMPAL  000022-R  $CMPCS 000110-R
$SAVAL 000300-R  $SRTI  000002-R

 Task builder statistics:

    Total work file references: 300.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 6466. words (25. pages)
    Size of work file: 1024. words (4. pages)

    Elapsed time:00:00:08
```

Example 8-3: Completion Routine $CMPCS from SYSLIB.OLB

```
        .TITLE  CMPAL
        .IDENT  /0204/

;
;               COPYRIGHT (c) 1987 BY
;       DIGITAL EQUIPMENT CORPORATION, MAYNARD
;        MASSACHUSETTS.  ALL RIGHTS RESERVED.
;
; THIS  SOFTWARE  IS  FURNISHED  UNDER  A LICENSE AND MAY BE USED
; AND  COPIED  ONLY IN  ACCORDANCE WITH THE TERMS OF SUCH LICENSE
; AND WITH  THE INCLUSION  OF THE ABOVE  COPYRIGHT  NOTICE.  THIS
; SOFTWARE, OR ANY OTHER  COPIES  THEREOF, MAY NOT BE PROVIDED OR
; OTHERWISE MADE  AVAILABLE TO ANY OTHER PERSON.  NO TITLE TO AND
; OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.
;
; THE INFORMATION  IN THIS DOCUMENT IS SUBJECT  TO CHANGE WITHOUT
; NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT  BY  DIGITAL
; EQUIPMENT CORPORATION.
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF
; ITS SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
;

        .ENABL  LC

;
;
; This module supports the "new" transfer vector format generated by
; the Task Builder for entering super mode libraries.  This format
; is optimized for speed and size and supports user data space tasks.
;
; The CSM dispatcher routine and the standard completion routines
; $CMPAL and $CMPCS are included in this module due to the close
; interaction between them.
;

;
; **-CSM Dispatcher-Dispatch CSM entry
;
; This module must be linked at virtual zero in the supervisor mode
; library.  It is entered via a four word transfer vector of the form:
;
;       MOV     #completion-routine,-(SP)
;       CSM     #routine
;
; Note: Immediate mode emulation of the CSM instruction is required
;       in the Executive.
```

(Continued on next page)

Example 8–3 (Cont.): Completion Routine $CMPCS from SYSLIB.OLB

```
;
; The CSM instruction transfers control to the address contained in
; supervisor mode virtual 10.  At this point the stack is the following:
;
;          (SP)  routine address
;         2(SP)  PC (past end of transfer vector)
;         4(SP)  PSW with condition codes cleared
;         6(SP)  Completion-routine address
;        10(SP)  Return address
;
; A routine address of 0 is special cased to support return to
; supervisor mode from a user mode debugging aid (ODT).  In this case
; stack is the following:
;
;          (SP)  zero
;         2(SP)  PC from CSM to be discarded
;         4(SP)  PSW from CSM to be discarded
;         6(SP)  Super mode PC supplied by debugger
;        10(SP)  Super mode PSW supplied by debugger
;
; To allow positioning at virtual zero, this code must be in the blank
; PSECT which is first in the TKBs PSECT ordering.

        .PSECT
        .ENABL  LSB

; Debugger return to super mode entry. Must start at virtual zero

        CMP     (SP)+,(SP)+     ; Clean off PSW and PC from CSM

;
; **-$SRTI-SUPER mode RTI
;
; This entry point performs the necessary stack management to allow
; an RTI from super mode to either super mode or user mode.
; In this case, the stack is the following:
;
;         (SP)   Super mode PC
;        2(SP)   Super mode PSW

$SRTI:: TST     2(SP)           ; Returning to user mode?
        BR      70$             ; Join common code
```

(Continued on next page)

Example 8-3 (Cont.): Completion Routine $CMPCS from SYSLIB.OLB

```
; CSM transfer address, this word must be at virtual 10 in super mode

        .WORD   CSMSVR          ; CSM dispatcher entry

; Dispatch CSM entry

CSMSVR: MOV     6(SP),2(SP)     ; Set completion routine address for RETURN
        JMP     @(SP)+          ; Transfer to super mode library routine

;
; **-$CMPAL-Completion routine which sets up NZVC in the PSW
;
; Copy all condition codes to stacked PSW.  Current stack:
;
;        (SP)   PSW with condtion codes cleared
;       2(SP)   Completion routine address (to be discarded)
;       4(SP)   Return address
;

$CMPAL::BPL     40$             ;
        BNE     20$             ;
        BVC     10$             ;
        BIS     #16,(SP)        ; Set NZV
        BR      $CMPCS          ;
10$:    BIS     #14,(SP)        ; Set NZ
        BR      $CMPCS          ;
20$:    BVC     30$             ;
        BIS     #12,(SP)        ; Set NV
        BR      $CMPCS          ;
30$:    BIS     #10,(SP)        ; Set N
        BR      $CMPCS          ;
40$:    BNE     60$             ;
        BVC     50$             ;
        BIS     #6,(SP)         ; Set ZV
        BR      $CMPCS          ;
50$:    BIS     #4,(SP)         ; Set Z
        BR      $CMPCS          ;
60$:    BVC     $CMPCS          ;
        BIS     #2,(SP)         ; Set V
```

(Continued on next page)

Example 8–3 (Cont.): Completion Routine $CMPCS from SYSLIB.OLB

```
;
; **-$CMPCS-Completion routine which sets up only C in the PSW
;
; Copy only carry to stacked PSW.  Current stack:
;
;        (SP)   PSW with condtion codes cleared
;       2(SP)   Completion routine address (to be discarded)
;       4(SP)   Return address
;
$CMPCS::ADC     (SP)            ; Set up carry
        MOV     4(SP),2(SP)     ; Set up return address for RTT
        MOV     (SP)+,2(SP)     ; And PSW. Returning to super mode?
70$:    BPL     80$             ; If PL yes
        MOV     #6,-(SP)        ; Number of bytes for (SP), PSW, and PC
        ADD     SP,(SP)         ; Compute clean stack value
        MTPI    SP              ; Set up previous stack pointer
80$:    RTT                     ; Return to previous mode and caller

        .DSABL  LSB

        .END
```

Example 8-4: Code for TSUP.MAC

```
        .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:  .WORD   0
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
```

(Continued on next page)

Example 8–4 (Cont.): Code for TSUP.MAC

```
        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
;+
;Task Builder replaced call to SORT subroutine in SUPLIB with 4-word
;context switching vector. Flow of control switches to SUPLIB via
;the vector and back via the completion routine $CMPCS. TSUP
;continues executing at the next instruction.
;-
        CLR     R2              ;
        MOV     #IARRAY,R0      ; GET ARRAY ADDRESS
30$:
        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
```

(Continued on next page)

Example 8-4 (Cont.): Code for TSUP.MAC

```
;
;Call to SUPLIB for SEARCH subroutine.
;
        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
```

TSUP prompts you to enter numbers at your terminal. It calls a subroutine in SUPER to sort the numbers. Then it displays the numbers you entered as array entries and prompts you to request a number to search for. TSUP calls a subroutine in SUPERLIB to search for the number. Finally, TSUP indicates at your terminal either that the number was not found or the array location in which the number is stored.

Example 8-5: Memory Allocation Map for TSUP

```
TSUP.TSK;1 Memory allocation map  TKB M43.00      Page 1
                     29-DEC-87   15:01


Partition name : GEN
Identification : 01
Task  UIC      : [301,55]
Stack    limits: 000274 001273 001000 00512.
PRG xfr address: 002130
Total address windows: 2.
Task  image  size  : 1344. words
Task address limits: 000000 005133
R-W disk blk limits: 000002 000007 000006 00006.

*** Root segment: TSUP

R/W mem  limits: 000000 005133 005134 02652.
Disk blk limits: 000002 000007 000006 00006.

Memory allocation synopsis:

Section                                        Title  Ident  File
-------                                        -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)     001274 002334 01244.
                              001274 001234 00668. TSUP   01     TSUP.OBJ;22
CMPAL :(RW,I,LCL,REL,CON)     000000 000474 00316.
PUR$D :(RO,I,LCL,REL,CON)     003630 000076 00062.
PUR$I :(RO,I,LCL,REL,CON)     003726 000752 00490.
$$RESL:(RO,I,LCL,REL,CON)     004700 000212 00138.
$$SLVC:(RO,I,LCL,REL,CON)     005112 000020 00016.

TSUP.TSK;1 Memory allocation map  TKB M43.00      Page 2
                    29-DEC-87   15:01


*** Task builder statistics:

    Total work file references: 2477.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 6988. words (27. pages)
    Size of work file: 1024. words (4. pages)

    Elapsed time:00:00:05
```

8.6.3.1 Building SUPER

To build SUPER in directory [301,55] on device SY:, use the following TKB or LINK command sequence:

```
TKB>SUPER/-HD/LI/PI,SUPER/MA,SUPER= RET
TKB>LB:[1,1]SYSLIB/LB:CMPAL,SY:[301,55]SUPER RET
TKB>/ RET
Enter Options:
TKB>STACK=0 RET
TKB>PAR=GEN:0:2000 RET
TKB>CMPRT=$CMPCS RET
TKB>GBLXCL=$SAVAL RET
TKB>// RET
>

$ LINK/TAS:SUPER/NOH/SHARE:LIB/CODE:PIC/MAP:SUPER/SYS/SYM:SUPER/OPT - RET
->LB:[1,1]SYSLIB/INC:CMPAL,SY:[301,55]SUPER RET
Option? STACK=0 RET
Option? PAR=GEN:0:2000 RET
Option? CMPRT=$CMPCS RET
Option? GBLXCL=$SAVAL RET
Option? RET
$
```

SUPER is built without a header or stack. It is position independent and has only one program section, named .BLK. The /LI switch or the /SHARE:LIB qualifier eliminates program section name conflicts between the library and the referencing task.

The completion routine module CMPAL is specified first in the input line. The library will run in partition GEN at 0 and is not more than 1K words.

The GBLXCL option excludes $SAVAL from the library's STB file. You exclude $SAVAL from the STB file because the referencing task, TSUP, also calls $SAVAL. If TSUP finds $SAVAL in the STB file of SUPER, it will not link a separate copy of $SAVAL into its task image from the system library. If TSUP could not link to a copy of $SAVAL that is mapped through user APRs, TSUP would call $SAVAL as a subroutine residing within the supervisor-mode library but without the necessary mode-switching vector and completion routine support. This option forces TKB to link $SAVAL from the system library into the task image for TSUP.

The memory allocation map in Example 8-2 shows the following information:

- SUPER begins at virtual 0.
- The completion routine, $CMPAL, is linked into the library from the system library at virtual 0.
- The entry point $CMPAL is located at virtual 22, SEARCH is located at 35, and SORT is located at 274. All of these entry points are relocatable.
- The SEARCH and SORT subroutines that were located at virtual address 352 and 274, respectively, in the virtual address space of SUPER have been relocated to the mode-switching vectors residing at 5112 and 5122, respectively, in TSUP.

8.6.3.2 Building TSUP

Use the following TKB or LINK command sequence to build a task, TSUP, that links to SUPER:

```
TKB>TSUP,TSUP=TSUP RET
TKB>/ RET
Enter Options:
TKB>RESSUP=SUPER/SV:0 RET
TKB>// RET

$ LINK/TAS/MAP/OPT SUPER RET
Option? RESSUP=SUPER/SV:0 RET
Option? RET
$
```

These two command sequences tell TKB to include in the logical address space of TSUP a user-owned supervisor-mode library named SUPER. TKB includes a 4-word mode-switching vector within the task image for each call to a subroutine within the library. The library is position independent and is mapped with supervisor I-space APR 0. This is a requirement for CSM libraries because the CSM library expects to find the entry point of the completion routine at location 10.

The memory allocation map for TSUP (Example 8-5) shows the following information:

- $CMPAL is linked from the STB file of the library and begins at location 0.
- The mode-switching vectors begin at 005136 and are 16_{10} bytes in length. This means that TSUP calls subroutines within the library two times (four words for each vector).
- The initiation routine $SUPL is located at 4700.
- The SEARCH and SORT subroutines that were located at virtual 112 and 32, respectively, in the virtual address space of SUPER have been relocated to the mode-switching vectors residing at 5136 and 5146, respectively, in TSUP.
- The system library module SAVAL, containing $SAVAL, has been linked into the task image instead of including $SAVAL from the library's STB file.

8.6.3.3 Running TSUP

After building SUPER and TSUP as indicated in the task-build command sequence discussed previously, you install SUPER and run TSUP. TSUP prompts you for a number, as follows:

```
ARRAY (x)
```

x

The position in which to store the number in the array. You enter a number. TSUP stores the number in the array and prompts you again for a number. This continues until you either have entered a 0, an invalid number, or 10 numbers. Then TSUP calls the SORT routine in SUPER.

You enter a number. TSUP calls the SEARCH routine in SUPER. Then TSUP outputs a message indicating whether the number was in the array.

8.6.4 The CSM Library Dispatching Process

When you build the referencing task and you specify the SV argument to the RESSUP or SUPLIB option, TKB includes a 4-word context-switching vector for each call to a subroutine in the library. This has been very generally discussed in Section 8.2. This section discusses the CSM library vector in detail.

CSM mode switching occurs as follows:

1. The vector is entered with the return address on top of the stack (TOS).
2. The vector pushes the completion-routine address on the stack.
3. A CSM instruction is executed with the supervisor-mode entry point as the immediate addressing mode parameter. The CSM instruction performs the following actions:
   a. Evaluates the source parameter and stores the entry point address in a temporary register
   b. Copies the user stack pointer to the supervisor stack pointer
   c. Places the current PSW and PC on the supervisor stack, clearing the condition codes in the PSW
   d. Pushes the entry point address on the supervisor stack
   e. Places the contents of location 10 in supervisor I-space into the PC

The stack looks like this when the processor begins to execute at the contents of virtual 10 in supervisor mode:

```
user sp ---->  return address
               completion routine address
                         PSW
                         PC

super sp ---->  entry point address
```

The most important aspect of how the CSM library mode-switching vector works is that the processor begins executing at the contents of virtual 10 in supervisor mode. This is why the completion routine must be located at virtual 0, so that virtual location 10 is within the completion routine.

8.7 Converting SCAL Libraries to CSM Libraries

You can easily convert your SCAL libraries to CSM libraries. Rebuilding a task on an RSX–11M–PLUS Version 2.0 system or later that linked to a library on a Version 1.0 system requires that you rebuild the library also. Rebuild the library specifying the completion routine as the first input module. If the library was not built to run at a starting address of 0 in its partition, rebuild it to begin at 0 so that TKB can find the completion routine.

8.8 Using Supervisor-Mode Libraries as Resident Libraries

Supervisor-mode libraries can double as conventional resident libraries. For position-independent supervisor-mode libraries, you rebuild the referencing task using the RESLIB option instead of the RESSUP option. Indicate the first available user-mode APR that you want to map the library. For CSM libraries, this will always change because you cannot map a shared region with APR 0. You do not have to rebuild the library.

For absolute supervisor-mode libraries, rebuild the referencing task using the RESLIB option instead of the RESSUP option. Rebuild the library only if the beginning partition address in the PAR option is incompatible with the address limits of your referencing task.

8.9 Multiple Supervisor-Mode Libraries

A user task can reference multiple supervisor-mode CSM libraries. However, all the CSM libraries must use the completion routine that begins at virtual 0 in supervisor-mode instruction space.

8.10 Linking a Resident Library to a Supervisor-Mode Library

You can link a conventional resident library to a supervisor-mode library using the following TKB or LINK command sequence:

```
TKB>F4PRES/-HD,F4PRES,LB:[1,1]F4PRES= RET
TKB>F4PRES/LB RET
TKB>/ RET
Enter Options:
TKB>STACK=0 RET
TKB>SUPLIB=FCSFSL:SV RET
TKB>PAR=F4PRES:140000:20000 RET
TKB>// RET
>

$ LINK/TAS:F4PRES/NOH/MAP:F4PRES/SYM:LB:[1,1]F4PRES/OPT - RET
-> F4PRES/LIB RET
Option? STACK=0 RET
Option? SUPLIB=FCSFSL:SV RET
Option? PAR=F4PRES:140000:20000 RET
Option? RET
$
```

These two command sequences show you how to link F4PRES to FCSFSL.

8.11 Linking Supervisor-Mode Libraries

You cannot link supervisor-mode libraries together, and you cannot link a supervisor-mode library to a resident user-mode library. Calling a user-mode library is not possible because its code is not mapped through the I-space APRs while in the supervisor-mode library. However, you can link user-mode libraries to a supervisor-mode library.

8.12 Writing Your Own Vectors and Completion Routines

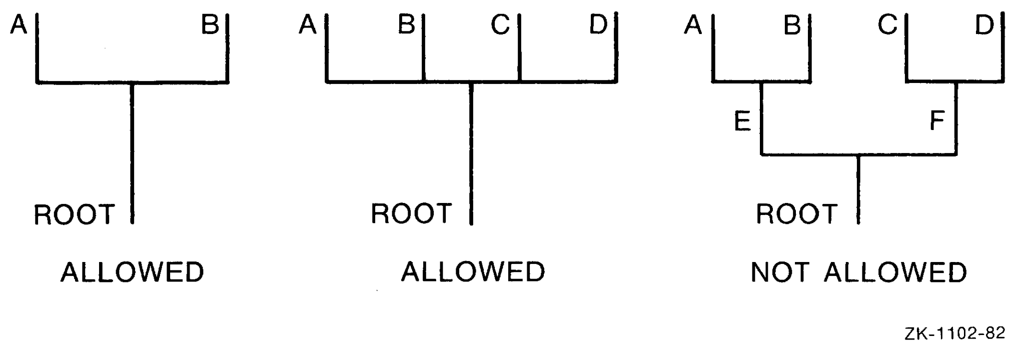
You can write your own mode-switching vectors and completion routines. This may be necessary for threaded code. If you use your own vectors, build them into the task and use the /-SV switch on the RESSUP or RESLIB option when you build the referencing task. If you create your own completion routines, write your completion routine to resemble the system-supplied completion routines (see Example 8-3) as much as possible. If you do not retain the last three lines of code as indicated in Example 8-3, the task may crash if the Executive processes an interrupt before the switch back to user mode has completed.

8.13 Overlaid Supervisor-Mode Libraries

It is possible to use overlaid supervisor-mode libraries. However, the following restrictions must be noted when building these libraries:

- The completion routine for the library must be in the root.
- Only one level of overlaying is allowed. This is illustrated in Figure 8-4.
- Although the Fast Task Builder (FTB) can link to supervisor-mode libraries, it cannot link to overlaid supervisor-mode libraries.

Figure 8-4: Overlay Configuration Allowed for Supervisor-Mode Libraries



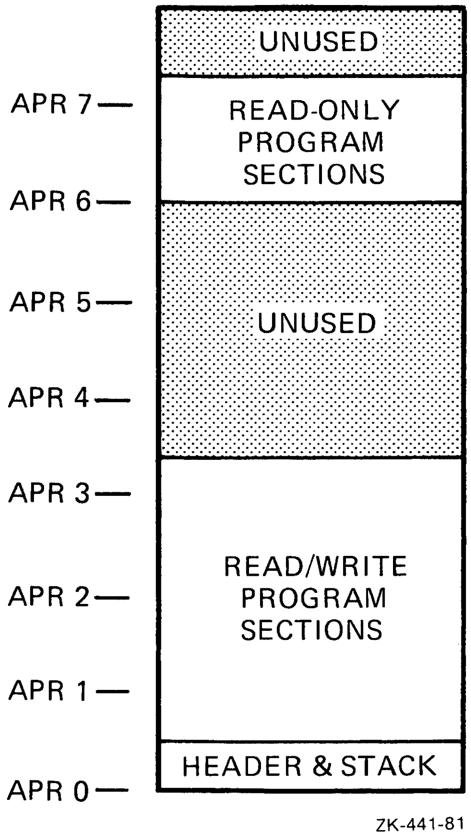
Chapter 9
Multiuser Tasks

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.

When you build a task, you designate it as multiuser 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) contains the read/write portion of the task; the other region (region 1) contains the read-only portion of the task.

As with all other tasks, TKB 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 in a single user task, however, the read-only portion of the task is hardware protected. In addition, TKB 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 9-1 illustrates this allocation.

Figure 9–1: Allocation of Program Sections in a Multiuser Task



For I- and D-space multiuser tasks, in addition to having the multiuser task divided into regions of read-only program sections and read/write program sections, these regions themselves are divided into I-space areas and D-space areas. All of the following combinations must be present in an I- and D-space multiuser task:

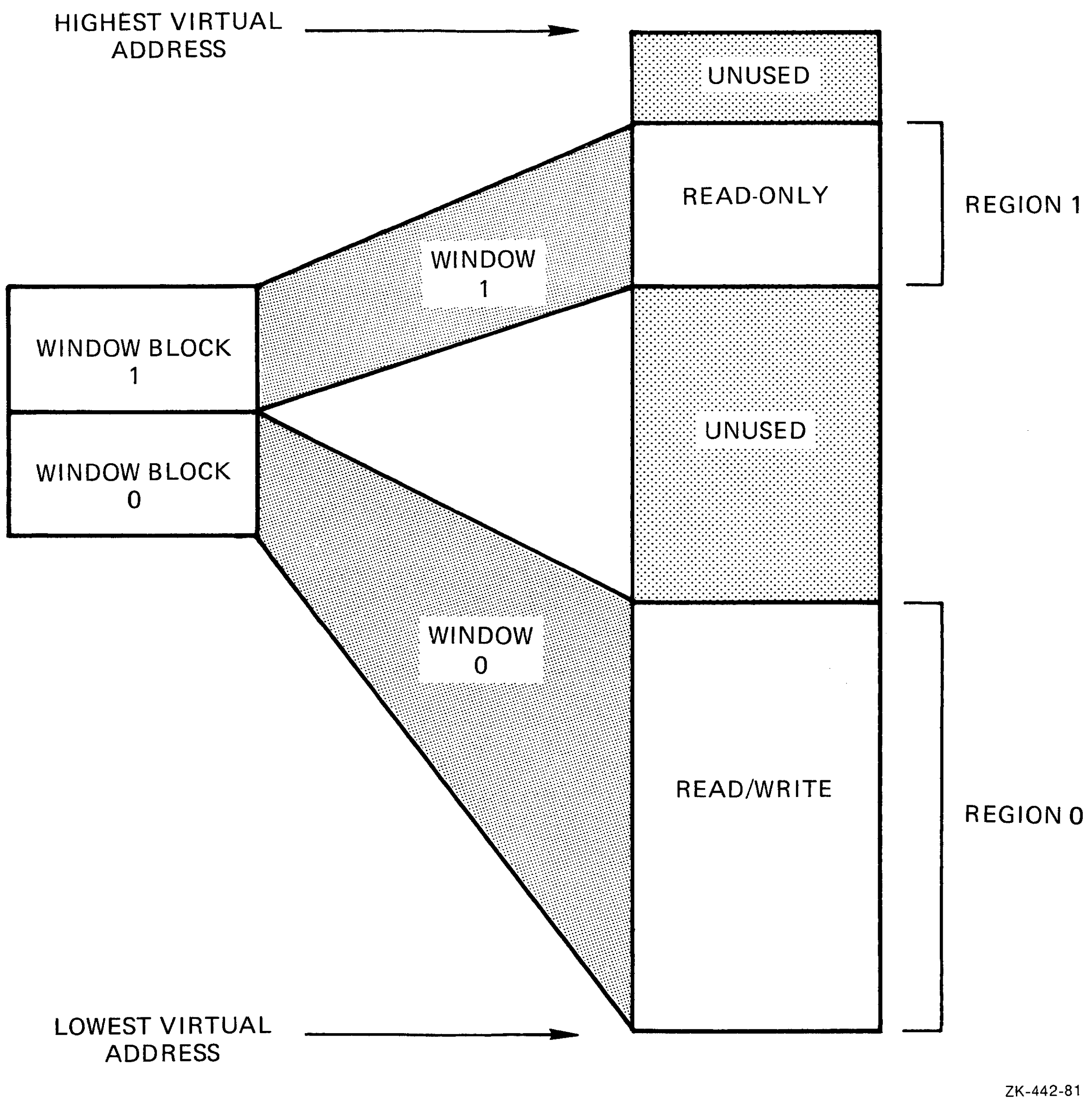
- .PSECT psectnamew, RO, I, ...
- .PSECT psectnamex, RW, I, ...
- .PSECT psectnamey, RO, D, ...
- .PSECT psectnamez, RW, D, ...

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

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

Figure 9-2 shows the window-to-region relationship of a multiuser task.

Figure 9–2: Windows for a Multiuser Task



9.1 Overlaid Multiuser Task

If a multiuser task is an overlaid task (described in Chapter 3), the read-only portion of the task can be made up of the following components:

- 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

9.2 Disk Image of a Multiuser Task

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 appear 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.)

9.3 I- and D-Space Multiuser Tasks

The APR and window block assignment in an I- and D-space multiuser task differs from that in a conventional multiuser task.

D-space APRs map the read/write and read-only program sections that have the data attribute. Similarly, I-space APRs map the read/write and read-only program sections that have the instruction attribute. Figure 9-3 shows the APR mapping for both kinds of program sections in an I- and D-space multiuser task.

TKB needs four window blocks to map an I- and D-space multiuser task. Window blocks 0 and 1 map region 0, which contains the read/write instruction and data program sections. Window blocks 2 and 3 map region 1, which contains the read-only instruction and data program sections. Figure 9-4 illustrates the mapping and assignment of these window blocks for an I- and D-space multiuser task.

9.4 Example 9-1: Building a Multiuser Task

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 5-3. 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 Example 9-1.

Figure 9–3: Example Allocation of Program Sections in an I- and and D-Space Multiuser Task

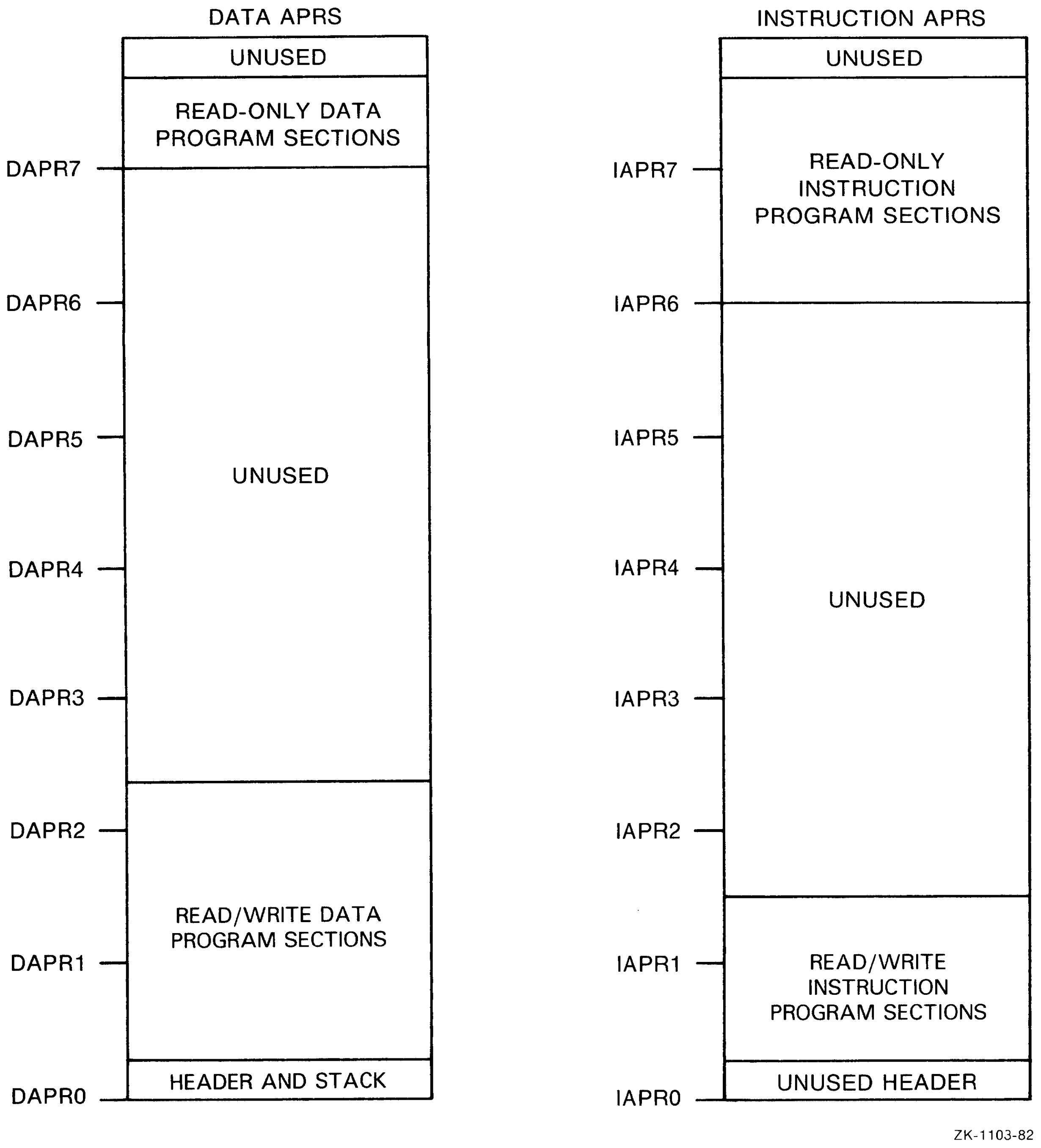
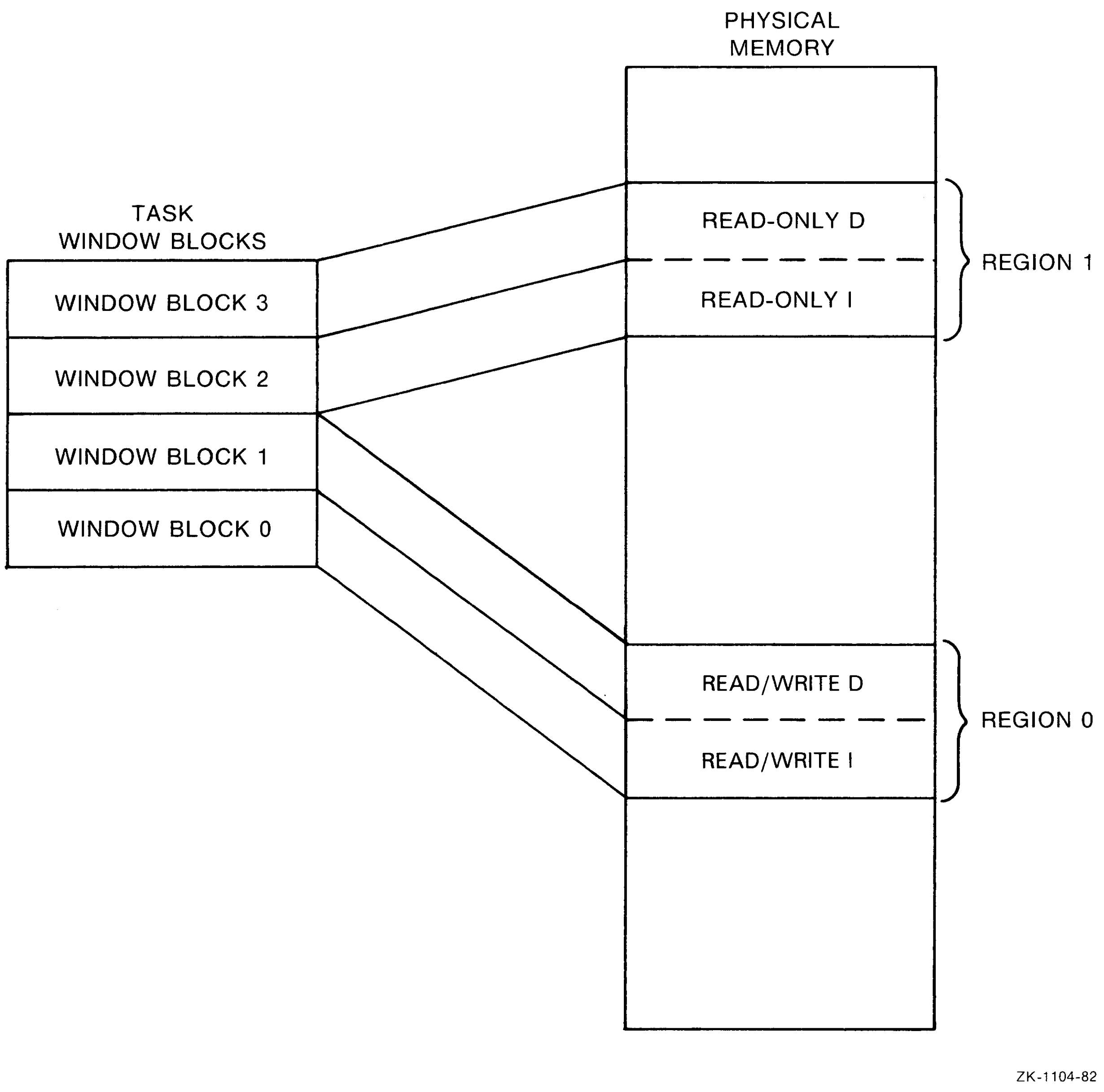


Figure 9–4: Windows for an I- and D-Space Multiuser Task



Example 9-1: Part 1, Source Listing for ROTASK.MAC

```
        .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
;+
;** 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
```

(Continued on next page)

Example 9–1 (Cont.): Part 1, Source Listing for ROTASK.MAC

```
;** FORTRAN CALLABLE SUBROUTINE TO ADD TWO INTEGERS
        .PSECT  AADD,RO,I,GBL,REL,CON

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

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

```
TKB>ROTASK/MU,ROTASK/-WI/-SP=ROTASK [RET]
TKB>/ [RET]
Enter Options:
TKB>ROPAR=RDONLY [RET]
TKB>// [RET]
>

$ LINK/TAS/SHARE:ROTASK/MAP:ROTASK/NOPRINT/NOWIDE/OPT ROTASK [RET]
Option? ROPAR=RDONLY [RET]
Option? [RET]
$
```

These two command sequences direct TKB to build a multiuser task image named ROTASK.TSK and to create an 80-column map file named ROTASK.MAP. TKB does not output a map to the line printer.

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 Example 9-1, Part 2. Note that TKB 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 TKB has allocated space for all the program sections with the read-only attribute, beginning with the highest available APR (in this case, APR 7).

Example 9-1: Part 2, Task Builder Map for ROTASK.TSK

```
ROTASK.TSK;1     Memory allocation map  TKB M43.00       PAGE 1
                       10-DEC-87   14:42

Partition name : GEN
Identification : 01
Task  UIC      : [7,62]
Stack    Limits: 000274 001273 001000 00512.
PRG xfr address: 001634
Task attributes: MU
Total address windows: 2.
Task  image  size  : 1088. words
Task address limits: 000000 004157
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 004157 004160 02160.
R-O mem  limits: 160000 160377 000400 00256.
Disk blk limits: 000002 000006 000005 00005.


Memory allocation synopsis:

Section                                          Title  Ident  File
-------                                          -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001274 002662 01458.
                          001274 000530 00344. ROTASK 01     ROTASK.OBJ;1
AADD  :(RO,I,LCL,REL,CON) 160000 000024 00020.
                          160000 000024 00020. ROTASK 01     ROTASK.OBJ;1
DDIV  :(RO,I,LCL,REL,CON) 160024 000026 00022.
                          160024 000026 00022. ROTASK 01     ROTASK.OBJ;1
MMUL  :(RO,I,LCL,REL,CON) 160052 000024 00020.
                          160052 000024 00020. ROTASK 01     ROTASK.OBJ;1
SSUB  :(RO,I,LCL,REL,CON) 160076 000024 00020.
                          160076 000024 00020. ROTASK 01     ROTASK.OBJ;1
$$RESL:(RO,I,LCL,REL,CON) 160122 000212 00138.

Global symbols:

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

*** Task builder statistics:

Total work file references: 2145.
Work  file  reads: 0.
Work  file writes: 0.
Size of core pool: 7086. words (27. PAGES)
Size of work file: 1024. words (4. PAGES)

Elapsed time:00:00:07
```

Chapter 10
TKB Switches

You use TKB switches, or LINK qualifiers, and TKB options to control the construction of your task image. This chapter provides detailed reference information on all the TKB switches. Chapter 11 describes the LINK qualifiers. Chapter 12 describes the TKB and LINK options.

10.1 TKB Switches

The following sections discuss switches as used in the syntax of file specifications, correct switch designation, switches that override other switches, a switch summary table, and the individual switches in alphabetical order.

10.1.1 File Specification Syntax

The syntax for a file specification, as given in Chapter 1, is:

```
dev:[directory]filename.type;version/sw1/sw2.../swn
```

Optionally, you can conclude a file specification with one or more switches (sw1,sw2,...swn). When you do not specify a switch, the Task Builder establishes a default setting for it.

10.1.2 Switch Designation

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 word NO, TKB negates the function of the two characters. For example, TKB 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.

10.1.3 Overriding Switches

In some cases, two particular switches cannot both be used in a file specification. When such a conflict occurs, TKB 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 [RET]
```

TKB 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.

10.1.4 Switch Summary Table

The switches that TKB recognizes are given in alphabetical order in Table 10-1. Sections 10.2 through 10.43 give detailed descriptions of each switch, including the following information:

- The switch format
- The file or files 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 10-1: TKB 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
/CL	Task is a command line interpreter.	TSK	/-CL
/CM	Memory-resident overlays are aligned on 256-word physical boundaries.	TSK	/-CM

Table 10–1 (Cont.): TKB Switches

Format	Meaning	Applies to File	Default
/CO	Causes TKB to build a shared common.	TSK STB	/CO
/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 KE11-A Extended Arithmetic Element.	TSK	/-EA
/EL	Specifies library size according to partition size.	TSK	/-EL
/FM	Tells TKB to allocate space in memory between the task and the external header for use by the fast-mapping feature of the Executive.	TSK	/-FM
/FO	Causes task to use overlay run-time system Fast Map module.	TSK	/-FO
/FP	Task uses the Floating Point Processor.	TSK	/FP
/FU	All co-tree 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
/ID	Task will use I- and D-space.	TSK	/-ID
/IP	Allows TKB to inform INSTALL that the task purposely overmaps the I/O page.	TSK	/-IP
/LB	Input file is a library file.	OLB	/-LB
/LI	Informs TKB to build a shared library.	TSK STB	/-LI
/MA	Map file includes information from the file.	MAP OBJ	/MA or /-MA[1]

[1]The default is /MA for an input file and /-MA for system library and resident library STB files.

Table 10–1 (Cont.): TKB Switches

Format	Meaning	Applies to File	Default
/MM	System on which the task is to run 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
/NM	Tells TKB to inhibit two diagnostic messages.	TSK	/-NM
/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
/SB	Task is built with the slow mode of the Task Builder.	TSK	/-SB
/SE	Messages can be directed to the task by means of the Executive directive Send.	TSK	/SE
/SG	Allocates task program sections alphabetically by access code (RW followed by RO).	TSK	/-SG
/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	Allocates task program sections in input order by access code.	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
/XH	Task is to have an external header.	TSK	[3]
/XT[:n]	TKB exits after n diagnostic errors.	TSK	/-XT

[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.

[3]The default is ultimately determined by the /XHR switch in the INSTALL command, which overrides the TKB setting except for /-XH.

10.2 /AC[:n]—Ancillary Control Processor

The /AC switch informs TKB that 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.

Format

```
file.TSK/AC:0=file.OBJ
```

This switch also informs TKB that your task is privileged. TKB 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 (APR) that you want the Executive to use to map your task's image when your task is running in user mode. Valid APRs 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 (TKB is not running in a system with KT-11 hardware), TKB merely tests the value of n for validity, but otherwise ignores it.

The default is /-AC.

Note

You should not use /AC and /PR on the same command line.

10.3 /AL—Allocate Checkpoint Space

The /AL switch informs TKB that your task is checkpointable. The system will checkpoint it to a space in your task's image file. However, the system uses the system checkpoint file first if you specified dynamic checkpointing.

Format

```
file.TSK/AL=file.OBJ
```

As well as making your task checkpointable, this switch directs TKB to allocate additional space in your task image file to contain the checkpointed task image.

The default is /-AL.

Notes

1. Do not use /AL and /CP on the same command line.
2. The /AL switch should not be used with the /-HD switch to build tasks. Examples of tasks that use the /-HD switch are the Executive, device drivers, and commons.

10.4 /CC—Concatenated Object Modules

The /CC switch controls the way TKB extracts modules from your input file.

Format

```
file.TSK=file.OBJ/-CC
```

By default, TKB includes in your task's image all the modules of your input file. If you negate this switch, TKB includes only the first module of your input file.

The default is /CC.

10.5 /CL—Command Line Interpreter

The /CL switch informs TKB that the task is a command line interpreter (CLI).

Format

```
file.TSK/CL=file.OBJ
```

Using /CL enables you to install a CLI without specifying /CLI=YES on the INSTALL command line. Use this switch when you task build the DCL task or any other CLI task. You can still install a CLI built without the /CL switch by specifying /CLI=YES when installing it.

The default is /-CL.

Note

The Fast Task Builder (FTB) does not support the /CL switch.

10.6 /CM—Compatibility Mode Overlay Structure

The /CM switch causes the Task Builder to build your task in compatibility mode.

Format

```
file.TSK/CM=file.OBJ
```

TKB aligns memory-resident overlay segments on 256-word boundaries for compatibility with other implementations of the mapping directives.

The default is /-CM.

10.7 /CO—Build a Common Block Shared Region

The /CO switch informs TKB that a shared common is being built. If you build a shared common, you should use the /CO switch and the /-HD switch.

If you use the /-PI switch for an absolute shared common, all the program sections in the common are marked absolute. Using the /-PI/-HD switches without the /CO switch causes TKB to build a shared library.

If you use the /PI switch for a relocatable shared common, all program sections in the common are marked relocatable.

In either case, the STB file contains all the program section names, attributes, length, and symbols. TKB links common blocks by means of program sections. Therefore, the STB file of a shared region built with the /CO switch contains all defined program sections.

Using the /PI/-HD switches without the /CO switch causes TKB to build a shared common.

The /CO switch cannot be negated.

Format

```
file.TSK/CO=file.OBJ
```

This switch causes TKB to include all program section declarations in the STB file.

The default is /CO when the /PI switch is used, but when the /-PI switch is used, TKB builds a shared library (/LI) instead of a shared common (/CO).

10.8 /CP—Checkpointable

The /CP switch causes TKB to mark your task as 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-PLUS MCR Operations Manual.*)

Format

```
file.TSK/CP=file.OBJ
```

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.

The default is /-CP.

Note

Using /AL also makes your task checkpointable.

10.9 /CR—Cross-Reference

The /CR switch directs TKB to add a cross-reference listing to the map file of your task.

Format

```
file.TSK,file.MAP/CR=file.OBJ
```

TKB creates a special work file (file.CRF) that 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 the *RSX-11M-PLUS Utilities Manual* for more information on CRF.)

The Example section below describes the cross-reference listing and its contents.

The default is /-CR.

Note

For this switch to be effective, CRF must be installed in your system.

Example

Example 10–1 shows a cross-reference listing for task OVR. The numbered items in the notes correspond to the numbers in the example.

Example 10–1: Cross-Reference Listing for OVR.TSK

```
CREF       CREATED BY  TKB    ON 27-JUL-87 AT 09:46     PAGE 1       ❶
GLOBAL CROSS REFERENCE                                  CREF    V01

SYMBOL  VALUE       REFERENCES...

AADD    020000-R    * AADD   @ CALC
ADDEXI  020060-R    * AADD
ARGBLK  001340-R    CALC     # MAIN
BUFF    001366-R    # MAIN   OUTPUT
CALC    003270-R    * CALC   @ MAIN
DIFR    001360-R    CALC     # MAIN
DIVEXI  020062-R    * DIVV                                               ❷
DIVR    001364-R    CALC     # MAIN
DIVV    020000-R    @ CALC   * DIVV
I       001350-R    INPUT    # MAIN
IE.EOF  177766      INPUT    # QIOSYM
INITL   005664-R    # INITL  ^ MAIN
INPUT   003364-R    * INPUT  @ MAIN
IOSB    001334-R    INPUT    # MAIN

CREF       CREATED BY   TKB    ON 27-JUL-87 AT 09:46      PAGE 2
SEGMENT CROSS REFERENCE                                     CREF   V01

SEGMENT NAME  RESIDENT MODULES

AADD    AADD
CALC    CALC
DIVV    DIVV
INPUT   ARITH   CATB    INPUT   QIOSYM   SAVRG                           ❸
LIBROT  INITL   SAVAL
MAIN    ALERR   AUTO    MAIN    OVCTR    OVDAT  OVRES  SAVR1
        VCTDF
MULL    MULL
OUTPUT  ARITH   CATB    CBTA    CDDMG    C5TA   DARITH EDDAT
        EDTMG   OUTPUT  QIOSYM  SAVRG
SUBB    SUBB
```

❶ 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.

❷ 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 is displayed more than once within each entry.

The suffix -R appears next to the value if the symbol is relocatable.

Prefix symbols accompanying each module name define the type of reference as follows:

Prefix Symbol	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 nonautoloadable definition
*	Module contains an autoloadable definition

❸ 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.

10.10 /DA—Debugging Aid

The /DA switch causes TKB to include a debugging aid in your task. The debugging aid controls the task's execution.

Format

```
file.TSK/DA=file.OBJ
```

If you apply this switch to your task image file, TKB includes the system debugging aid LB0:[1,1]ODT.OBJ into your task image. If you use the /DA switch with the /ID switch, TKB includes LB:[1,1]ODTID.OBJ in the task.

TKB passes control to the debugging program when you or the system starts task execution.

If you apply this switch to one of your input files, TKB 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. The default file type for the debugging aid is OBJ.

In either case, /DA has the following effects on your task image:

- The transfer address of the debugging aid overrides the task transfer address.
- TKB 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 Radix–50 characters representing the version of your task. TKB derives the version from the first .IDENT directive it encounters in your task. If no .IDENT directive is in your task, this value is 01.
R4	The second three Radix–50 characters representing the version of your task.

The default is /-DA.

10.11 /DL—Default Library

The /DL switch causes the input file to be a replacement for the system object module library. The default file type for the input file is OBJ.

Format

```
file.TSK=file.OBJ,file.OLB/DL
```

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. TKB 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.

The default is /-DL.

10.12 /EA—Extended Arithmetic Element

The /EA switch informs TKB that your task uses the KE11-A Extended Arithmetic Element.

Format

```
file.TSK/EA=file.OBJ
```

TKB allocates three words in your task's header for saving the state of the extended arithmetic element.

The default is /-EA.

Note

You should not use /EA and /FP on the same command line.

10.13 /EL—Extend Library

The /EL switch places the upper address limit as determined by the PAR option in the library's label block, though the actual size of the library may be smaller. This switch is useful when you build vectored libraries such as RMS, which are subject to size changes.

Format

```
file.TSK/LI/-HD/EL=file.OBJ
```

This switch specifies the maximum possible size for the library according to the size specified in the PAR option. The switch specifies a larger library virtual address range than is actually present in the library to allow RMS to map its vectored library segments.

The default is /-EL.

10.14 /FM—Fast Map

The /FM switch informs TKB that space must be allocated in memory between the task and the external header for use by the fast-mapping feature of the Executive.

Format

```
file.TSK/FM,,=file.OBJ
```

The /FM switch corresponds to the INSTALL command qualifier /FMAP=YES.

The default is /-FM.

Note

The Fast Task Builder (FTB) does not support the /FM switch.

10.15 /FO—Fast OTS

The /FO switch directs TKB to build your task with the overlay run-time system Fast Mapping module FSTMAP to map autoloaded memory-resident overlays.

Format

```
file.TSK/FO,,=file.OBJ
```

The default is /-FO.

Notes

1. The /FO switch requires the /FM and /XH switches (/XH is the default).
2. The Fast Task Builder (FTB) does not support the /FO switch.

10.16 /FP—Floating Point

The /FP switch informs TKB that your task uses the Floating Point Processor.

Format

```
file.TSK/FP=file.OBJ
```

TKB allocates 25 words in your task's header for saving the state of the Floating Point Processor. The default is /FP.

Notes

1. You should not use /FP and /EA on the same command line.
2. The /FP switch allocates space in the task header to save the floating point status if your task is context-switched. For information on changing the Task Builder's defaults, refer to Appendix F.

10.17 /FU—Full Search

The /FU switch controls the Task Builder's search for undefined symbols when it is processing modules from the default library.

Format

```
file.TSK/FU=file.ODL/MP
```

When TKB 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, TKB searches all co-tree segments for a matching definition or reference. See Sections 3.2.2 and 3.2.3 in Chapter 3 for more details.

The default is /-FU.

10.18 /HD—Header

The /HD switch directs TKB to include a header in the task image.

Format

```
file.TSK/-HD,,file.STB=file.OBJ
```

TKB 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.

The default is /HD.

Note

Do not use the /-HD switch and the /ID switch in the same build.

10.19 /ID—I- and D-Space Task

The /ID switch directs TKB to mark your task as one that uses I-space APRs and D-space APRs in user mode. TKB separates I-space program sections from D-space program sections.

Format

```
file.TSK/ID=file.OBJ
```

TKB includes the data structures in the task label block that informs the INSTALL task that the task has separate I-space and D-space.

The default is /-ID.

Note

Do not use the /-HD switch and the /ID switch in the same build.

10.20 /IP—Task Maps I/O Page

The /IP switch causes TKB to map your task to the I/O page. The /-IP switch informs TKB that the task is purposely over 12K words and does not need to be mapped to the I/O page.

Format

```
file.TSK/PR/-IP=file.OBJ
```

TKB sets a bit in the task's label block that informs the INSTALL task that the task intentionally does not map the I/O page. When this bit is set, INSTALL does not display an error message when it detects that the privileged task extends into APR 7.

The default is /IP.

10.21 /LB—Library File

The /LB switch specifies that TKB is to use an object module library file when it builds your task. The Task Builder's interpretation of this switch depends upon whether you supply arguments with the switch.

The default file type is OLB. The default form of the switch is /-LB.

Format

```
file.TSK=file.OBJ,file.OLB/LB
```

or

Format

```
file.TSK=file.OBJ,file.OLB/LB:mod-1:mod-2...:mod-8
```

The file to which the /LB switch is attached is an object module library file. If you apply this switch without arguments, TKB assumes that your input file is a library file of relocatable object modules. TKB searches the file immediately to resolve undefined references in any modules preceding the library specification. It also 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, TKB extracts from the library the modules named as arguments of the switch regardless of whether the modules contain definitions for unresolved references.

If you want TKB 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. For example:

```
file.TSK=file.OLB/LB:mod-1:mod-2,file.OLB/LB
```

The position of the library file within the TKB command sequence is important. The following rules apply:

- The library file must follow to the right of the input file or files 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 use of the /LB switch. The following command illustrates incorrect use:

  ```
  TKB>file.TSK=lib.OLB/LB,file1.OBJ
  ```

- If you are using the Task Builder's multiline input format 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
  TKB>//
  ```

- When you are building an overlay structure, 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 AO-LIB-(A1,A2-(A21,A22))

BFCTR:    .FCTR BO-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 3.)

Notes

1. You should not use the /LB switch and the /CC switch in the same command line.
2. You can use the /SS switch with the /LB switch (with or without arguments) to perform a selective search for global definitions.

10.22 /LI—Build a Library Shared Region

The /LI switch makes TKB build a shared library. However, you must use the /-HD switch with the /LI switch to build the shared library. The /LI switch cannot be negated.

Format

```
file.TSK/LI=file.OBJ
```

TKB includes only one program section declaration in the STB file.

If you use the /-PI switch for an absolute library, TKB gives the program section the same name as the library root, makes the library position dependent, and defines all symbols as absolute. Also, if you use the /-PI switch without the /LI switch, TKB assumes /LI to be the default.

If you use the /PI switch for a relocatable library, TKB names the program section the same as the root segment of the library. TKB forces this name to be the first and only declared program section in the library. TKB declares all global symbols in the STB file relative to that program section. Also, if you use the /PI switch without the /LI switch, TKB assumes that a shared common is to be built (/CO is the default).

The default is to not build a shared library unless the /-PI switch is used.

When the /-PI switch is used, TKB builds a shared library (/LI). When the /PI switch is used, TKB builds a shared common (/CO).

10.23 /MA—Map Contents of File

The /MA switch instructs TKB to include information from your input file in the memory allocation output file.

Format

```
file.TSK,file.MAP=file.OBJ,file.OBJ/-MA
```

If you negate this switch and apply it to an input file, TKB excludes from the map and cross-reference listings all global symbols defined or referred to in the file. In addition, TKB does not list the file in the "file contents" section of the map.

If you apply this switch to the map file, TKB includes in the map file the names of routines it has added to your task from the system library. It also includes in the map file information contained in the symbol definition file of any shared region to which the task refers.

The default is /MA for input files, and /-MA for system library and resident library STB files.

10.24 /MM[:n]—Memory Management

The /MM switch informs TKB whether the system on which your task is to run has memory management hardware. Specify n as the decimal numbers 28 or 30.

Format

```
file.TSK/MM[:n]=file.OBJ
```

If you use n with the /-MM switch (for an unmapped system), n specifies the highest physical address in K-words of the task or system being built. If you do not specify n with /-MM, the default highest address of the task or system is 28K.

If you specify n with /MM, n is ignored.

When you do not apply /MM or /-MM to your task image file, TKB allocates memory according to the mapping status of the system on which your task is being built. The maximum task size for a mapped system is always 32K words.

Note

When you use /-MM, TKB does not recognize the memory-resident overlay operator (!). TKB checks the operator for correct syntax, but it does not create any resident overlay segments.

10.25 /MP—Overlay Description

The /MP switch specifies that the input file is an Overlay Description Language (ODL) file.

Format

file.TSK=file.ODL/MP

TKB 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 Chapter 1), TKB requests option information at the console terminal by displaying the following prompt:

```
Enter Options:
```

The default is /-MP.

Notes

1. If you use the multiline command format in TKB when you specify an ODL file, TKB automatically prompts for option input. Therefore, you must not use the single slash (/) to direct TKB 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. The default file type is ODL.

10.26 /MU—Multiuser

The /MU switch specifies to TKB that the task is a multiuser task.

Format

```
file.TSK/MU=file.OBJ
```

TKB 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. Multiuser tasks are described in Chapter 9.

The default is /-MU.

10.27 /NM—No Diagnostic Messages

The /NM switch controls the displaying of diagnostic messages.

Format

file.TSK/NM=file.OBJ

The /NM switch prevents the following messages from being displayed:

```
n Undefined symbols segment seg-name
```

and

```
Module module-name multiply defines P-section p-sect-name
```

The default is /-NM.

10.28 /PI—Position Independent

The /PI switch informs TKB that the task's shared region contains only position-independent code or data. Use this switch with /-HD and either /CO or /LI.

Format

```
file.TSK/PI=file.OBJ
```

or

Format

```
file.TSK,,file.STB/PI=file.OBJ
```

TKB sets the position-independent code (PIC) attribute flag in the label block flag word of the shared region.

Be aware that if you specify /PI without using the /CO or /LI switch, TKB builds a shared common (/CO default). If you specify /-PI without using the /CO or /LI switch, TKB builds a shared library (/LI default).

The default is /-PI.

10.29 /PM—Postmortem Dump

The /PM switch instructs the system to automatically list the contents of the memory image if your task terminates abnormally.

Format

```
file.TSK/PM=file.OBJ
```

TKB sets the postmortem dump flag in your task's label block flag word.

The default is /-PM.

Notes

1. If your task issues an ABRT$ (Abort Task) directive, the system will not dump the task image even though TKB has set the postmortem dump flag in your task's label block 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 task must be installed in your system and be able to get into physical memory for this switch to be effective.

10.30 /PR[:n]—Privileged

The /PR switch informs TKB that 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. Privileged tasks are described in Chapter 6.

Format

```
file.TSK/PR:n=file.OBJ
```

TKB 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 (APR) that you want the Executive to use to map your task image when your task is running in user mode. Valid APRs are 0, 4, and 5. If you do not specify one of these values, TKB 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 implied (by the presence of KT-11 hardware on the system upon which TKB is running), TKB merely tests the value (:n) of the switch for validity; otherwise, TKB ignores it.

The default is /-PR.

Note

You should not use /PR and /AC on the same command line.

10.31 /RO—Resident Overlay

The /RO switch enables TKB's recognition of the memory-resident overlay operator (!).

Format

```
file.TSK/-RO=file.ODL/MP
```

The memory-resident overlay operator (!), when present in the overlay description file, indicates to TKB that it is to construct a task image that contains one or more memory-resident overlay segments. If you negate this switch, TKB checks the operator for correct syntactical usage, but otherwise ignores it. With the memory-resident overlay operator thus disabled, TKB builds a disk-resident overlay from the overlay description file.

The default is /RO.

10.32 /SB—Slow Task Builder

The /SB switch causes a task to be built with the slow mode of the Task Builder. The slow mode and the default Task Builder are included together in one image. Use the /SB switch to select the slow mode of TKB.

You should use the slow mode only if a task build produced the following message:

```
No virtual memory storage available
```

Format

file.TSK/SB,,=file.OBJ

/SB causes the slow mode of the Task Builder to be used. The default Task Builder and the Fast Task Builder use a symbol table structure that can be searched quickly, but that requires more work-file space than the slow mode. If you receive the error message shown, you have the choice of reducing work-file size, as described in Appendix F of this manual, or of using the slow mode.

Except for the use of different symbol table structures, the default Task Builder and the slow mode are identical.

The default is /-SB.

10.33 /SE—Send

The /SE switch determines whether messages can be directed to your task by means of the following Executive directives:

- Send Data (SDAT$)
- Send, Request, and Connect (SDRC$)
- Send Data Request and Pass Offspring Control Block (SDRP$)
- Variable Send Data (VSDA$)
- Variable Send, Request, and Connect (VSRC$)

Refer to the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual* for information on these directives.

Format

```
file.TSK/-SE=file.OBJ
```

By default, messages can be directed to your task by means of one of the Send directives. If you negate this switch, the system inhibits the queuing of messages to your task.

The default is /SE.

10.34 /SG—Segregate Program Sections

The /SG switch allocates virtual address space to all read/write (RW) program sections and then to all read-only (RO) program sections.

Format

```
file.TSK/SG=file.OBJ
```

The /SG switch gives you control over the ordering of program sections. By using the /SG switch, you cause TKB to order program sections alphabetically by name within access code (RW followed by RO). If you specify the /SQ switch with the /SG switch, TKB orders program sections in their input order by access code. See the description of the /SQ switch.

You use the negated switch, /-SG, to make TKB interleave the RW and RO program sections. Thus, the combination /-SG/SQ results in a task with its program sections allocated in input order and its RW and RO sections interleaved. Additionally, you can use /-SQ/-SG to make TKB order program sections alphabetically with RW and RO sections interleaved. However, /-SG is the default.

When building multiuser tasks, the /MU switch causes TKB to default to /SG. Therefore, to correctly build read-only tasks, you can use the /MU switch only.

The default is /-SG.

10.35 /SH—Short Map

The /SH switch instructs TKB to produce the short version of the memory allocation file.

Format

```
file.TSK,file.MAP/SH=file.OBJ
```

TKB does not produce the "file contents" section of the memory allocation file.

The default is /SH.

Example

The memory allocation file consists of the following items:

- Page header
- Task attributes section
- Overlay description (if applicable)
- Root segment allocation
- Tree segment description (if applicable)
- Undefined references (if applicable)
- Task Builder statistics

An example of the memory allocation file (map) is shown in Example 10–2. The numbered and lettered items in the notes correspond to the numbers and letters in the example.

Example 10–2: Memory Allocation File (Map) Example

```
OVR.TSK;1     Memory Allocation Map  TKB M43.00       Page 1          ❶
                        3-JUN-87   11:28

Partition name  : GEN [a, b]
Identification  : 01  [c]
Task  UIC       : [303,3] [d, e]
Stack     limits: 000260 001257 001000 00512. [f]

Prg xfr address: 001264 [g, h, i]                                     ❷
Total address windows: 1. [j, k, l]
Task  image  size  : 7488. words [m, n]
Task address limits: 000000 035133 [o]
R-W disk blk limits: 000002 000073 000072 00058. [p, q]

OVR.TSK Overlay description:

Base    Top         Length
----    ---         ------
000000  005055  005056  02606.  ROOTM
005060  021103  014024  06164.  MULOV                                 ❸
005060  021103  014024  06164.  ADDOV
021104  035127  014024  06164.  SUBOV
021104  035131  014026  06166.  DIVOV

OVR.TSK      Memory allocation map  TKB M43.00       Page 2
ROOTM                   3-JUN-87   11:28

*** Root segment: ROOTM [a]

R/W mem  limits: 000000 005055 005056 02606. [b]
Disk blk limits: 000002 000007 000006 00006. [c]

Memory allocation synopsis:                                           ❹

Section                [d]                         Title  Ident  File
-------                                            -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001260 001174 00636.
                          001260 000102 00066.     ROOTM  01 [e] ROOTM.OBJ;1
                          001362 000260 00176.     PRNOV  01     PRNOV.OBJ;1
                          001642 000042 00034.     SAVOV  01     SAVOV.OBJ;1
```

(Continued on next page)

Example 10–2 (Cont.): Memory Allocation File (Map) Example

```
ANS   :(RW,D,GBL,REL,OVR) 002454 000002 00002.
                          002454 000002 00002.  ROOTM  01     ROOTM.OBJ;1
                          002454 000002 00002.  PRNOV  01     PRNOV.OBJ;1
                             .
                             .
                             .
Global symbols:

AADD    004032-R  DIVV  004052-R  PRINT  001550-R  SUBB   004042-R
                  MULL  004022-R  SAVAL  001642-R            [f]

AADD    004032-R  DIVV  004052-R  PRINT  001550-R  SUBB   004042-R
                  MULL  004022-R  SAVAL  001642-R
                           .
                           .
                           .
File: ROOTM.OBJ;1  Title: ROOTM   Ident: 01  [g]
<. BLK.>: 001260 001361 000102 00066. [h]
<ANS   >: 002454 002455 000002 00002.

File: PRNOV.OBJ;1  Title: PRNOV   Ident: 01  [g]
<. BLK.>: 001362 001641 000260 00176. [h]
   PRINT  001550-R  [i]
<ANS   >: 002454 002455 000002 00002.

File: SAVOV.OBJ;1  Title: SAVOV   Ident: 01  [g]      [j]     [k]
<. BLK.>: 001642 001703 000042 00034. [h]
   SAVAL  001642-R  [i]
```

❹

```
OVR.TSK       Memory allocation map  TKB M43.00     Page 4
MULOV                     3-JUN-87   11:28


*** Segment: MULOV


R/W mem  limits: 005060 021103 014024 06164.
Disk blk limits: 000010 000024 000015 00013.

Memory allocation synopsis:

Section                                         Title  Ident  File
-------                                         -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 005060 014024 06164.
                          005060 014024 06164.  MULOV  01     MULOV.OBJ;1
$$ALVC:(RW,I,LCL,REL,CON) 021104 000000 00000.
$$RTS :(RW,I,GBL,REL,OVR) 004756 000002 00002.
```

❺

(Continued on next page)

Example 10-2 (Cont.): Memory Allocation File (Map) Example

```
Global symbols:                                                    ┐
                                                                   │
MULL   021060-R                                                    │
                                                                   │ ❺
                           .                                       │
                           .                                       │
                           .                                       ┘

*** Task builder statistics:                                       ┐
                                                                   │
    Total work file references: 8570.    [a]                       │
    Work  file  reads: 0.   [b,                                    │
    Work  file writes: 0.       c]                                 │ ❻
    Size of core pool: 6662. words (26. pages)  [d]                │
    Size of work file: 3328. words (13. pages)  [e]                │
                                                                   │
    ELAPSED TIME:00:00:14                                          ┘
```

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 TKB that created the map.

❷ The task attributes section contains the following information:

a. Task name—The name specified in the TASK option. If you do not use the TASK option, TKB suppresses this field.

b. Partition name—The partition specified in the PAR option. If you do not specify a partition, the default is partition GEN.

c. Identification—The task version as specified in the .IDENT assembler directive. If you do not specify the task identification, the default is 01.

d. Task UIC—The task UIC as specified in the UIC option. If you do not specify the UIC, the default is the terminal UIC.

e. Task priority—The priority of the task as specified in the PRI option. If you do not specify PRI, the default is 50 and is not shown on the map.

f. Stack limits—The low and high octal addresses of the stack, followed by its length in octal and decimal bytes.

g. ODT transfer address—The starting address of the ODT debugging aid. If you do not specify the ODT debugging aid, this field is suppressed.

h. 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, TKB automatically establishes a transfer address of 000001 for it. TKB also suppresses this field in the map if you do not specify a transfer address.

i. Task attributes—These attributes are listed only if they differ from the defaults. One or more of the following attributes 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's initial PSW has T-bit enabled.
ID	Task is I- and D-space task.

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 Appendix B); mapped array is not shown if it does not apply.

l. Task extension—The increment of physical memory (decimal words) allocated through the EXTTSK or PAR option. Without these options, task extension is not shown.

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. Total task size is not shown in this example.

o. Task address limits—The lowest and highest virtual addresses allocated to the task, exclusive of virtual addresses allocated to virtual program sections 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; and 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; and 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.

❸ 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. TKB prints the overlay description only when an overlaid task is created.

❹ 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); and 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; and 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 the system library. (For more information, see the description of the /MA switch in Section 10.23.)

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 is always of zero length.

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 the 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 TKB command sequence. TKB 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, and 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, and 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 3), this value is the address of an autoload vector. The autoload vector in turn contains the actual address of the symbol.

   A -R is appended to the value if the symbol is relocatable.

j. Program section—The contents of this field is 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 TKB performance:

a. Work file references—The number of times that TKB 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 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 required to process the overlay description, parse the list of input file names, and create the cross-reference listing (if specified).

See Appendix F for a more detailed discussion of the work file.

10.36 /SL—Slave

The /SL switch directs TKB to mark your task as a slave to an initiating task.

Format

```
file.TSK/SL=file.OBJ
```

TKB 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 identity and privileges of the sending task.

For RSX-11M-PLUS systems, refer to your system generation manual for more information on multiuser protection and slave tasks.

The default is /-SL.

10.37 /SP—Spool Map Output

The /SP switch determines whether TKB calls the print spooler to print your memory allocation (map) file after the task is built.

Format

```
file.TSK,file.MAP/-SP=file.OBJ
```

By default, when you specify a map file in a TKB command sequence, TKB creates a map file on device SY0 and then queues the file to be printed on LP0 if the system task QMGPRT.TSK is installed with the PRT... name.

If you negate this switch, TKB creates the map file on device SY0 but does not call the print spooler to output it to LP0.

The default is /SP.

Note

The PRT task must be installed to process the request to print the map.

10.38 /SQ—Sequential

The /SQ switch causes TKB to construct your task image from the program sections you specified, in the order that you input them.

Format

```
file.TSK/SQ=file.OBJ
```

If you use this switch, TKB collects all the references to a given program section from your input object modules, groups them according to their access code (RW followed by RO) and, within these groups, allocates memory for them in the order that you input them. However, the /SG switch also affects program-section ordering and can be used with the /SQ switch. See the /SG switch for further details.

Without the /SQ switch, TKB reorders the program sections alphabetically.

You use this switch to satisfy any adjacency requirements that existing code may have when you are converting it to run under RSX. Using this feature is otherwise discouraged for the following reasons:

- Standard library routines (such as FORTRAN I/O handling routines and FCS modules from the system library) do 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.

The default is /-SQ.

10.39 /SS—Selective Search

The /SS switch directs TKB to include in its internal symbol table only those global symbols for which there is a previously undefined reference.

Format

```
file.TSK=file.OBJ,file.OBJ/SS
```

or

Format

```
file.TSK=file.OBJ,file.STB/SS
```

or

Format

```
file.TSK=file.OBJ,file.OLB/LB/SS
```

When processing an input file, TKB normally includes in its internal symbol table each global symbol it encounters within the file whether or not there are references to it. With the /SS switch attached to an input file, TKB checks each global symbol it encounters within that file against its list of undefined references. If TKB finds a match, it includes the symbol in its symbol table.

The default is /-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 10-2.

Table 10–2: Input Files for SEL.TSK

Input File Name	Global Definition	Global Reference
IN1		A
	A	
IN2	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 TKB to resolve the reference to global symbol A in IN1 to the definition for A in IN2. Assume further that you want TKB 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, TKB resolves the references correctly. The following command line illustrates the correct sequence:

```
TKB>SEL.TSK=IN1.OBJ,IN2.OBJ/SS,IN3.OBJ,IN4.OBJ/SS RET
```

TKB processes input files from left to right; therefore, in processing the previous command line, TKB processes file IN1 first and encounters the reference to symbol A. There is no definition for A within IN1; therefore, TKB marks A as undefined and moves on to process file IN2. Because the /SS switch is attached to IN2, TKB limits its search of IN2 to symbols it has previously listed as undefined, in this case, symbol A. TKB finds a definition for A and places A in its symbol table. Because there are no undefined references to symbols B or C, TKB does not place either of these symbols in its symbol table.

Note

It is important to realize that the /SS switch affects only the way the Task Builder constructs its internal symbol table. The routines for which symbols B and C are entry points are included in the task image even though there are no references to them.

TKB moves on to IN3. It encounters the references to symbol C. Because TKB did not include symbol C from IN2 in its symbol table, it cannot resolve the reference to C in IN3. TKB marks symbol C as undefined and moves on to IN4.

When TKB 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, TKB includes only C in its symbol table.

When TKB has completed its processing of the above command line, 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 TKB 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, TKB includes in its internal symbol table only those symbols in the Executive that the task references. An example of a TKB command line that illustrates this is as follows:

```
TKB>OUTFILE.TSK/PR:5=INFILE.OBJ,RSX11M.STB/SS RET
```

This command line directs TKB 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 TKB to extract from RSX11M.STB only those symbols for which there are references within OUTFILE.TSK.

10.40 /TR—Traceable

The /TR switch directs TKB to make your task traceable.

Format

```
file.TSK/TR=file.OBJ
```

TKB sets the T-bit in the initial PSW of your task. When your task is executed, a trace trap occurs when each instruction is completed.

The default is /-TR.

10.41 /WI—Wide Listing Format

The /WI switch controls the width of your map file.

Format

```
file.TSK,file.MAP/-WI=file.OBJ
```

By default, TKB formats a map file 132 columns wide. When you negate this switch, TKB makes the map file 80 columns wide.

The default is /WI.

10.42 /XH—External Header

The /XH switch informs TKB that the task will have an external header. The /-XH switch prevents the task from having an external header.

Format

```
file.TSK/-XH= file.OBJ
```

The effect of the /XH switch is twofold: the header space in the task image is not destroyed when the task is checkpointed, and Executive pool space is conserved. A task built with the /XH switch does not have a header in Executive pool, but has a copy of its header, which the Executive uses, in space allocated physically contiguous to and below the task image. When the task is checkpointed, the system writes the entire task image and the header copy below the task into a checkpoint file. The header in the task image is left unchanged.

Note that if the task is also built with the /FP switch, the floating-point save area is not included in the task image but is in the header copy found below the task image.

The INSTALL keyword /XHR interacts with the /XH switch. If you use /-XH, the task will always have a pool-resident header unless you rebuild the task to have an external header. If you use /XH, the task will have an external header, but the INSTALL keyword /XHR can override this. The default use of /XH by TKB is /XH (external header) unless this is changed by the INSTALL command.

The default is /XH; overridden by INSTALL/XHR.

10.43 /XT[:n]—Exit on Diagnostic

The /XT switch specifies the number of acceptable errors. More than n errors are not acceptable.

Format

```
file.TSK/XT:n=file.OBJ
```

TKB exits after encountering n errors. The number of errors can be specified as a decimal or octal number, using the following conventions:

n.	Indicates a decimal number (the decimal point must be included).
#n or n	Indicates an octal number.

If you do not specify n, TKB assumes that n is 1.

The default is /-XT.

Chapter 11

LINK Qualifiers

You use LINK qualifiers in the command line when building a task to specify the characteristics of the task or the input files. This chapter assumes a standard format for the LINK command line for the sake of clarity. However, many command qualifiers can be used also as qualifiers in the input filespec section of the LINK command and still create the required characteristics in the task. A few qualifiers must be used on input filespecs and can be used nowhere else.

11.1 Using the LINK Qualifiers

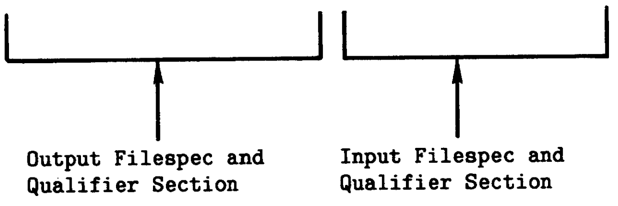
The following sections describe the syntax and use of LINK qualifiers. Section 11.1.1 describes the standard format for the LINK command line to be presented in this chapter. Section 11.1.2 describes the appropriate specification of and abbreviation of LINK qualifiers. Section 11.1.3 describes qualifiers that override other qualifiers. Section 11.1.4 contains Table 11-1, which lists all the qualifiers alphabetically, summarizes their use, shows what they affect, and shows their defaults.

The individual descriptions of each qualifier begin at Section 11.2.

11.1.1 LINK Command Line Syntax

For the sake of brevity, consistency, and clarity, the LINK command is described as having two sections: an output filespec and output qualifier section, and an input filespec and input qualifier section. Those qualifiers that can be used as output qualifiers will be referred to as command qualifiers. Those qualifiers that must be used as input filespec qualifiers will be so designated. The LINK command can thus be shown as follows:

```
$ LINK/OUTFILESPEC(S)/QUALIFIERS INFILESPECS/QUALIFIERS
```



The principal parts of the LINK output filespec section are as follows:

- LINK, the command itself
- One, two, or all three of the following output file designators, with or without filespecs: /TASK:filespec, /MAP:filespec, and /SYMBOL_TABLE:filespec
- One or more output file qualifiers or command qualifiers

The LINK input filespec section consists of one or more input filespecs and input file qualifiers.

If you do not specify a name for the output file, the LINK command, by default, produces an output task image named the same as the first input file. You can use /TASK:filename to specify a specific name, or /NOTASK to eliminate producing the task image. For example, no task is required if you want only a map of the task.

The /MAP qualifier produces a map file. The map file may be named by using /MAP:filename; otherwise the name of the map file will be the same as the first input file. You may use /MAP either as a command qualifier or as an input filename qualifier. If you specify /MAP, TKB spools the map file to the printer. If you do not want spooling, you can use the /NOPRINT qualifier to prevent it.

The /SYMBOL_TABLE qualifier (/SYM) must be specified to produce a symbol definition file (STB). You may name the file by using /SYM:filename.

The input filespec and qualifier section of the LINK command consists of an input file specification or specifications, separated by commas. And, if necessary, there are one or more qualifiers following any of the file names. For example:

```
$ LINK/TAS/MAP/SYM INFILE1,INFILE2/QUALIFIER,INFILE3 RET
```

Some qualifiers can operate correctly whether used as LINK command qualifiers or input filespec qualifiers. Others must be used as input filespec qualifiers; these are as follows:

/CONCATENATE
/DEFAULT_LIBRARY
/GLOBALS
/INCLUDE
/LIBRARY
/SELECTIVE_SEARCH

Those qualifiers that must be used on an input filespec are so designated in Table 11-1 (Section 11.1.4) and in the detailed qualifier description following Section 11.2. Use the others as LINK command qualifiers.

11.1.2 Qualifier Designation

You specify a qualifier by using a slash (/) followed by the qualifier name or its acceptable abbreviation. For example, in specifying a library, you would use /LIBRARY as an input file qualifier. Alternatively, you could use /LIB as the abbreviated form.

To use the negated version of a qualifier, if it has one, use the slash (/), followed by NO, followed by the qualifier name. For example, to prevent TKB from producing a task, you would use /NOTASK as a LINK command qualifier. The abbreviated form is /NOTAS.

As previously stated, LINK qualifiers may be abbreviated. In general, use the first three letters of the qualifier name to abbreviate a qualifier. However, be careful when you abbreviate LINK qualifiers that you use enough letters to make the abbreviation distinct from another possible abbreviation. For example, LINK will give you the message that /PRI is ambiguous because /PRI may be the abbreviation of /PRINT or /PRIVILEGED. To abbreviate these two qualifiers you must use either /PRIN or /PRIV. Appropriate abbreviations are listed in the Qualifier Summary Table in Section 11.1.4.

11.1.3 Overriding Qualifiers

In some cases, the use of two particular qualifiers may be illogical or redundant and should not be used together. In these cases, TKB selects one of the qualifiers to override the other. These qualifiers are as follows:

Qualifier	Used With	Overriding Qualifier
/ANCILLARY_PROC:n	/PRIVILEGED:n	/ANCILLARY_PROC:n
/CODE:FPP	/CODE:EAE	/CODE:FPP
/LIBRARY	/CONCATENATE	/LIBRARY

11.1.4 Qualifier Summary Table

The qualifiers used by LINK and passed to TKB are given in alphabetical order in Table 11-1. The qualifiers are described in their positive form. For example, /[NO]HEADER is followed by a description of /HEADER, its positive form. If a qualifier can be negated (as in /NOHEADER), the negative form produces the opposite effect. Some qualifiers have no negative form; to refrain from producing the effect described for one of these qualifiers, simply do not use the qualifier.

Please read the table carefully. Defaults for qualifiers may be the negative form, the positive form, or no qualifier at all (Not /...). The default for each qualifier is shown with its acceptable abbreviation.

A summary of the LINK qualifiers in Table 11-1 follows next.

Table 11-1: Link Qualifiers

Format	Meaning	File Affected	Default and Abbreviation
/ANCILLARY_PROCESSOR[:n]			
	Task is an Ancillary Control Processor and privileged.	TSK	Not /ANC
/BASIC			
	Input file is a command file created by BASIC-PLUS-2.	Input file	Not /BAS
/[NO]CHECKPOINT:SYSTEM			
	Specifies a checkpointable task and its checkpoint location on the system checkpoint file. /CHECKPOINT and /CHECKPOINT:SYSTEM are equivalent.	TSK	/NOCHE:SYS
/[NO]CHECKPOINT:TASK			
	Specifies a checkpointable task and its checkpoint location in added space in the task image on disk.	TSK	/NOCHE:TAS
/CODE:CLI			
	Specifies that task is a command line interpreter.	TSK	Not /CODE:CLI
/CODE:DATA_SPACE			
	Specifies an I- and D-space task.	TSK	Not /COD:DAT
/CODE:EAE			
	Specifies that the task uses the KE11-A Extended Arithmetic Element.	TSK	Not /COD:EAE
/CODE:FAST_MAP			
	Specifies that space must be allocated in memory between the task and external header for use by the fast-mapping feature of the Executive.	TSK	Not /CODE:FAST_MAP
/CODE:FPP			
	Specifies that the task uses the Floating Point Processor.	TSK	/COD:FPP

Table 11–1 (Cont.): Link Qualifiers

Format	Meaning	File Affected	Default and Abbreviation
/CODE:OTS_FAST			
	Specifies that the overlay run-time system (OTS) fast-mapping module FSTMAP be included in the task.	TSK	/Not /CODE:OTS_FAST
/CODE:PIC			
	Specifies that the task contains relocatable code. Usually used for a library or common with /NOHEAD.	STB TSK	Not /COD:PIC
/CODE:POSITION_INDEPENDENT			
	Same as /CODE:PIC	TSK STB	Not /COD:POS
/COMPATIBLE			
	Specifies alignment of overlay segments so as to be compatible with memory management directives.	TSK	Not /COM
/[NO]CONCATENATE			
	Specifies that all modules of the input file be concatenated. /NOCON specifies that only the first object module of the input file be used. Applicable only to an input file.	OBJ	/CON
/CROSS_REFERENCE			
	Specifies that a list of cross-referenced global symbols be appended to the map file.	MAP	Not /CRO
/DEBUG[:filepsec]			
	Specifies that the task is to contain the system default debugger. An object file (user-written debugger) may be named.	TSK OBJ	Not /DEB
/DEFAULT_LIBRARY			
	Specifies a library to be searched for unresolved global symbols other than the system library. Applicable only to an input file.	Input file OLB	Not /DEF
/ERROR_LIMIT[:n]			
	Specifies the number of errors at which TKB exits.	TSK	Not /ERR

Table 11-1 (Cont.): Link Qualifiers

Format	Meaning	File Affected	Default and Abbreviation
/[NO]EXECUTABLE[:filespec]			
	Specifies that TKB produce a task. The task may be named by filespec. /NOEXECUTABLE specifies that no task be built. Synonym for /TASK.	TSK	/EXE or /TAS
/[NO]EXTERNAL			
	Specifies that the task be built with an external header.	TSK	/EXT
/FAST			
	Specifies that the Fast Task Builder (FTB) be used.	All	Not /FAS
/FULL_SEARCH			
	Specifies that TKB search all co-tree overlay segments for matching symbols when processing the default object module library.	TSK	Not /FUL
/[NO]GLOBALS			
	Specifies that global symbols in the input file be included in the map file. Applicable only to an input file.	OBJ	/GLO
/[NO]HEADER			
	Specifies that the task contains a header. /NOHEADER specifies a task without a header for a library or common.	TSK	/HEA
/INCLUDE:(module1[,module2,...,modulen])			
	Includes the specified modules from a library file. At least one must be specified. See also /LIBRARY. Applicable only to an input file.	OLB	Not /INC
/[NO]IO_PAGE			
	Specifies that the task is to be mapped to the I/O page. Use with the /PRIVILEGED qualifier.	TSK	/IO

Table 11-1 (Cont.): Link Qualifiers

Format	Meaning	File Affected	Default and Abbreviation
/LIBRARY			
	Specifies that the input file is an object module library. Applicable only to an input OLB file. The OLB file on which /LIB is used should be the last one, if other input files are also specified.	OLB	Not /LIB
/LONG			
	Specifies that the file contents section of the input file is to be included in the task map.	MAP	Not /LON
/MAP[:filespec]			
	Specifies that TKB must produce a map file for the task. A short map is produced (not /LONG). The map may be named with filespec.	MAP	Not /MAP
/[NO]MEMORY_MANAGEMENT[:n]			
	Specifies that TKB build a task with the specified mapping for the system on which the task is to be run. Specify n as the highest physical task address on systems without memory management.	TSK	Mapping status of building system
/OPTION[:filespec]			
	Specifies to DCL that LINK is to prompt you for options. TKB options are described in Chapter 12.	TSK	Not /OPT No options allowed
/OVERLAY_DESCRIPTION			
	Specifies that the single input file contains the Overlay Description Language.	ODL	Not /OVE
/POSTMORTEM			
	Specifies that a postmortem dump is taken if the task terminates abnormally.	TSK	Not /POS
/[NO]PRINTER			
	Specifies that the MAP file is to be spooled to the line printer.	MAP	/PRIN

Table 11-1 (Cont.): Link Qualifiers

Format	Meaning	File Affected	Default and Abbreviation
/PRIVILEGED[:n]			
	Specifies that TKB is to build a privileged task. Specify the kind of privilege and mapping with n.	TSK	Not /PRIV
/[NO]RECEIVE			
	Specifies that the task may receive messages by the Executive directive Send.	TSK	/REC
/[NO]RESIDENT_OVERLAYS			
	Specifies that TKB recognize the memory-resident overlay operator in the ODL file. Use with the /OVERLAY_DESCRIPTION qualifier.	TSK	/RES
/SAVE			
	Saves the indirect command file that DCL uses to pass commands to TKB when you use LINK. The file is named ATLNK.CMD.	CMD	Not /SAV
/[NO]SEGREGATE			
	Segregates program sections by access code. Read/write followed by read-only. Interacts with the /SEQUENTIAL qualifier. See the detailed description.	TSK	/NOSEG
/SELECTIVE_SEARCH			
	Includes in the TKB internal symbol table only those symbols that match those in the TKB list of undefined references. Applicable only to an input file.	OBJ	Not /SEL
/SEQUENTIAL			
	Specifies that TKB allocate program sections in input order. Interacts with the /SEGREGATE qualifier. See the detailed description.	TSK	Not /SEQ
/SHAREABLE[:COMMON]			
	Specifies to TKB that a shared common be built. Use with the /NOHEADER qualifier. Interacts with /CODE:PIC qualifier. See detailed description.	TSK	Varies; see description

Table 11-1 (Cont.): Link Qualifiers

Format	Meaning	File Affected	Default and Abbreviation
/SHAREABLE[:LIBRARY]			
	Specifies to TKB that a shared library be built. Use with the /NOHEADER qualifier. Interacts with the /CODE:PIC qualifier. See detailed description.	TSK	Varies: see description
/SHAREABLE[:TASK]			
	Specifies that TKB build a multiuser task.	TSK	Not /SHA:TAS
/SLAVE			
	Specifies that this task is a slave task to a task that initiates it.	TSK	Not /SLA
/SLOW			
	Invokes the slow mode of the Task Builder.	TSK	Not /SLO
/SYMBOL_TABLE[:filespec]			
	Specifies that TKB produce a symbol table file (STB).	STB	Not /SYM
/[NO]SYSTEM_LIBRARY_DISPLAY			
	Specifies that TKB include global symbols in the map defined or referenced by the task. It also includes names of routines from the system library and symbols from shared regions to which the task refers.	MAP	/NOSYS
/[NO]TASK[:filespec]			
	Specifies that a task be built. Specify a name with filespec.	TSK	/TAS
/TKB			
	Invokes the default Task Builder.		/TKB
/TRACE			
	Causes the task to be traceable. A trace trap occurs after each instruction.	TSK	Not /TRA
/[NO]WARNINGS			
	Suppresses two diagnostic messages.	TSK	/WAR
/[NO]WIDE			
	Formats a map to print 132 characters wide.	MAP	/WID

11.2 /ANCILLARY_PROCESSOR[:n]

Use /ANCILLARY_PROCESSOR to specify a task as an Ancillary Control Processor, which is a privileged task.

Format

```
$ LINK/TAS/ANCILLARY_PROCESSOR:n/MAP/SYM inputfile...
```

This qualifier informs TKB that the task you are building 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.

/ANCILLARY_PROCESSOR also informs TKB that the task is privileged. TKB sets the AC attribute flag and the privileged attribute flag in the task's label block flag word.

The number n is an octal number that specifies the first KT-11 Active Page Register (APR) that you want the Executive to use to map the task's image when the task is running in user mode. Valid APRs are 0, 4, and 5. If you do not specify n, TKB assumes a value of 5.

If you do not explicitly specify that your task is to run on a mapped system (with the /MEMORY_MANAGEMENT qualifier) and mapping is not otherwise implied (because TKB is not running on a system with KT-11 hardware), TKB tests the value of n for validity, but otherwise ignores it.

The default is not /ANC; an Ancillary Control Processor is not specified.

11.3 /BASIC

Use /BASIC as an input file qualifier when the input file was created by the BASIC-PLUS-2 compiler.

Format

```
$ LINK inputfile/BASIC
```

/BASIC identifies the input file as a command file produced by the BASIC-PLUS-2 compiler. The BUILD command of the compiler creates a file of commands for the use of TKB and an object file. The command file supplies the appropriate switches and options to build a task image from the object file. This qualifier is not valid in any other situation.

You should not modify the command file created by the BASIC-PLUS-2 compiler, nor should you attempt to write the command file yourself. Fatal errors may occur when user-edited BASIC-PLUS-2 command files are supplied to TKB.

The default is not /BASIC; TKB assumes no input command file from the BASIC-PLUS-2 compiler.

11.4 /[NO]CHECKPOINT:SYS

Use /CHECKPOINT to create a checkpointable task image.

Format

```
$ LINK/TAS/CHECKPOINT:SYSTEM/MAP/SYM inputfile
```

/CHECKPOINT:SYSTEM marks your task as checkpointable.

/CHECKPOINT:SYSTEM causes the system to checkpoint the task to space that you have allocated in the system checkpoint file on the system disk. The system writes the task to the system checkpoint file on secondary storage when its physical memory is required by a task of higher priority. You must preallocate that space with the DCL command SET DEVICE. /CHECKPOINT and /CHECKPOINT:SYSTEM are equivalent qualifiers.

The task's checkpointability has an impact on the efficient operation of the entire system. If a task cannot be checkpointed, it may block other more important tasks from running. If it can be checkpointed, the checkpoint space on the device is not available for other use. If it is checkpointable on the system device, there may at times be no room for the task on the device. As a general rule, all user tasks should be checkpointable. However, there are exceptions to this rule. Consult with your system manager on the proper use of /CHECKPOINT.

The DCL command INSTALL overrides the use of the /CHECKPOINT qualifier.

The default is /NOCHECKPOINT:SYSTEM.

Notes

1. Do not use /CHECKPOINT:SYSTEM and /CHECKPOINT:TASK in the same command line.
2. Do not use /CHECKPOINT:TASK in the same command line with /NOHEADER to build tasks. Examples of tasks that use the /NOHEADER switch are the Executive, device drivers, and commons or libraries.

11.5 /[NO]CHECKPOINT:TAS

Use /CHECKPOINT to create a checkpointable task image.

Format

```
$ LINK/TAS/CHECKPOINT:TASK/MAP/SYM inputfile
```

/CHECKPOINT:TASK causes the system to checkpoint the task to a space in the task image file. However, the system uses the system checkpoint file first if you specified dynamic checkpointing with SET DEVICE. Also, /CHECKPOINT:TASK tells TKB to allocate additional space in the task image to accommodate the checkpointed task.

The task's checkpointability has an impact on the efficient operation of the entire system. If a task cannot be checkpointed, it may block other more important tasks from running. If it can be checkpointed, the checkpoint space on the device is not available for other use. If it is checkpointable on the system device, there may at times be no room for the task on the device. As a general rule, all user tasks should be checkpointable. However, there are exceptions to this rule. Consult with your system manager on the proper use of /CHECKPOINT.

The DCL command INSTALL overrides the use of the /CHECKPOINT qualifier.

The default is /NOCHECKPOINT:TASK.

Notes

1. Do not use /CHECKPOINT:SYSTEM and /CHECKPOINT:TASK in the same command line.
2. Do not use /CHECKPOINT:TASK in the same command line with /NOHEADER to build tasks. Examples of tasks that use the /NOHEADER switch are the Executive, device drivers, and commons or libraries.

11.6 /CODE:CLI

Use /CODE:CLI to inform TKB that the task is a command line interpreter.

Format

```
$ LINK/TAS/CODE:CLI/MAP/SYM inputfile
```

Using /CODE:CLI enables you to install a CLI without specifying /CLI=YES on the INSTALL command line. Use this qualifier when you task build the DCL task or any other CLI task. You can still install a CLI built without /CODE:CLI by specifying /CLI=YES when installing it.

The default is not /CODE:CLI.

Note

The Fast Task Builder (FTB) does not support /CODE:CLI.

11.7 /CODE:DATA_SPACE

Use /CODE:DATA_SPACE to inform TKB that the task uses D-space APRs for its data and I-space APRs for its instructions. TKB separates the I-space program sections from the D-space program sections that reside in the task.

Format

```
$ LINK/TAS/CODE:DATA_SPACE/MAP/SYM inputfile
```

TKB separates the I-space program sections in the task from the D-space program sections. TKB maps the I-space program sections with the I-space APRs and maps the D-space program sections with the D-space APRs. Therefore, you must separate the instructions and data in your task by defining them with separate program sections (see Chapter 7). A bit in the task's label block informs INSTALL that the task has separate I- and D-space.

The default is not /CODE:DATA_SPACE.

Note

Do not use the /CODE:DATA_SPACE and the /NOHEADER qualifiers in the same build.

11.8 /CODE:EAE

Use /CODE:EAE to inform TKB that the task uses the KE11-A Extended Arithmetic Element.

Format

```
$ LINK/TAS/CODE:EAE/MAP/SYM inputfile
```

TKB allocates three words in the task's header for saving the state of the arithmetic element. You should not use /CODE:EAE and /CODE:FPP on the same command line.

The default is not /CODE:EAE.

11.9 /CODE:FAST_MAP

Use /CODE:FAST_MAP to inform TKB that space must be allocated in memory between the task and the external header for use by the fast-mapping feature of the Executive.

Format

```
$ LINK/TAS/CODE:FAST_MAP/MAP/SYM inputfile
```

The default is not /CODE:FAST_MAP.

Note

The Fast Task Builder (FTB) does not support /CODE:FAST_MAP.

11.10 /CODE:FPP

Use /CODE:FPP to inform TKB that the task uses the Floating Point Processor.

Format

```
$ LINK/TAS/CODE:FPP/MAP/SYM inputfile
```

TKB allocates 25 words in the task's header for saving the state of the Floating Point Processor if the task is context-switched. For information on changing TKB defaults, refer to Appendix F.

The default is /CODE:FPP on an RSX-11M-PLUS system, and /CODE:FPP on Micro/RSX systems with the Floating Point Processor.

11.11 /CODE:OTS_FAST

Use /CODE:OTS_FAST to instruct TKB to link in the overlay run-time system (OTS) FSTMAP module.

Format

```
$ LINK/TAS/CODE:OTS_FAST/MAP/SYM inputfile
```

The default is not /CODE:OTS_FAST.

Notes

1. The /CODE:OTS_FAST qualifier requires the /EXTERNAL and /FAST qualifiers.
2. The Fast Task Builder (FTB) does not support /CODE:OTS_FAST.

11.12 /CODE:PIC

Use /CODE:PIC to inform TKB that the task's shared region contains only position-independent code or data. This qualifier is equivalent to /CODE:POSITION_INDEPENDENT.

Format

```
$ LINK/TAS/CODE:PIC/MAP/SYM inputfile
```

This qualifier must be used with the /NOHEADER qualifier and with either the /SHAREABLE:COMMON or /SHAREABLE:LIBRARY qualifier. TKB sets the PIC code attribute flag in the label block flag word of the shared region.

Be aware that if you specify /CODE:PIC without using the /SHAREABLE:COMMON or /SHAREABLE:LIBRARY qualifier, TKB builds a shared common (/SHAREABLE:COMMON default). If you specify /[NO]HEADER without /CODE:PIC, /SHAREABLE:COMMON, or /SHAREABLE:LIBRARY, TKB creates a shared library (/SHAREABLE:LIBRARY default).

The default is not /CODE:PIC.

11.13 /CODE:POSITION_INDEPENDENT

Use /CODE:POSITION_INDEPENDENT to inform TKB that the task's shared region contains only position-independent code. This qualifier is equivalent to /CODE:PIC.

Format

```
$ LINK/TAS/CODE:POSITION_INDEPENDENT/MAP/SYM inputfile
```

This qualifier must be used with the /NOHEADER qualifier and with either the /SHAREABLE:COMMON or /SHAREABLE:LIBRARY qualifier. TKB sets the PIC code attribute flag in the label block flag word of the shared region.

Be aware that if you specify /CODE:POSITION_INDEPENDENT without using the /SHAREABLE:COMMON or /SHAREABLE:LIBRARY qualifier, TKB builds a shared common (/SHAREABLE:COMMON default). If you specify /[NO]HEADER without /CODE:PIC, /SHAREABLE:COMMON, or /SHAREABLE:LIBRARY, TKB creates a shared library (/SHAREABLE:LIBRARY default).

The default is not /CODE:POSITION_INDEPENDENT.

11.14 /COMPATIBLE

Use /COMPATIBLE to align memory-resident overlay segments on 256-word boundaries, if your task uses Executive mapping directives and memory-resident overlays.

Format

```
$ LINK/TAS/COMPATIBLE/MAP/SYM inputfile
```

When memory-resident overlay segments are read in, TKB aligns them on 256-word boundaries. This is to make the memory-resident segment compatible with the Executive mapping directives, which operate on a basis of 256-word boundaries.

Using this directive could cause some loss of the use of virtual addressing space in your task. For example, if the root of your task extends just a few words beyond a 256-word boundary, and you use this qualifier, a large space will be left between the end of the root and the first memory-resident overlay segment. A loss of space may also occur between multiple overlay segments. To avoid this loss, try to code your task so that the end of the root or of segments is near and just below a 256-word boundary.

The default is not /COMPATIBLE; TKB does not align memory-resident overlay segments on 256-word boundaries, but on 32-word boundaries.

11.15 /[NO]CONCATENATE

Use /NOCONCATENATE to include only the first module of the input file to which this qualifier is attached.

Use /CONCATENATE to include all the modules of the input file to which this qualifier is attached. This is the default operation.

Format

```
$ LINK/TAS/MAP/SYM inputfile/NOCONCATENATE
```

By default, TKB includes in the task all the modules in the input file. In other words, it concatenates all the modules. The /NOCONCATENATE qualifier provides a way to include only the first module of the input file as part of the task.

The default is /CONCATENATE; TKB includes in the task all the modules in the input file.

11.16 /CROSS_REFERENCE

Use /CROSS_REFERENCE to add a cross-reference listing that includes segment, module, and global symbol information to the map (MAP) file.

Format

```
$ LINK/TAS/MAP/CROSS_REFERENCE/SYM inputfile
```

If you include this qualifier, LINK automatically includes the /MAP qualifier as well and, therefore, produces a map file. You need not use the /MAP qualifier unless you want to supply a name to the map file that is different from the input file name.

/CROSS_REFERENCE instructs TKB to create a special work file (file.CRF) that contains segment, module, and global symbol information. TKB 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 the *RSX-11M-PLUS Utilities Manual* for more information on CRF.

For this qualifier to operate correctly, CRF must be installed in your system.

The Example section describes the cross-reference listing and its contents.

The default is not /CROSS_REFERENCE; TKB does not include cross-reference information in the map file.

Example

Example 11-1 shows a cross-reference listing for task OVR. The numbered items in the notes correspond to the numbers in the example.

Example 11-1: Cross-Reference Listing for OVR.TSK

```
CREF        CREATED BY  TKB    ON 27-JUL-87 AT 09:46     PAGE 1            ❶

GLOBAL CROSS REFERENCE                                   CREF    V01

SYMBOL  VALUE       REFERENCES...

AADD    020000-R  * AADD      @ CALC
ADDEXI  020060-R  * AADD
ARGBLK  001340-R    CALC      # MAIN
BUFF    001366-R  # MAIN        OUTPUT
CALC    003270-R  * CALC      @ MAIN
DIFR    001360-R    CALC      # MAIN                                       ❷
DIVEXI  020062-R  * DIVV
DIVR    001364-R    CALC      # MAIN
DIVV    020000-R  @ CALC      * DIVV
I       001350-R    INPUT     # MAIN
IE.EOF  177766      INPUT     # QIOSYM
INITL   005664-R  # INITL     ^ MAIN
INPUT   003364-R  * INPUT     @ MAIN
IOSB    001334-R    INPUT     # MAIN

CREF        CREATED BY  TKB    ON 27-JUL-87 AT 09:46     PAGE 2

 SEGMENT CROSS REFERENCE                                  CREF    V01

SEGMENT NAME  RESIDENT MODULES

AADD          AADD
CALC          CALC
DIVV          DIVV
INPUT         ARITH   CATB    INPUT   QIOSYM  SAVRG                        ❸
LIBROT        INITL   SAVAL
MAIN          ALERR   AUTO    MAIN    OVCTR   OVDAT   OVRES   SAVR1
              VCTDF
MULL          MULL
OUTPUT        ARITH   CATB    CBTA    CDDMG   C5TA    DARITH  EDDAT
              EDTMG   OUTPUT  QIOSYM  SAVRG
SUBB          SUBB
```

❶ 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.

❷ 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 is displayed more than once within each entry.

The suffix -R appears next 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 nonautoloadable definition
*	Module contains an autoloadable definition

❸ 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.

11.17 /DEBUG[:filespec]

Use /DEBUG to include a debugging aid in your task.

Format

```
$ LINK/TAS/DEBUG[:filename]/MAP/SYM inputfile
```

If you apply /DEBUG without a file name as a LINK output qualifier or input qualifier, TKB includes the system debugging aid LB0:[1,1]ODT.OBJ into the task image. If you use /DEBUG in the same way with the /CODE:DATA_SPACE qualifier, TKB includes LB:[1,1]ODTID.OBJ in the task image.

If you apply /DEBUG with a file name, you are specifying an input file with a file type of OBJ that is a user-written debugger. LINK passes the specified file name to TKB as one of the input files.

TKB passes control to the debugger when the task starts execution.

Debugging aids can trace a task, run the task in single-step mode, print out relevant debugging information, or monitor the task's performance for evaluation.

/DEBUG without a file name has the following effects on the task image:

- The transfer address of the debugging aid overrides the task transfer address.
- TKB initializes the header of the 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 Radix–50 characters representing the version of the task. TKB derives the version from the first .IDENT directive it encounters in the task. If no .IDENT directive is in the task, this value is 01.
R4	The second three Radix–50 characters representing the version of the task.

The default is not /DEBUG; no debugging aids are present for inclusion in the task.

11.18 /DEFAULT_LIBRARY

Use /DEFAULT_LIBRARY to substitute another library for the system object module library.

Format

```
$ LINK/TAS/MAP/SYM inputfile1/DEFAULT_LIBRARY,inputfile2,...
```

Attaching /DEFAULT_LIBRARY to an input file causes that file to be a replacement library for the system object module library. The input file default file type is OBJ.

The specified library file replaces the file LB0:[1,1]SYSLIB.OLB as the library file that TKB searches to resolve undefined global references. The default device for the replacement file is SY0. TKB refers to it only when undefined symbols remain after it has processed all the files you have specified. You can apply the qualifier to only one input file.

The default is not /DEFAULT_LIBRARY; no default library is specified or used.

11.19 /ERROR_LIMIT[:n]

Use /ERROR_LIMIT to abort the task-building process after n diagnostic errors.

Format

```
$ LINK/TAS/ERROR_LIMIT:n/MAP/SYM inputfile
```

/ERROR_LIMIT causes TKB to abort after n diagnostic errors have been reached. More than n errors are not acceptable. Enter n as a decimal number, with or without the decimal point.

If you do not specify n, TKB assumes a value of 5.

The default is not /ERROR_LIMIT; TKB does not abort on any particular error limit.

11.20 /[NO]EXECUTABLE[:filespec]

Use /[NO]EXECUTABLE to specify a task name, or no task.

Format

```
$ LINK/MAP/SYM inputfile/EXECUTABLE
```

If you use /EXECUTABLE by itself as a LINK output qualifier, it will have no effect because the creation of a task file is the default operation. If you use /EXECUTABLE by itself as an input file qualifier, TKB gives the task the same name as that of the file to which /EXECUTABLE is attached.

/EXECUTABLE:filename specifies a name different from that of the first input file encountered in the input file string. You can use /EXECUTABLE in this way to give a specific name to a task. If you use /EXECUTABLE:filename attached to an input file, or as an output qualifier, the task is named by the specified file name.

/NOEXECUTABLE specifies that TKB is not to create a task file. Use it to create a map or symbol definition file only, or to go through the task-build operation to check for errors.

/EXECUTABLE is a synonym for the /TASK qualifier.

The default is /EXECUTABLE as an output qualifier; a task file is created with the same name as that of the first input file.

11.21 /[NO]EXTERNAL

Use /EXTERNAL to specify that a task is to have an external header (a header outside of Executive pool space).

Format

```
$ LINK/TAS/[NO]EXTERNAL/MAP/SYM inputfile
```

The effect of the /EXTERNAL qualifier is twofold: the header space in the task image is not destroyed when the task is checkpointed, and Executive pool space is conserved. A task built with the /EXTERNAL qualifier does not have a header in Executive pool, but has a copy of its header, which the Executive uses, in space allocated physically contiguous to and below the task image. When the task is checkpointed, the system writes the entire task image and the header copy below the task into a checkpoint file. The header in the task image is left unchanged.

Note that if the task is also built with the /CODE:FPP qualifier, the floating-point save area is not included in the task image but is in the header copy found below the task image.

The INSTALL qualifier /EXTERNAL_HEADER interacts with the LINK /EXTERNAL qualifier. If you use /NOEXTERNAL, the task will always have a pool-resident header, unless you rebuild the task to have an external header. If you use /EXTERNAL, the task will have an external header, but the INSTALL /NOEXTERNAL_HEADER qualifier can override this. The default use of /EXTERNAL by TKB is /EXTERNAL (external header) unless this is changed by the INSTALL command.

The default is /EXTERNAL; overridden by INSTALL /EXTERNAL_HEADER.

11.22 /FAST

Use /FAST to specify that you want to use the Fast Task Builder to build your task.

Format

```
$ LINK/TAS/FAST/MAP inputfile
```

/FAST specifies that you want to use the Fast Task Builder (FTB). The Fast Task Builder supports a limited number of LINK qualifiers, Task Builder capabilities, and options. See Appendix G for a full description of the Fast Task Builder capabilities.

The default is not /FAST; the standard Task Builder (TKB) is used.

11.23 /FULL_SEARCH

Use /FULL_SEARCH when you want TKB to search all co-tree segments for a matching definition or reference that matches a symbol found in the default library.

Format

```
$ LINK/TAS/FULL_SEARCH/MAP/SYM inputfile
```

In processing modules from the default object module library and encountering undefined symbols within those modules, TKB normally limits its search for definitions to the root of the main tree and to the current tree. Thus, unintended global references among co-tree overlay segments are eliminated. When you append the /FULL_SEARCH qualifier to the task image file of an overlaid task, TKB searches all co-tree segments for a matching definition or reference. See Chapter 3 for more details.

The default is not /FULL_SEARCH; TKB limits its search to the root of the main tree and to the current tree.

11.24 /[NO]GLOBALS

Use /NOGLOBALS to exclude all global symbols defined or referenced in the input file from the map and cross-reference listings. This qualifier also eliminates the "file contents" section of the map file.

Format

```
$ LINK/TAS/MAP/SYM inputfile/[NO]GLOBALS
```

TKB eliminates all global symbols defined or referenced in the input file from the map and cross-reference listings. TKB also does not include the "file contents" section in the map file.

The default is /GLOBALS; global symbols are included in the map file.

11.25 /[NO]HEADER

Use the /NOHEADER qualifier when building a resident library, common, or loadable driver.

Format

```
$ LINK/TAS/[NO]HEADER/MAP/SYM inputfile
```

TKB does not construct a header for your task image. Libraries, commons, and loadable drivers do not use task headers. Note that you must also use the STACK=0 option with this qualifier.

The default is /HEADER; TKB includes a header in the task image.

Note

Do not use the /CODE:DATA_SPACE and the /NOHEADER qualifiers in the same build.

11.26 /INCLUDE:(module1[,module2,...,modulen])

Use /INCLUDE to include specific modules from the library file into the task image.

Format

```
$ LINK/TAS/MAP/SYM inputfile/INCLUDE:MOD1,MOD2,...MOD8
```

The /INCLUDE qualifier causes TKB to extract from the library the modules named as arguments of the qualifier regardless of whether the modules contain definitions for unresolved references.

You can use the /SELECTIVE_SEARCH qualifier in the command sequence with both the /LIBRARY or /INCLUDE qualifiers to perform a selective search for global definitions.

The default is not /INCLUDE; TKB does not include modules from a library file.

Notes

1. The library file has a default file type of OLB.
2. The library file from which you extract modules can be anywhere in the input file string.
3. If you are using multiline input in the LINK command line, the library file with the /INCLUDE qualifier can appear more than once. For example:

   ```
   $ LINK/TAS/MAP/SYM inputfile1,inputfile2/INCLUDE:MOD1,- RET
   ->inputfile3,inputfile4.OLB/INCLUDE:MOD1,... RET
   ```

4. You should not use the /INCLUDE qualifier and the /CONCATENATE qualifier in the same command sequence.
5. If you want TKB 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 the task image listed as arguments to the /INCLUDE qualifier; and a second time, with the /LIBRARY qualifier. For example:

   ```
   $ LINK/TAS/MAP/SYM  INLIB1/INCLUDE:MOD1,MOD2,- RET
   ->inputfile2,INLIB1/LIBRARY RET
   ```

11.27 /[NO]IO_PAGE

Use /NOIO_PAGE to inform TKB that the task is over 12K words in size and need not map the I/O page.

Conversely, if you use the /IO_PAGE qualifier, TKB assumes that the task is over 12K words in size and must map the IO page.

Format

```
$ LINK/TAS/MAP/SYM/[NO]IO_PAGE inputfile
```

TKB maps the task with the appropriate APRs and maps the I/O page with APR 7. Using the /NOIO_PAGE qualifier causes TKB to set a bit in the task's label block that informs INSTALL that the task intentionally does not map the I/O page. When this bit is set, INSTALL does not display an error message when it detects that the privileged task extends into APR 7.

The default is /IO_PAGE; TKB assumes that the task is larger than 12K words and is to map the I/O page.

11.28 /LIBRARY

Use /LIBRARY on an input file that is a library and contains relocatable object modules.

Format

```
$ LINK/TAS/MAP/SYM inputfile1,inputfile2,...inputfileN/LIBRARY
```

TKB searches the file immediately to resolve undefined references in any modules preceding the library specification. TKB also extracts from the library, for inclusion in the task image, any modules that contain definitions for such references.

The default is not /LIBRARY; TKB does not use the input file as a library.

Notes

1. The library file must contain relocatable object modules.
2. The library file must be to the right in the string of input files that contain references to be defined in the library.
3. If you use LINK with multiline input and you specify a given library more than once during the command sequence, you must attach the /LIBRARY qualifier to the library each time that you specify the library. For example:

```
$ LINK/TAS/MAP/SYM inputfile1,INLIB/LIBRARY,- RET
->inputfile2,inputfile3,INLIB/LIBRARY RET
```

4. If you are building an overlay structure, you must specify object module libraries for an overlay structure within the ODL file for the structure. To do this, you must use the .FCTR directive to specify the library. For example:

```
        .ROOT CONTRL-LIB-(AFCTR,BFCTR,C)
AFCTR:  .FCTR AO-LIB-(A1,A2-(A21,A22))

BFCTR:  .FCTR BO-LIB-(B1,B2)
LIB:    .FCTR LB:[303,3]LIBOBJ.OLB/LB
        .END
```

The technique used in this ODL file allows you to control the placement of object module library routines into the segments of the task's overlay structure. The Task Builder /LB switch used in the ODL file example is the equivalent of the LINK /LIBRARY qualifier. However, you cannot use LINK qualifiers in an ODL file; you must use TKB switches. (TKB switches are described in Chapter 10. For more information on overlaid tasks, see Chapter 3.)

11.29 /LONG

Use /LONG to produce the long version of the memory allocation file. The long version differs from the short version in that it includes the "file contents" section.

Format

```
$ LINK/TAS/MAP/SYM/LONG inputfile
```

If you use /LONG without /MAP, LINK directs TKB to produce a map file as though you had specified /MAP in the LINK command line.

The memory allocation file consists of the following items:

- Page header
- Task attributes section
- Overlay description (if applicable)
- Root segment allocation
- Tree segment description (if applicable)
- Undefined references (if applicable)
- Task Builder statistics

The default is not /LONG; TKB does not produce the "file contents" section of the memory allocation file.

An example of the memory allocation file (MAP) is shown in Example 11-2. The numbered and lettered items in the notes following correspond to the numbers and letters in the example.

Example 11–2: Memory Allocation File (Map) Example

```
OVR.TSK;1     Memory Allocation Map  TKB M43.00      Page 1        ❶
                      15-DEC-87   11:28

Partition name : GEN    [a]  [b]
Identification : 01        [c]
Task  UIC      : [303,1]        [d]  [e]
Stack    limits: 000260 001257 001000 00512   [f]
Prg xfr address: 001264    [g]  [h]  [i]                            ❷
Total address windows: 1.    [j]  [k]  [l]
Task  image  size  : 7488. words    [m]  [n]
Task address limits: 000000 035133    [o]
R-W disk blk limits: 000002 000073 000072 00058.   [p]  [q]

OVR.TSK Overlay description:

Base    Top        Length
----    ---        ------
000000  005055  005056  02506.    ROOTM                             ❸
005060  021103  014024  06164.      MULOV
005060  021103  014024  06164.      ADDOV
021104  035127  014024  06164.        SUBOV
021104  035131  014026  06166.        DIVOV

OVR.TSK       Memory allocation map  TKB M43.00      Page 2
ROOTM                 28-DEC-87    09:10

*** Root segment: ROOTM    [a]

R/W mem  limits: 000000 005055 005056 02606.   [b]
Disk blk limits: 000002 000007 000006 00006.   [c]

Memory allocation synopsis:

Section                   [d]                         Title  Ident  File    ❹
-------                                               -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001260 001174 00636.
                          001260 000102 00066.   ROOTM   01[e]  ROOTM.OBJ;1
                          001362 000260 00176.   PRNOV   01     PRNOV.OBJ;1
                          001642 000042 00034.   SAVOV   01     SAVOV.OBJ;1
                          002454 000002 00002.
```

(Continued on next page)

Example 11–2 (Cont.): Memory Allocation File (Map) Example

```
ANS    :(RW,D,GBL,REL,OVR) 002454 000002 00002.
                           002454 000002 00002.  ROOTM   01    ROOTM.OBJ;1
                           002454 000002 00002.  PRNOV   01    PRNOV.OBJ;1
                                  .
                                  .
                                  .

Global symbols:

AADD    004032-R  DIVV   004052-R  PRINT  001550-R  SUBB    004042-R   [f]
                  MULL   004022-R  SAVAL  001642-R
                                  .
                                  .
                                  .
                                                                              ❹
File: ROOTM.OBJ;1  Title: ROOTM    Ident: 01    [g]
<. BLK.>: 001260 001361 000102 00066.    [h]
<ANS    >: 002454 002455 000002 00002.

File: PRNOV.OBJ;1  Title: PRNOV    Ident: 01    [g]
<. BLK.>: 001362 001641 000260 00176.    [h]
   PRINT  001550-R   [i]
<ANS    >: 002454 002455 000002 00002.

File: SAVOV.OBJ;1  Title: SAVOV    Ident: 01
<. BLK.>: 001642 001703 000042 00034.    [j]
   SAVAL  001642-R
                                              [k]

OVR.TSK           Memory allocation map  TKB M43.00 Page 3
MULOV                 28-DEC-87   09:10


*** Segment: MULOV
                                                                              ❺

R/W mem  limits: 005060 021103 014024 06164.
Disk blk limits: 000010 000024 000015 00013.
```

(Continued on next page)

Example 11-2 (Cont.): Memory Allocation File (Map) Example

```
Memory allocation synopsis:

Section                                          Title  Ident  File
-------                                          -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 005060 014024 06164.
                          005060 014024 06164.  MULOV  01     MULOV.OBJ;1
$$ALVC:(RW,D,LCL,REL,CON) 021104 000000 00000.
$$RTS :(RW,I,BGL,REL,OVR) 004756 000002 00002.                              ❺

Global symbols:

   MULL   021060-R

                                   .
                                   .
                                   .

*** Task builder statistics:

    Total work file references: 8570.   [a]
    Work  file  reads: 0.   [b,
    Work  file writes: 0.      c]                                            ❻
    Size of core pool: 6662. words (26. pages)   [d]
    Size of work file: 3328. words (13. pages)   [e]

    Elapsed time:00:00:12
```

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 TKB that created the map.

❷ The task attributes section contains the following information:

a. Task name—The name specified in the TASK option. If you do not use the TASK option, TKB suppresses this field.

b. Partition name—The partition specified in the PAR option. If you do not specify a partition, the default is partition GEN.

c. Identification—The task version as specified in the .IDENT assembler directive. If you do not specify the task identification, the default is 01.

d. Task UIC—The task UIC as specified in the UIC option. If you do not specify the UIC, the default is the terminal UIC.

e. Task priority—The priority of the task as specified in the PRI option. If you do not specify PRI, the default is 50 and is not shown on the map.

f. Stack limits—The low and high octal addresses of the stack, followed by its length in octal and decimal bytes.

g. ODT transfer address—The starting address of the ODT debugging aid. If you do not specify the ODT debugging aid, this field is suppressed.

h. 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, TKB automatically establishes a transfer address of 000001 for it. TKB also suppresses this field in the map if you do not specify a transfer address.

i. Task attributes—These attributes are listed only if they differ from the defaults. One or more of the following attributes 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 Executive directive Send will be rejected by the Executive.
SL	Task can be slaved.
TR	Task's initial PSW has T-bit enabled.
ID	Task is I- and D-space task.

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 Appendix B); mapped array is not shown if it does not apply.

l. Task extension—The increment of physical memory (decimal words) allocated through the EXTTSK or PAR option. Without these options, task extension is not shown.

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. Total task size is not shown in this example.

o. Task address limits—The lowest and highest virtual addresses allocated to the task, exclusive of virtual addresses allocated to virtual program sections 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; and 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; and 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.

❸ 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. TKB prints the overlay description only when an overlaid task is created.

❹ 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); and 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; the 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; the starting virtual address of the program section; and the 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 the system library. (For more information, see the description of the /SYSTEM_LIBRARY_DISPLAY qualifier in this chapter.)

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 is always of zero length.

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 the 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 /LONG in the LINK command sequence. TKB 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, and 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, and 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 3), this value is the address of an autoload vector. The autoload vector in turn contains the actual address of the symbol.

   A -R is appended to the value if the symbol is relocatable.

j. Program section—The contents of this field are 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 TKB performance:

a. Work file references—The number of times that TKB 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 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 required to process the overlay description, parse the list of input file names, and create the cross-reference listing (if specified).

See Appendix F for a more detailed discussion of the work file.

11.30 /MAP[:filespec]

Use /MAP to produce a memory allocation (map) file. (The /LONG qualifier produces a map file without the need of using /MAP also.)

Format

```
$ LINK/TAS/MAP[:filespec]/SYM inputfile
```

The map file may be named by using /MAP:filename; otherwise, the name of the map file is the same as the first input file. You may use /MAP either as a command qualifier or as an input filename qualifier. If you specify /MAP, TKB spools the map file to the printer if the system task QMGPRT.TSK is installed with the PRT... name. TKB does not spool the map file to the printer if you use the /NOPRINT qualifier with the /MAP qualifier.

The default is not /MAP; TKB does not produce a map file.

11.31 /[NO]MEMORY_MANAGEMENT[:n]

Use /MEMORY_MANAGEMENT to inform TKB whether the system on which the task is to run has memory management hardware. Specify n as the decimal numbers 28 or 30.

Format

```
$ LINK/TAS/MEMORY_MANAGEMENT[:n]/MAP/SYM inputfile
```

If you use n with /NOMEMORY_MANAGEMENT (for an unmapped system), n specifies the highest physical address in K-words of the task or system being built. If you do not specify n with /NOMEMORY_MANAGEMENT, the default highest address of the task or system is 28K.

If you specify n with /MEMORY_MANAGEMENT, n is ignored.

If you use /NOMEMORY_MANAGEMENT, TKB does not recognize the memory-resident overlay operator (!). TKB checks the operator for correct syntax, but it does not create any resident overlay segments.

The default is not /MEMORY_MANAGEMENT; if you do not apply either /MEMORY_MANAGEMENT or /NOMEMORY_MANAGEMENT to your task image file, TKB allocates memory according to the mapping status of the system on which your task is being built. The maximum task size for a mapped system is 32K words, or 64K words for a task that uses I- and D-space. The default highest address for a task or system in an unmapped system is 28K.

11.32 /OPTION[:filespec]

Use /OPTION in the LINK command to tell LINK to prompt you for Task Builder options.

Format

$ LINK/TAS/MAP/SYM/OPTION[:filespec] inputfile

After you enter the LINK command line with /OPTION included, LINK prompts you for options so that the entire sequence looks like the following:

```
$ LINK/TAS/MAP/SYM/OPT inputfile RET
Option? enter option here, or options separated by commas RET
Option? RET
$
```

If you enter multiple options, you may enter each one on a separate line; LINK will prompt you again. To end option input, press the RETURN key only.

Another way of entering multiple options is to enter many of them on a single option line. If you do, you must separate them with a comma. To end option input, press the RETURN key only.

As an option entry after the Option? prompt, you may specify a file that contains many options, each one on a separate line. This file must have a file type of CMD. Comments in this file are noted by beginning the comment line with a semicolon. There must not be any slashes in this file.

Alternatively, you may have a file of options that you can name in filespec, as shown in the format. This file must have a file type of CMD. This file should be formatted as a list of options, one option on each line. There must not be any slashes in this file.

An OPTION.CMD file could look like the following example:

```
STACK=0
; NO STACK BECAUSE THIS IS A LIBRARY
PAR=PETROV:160000:40000

; PARTITION SIZE AND ADDRESS
GBLREF=L$DIME
; MAKE REFERENCES GLOBAL
```

A LINK command line to enter this option file could look like the following example:

```
$ LINK/TAS/NOHEAD/MAP/SYM/OPT RET
File(s)? inputfile1,inputfile2 RET
Option? @OPTION RET
Option? RET
$
```

If, while entering options, you decide that the LINK command or the options you have already entered are incorrect, you can enter the ABORT option as the last option. Then, press the RETURN key. When TKB receives the ABORT option, it aborts its operation and allows you to start again without having a task build take place.

The default is not /OPTION; LINK does not prompt for or accept options.

11.33 /OVERLAY_DESCRIPTION

Use /OVERLAY_DESCRIPTION to inform TKB that the input file (ODL) contains the Overlay Description Language that describes the overlaying to take place with the input files named in the ODL file.

Format

```
$ LINK/TAS/MAP/SYM/OVERLAY_DESCRIPTION inputfile
```

/OVERLAY_DESCRIPTION informs TKB that the single input file contains Overlay Description Language. TKB uses this language, which contains file names, to build overlay segments in certain configurations, which are also specified in the file.

The input file must have a file type of ODL and it must be the only file specified.

The Overlay Description Language is described in Chapters 3 and 4.

The default is not /OVERLAY_DESCRIPTION.

11.34 /POSTMORTEM

Use /POSTMORTEM to dump the contents of memory if the task terminates abnormally.

Format

```
$ LINK/TAS/POSTMORTEM/MAP/SYM inputfile
```

The /POSTMORTEM qualifier sets the postmortem dump flag in the task's label block flag word. This flag is checked when the task abnormally terminates and, if it is on, the contents of memory are dumped.

If your task issues an ABRT$ (Abort Task) directive, the system will not dump the task image even though TKB has set the postmortem dump flag in the task's label block flag word. In this case, the system assumes that a postmortem dump is not necessary because you already know why your task was aborted.

The PMD task must be installed in your system and be able to get into physical memory for this qualifier to be effective.

The default is not /POSTMORTEM; no memory dump takes place if the task terminates abnormally.

11.35 /[NO]PRINTER

Use /NOPRINTER with the /MAP qualifier to prevent spooling the map file to the printer. /PRINTER is the default and it need not be specified.

Format

```
$ LINK/TAS/MAP:filename/[NO]PRINTER/SYM inputfile
```

/PRINTER controls whether or not the map file is spooled to the line printer. You should use /[NO]PRINTER with the /MAP qualifier. If you do not want the map file to print, use /NOPRINTER. The map file prints only if the system task QMGPRT.TSK is installed with the PRT... name.

The default is /PRINTER; the Task Builder spools the map file to the printer.

11.36 /PRIVILEGED[:n]

Use /PRIVILEGED to inform TKB that the task is privileged with respect to memory access rights and device access rights. The octal number n can be 0, 4, or 5, which specify privilege 0, 4, or 5, respectively. Privileged tasks are described in Chapter 6.

Format

```
$ LINK/TAS/PRIVILEGED[:n]/MAP/SYM inputfile
```

If you specify /PRIVILEGED:0, the task does not have access to the I/O page or the Executive. However, if you specify /PRIVILEGED:4 or /PRIVILEGED:5, your task does have access to the I/O page and the Executive, in addition to its own partition.

The value of n is an octal number that specifies the first Active Page Register (APR) that you want the Executive to use to map the task image when the task is running in user mode. Valid APRs are 0, 4, and 5. If you do not specify one of these values, TKB assumes a value of 5.

If you do not explicitly specify that your task is to run on a mapped system (with the /MEMORY_MANAGEMENT qualifier) and it is not implied (by the presence of KT-11 hardware on the system upon which TKB is running), TKB merely tests the value (:n) of the switch for validity; otherwise, TKB ignores it.

You should not use /PRIVILEGED and /ANCILLARY_PROCESSOR on the same command line.

The default is not /PRIVILEGED; TKB assumes that a normal task is being built.

11.37 /[NO]RECEIVE

Use /[NO]RECEIVE to determine whether your task can receive messages directed to it by the following Executive directives:

- Send Data (SDAT$)
- Send, Request, and Connect (SDRC$)
- Send Data Request and Pass Offspring Control Block (SDRP$)
- Variable Send Data (VSDA$)
- Variable Send, Request, and Connect (VSRC$)

Format

```
$ LINK/TAS/[NO]RECEIVE/MAP/SYM inputfile
```

/RECEIVE determines whether messages can be directed to the task by means of the Send directives. (Refer to the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual* for information on the Send directives.)

By default, messages can be directed to your task by means of the Send directives. If you negate this qualifier (/NOREC), the system inhibits the queuing of messages to your task.

The default is /RECEIVE; the task can normally receive messages from the Send directives.

11.38 /[NO]RESIDENT_OVERLAYS

Use /RESIDENT_OVERLAYS to enable TKB's recognition of the memory-resident overlay operator (!) in the Overlay Description Language file.

Format

```
$ LINK/TAS/[NO]RESIDENT_OVERLAY/MAP/SYM inputfile
```

The memory-resident overlay operator, when present in the overlay description file, indicates to TKB that it is to construct a task image that contains one or more memory-resident overlay segments. If you negate this switch (/NORES), TKB checks the operator for correct syntactical usage, but otherwise ignores it. With the memory-resident overlay operator thus disabled, TKB builds a disk-resident overlay from the overlay description file.

The default is /RESIDENT_OVERLAY; TKB normally recognizes the memory-resident overlay operator.

11.39 /SAVE

Use /SAVE to save in your directory the command file that DCL creates from the LINK command.

Format

```
$ LINK/TAS/MAP/SYM/SAVE inputfile
```

If you use /SAVE in the LINK command line, the command file created by DCL from the LINK command line appears in your directory after the task build. This file is named ATLNK.CMD. It contains the resulting TKB commands translated from the LINK commands you entered, and it contains legitimate TKB command syntax.

ATLNK.CMD is always the same name, so you could have many files of this name if you enter multiple LINK commands. Therefore, it is perhaps best to rename this file to a meaningful name—to COOKIE.CMD, for example. Later, if you want to build the same task in the same way, you can enter the following command line:

```
$ LINK @COOKIE [RET]
```

Also, ATLNK.CMD is a place to look to see how DCL and LINK interpreted and translated the LINK command line that you entered.

The default is not /SAVE; ATLNK.CMD is not saved in your directory.

11.40 /[NO]SEGREGATE

Use /SEGREGATE to allocate virtual address space to program sections contiguously; first to the read/write (RW) program sections, and then to the read-only (RO) program sections.

Format

```
$ LINK/TAS/[NO]SEGREGATE/MAP/SYM inputfile
```

The /SEGREGATE qualifier gives you control over the ordering of program sections. By using the /SEGREGATE qualifier, you cause TKB to order program sections alphabetically by name within access code (RW followed by RO). If you specify the /SEQUENTIAL qualifier with the /SEGREGATE qualifier, TKB orders program sections in their input order by access code. See the description of the /SEQUENTIAL qualifier.

You use the negated qualifier, /NOSEGREGATE, to make TKB interleave the RW and RO program sections. Thus, the combination /NOSEGREGATE/SEQUENTIAL results in a task with its program sections allocated in input order and its RW and RO sections interleaved. Additionally, you can use /NOSEGREGATE and /NOSEQUENTIAL to make TKB order program sections alphabetically with RW and RO sections interleaved. However, /NOSEGREGATE is the default.

When building multiuser tasks, the /SHAREABLE:TASK qualifier causes TKB to default to /SEGREGATE. Therefore, to correctly build read-only tasks, you can use the /SHAREABLE:TASK qualifier only.

The default is /NOSEGREGATE; TKB does not segregate program sections by access code.

11.41 /SELECTIVE_SEARCH

Use /SELECTIVE_SEARCH to include in TKB's internal symbol table only those global symbols for which there is a previously undefined reference.

Format

```
$ LINK/TAS/MAP/SYM inputfile/SELECTIVE_SEARCH
```

When processing an input file, TKB normally includes in its internal symbol table each global symbol it encounters within the file, whether or not there are references to it. With the /SELECTIVE_SEARCH qualifier attached to an input file, TKB checks each global symbol it encounters within that file against its list of undefined references. If TKB finds a match, it includes the symbol in its symbol table.

The default is not /SELECTIVE_SEARCH; TKB does not include in its internal symbol table those global symbols with previously undefined references.

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 11-2.

Table 11-2: Input Files for SEL.TSK

Input File Name	Global Definition	Global Reference
IN1		A
	A	
IN2	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 TKB to resolve the reference to global symbol A in IN1 to the definition for A in IN2. Assume further that you want TKB to resolve the reference to global symbol C in IN3 to the definition for C in IN4. By selecting the line of the input files properly and applying the /SELECTIVE_SEARCH qualifier to files IN2 and IN4, TKB resolves the references correctly. The following command line illustrates the correct sequence:

```
$ LINK/TAS:SEL IN1,IN2/SEL,IN3,IN4/SEL RET
```

TKB processes input files from left to right; therefore, in processing the above command line, TKB processes file IN1 first and encounters the reference to symbol A. There is no definition for A within IN1; therefore, TKB marks A as undefined and moves on to process file IN2. Because the /SELECTIVE_SEARCH qualifier is attached to IN2, TKB limits its search of IN2 to symbols it has previously listed as undefined, in this case, symbol A. TKB finds a definition for A and places A in its symbol table. Because there are no undefined references to symbols B or C, TKB does not place either of these symbols in its symbol table.

Note

It is important to realize that the /SELECTIVE_SEARCH qualifier affects only the way the Task Builder constructs its internal symbol table. The routines for which symbols B and C are entry points are included in the task image even though there are no references to them.

TKB moves on to IN3. It encounters the references to symbol C. Because TKB did not include symbol C from IN2 in its symbol table, it cannot resolve the reference to C in IN3. TKB marks symbol C as undefined and moves on to IN4.

When TKB processes IN4, it encounters the definition for C in that file and includes it in the table. Again, because the /SELECTIVE_SEARCH qualifier is attached to IN4, TKB includes only C in its symbol table.

When TKB has completed its processing of the above command line, 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

This example does not represent good programming practice. It is included here to illustrate the effect of the /SELECTIVE_SEARCH qualifier on TKB during a search sequence.

The /SELECTIVE_SEARCH qualifier 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 /SELECTIVE_SEARCH qualifier to it, TKB includes in its internal symbol table only those symbols in the Executive that the task references. An example of a LINK command line that illustrates this is as follows:

```
$ LINK/TAS:OUTFILE/PRI:5 INFILE,RSX11M.STB/SEL [RET]
```

This command line directs TKB 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 /SELECTIVE_SEARCH qualifier applied to it directs TKB to extract from RSX11M.STB only those symbols for which there are references within OUTFILE.TSK.

11.42 /SEQUENTIAL

Use /SEQUENTIAL to cause TKB to construct the task image from the program sections you specified, in the order in which you input them.

Format

```
$ LINK/TAS/MAP/SYM/SEQUENTIAL inputfile
```

/SEQUENTIAL causes TKB to collect all the references to a given program section from your input object modules, group them according to their access code (RW followed by RO), and, within these groups, allocate memory for them in the order in which you input them. However, the /SEGREGATE qualifier affects program-section ordering and can be used with the /SEQUENTIAL qualifier. See the /SEGREGATE qualifier for further details.

Without the /SEQUENTIAL qualifier, TKB reorders the program sections alphabetically.

You use /SEQUENTIAL to satisfy any adjacency requirements that existing code may have when you are converting it to run under RSX. Using this qualifier otherwise is discouraged for the following reasons:

- Standard library routines (such as FORTRAN I/O handling routines and FCS modules from the system library) do not work properly.
- Sequential ordering 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.

The default is not /SEQUENTIAL; sequential program-section ordering does not take place.

11.43 /SHAREABLE:COMMON

Use /SHAREABLE:COMMON to inform TKB that a shared common is being built. You must use the /NOHEADER qualifier with /SHAREABLE:COMMON.

Format

```
$ LINK/TAS/NOHEADER/SHAREABLE:COMMON/MAP/SYM inputfile
```

Shared commons, by Task Builder definition, contain only data.

If you do not use the /CODE:PIC qualifier for an absolute shared common, all the program sections in the common are marked absolute. Using the /NOHEADER qualifier without the /SHAREABLE:COMMON and /CODE:PIC qualifiers causes TKB to build a shared library.

If you use the /CODE:PIC qualifier for a relocatable shared common, all program sections in the common are marked relocatable.

In either case, TKB includes all program-section declarations in the STB file. The STB file contains all the program section names, attributes, length, and symbols. TKB links common blocks by means of program sections. Therefore, the STB file of a shared region built with the /SHAREABLE:COMMON qualifier contains all defined program sections.

Using the /CODE:PIC and /NOHEADER qualifiers without the /SHAREABLE:COMMON qualifier causes TKB to build a shared common.

The /SHAREABLE:COMMON qualifier cannot be negated.

The default is not /SHAREABLE:COMMON. With the /CODE:PIC qualifier, TKB builds a shareable common. Without the /CODE:PIC qualifier, TKB builds a shareable library.

11.44 /SHAREABLE:LIBRARY

Use /SHAREABLE:LIBRARY to build a shared library. You must use also the /NOHEADER qualifier to build a shared library.

Format

```
$ LINK/TAS/SHAREABLE:LIBRARY/NOHEADER/MAP/SYM inputfile
```

TKB includes only one program-section declaration in the STB file.

If you do not use the /CODE:PIC qualifier for an absolute library, TKB gives the program section the same name as the library root, makes the library position independent, and defines all symbols as absolute. Also, if you do not use either the /CODE:PIC or the /SHAREABLE:LIBRARY qualifier, TKB assumes /SHAREABLE:LIBRARY to be the default and builds a shareable library.

If you use the /CODE:PIC qualifier for a relocatable library, TKB names the program section the same as the root segment of the library. TKB forces this name to be the first and only declared program section in the library. TKB declares all global symbols in the STB file relative to that program section.

If you use the /CODE:PIC qualifier without the /SHAREABLE:LIBRARY qualifier, TKB assumes that a shared common is to be built (/SHAREABLE:COMMON default occurs with /CODE:PIC only).

The /SHAREABLE:LIBRARY qualifier cannot be negated.

The default is not /SHAREABLE:LIBRARY. With the /CODE:PIC qualifier, TKB builds a shareable common. Without the /CODE:PIC qualifier, TKB builds a shareable library.

11.45 /SHAREABLE:TASK

Use /SHAREABLE:TASK to inform TKB that you are building a multiuser task.

Format

```
$ LINK/TAS/SHAREABLE:TASK/MAP/SYM inputfile
```

TKB separates the task's read-only and read/write program sections. It then places the read-only program sections in the task's upper virtual address space and the read/write program sections in the task's lower virtual address space.

The default is not /SHAREABLE:TASK; TKB does not build a multiuser task.

11.46 /SLAVE

Use the /SLAVE qualifier to direct TKB to mark your task as a slave to an initiating task.

Format

```
$ LINK/TAS/SLAVE/MAP/SYM inputfile
```

TKB 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 identity and privileges of the sending task.

The default is not /SLAVE; TKB does not produce a slave task.

11.47 /SLOW

Use /SLOW to specify the slow mode of the Task Builder. The default Task Builder and the slow mode are included together in one image.

You should use the slow mode only if the task build produced the following message:

```
No virtual memory storage available
```

Format

```
$ LINK/TAS/MAP/SYM/SLOW inputfile
```

/SLOW causes the slow mode to be used. The default Task Builder and the Fast Task Builder use a symbol table structure that can be searched quickly, but that requires more work-file space than the slow mode. If you receive the error message shown, you have the choice of reducing work-file size, as described in Appendix F of this manual, or of using the slow mode.

Except for the use of different symbol table structures, the default Task Builder and the slow mode are identical. All qualifiers to LINK and all TKB options are available for the slow mode.

The default is not /SLOW; the default Task Builder is invoked and run.

Note

The /SLOW qualifier does not function when used with an indirect command file. If you invoke TKB with an indirect command file, you have to accept the use of the standard TKB or add the /SB switch in the indirect command file.

11.48 /SYMBOL_TABLE[:filespec]

Use /SYMBOL_TABLE to specify that you want a symbol definition output file.

Format

```
$ LINK/TAS/MAP/SYMBOL_TABLE[:filespec] inputfile
```

The presence of /SYMBOL_TABLE indicates to TKB that you want a symbol definition file as one of the output files. If this qualifier is absent, TKB does not produce the symbol definition file. You may specify a file specification; otherwise, the file name defaults to that of the first input file encountered in the string of input files. The default file type is STB.

The default is not /SYMBOL_TABLE; TKB does not produce a symbol definition file.

11.49 /[NO]SYSTEM_LIBRARY_DISPLAY

Use /SYSTEM_LIBRARY_DISPLAY to include global symbols in the map file.

Format

```
$ LINK/TAS/MAP/SYSTEM_LIBRARY_DISPLAY/SYM inputfile
```

TKB includes in the map file the names of routines it has added to your task from the system library. It also includes in the map file global symbols contained in the symbol definition file of any shared region to which the task refers. Those global symbols are the ones defined or referenced by the task.

If you use /SYSTEM_LIBRARY_DISPLAY, LINK automatically specifies that TKB include a map file. Therefore, you need not specify a /MAP qualifier, unless you want to name the map file instead of letting the map file name default to the name of the first input file.

The default is /NOSYSTEM_LIBRARY_DISPLAY; TKB does not include global symbols in the map file.

11.50 /[NO]TASK[:filespec]

Use /[NO]TASK to specify a task name, or no task.

Format

```
$ LINK/MAP/SYM inputfile/TASK[:filename]
```

or

Format

```
$ LINK/MAP/SYM/NOTASK
```

Used by itself as a LINK output qualifier, /TASK has no effect because the creation of a task file is the default operation. If you use /TASK by itself as an input file qualifier, TKB gives the task the same name as that of the file to which /TASK is attached.

/TASK:filename specifies a name different from that of the first input file encountered in the input file string. You can use /TASK in this way to give a specific name to a task. If you use /TASK:filename attached to an input file, or as an output qualifier, the task is named by the specified file name.

/NOTASK specifies that TKB is not to create a task file. /NOTASK is useful when you want to create a map or symbol definition file only, or to go through the task-build operation just to check for errors.

/TASK is a synonym for the /EXECUTABLE qualifier.

The default is /TASK as an output qualifier; a task file is created with the same name as that of the first input file.

11.51 /TKB

Use /TKB to specify the standard, default Task Builder. This qualifier is included here for DCL completeness. It is the default operation.

Format

```
$ LINK/TAS/MAP/SYM/TKB inputfile
```

/TKB invokes the default Task Builder. This qualifier is the default operation and need not be specified.

The default is /TKB.

11.52 /TRACE

Use /TRACE to direct TKB to make the task traceable.

Format

```
$ LINK/TAS/TRACE/MAP/SYM inputfile
```

TKB sets the T-bit in the initial PSW of your task. When your task is executed, a trace trap occurs at the completion of each instruction.

The default is not /TRACE; TKB does not set the T-bit and a trace trap does not occur.

11.53 /[NO]WARNINGS

Use /WARNINGS to allow two diagnostic messages to be displayed. /NOWARNINGS prevents these messages from being displayed.

Format

```
$ LINK/TAS/[NO]WARNINGS/MAP/SYM inputfile
```

/WARNINGS allows the following messages to be displayed:

```
n Undefined symbols segment seg-name
```

and

```
Module module-name multiply defines P-section p-sect-name
```

The default is /WARNINGS; the messages occur.

11.54 /[NO]WIDE

Use /[NO]WIDE to control the width of the map file.

Format

```
$ LINK/TAS/MAP/SYM/[NO]WIDE inputfile
```

By default, TKB formats a map file with a width of 132 columns. When you negate this qualifier (by specifying /NOWIDE), TKB makes the map file 80 columns wide.

The default is /WIDE; a 132-column map file is produced.

Chapter 12
Options

Task Builder options provide you with the means to give TKB information about the characteristics of your task.

These options, which are listed in Table 12-1, can be divided into seven categories. The identifying abbreviation and a brief description of each category are listed below:

Abbreviation	Description
contr	You use control options to affect TKB execution. ABORT is the only member of this category. You can direct the Task Builder to abort the task build with this option.
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.
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.
share	You use storage-sharing options to indicate to TKB that your task intends to access a shared region.
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.
alter	You use the content-altering options to define a global symbol and value, or to introduce patches in your task image.
synch	You use synchronous trap options to define synchronous trap vectors.

Some TKB 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 12-1 lists all the options alphabetically and gives a brief description of each.

Table 12-1: Task Builder Options

Option	Meaning	Interest	Category
ABORT	Directs TKB to terminate a task build	H,M	contr
ABSPAT	Declares absolute patch values for conventional tasks, or I-space in I- and D-space tasks	M	alter
ACTFIL	Declares number of files open simultaneously	H	alloc
ASG	Declares device assignment to logical units	H,M	device
CLSTR	Declares a group of shared regions accessed by the task and residing in the same virtual address space in the task	H,M	share
CMPRT	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
DSPPAT	Declares absolute patch values for conventional tasks, or D-space in I- and D-space tasks	M	alter
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
GBLINC	Includes symbols in the STB file	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	Declares global symbols to be excluded from the STB file	H,M	alter

Table 12–1 (Cont.): Task Builder Options

Option	Meaning	Interest	Category
IDENT	Declares the identification of the task	H,M	ident
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
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	Declares task's intention to access a resident supervisor-mode library	H,M	share
RNDSEG	Declares task's intention to round the size of a named segment up to the nearest APR boundary while building a resident library	H,M	share
ROPAR	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	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

Table 12–1 (Cont.): Task Builder Options

Option	Meaning	Interest	Category
VARRAY	Declares the task's intention to access a virtual array directly without passing arguments to a subroutine	H	alloc
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

12.1 ABORT—Abort the Task Build

You use the ABORT option when you discover that an earlier error in the terminal sequence causes TKB to produce an unusable task image.

The Task Builder, on recognizing the ABORT option, stops accepting input and restarts for another task build.

Syntax

ABORT=n

Parameter

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 press CTRL/Z at any time, it causes TKB to stop accepting input and begin building the task.

The ABORT option is the only correct way for you to restart TKB if you discover an error and decide you do not want the Task Builder output.

12.2 ABSPAT—Absolute Patch

You use the ABSPAT option to declare a series of object-level patches starting at a specified base address. You may use this option for conventional or I- and D-space tasks. If used for an I- and D-space task, this option patches I-space. (See also the DSPPAT option.) You can specify up to eight patch values.

Syntax

ABSPAT=seg-name:address:val1:val2...:val8

Parameters

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 to 177777 to be stored at address.

val2
An octal number in the range of 0 to 177777 to be stored at address+2.

.
.
.

val8
An octal number in the range of 0 to 177777 to be stored at address+16.

Note

All patches must be within the segment address limits or TKB generates the following error message:

```
TKB--*DIAG*--Load address out of range in module name
```

12.3 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, TKB allocates approximately 512 bytes.

If you specify fewer 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 a language Object Time System (OTS), such as FORTRAN, 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

Parameter

file-max

A decimal integer indicating the maximum number of files that can be open at the same time.

Default

ACTFIL=4

12.4 ASG—Device Assignment

You use the ASG option to declare the physical device that is assigned to one or more logical units.

Syntax

ASG=device-name:unit-num1:unit-num2...:unit-num8

Parameters

device-name

A 2-character alphabetic device name followed by a 1- to 3-digit decimal unit number. If your task uses more than six logical units, you must use the UNITS option to specify the number of logical units that your task will use.

unit-num1
unit-num2

.
.
.

unit-num8

Decimal integers indicating the logical unit numbers.

Default

ASG=SY0:1:2:3:4,TI0:5,CL0:6

Note

The decimal unit number you use cannot exceed 255, minus the logical units used by your task. For example, if your task uses three logical units in addition to the six logical units TKB provides you by default, your decimal unit number cannot exceed 246.

12.5 CLSTR—System-Owned Cluster of Resident Libraries or Commons

You use the CLSTR option to link your program to one to six shared regions (such as FMS, RMS, FORTRAN, or BASIC-PLUS-2) with a minimum of lost virtual address space for your task. CLSTR allows two to six shared regions to reside in the same virtual address space in your task.

To obtain the required overlay run-time structures in your task, you must define all libraries (except possibly the first) by using memory-resident overlays. Although it can be an overlaid library, the first library need not be overlaid and can be a single-segment structure.

If the first library is overlaid with a null root, the overlay run-time system cannot distinguish between the first library and the other libraries in the cluster (those named in the CLSTR option after the first). Therefore, if the first library called is not the first library named in the CLSTR option, severe performance degradation may be noticed because of excessive mapping and unmapping of the libraries. Therefore, to avoid performance degradation if the first library is overlaid with a null root, make certain that the first library called is the first library named in the CLSTR option.

You use CLSTR to declare a cluster or group of system-owned resident libraries or commons that your task intends to access and have reside at the same virtual address in the address space of your task.

The term "system-owned" means that TKB expects to find the commons or libraries named in the option and the symbol table associated with them in directory [1,1] on device LB.

Syntax

CLSTR=library_1,library_2,...library_n:access-code:apr

Parameters

library_n

The library names must be 1- to 6-character Radix–50 names. TKB expects to find a symbol definition file of the same name for each specified shared region in directory [1,1] on device LB. The first specification denotes the first or the default library, which is the library to which the task is mapped when the task starts up and remaps after any call to another library.

The total number of libraries to which a task may map is seven. The number of the component libraries in clusters is limited to a maximum of six. A cluster must contain a minimum of two libraries. It is possible to have two clusters of three libraries each or three clusters of two libraries each; any combination of number of clusters and libraries must equal at least two or a maximum of six. If six libraries are used in clusters, the task may map to only one other, separate library.

access-code

The code RW (read/write) or the code RO (read-only) indicates the type of access the task requires. All shared regions in the cluster have the same type of access.

Note

A privileged task can change data in or move data to a resident common even though the task has been linked to the common with read-only access.

apr

The apr is an integer in the range of 1 to 7 that specifies the first Active Page Register (APR) that you want TKB to reserve for the cluster of shared regions. You can specify it for a cluster made up of only position-independent shared regions. If you omit the APR parameter and all shared regions are position independent, TKB selects the highest available APR to map the cluster. A cluster can be made up of both position-independent and absolute shared regions. If one absolute shared region is present with position-independent shared regions, the position-independent shared regions assume the same base address as that of the absolute shared region. However, if you specify more than one absolute shared region, all must be built with the same base address.

Default

None

Notes

1. All but the first shared region in a cluster must be memory-resident overlaid libraries. The first shared region specified in the cluster option can be a single-segment structure (nonoverlaid) or an overlaid library.
2. The first library specified in the CLSTR option should be the first library called by the task. If a memory-resident overlaid library has been built with a null root, the install process merely checks for the presence of the required libraries but does not map to them. When the task executes, the first library called becomes the default library, regardless of the order in which the libraries are specified in the CLSTR option statement.

   If the first cluster element specified in the CLSTR statement is not the first library called and that library uses the stack for parameter passing, the information passed between the cluster library elements and the overlay run-time system becomes corrupted, which results in unpredictable behavior of the executing task.

12.6 CMPRT—Completion Routine

You use the CMPRT option to identify a shared region as a supervisor-mode library. The CMPRT option requires an argument that specifies the entry point of the completion routine in the library. The completion routine switches the processor from supervisor mode to user mode and returns program control to the user task after the supervisor-mode library subroutine that was called from the user task has executed.

The following completion routines are available in the system library:

- $CMPCS restores only the carry bit in the user-mode PSW.
- $CMPAL restores all the condition code bits in the user-mode PSW.

These routines perform all the necessary overhead to switch the processor from supervisor mode to user mode and return program control to the user task at the instruction following the call to a supervisor-mode library subroutine.

Although you can write your own completion routines, it is best to use either $CMPCS or $CMPAL whenever possible. Chapter 8 discusses completion routines in detail.

Syntax

```
CMPRT=name
```

Parameter

name
A 1- to 6-character Radix–50 name identifying the completion routine.

Default

None

12.7 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 identifies a shared region that contains only data, and the LIBR option identifies a shared region that contains only code.

If you use the COMMON option with an I- and D-space task, the common is mapped with D-space APRs only and therefore must contain data only.

If you use the LIBR option with an I- and D-space task, the library is overmapped with both I-space and D-space APRs.

The term "system-owned" means that TKB expects to find the common or library named with the option and the symbol definition file associated with it in directory [1,1] on device LB.

Syntax

COMMON=name:access-code[:apr]

or

LIBR=name:access-code[:apr]

Parameters

name

The 1- to 6-character Radix–50 name specifying the common or library. TKB expects to find a symbol definition file having the same name as that of the common or library with a file type of STB in directory [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 change data in or move data 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 to 7 that specifies the first Active Page Register (APR) that you want TKB to reserve for the shared region. TKB 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, TKB selects 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 arguments in the PAR option when the region is built. Refer to PAR in Section 12.21.

Default

None

12.8 DSPPAT—Absolute Patch for D-Space

You use the DSPPAT option to declare a series of object-level patches starting at the specified base address. This option is for making patches to the D-space of an I- and D-space task. You may also use this option to patch a conventional task at any location. You can specify up to eight patch values.

Syntax

DSPPAT=seg-name:address:val1:val2:...:val8

Parameters

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 to 177777 to be stored at address.

val2
An octal number in the range of 0 to 177777 to be stored at address+2.

.
.
.

val8
An octal number in the range of 0 to 177777 to be stored at address+16.

Note

All patches must be within the segment address limits or TKB generates the following error message:

```
TKB--*DIAG*--Load address out of range in module-name
```

12.9 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), TKB extends the section by the number of bytes you specify in the EXTSCT option. If the program section has the attribute OVR (overlay), TKB extends the section only if the length you specify in the EXTSCT option is greater than the length of the program section.

Syntax

EXTSCT=psect-name:extension

Parameters

psect-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.

Default

None

Example

In the following example, the program section BUFF is 200 bytes long.

```
EXTSCT=BUFF:250
```

The number of bytes by which TKB extends the program section BUFF depends on the CON /OVR attribute as follows:

- For CON, the extension is 250 bytes.
- For OVR, the extension is 50 bytes.

TKB extends the program section if it encounters the program section name in an input object file or in the overlay description file.

12.10 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.

This option extends only the D-space of an I- and D-space task.

Syntax

EXTTSK=length

Parameter

length

A decimal number in the range 0 <n <65,535 specifying the increase in task memory allocation (in words).

Default

The task is extended to the size specified in the PAR option (see Section 12.21).

Notes

1. You should not use the EXTTSK option to extend a task containing memory-resident overlays because the system does 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 or the /CHECKPOINT:TASK qualifier, the checkpoint file within the task image is the size of the task plus the size of the extended task area.
3. Be careful when extending an I- and D-space task that is linked to a library which contains both data and instructions. Normally, libraries are mapped in both I-space and D-space, allowing data and instructions to be intermixed. The extension length must not extend into the area mapped for the library or the library will be mapped in I-space only.

12.11 FMTBUF—Format Buffer Size

You use the FMTBUF option to declare the length of the internal working storage area that you want TKB to allocate within your task for compiling format specifications at run time. 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 TKB extends has the reserved name $$OBF1.

Syntax

FMTBUF=max-format

Parameter

max-format

A decimal integer, larger than the default, that specifies the number of characters in the longest format specification.

Default

FMTBUF=132

12.12 GBLDEF—Global Symbol Definition

You use the GBLDEF option to declare the definition of a global symbol.

TKB considers this symbol definition to be absolute. It overrides any definition in your input object modules.

Syntax

GBLDEF=symbol-name:symbol-value

Parameters

symbol-name

A 1- to 6-character Radix–50 name of the defined symbol.

symbol-value

An octal number in the range of 0 to 177777 assigned to the defined symbol.

Default

None

12.13 GBLINC—Include Global Symbols

You use the GBLINC option to direct the Task Builder to include the symbol or symbols specified in the STB file being generated by the link operation in which this option appears. This option is intended for use when creating shared regions, especially shared libraries, when you want to force particular modules to be linked to your task that references this library. The global symbol references specified by this option must be satisfied by some module or GBLDEF specification when you build the task.

Syntax

GBLINC=symbol-name,symbol-name,....,symbol-name

Parameter

symbol-name

The symbol to be included.

Default

None

12.14 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

Parameters

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 to 177777 to be stored at the octal address of the first patch.

val2
An octal number in the range of 0 to 177777 to be stored at the first address+2.

.
.
.

val8
An octal number in the range of 0 to 177777 to be stored at the first address+16.

Default

None

Note

All patches must be within the segment address limits or TKB generates a fatal error.

12.15 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 option is used for memory-resident overlays of shared regions.

Syntax

GBLREF=symbol-name,symbol-name...,symbol-name

Parameter

symbol-name
A 1- to 6-character Radix–50 name of a global symbol reference.

Default

None

12.16 GBLXCL—Exclude Global Symbols

You use the GBLXCL option to direct TKB to exclude from the symbol definition file of a shared region the symbol or symbols specified in the option.

Syntax

GBLXCL=symbol-name,symbol-name...,symbol-name

Parameter

symbol-name

The symbol or symbols to be excluded.

Default

None

12.17 IDENT—Task Identification

You use the IDENT option to change the identification of the task from the one originally specified in the .IDENT MACRO–11 statement in the MAC file to the one specified in the option.

If the IDENT option is not used, TKB uses the identification of the first input MAC file that it encounters.

Syntax

IDENT=name

Parameter

name

Any 1- to 6-character Radix–50 name for use as task identification. You may use any Radix–50 character that is correct for use in the MACRO–11 .IDENT statement.

Default

TKB does not supply a default name. A name must be specified if the IDENT option is used.

12.18 LIBR—System-Owned Library

Refer to the description of the COMMON option in Section 12.7.

12.19 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 a language Object Time System (OTS), such as FORTRAN, 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

Parameter

max-record

A decimal integer, larger than the default, that specifies the maximum record size in bytes.

Default

MAXBUF=133

12.20 ODTV—ODT SST Vector

You use the ODTV option to declare that a global symbol is the address of the ODT synchronous system trap (SST) vector. You must define the global symbol in the main root segment of your task.

Syntax

ODTV=symbol-name:vector-length

Parameters

symbol-name
A 1- to 6-character Radix–50 name of a global symbol.

vector-length
A decimal integer in the range of 1 to 32 specifying the length of the SST vector in words.

Default

None

12.21 PAR—Partition

You use the PAR option to identify the partition for which your task is built.

You can install your task in any system partition or user partition large enough to contain it.

Syntax

PAR=pname[:base:length]

Parameters

pname

The name of the partition.

base

The octal byte address defining the start of the partition. The base must be 0 for a task or a 4K-word boundary for a shared region.

length

The octal number of bytes contained in the partition.

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 automatically extends the length of your task to match the length of the partition. This procedure is equivalent to using the EXTTSK option to increase the task memory. If your task size is greater than the partition size that you specify, TKB generates the following error message:

```
TKB--*DIAG*-Task has illegal memory limits
```

Whether or not the target system is mapped, the Task Builder does not extend the length of a shared region, or any task built without a header, to match the specified partition length.

If you do not specify the base and length, TKB tries 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, TKB can obtain the base and length.

TKB binds the task to the addresses defined by the partition base. If the partition is user controlled, TKB verifies that the task does not exceed the length specification.

A TKB command sequence or build file for a memory-resident overlaid library must contain the statement PAR=xxx, where xxx is the same name as that of the region being built.

Default

PAR=GEN

12.22 PRI—Priority

You use the PRI option to declare your task's execution priority.

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, the system uses this priority.

Syntax

PRI=priority-number

Parameter

priority-number
A decimal integer in the range of 1 to 250.

Default

Established by INSTALL; refer to the *RSX-11M-PLUS MCR Operations Reference Manual,* the *RSX-11M-PLUS Command Language Manual,* or the *Micro/RSX User's Guide.*

12.23 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 identifies a shared region that contains only data and the RESLIB option identifies a shared region that contains only code.

If you use the RESCOM option with an I- and D-space task, the common is mapped with D-space APRs only and therefore must contain data only.

If you use the RESLIB option with an I- and D-space task, the library is overmapped with both I-space and D-space APRs.

The term "user-owned" means that the resident common or library and the symbol definition file associated with it can reside in any directory that you choose. You can specify the directory 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]

Parameters

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 change data in or move data into 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 to 7 that specifies the first Active Page Register (APR) that you want TKB to reserve for the common or library. You can specify it only for position-independent shared regions. If the APR parameter is omitted and the shared region is position independent, TKB selects 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 arguments in the PAR option when the region is built. Refer to PAR in Section 12.21.

Default

When you omit portions of the file specification, the following defaults apply:

- Directory—Taken from the current terminal UIC
- Device—SY0
- File type—TSK
- File version—Latest

Notes

1. The Task Builder expects to find a symbol definition file having the same name as that of the memory image file, but with a file type of STB, on the same device and in the same directory as that of the memory image file.
2. Regardless of the version number you give in the file specification, TKB uses the latest version of the STB file.

12.24 RESLIB—Resident Library

Refer to the description of the RESCOM option in Section 12.23.

12.25 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 in any directory that you choose. You can specify the directory and the remaining portions of the file specification. You must not place comments on the line with RESSUP.

Syntax

RESSUP=file-specification/[-]SV[:apr]

Parameters

file-specification

The memory image file of the supervisor-mode library. The file specification has the standard format discussed in Chapter 1.

/[-]SV

The code /SV or /-SV to indicate whether TKB includes mode-switching vectors within the user task. If you specify /SV, TKB includes a 4-word mode-switching vector within the address space of the user task for each call to a supervisor-mode library subroutine. If you specify /-SV, you must provide your own mode-switching vector. Providing your own mode-switching vectors is useful if your library contains threaded code. However, it is best to use the system-supplied vectors whenever possible.

apr

An integer in the range of 0 to 7 that specifies the first supervisor Active Page Register (APR) that you want TKB to reserve for your supervisor-mode library. You can specify an APR only for position-independent supervisor-mode libraries. The default is the lowest available APR.

The library at virtual 0 must have the CSM (change supervisor mode) dispatcher present in the system-supplied completion routine described in Chapter 8.

Default

When you omit portions of the file specification, the following defaults apply:

- Directory—Taken from the current terminal UIC
- Device—SY0
- File type—TSK
- File version—Latest

Notes

1. The Task Builder expects to find a symbol definition file having the same name as that of the memory image file, but with a file type of STB, on the same device and in the same directory as that of the memory image file.
2. Regardless of the version number you give in the file specification, TKB uses the latest version of the STB file.

12.26 RNDSEG—Round Segment

You use the RNDSEG option to cause TKB to round the size of a named segment up to the nearest APR boundary while building a resident library.

When you install a resident library, INSTALL makes an entry for the resident library in the Common Block Directory (CBD). The system loads the resident library when a task that uses it runs.

The length parameter for the common block, as described in the label block for the task image, must match the corresponding parameter in the system CBD. If the task's label block data does not match the system data for that task, the task cannot be installed.

If you do not use RNDSEG and the common block length for the newly rebuilt library is not the same as the common block length previously recorded in the CBD, you have to relink the task with the new library before you can install it.

Syntax

RNDSEG=seg-name

seg-name
The 1- to 6-character Radix–50 name of the segment.

Default

None

Notes

1. The RNDSEG option operates only during a library build. Attempting to use the option while building any other form of task will result in the following diagnostic error message:

   ```
   TKB -- *DIAG* - Library build not requested - ignoring option
   RNDSEG=SEG1
   ```

2. If you attempt to specify a nonexistent segment name, the following diagnostic error will be generated and the build will continue:

   ```
   TKB -- *DIAG* - Segment not found to address round
   RNDSEG=NOSEG
   ```

12.27 ROPAR—Read-Only Partition

You use the ROPAR option to declare the partition in which the read-only portion of your multiuser task is to reside.

Syntax

ROPAR=parname

Parameter

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.

12.28 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

Parameter

stack-size
A decimal integer specifying the number of words required for the stack.

Default

STACK=256

12.29 SUPLIB—Supervisor-Mode Library

You use the SUPLIB option to declare that your task intends to access a system-owned supervisor-mode library. The term "system-owned" means that TKB expects to find the supervisor-mode library and the symbol definition file associated with it in directory [1,1] on device LB.

Syntax

SUPLIB=name:[-]SV[:apr]

Parameters

name

The 1- to 6-character Radix–50 name specifying the system-owned, supervisor-mode library. TKB expects to find a symbol definition file having the same name as that of the library with a file type of STB in directory [1,1] on device LB.

[-]SV

The code SV or -SV to indicate whether TKB includes mode-switching vectors within the user task. If you specify SV, TKB includes a 4-word mode-switching vector within the address space of the user task for each call to a supervisor-mode library subroutine. If you specify -SV, you must provide your own mode-switching vector. Providing your own mode-switching vectors is useful if your library contains threaded code. However, it is best to use the system-supplied vectors whenever possible.

apr

An integer in the range of 0 to 7 that specifies the first supervisor Active Page Register (APR) that TKB is to reserve for the library. You can specify an APR only for position-independent supervisor-mode libraries. The default is the lowest available APR.

The library at virtual 0 must have the CSM (change supervisor mode) dispatcher present in the system-supplied completion routine described in Chapter 8.

Default

None

12.30 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

Parameter

task-name
A 1- to 6-character name identifying your task.

Default

The first six characters of the task image file name identify the task when the task is installed.

12.31 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

Parameters

symbol-name

A 1- to 6-character Radix–50 name of a global symbol.

vector-length

A decimal integer in the range of 1 to 32 specifying the length of the SST vector in words.

Default

None

12.32 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]

Parameters

group

An octal number in the range of 1 to 377, or a decimal number in the range of 1 to 255. Decimal numbers must be followed by a period (.).

member

An octal number in the range of 1 to 377, or a decimal number in the range of 1 to 255. Decimal numbers must be followed by a period (.).

Default

The UIC that the Task Builder is running under (normally the terminal UIC).

12.33 UNITS—Logical Unit Usage

You use the UNITS option to declare the number of logical units that your task uses.

Syntax

UNITS=max-units

Parameter

max-units

A decimal integer in the range of 0 to 250 specifying the maximum number of logical units. A 2-word block is allocated in the task's header for every logical unit. A task that uses many logical units can use a significant portion of dynamic memory because the header is in dynamic memory when the task is executing. The /XH switch or /EXTERNAL qualifier affects pool usage by the task header.

Default

UNITS=6

12.34 VARRAY—Virtual Array Specification and Usage

A virtual array in FORTRAN is a defined area outside of the virtual address space of a task, but it is within the task's logical address space. TKB assigns the name $VIRT to the virtual array and positions it in memory adjacent to the task header. The VARRAY=OVR option specifies an overlaid virtual array such that the virtual array may be used in a way similar to the use of a FORTRAN COMMON. Using the virtual array in this way means that each segment of an overlaid task that uses the virtual array defines the array in the same way as it is defined in the root segment. Thereafter, the segment may access the array directly without passing arguments, as is necessary when the array has the concatenated attribute (the default, VARRAY=CON).

To use the VARRAY option with the OVR attribute as VARRAY=OVR, you must first define the array (for example, VIRTUAL DATA(10)), in the root segment of the task. Then, you must define the array in the same way in each segment of the overlaid task that uses the virtual array. Example 12-1, A Task Using a Virtual Array with the OVR Attribute, shows a way in which a virtual array may be directly accessed by segments in a task. Example 12-1 also shows the TKB command line and overlay description file for building the task.

Using the VARRAY option with the CON attribute as VARRAY=CON (the default operation) results in a virtual array subject to the restrictions and uses that are described in the reference manual for the specific kind of FORTRAN that you are using.

Syntax

VARRAY=option

Parameter

option
Either OVR or CON.

Default

VARRAY=CON

Example 12–1: A Task Using a Virtual Array with the OVR Attribute

```
C
C Program to test the Task Builder option VARRAY
C
     PROGRAM MAIN
     IMPLICIT INTEGER *2 (A-Z)
     VIRTUAL DATA(10)
     CALL INPUT
     CALL CALC
     CALL OUTPUT
     CALL EXIT
     END

     SUBROUTINE INPUT
     IMPLICIT INTEGER *2 (A-Z)
     VIRTUAL DATA(10)
     TYPE 10
10   FORMAT (1X,'Input I '$)
     ACCEPT 20,DATA(1)
20   FORMAT (I2)
     TYPE 30
30   FORMAT (1X,'Input J '$)
     ACCEPT 20,DATA(2)
     RETURN
     END

     SUBROUTINE CALC
     IMPLICIT INTEGER *2 (A-Z)
     VIRTUAL DATA(10)
     DATA(3) = DATA(1) + DATA(2)                  !I + J
     DATA(4) = DATA(1) - DATA(2)                  !I - J
     DATA(5) = DATA(1) * DATA(2)                  !I * J
     DATA(6) = DATA(1) / DATA(2)                  !I / J
     RETURN
     END
```

(Continued on next page)

Example 12–1 (Cont.): A Task Using a Virtual Array with the OVR Attribute

```
      SUBROUTINE OUTPUT
      IMPLICIT INTEGER *2 (A-Z)
      VIRTUAL DATA(10)
      TYPE 10,DATA(3),DATA(4),DATA(5),DATA(6)
10    FORMAT (1X,'I + J =',I6,/,1X,'I - J =',I6,/
      1,1X'I * J =',I6,/,1X,'I / J =',I6)
      RETURN
      END


;
; Command file MAINFT.CMD to build MAINFT.TSK
;
MAINFT/FP,MAINFT/MA/-WI=MAINFT/MP
VARRAY=OVR
//
;
;
; Overlay description file MAINFT.ODL for MAINFT.TSK
;
        .ROOT   $MAIN-*($INPT,$CALC,$OUTP)
$MAIN:  .FCTR   MAIN-LB:[1,1]F770TS/LB
$INPT:  .FCTR   INPUT-LB:[1,1]F770TS/LB
$CALC:  .FCTR   CALC-LB:[1,1]F70TS/LB
$OUTP:  .FCTR   OUTPUT-LB:[1,1]F770TS/LB
        .END
```

12.35 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 Chapter 5 for more information on virtual program sections.

Syntax

VSECT=p-sect-name:[base:window][:physical-length]

Parameters

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 to 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.

physical-length

An octal value specifying the minimum amount of physical memory to be allocated to the section in units of 64-byte blocks. TKB 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, TKB assumes a value of 0.

Default

Physical-length defaults to 0.

12.36 WNDWS—Number of Address Windows

You use the WNDWS option to declare 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

Parameter

n

A value in the range of 1 to 23.

Default

WNDWS=0

Appendix A
Task Builder Input Data Formats

An object module is the fundamental unit of input to the Task Builder (TKB). You create an object module by using any of the standard language processors (for example, MACRO–11 or FORTRAN) or by using TKB itself (the symbol definition file). The RSX–11M–PLUS and Micro/RSX librarian (LBR) gives you the capability to combine a number of object modules into a single library file.

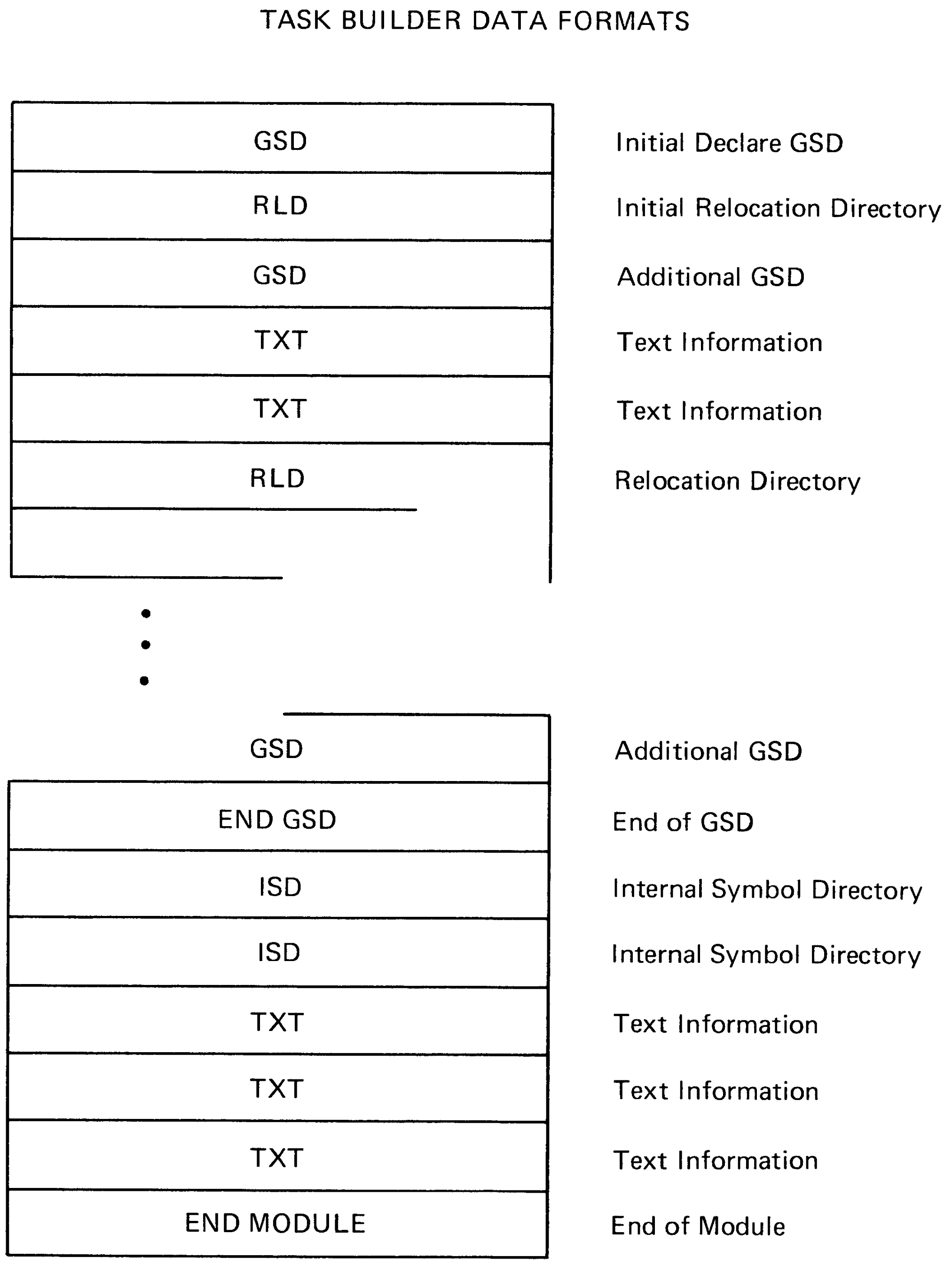
An object module consists of variable-length records of information that describe the contents of the module. These records guide TKB in translating the object language into a task image. Six record (block) types are included in the object language, as follows:

- 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)

TKB 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.

Figure A-1: General Object Module Format



ZK-444-81

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. Variable-length records in an object file should not be longer than 128_{10} bytes. If TKB attempts to read an object record longer than 128 bytes, the following error message results:

```
TKB -- *FATAL*-I/O error on input file  file-name
```

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 TKB to assign addresses to global symbols and to allocate the virtual address space required by a task.

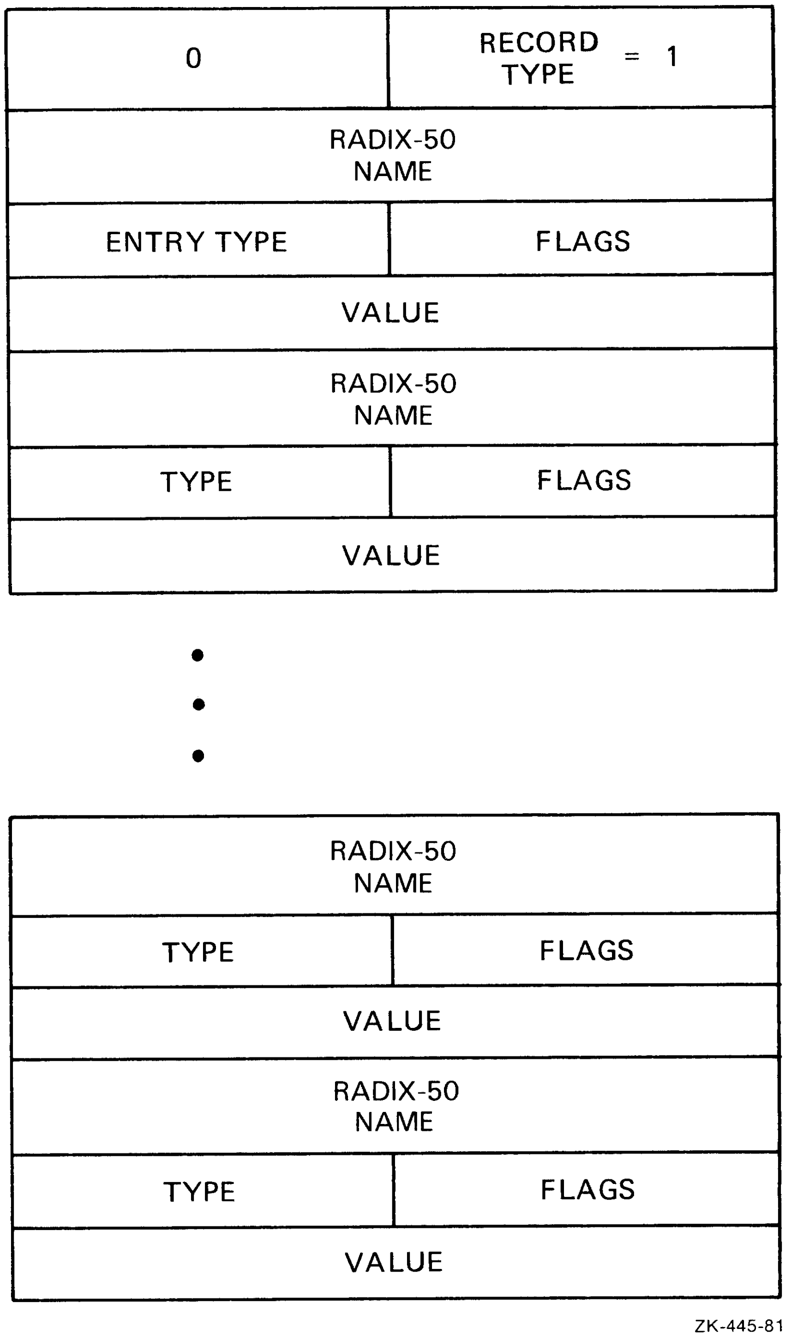
GSD records are the only records processed by TKB in its first pass. Therefore, you can save substantial time by placing all GSD records at the beginning of a module (because the Task Builder has to read less of the file).

GSD records contain nine types of entries, as follows:

- 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)

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.

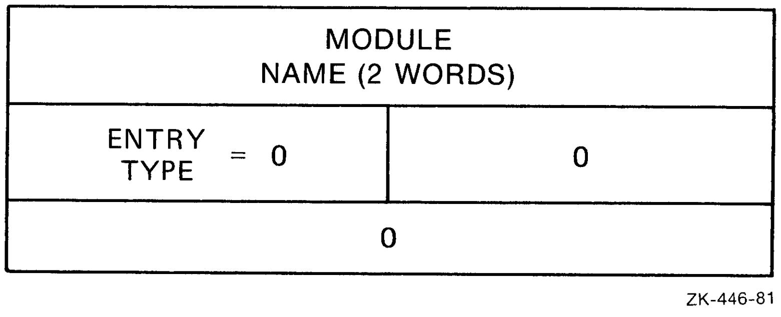
Figure A–2: Global Symbol Directory Record Format



A.1.1 Module Name (Type 0)

The module name entry (two words) 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 name entry format.

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 by program sections. For compatibility with other systems, TKB 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 follows:

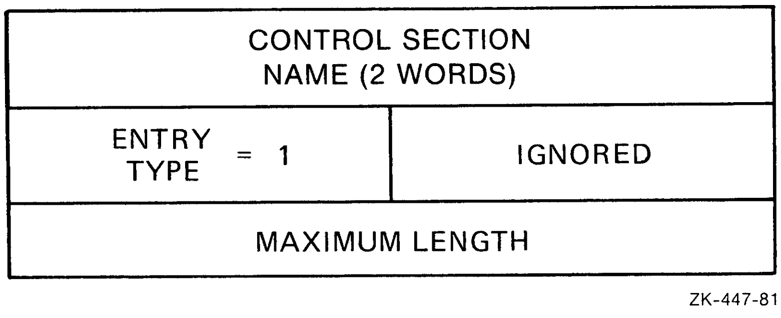
  ```
  .PSECT name, GBL,REL,OVR,RW,I,LOW
  ```

- For an ASECT, the program section is defined as follows:

  ```
  .PSECT . ABS.,GBL,ABS,I,OVR,RW,LOW
  ```

TKB processes ASECTs and CSECTs as program sections with the fixed attributes defined above. Figure A-4 illustrates the control section name entry format.

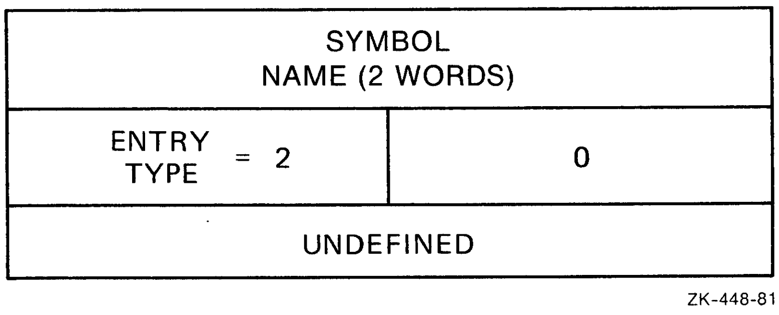
Figure A-4: Control Section Name Entry Format



A.1.3 Internal Symbol Name (Type 2)

The internal symbol name entry (two words) declares the name of an internal symbol (with respect to the module). TKB does not support internal symbol tables; therefore, the detailed format of this entry is undefined. If TKB 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



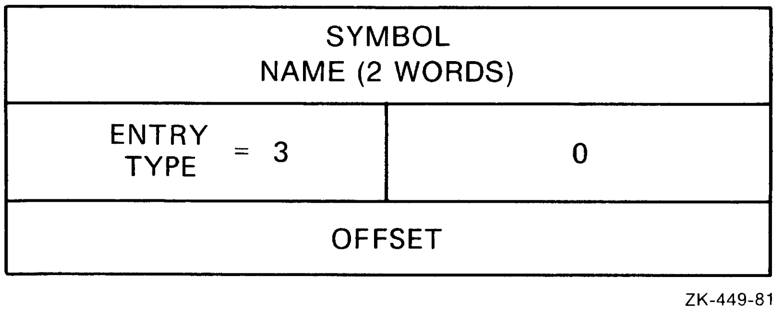
A.1.4 Transfer Address (Type 3)

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).

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 of the third word 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.

Figure A–7: Global Symbol Name Entry Format

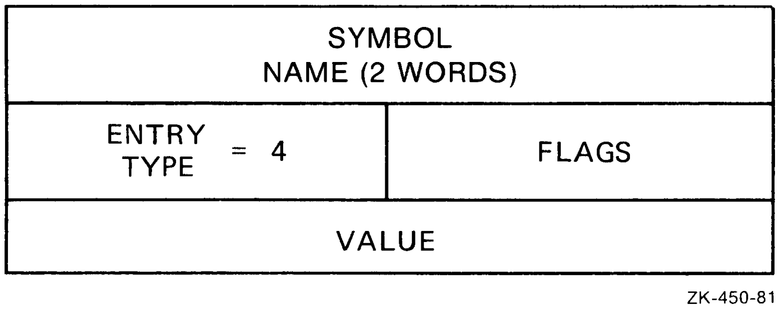


Table A-1 lists the bit assignments of the flag byte of the symbol-declaration entry.

Table A–1: Symbol Declaration Flag Byte—Bit Assignments

Bit Number	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. TKB 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.

You must construct GSD records 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.

Figure A-8: Program Section Name Entry Format

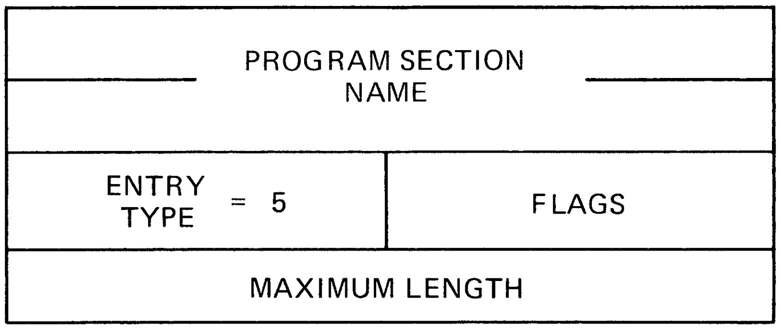


Table A-2 lists the bit assignments of the flag byte of the program section name entry.

Table A-2: Program Section Name Flag Byte—Bit Assignments

Bit Number	Name	Setting	Meaning
0	Save	0	Normal program section.
		1	The program section is forced into the root of the task.
1	Library program section	0	Normal program section.
		1	The program section is relocatable and refers to a shared region.
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.
3			Not used; reserved for future DIGITAL use.
4	Access	0	The program section has read/write access.
		1	The program section has read-only access.
5	Relocation	0	The program section is absolute and requires no relocation.

Table A-2 (Cont.): Program Section Name Flag Byte—Bit Assignments

Bit Number	Name	Setting	Meaning
		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. TKB collects references to the program section across segment boundaries. The Task Builder determines the segment in which storage is allocated for a global program section 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 ODL .PSECT directive.
7	Type	0	The program section contains instruction (I) references.
		1	The program section contains data (D) references.

Note

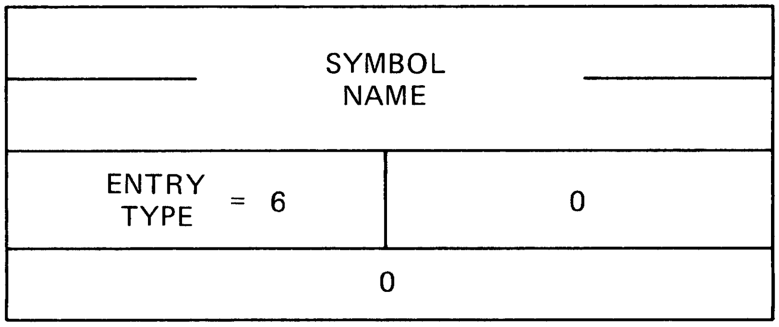
The length of all absolute sections is 0.

A.1.7 Program Version Identification (Type 6)

The program version identification entry declares the version of the module. TKB saves the version identification of the first module that defines a nonblank version. It then includes this identification on the memory allocation file (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.

Figure A-9: Program Version Identification Entry Format



ZK-452-81

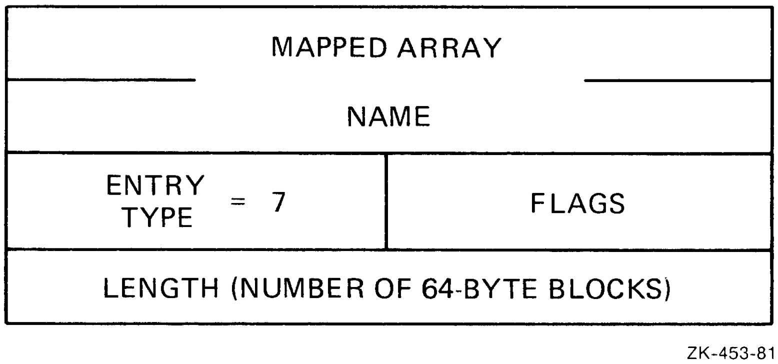
A.1.8 Mapped Array Declaration (Type 7)

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 0.

One additional window block is allocated whenever a mapped array is declared.

Figure A-10 illustrates the mapped array declaration entry format.

Figure A-10: Mapped Array Declaration Entry Format



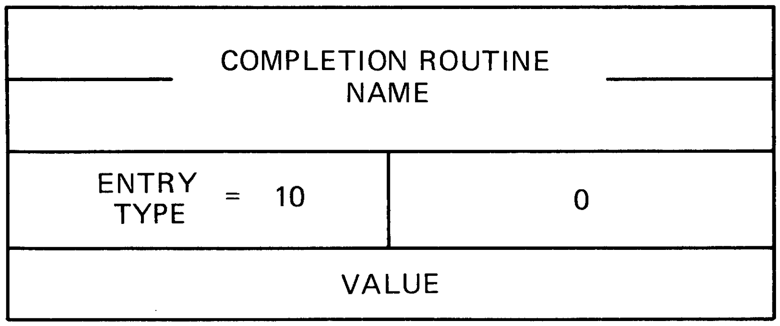
ZK-453-81

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

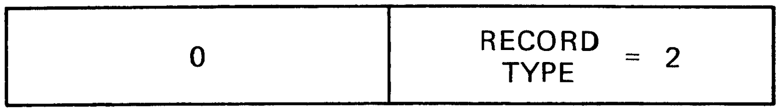


ZK-454-81

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 in length.

Figure A-12: End of Global Symbol Directory Record Format



ZK-455-81

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.

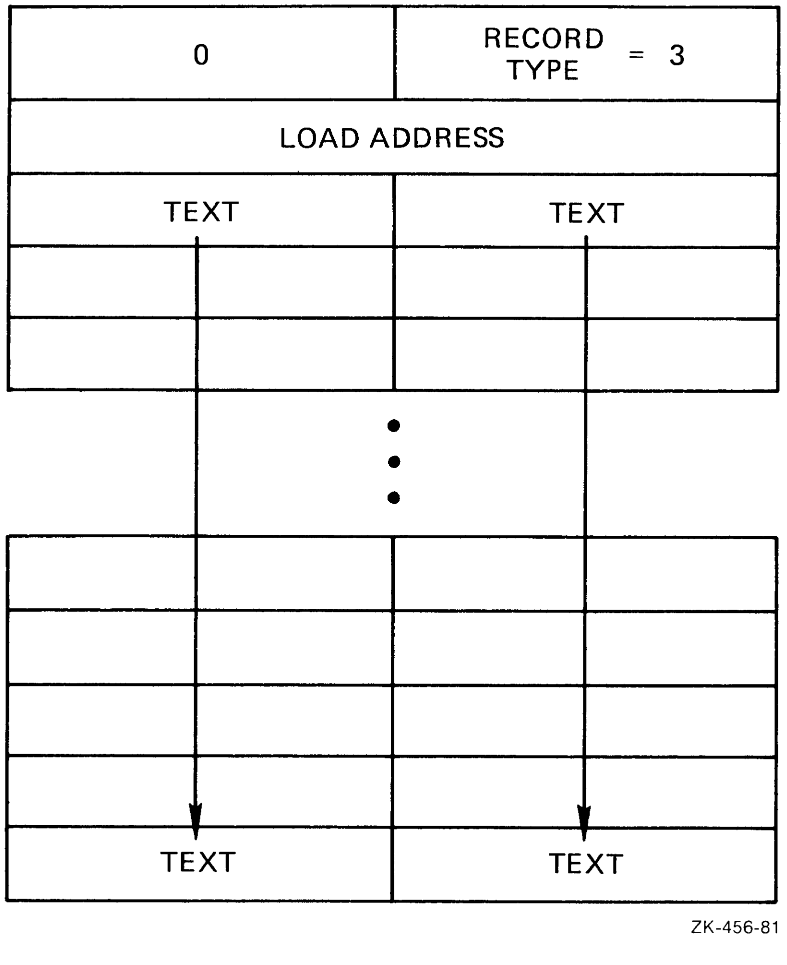
TXT records can contain words and/or bytes of information whose final contents have not yet been determined. This information will be bound by a relocation directory record that immediately follows the text record (see Section A.4). If the TXT record needs no modification, then no relocation directory 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.

TKB 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 entries, classified as relocation or location modification entries. The types of entries are as follows:

- 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 that 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.

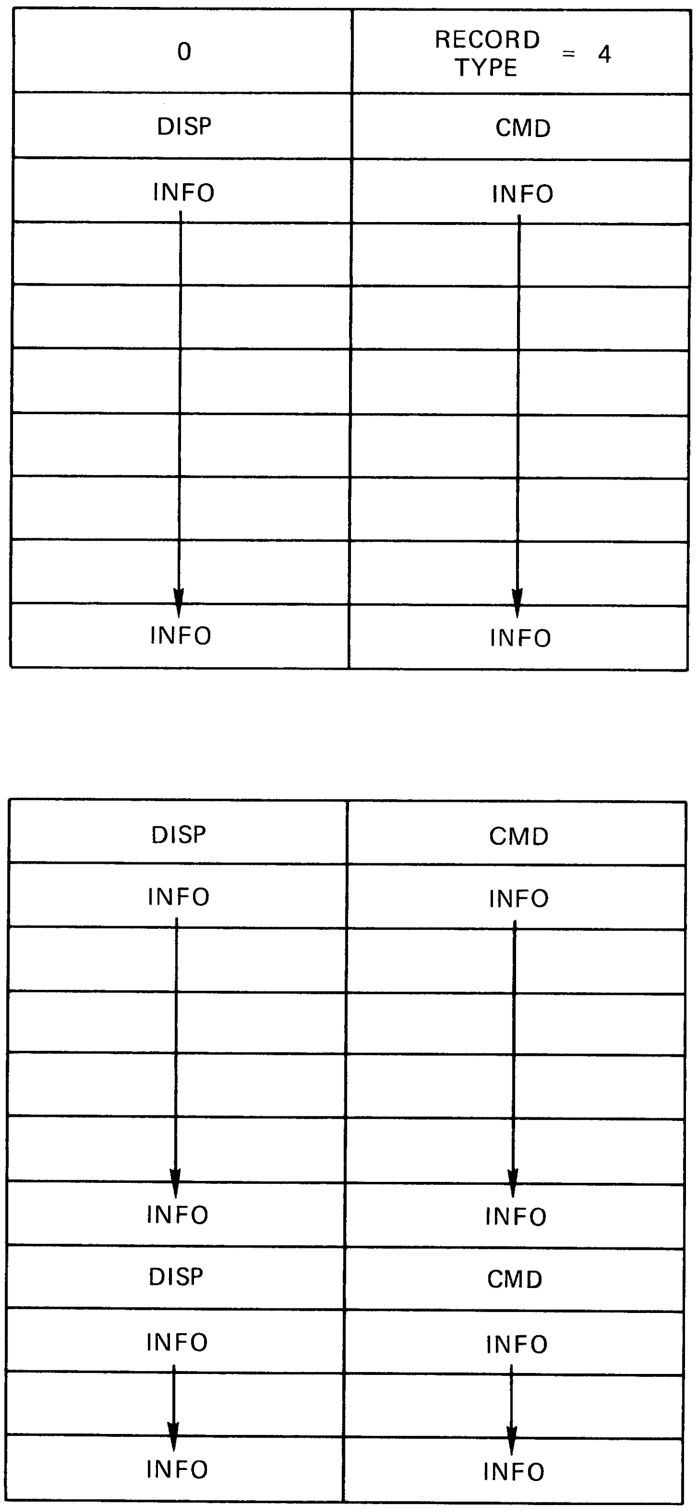
Table A-3 lists the bit assignments of the command byte of each RLD entry.

Table A-3: Relocation Directory Command Byte—Bit Assignments

Bit Number	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 one byte. TKB 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. TKB 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:

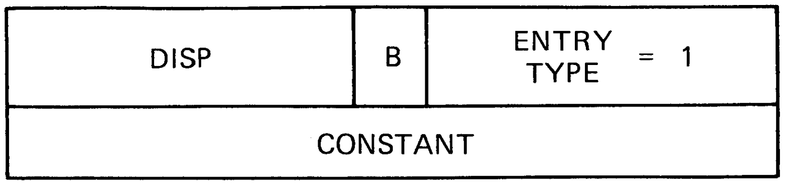
```
A:      MOV     #A,R0
```

or

```
        .WORD   A
```

Figure A-15 illustrates the internal relocation entry format.

Figure A-15: Internal Relocation Entry Format



ZK-458-81

A.4.2 Global Relocation (Type 2)

The global relocation entry relocates a direct pointer to a global symbol. TKB obtains the definition of the global symbol and writes the result into the task image file at the calculated address.

For example:

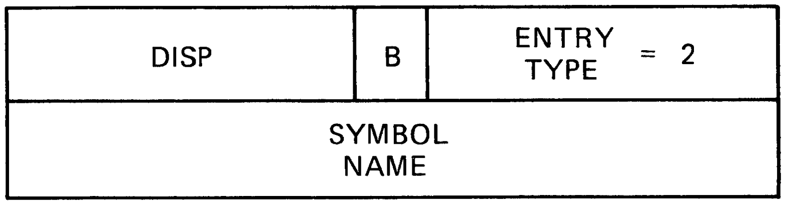
```
MOV     #GLOBAL,R0
```

or

```
.WORD   GLOBAL
```

Figure A-16 illustrates the global relocation entry format.

Figure A-16: Global Relocation Entry Format



ZK-459-81

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. TKB 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:

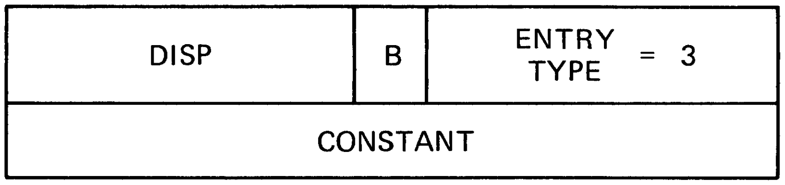
```
CLR     177550
```

or

```
MOV     177550,R0
```

Figure A-17 illustrates the internal displaced relocation entry format.

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. TKB 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 intô the task image file at the calculated address.

For example:

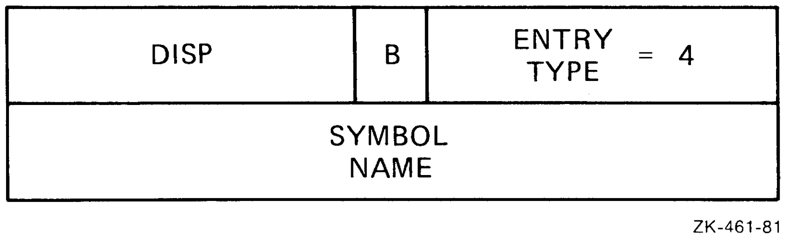
```
CLR     GLOBAL
```

or

```
MOV     GLOBAL,R0
```

Figure A-18 illustrates the global displaced relocation entry format.

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. TKB 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:

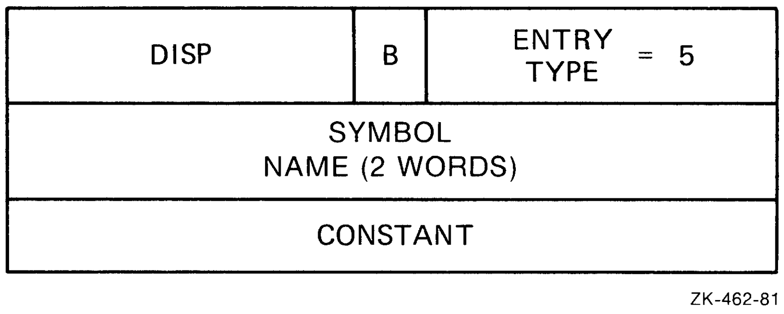
```
MOV     #GLOBAL+2,R0
```

or

```
.WORD   GLOBAL-4
```

Figure A-19 illustrates the global additive relocation entry format.

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. TKB 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 resulting additive value, and writes the result into the task image file at the calculated address.

For example:

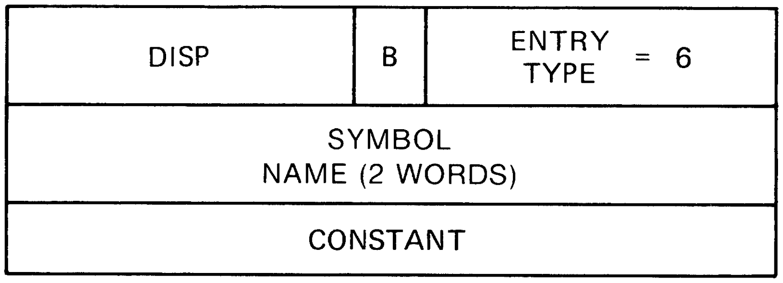
```
CLR     GLOBAL+2
```

or

```
MOV     GLOBAL-5,RO
```

Figure A-20 illustrates the global additive displaced relocation entry format.

Figure A-20: Global Additive Displaced Relocation Entry Format



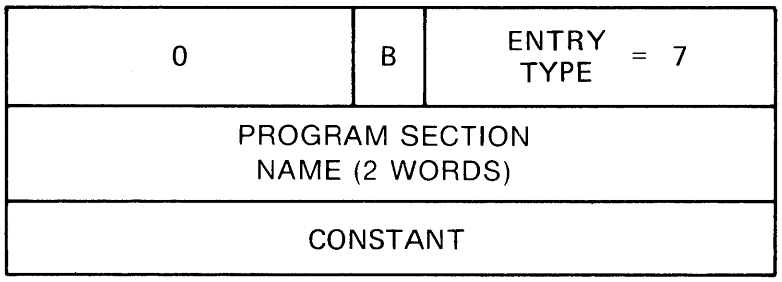
ZK-463-81

A.4.7 Location Counter Definition (Type 7)

The location counter definition entry declares a current program section and location counter value. TKB 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



ZK-464-81

A.4.8 Location Counter Modification (Type 10)

The location counter modification entry modifies the current location counter. TKB adds the current program section base to the specified constant and stores the result as the current location counter.

For example:

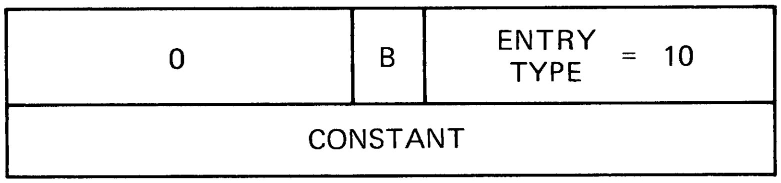
```
.=.+N
```

or

```
.BLKB   N
```

Figure A-22 illustrates the location counter modification entry format.

Figure A–22: Location Counter Modification Entry Format



ZK-465-81

A.4.9 Program Limits (Type 11)

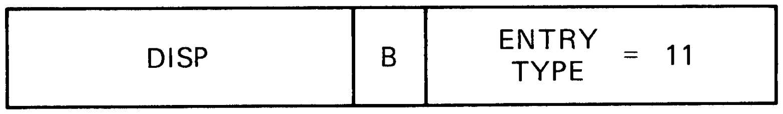
The program limits entry is generated by the .LIMIT assembler directive. TKB obtains the first address above the header (normally the beginning of the stack) and the highest address allocated to the task. It then writes these two addresses into the task image file at the calculated address and at the calculated address plus 2, respectively.

For example:

```
.LIMIT
```

Figure A-23 illustrates the program limits entry format.

Figure A–23: Program Limits Entry Format



ZK-466-81

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. TKB obtains the current base address of the specified program section and writes it into the task image file at the calculated address.

For example:

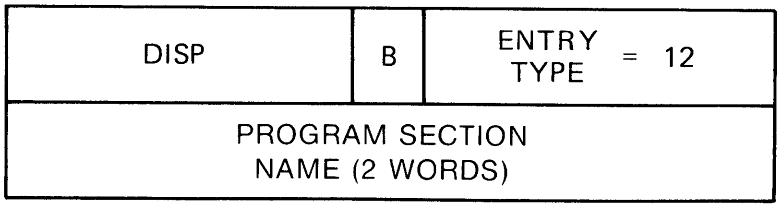
```
        .PSECT A
B:
        .
        .
        .
        .PSECT C
        MOV #B,RO
```

or

```
        .WORD B
```

Figure A-24 illustrates the program section relocation entry format.

Figure A–24: Program Section Relocation Entry Format



ZK-467-81

A.4.11 Program Section Displaced Relocation (Type 14)

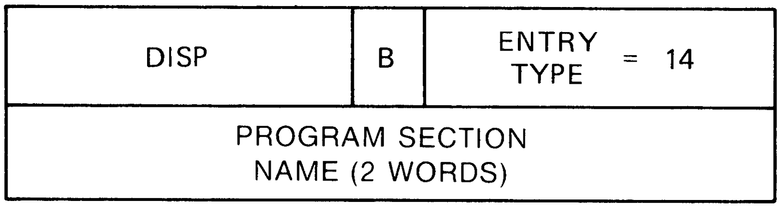
The program section displaced relocation entry relocates a relative reference to the beginning address of another program section within a module. TKB obtains the current base address of the specified program section, subtracts the address plus 2 that the relocated value is to be written into from the 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



ZK-468-81

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. TKB 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:

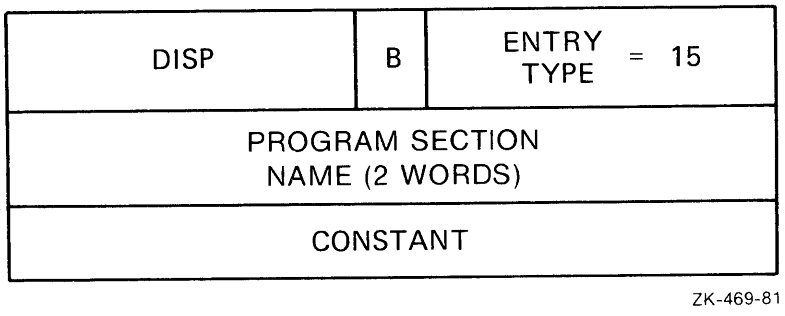
```
        .PSECT A
B:
        .
        .
        .
C:
        .
        .
        .
        .PSECT D
MOV     #B+10,R0
MOV     #C,R0
```

or

```
        .WORD B+10
        .WORD C
```

Figure A-26 illustrates the program section additive relocation entry format.

Figure A-26: Program Section Additive Relocation Entry Format



A.4.13 Program Section Additive Displaced Relocation (Type 16)

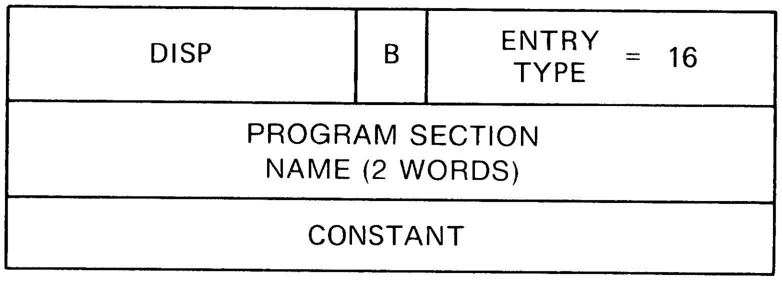
The program section additive displaced relocation entry relocates a relative reference to an address in another program section within a module. TKB obtains the current base address of the specified program section, adds this address to the specified constant, subtracts the address plus 2 that the relocated value is to be written into from the resulting 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,RO
        MOV C,RO
```

Figure A-27 illustrates the program section additive displaced relocation entry format.

Figure A-27: Program Section Additive Displaced Relocation Entry Format



ZK-470-81

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 as follows:

- No operation—Value 0
- Addition (+)—Value 1
- Subtraction (-)—Value 2
- Multiplication (*)—Value 3
- Division (/)—Value 4
- Logical AND (&)—Value 5
- Logical inclusive OR (!)—Value 6
- Negation (-)—Value 10

- Complement (^C)—Value 11
- Store result (command termination)—Value 12
- Store result with displaced relocation (command termination)—Value 13
- Fetch global symbol—Value 16 (It is followed by four bytes containing the symbol name in Radix–50 representation.)
- Fetch relocatable value—Value 17 (It is followed by one byte containing the program section number, and two bytes containing the offset within the program section.)
- Fetch constant—Value 20 (It is followed by two bytes containing the constant.)
- Fetch resident library base address—Value 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 two's complement arithmetic. The results are equivalent to expressions that the assembler evaluates internally. The following rules should be noted:

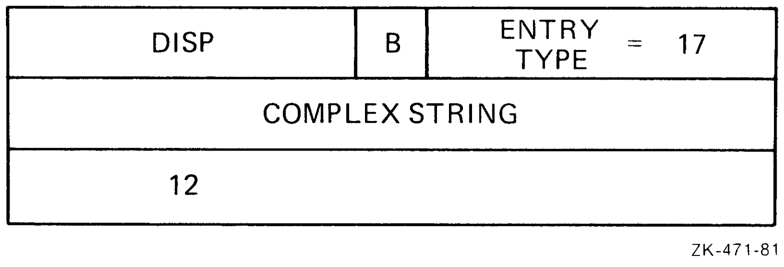
1. An attempt to divide by 0 yields a 0 result. The Task Builder issues a nonfatal diagnostic error message.
2. All results are truncated from the left to fit into 16 bits. No diagnostic error message is issued if the number is too large. If the result modifies a byte, TKB checks for truncation errors as described in Table A–3.
3. All operations are performed on relocated (additive) or absolute 16-bit quantities. Displacement of the program counter (PC) is applied to the result only.

For example:

```
        .PSECT  ALPHA
A:
        .
        .
        .
        .PSECT  BETA
B:
        .
        .
        .
        MOV     #A+B-G1/G2&^C177120!G3>>,R1
```

Figure A-28 illustrates the complex relocation entry format.

Figure A-28: Complex Relocation Entry Format



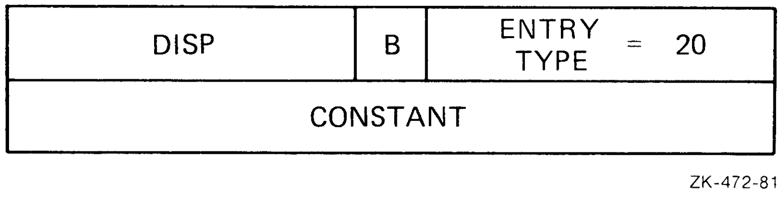
A.4.15 Resident Library Relocation (Type 20)

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), TKB 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, TKB uses the task base address.

Figure A-29 illustrates the library relocation entry format.

Figure A-29: Resident Library Relocation Entry Format



A.5 Internal Symbol Directory Record

Internal symbol directory (ISD) records have two purposes, as follows:

- To pass information to symbolic debuggers by means of the STB file
- To create autoload vectors dynamically for the entry points of the library

TBK looks for global symbol definitions in the input object modules and looks for ISD records if you specify the /DA switch or /DEBUG qualifier; otherwise, TKB ignores the ISD records. Some ISD records require no relocation and TKB can copy them directly into the STB file. Others will require modification; after being modified, they can be written to the STB file. In addition, TKB may need to generate some ISD records of its own in the STB file.

Except for autoloadable library entry points, TKB puts ISD records into the STB file only if you use the /DA switch or the /DEBUG qualifier. When TKB outputs the STB file, it writes three major types of ISD records, as follows:

- Type 1 records, TKB-generated ISDs. The form of these records is language independent.

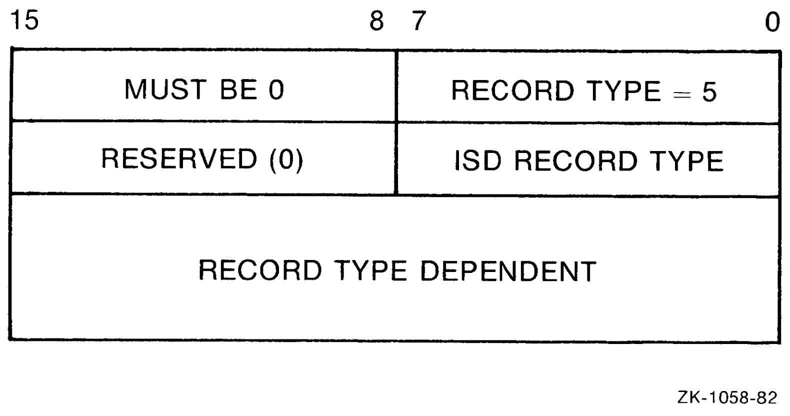
- Type 3 records, written for any type 2 records in an input object module. TKB does this after adding data and then changing the type to 3. Type 2 relocatable/relocated records are those that contain both language-dependent and language-independent sections. Language processors generate these records and TKB modifies them. They contain information that can be used to find the absolute task image address of source program entities (variables, program statements, and so on).
- Type 4 records, written to the STB file without modification. Type 4 records are literal records that contain language-dependent information. Apart from the first few bytes, TKB ignores the rest of the record.

These record formats are described in the following sections.

A.5.1 Overall Record Format

ISD records have the same basic structure as all object language records. Because of the variety of different types, the skeleton structure must include additional fields that are common to all ISD record types. The general format of all ISD records is shown in Figure A-30.

Figure A-30: General Format of All ISD Records



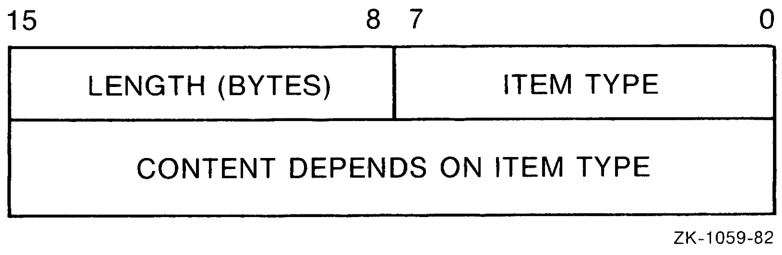
ISD record types fall into general categories. The categories are as follows:

- 0—Illegal.
- 1—TKB-generated.
- 2—Compiler-generated relocatable.
- 3—Relocated (type 2 after TKB processing).
- 4-127—Not defined and reserved for future use.
- 128-255—Literal records: the type code identifies the generating language processor and the internal structure.

A.5.2 TKB-Generated Records (Type 1)

This record type contains a string of individual items, each with its own format. The items are either start-of-segment items, task identification items, or autoloadable entry point items. The TKB-generated record is similar to the structure of an RLD or GSD record. The general format is shown in Figure A-31.

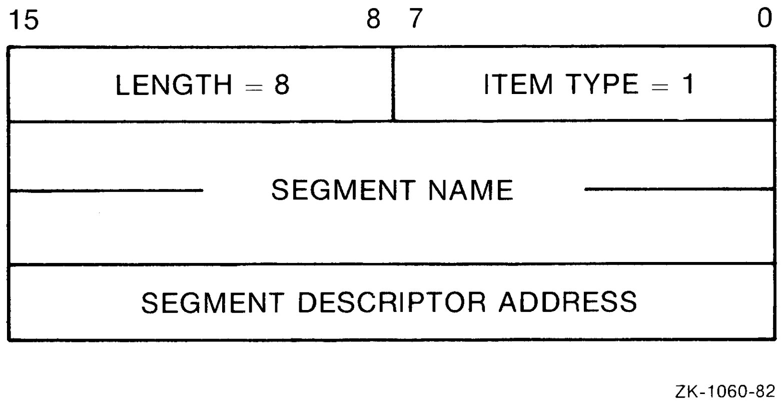
Figure A-31: General Format of a TKB-Generated Record



A.5.2.1 Start-of-Segment Item (1)

The format of the start-of-segment item is shown in Figure A-32.

Figure A-32: Format of TKB-Generated Start-of-Segment Item (1)



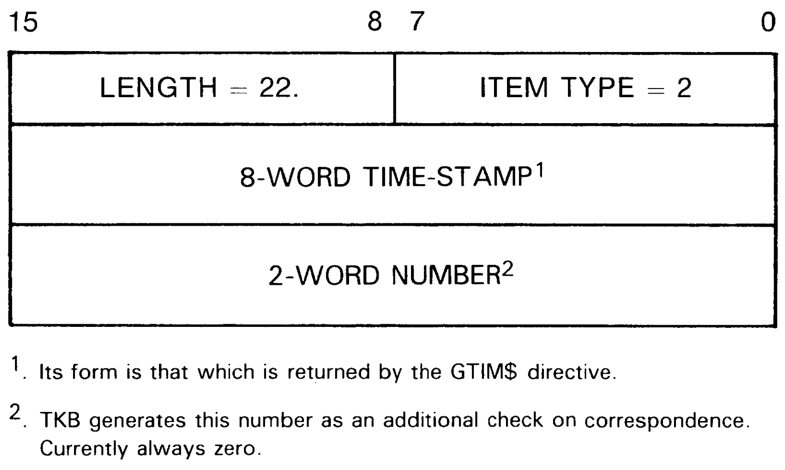
A.5.2.2 Task Identification Item (2)

The task identification item ensures that an STB file and the task image being debugged were generated at the same time. Otherwise, symbols that are found may not correspond to the actual task.

The task identification item exists to make the correlation between the STB file and its related task possible. The contents of this item correspond exactly to the first 10 words of an area in a task image file, which is in the TKB-created program section called $$DBTS.

The format of the task identification item is shown in Figure A-33.

Figure A-33: Format of TKB-Generated Task Identification Item (2)

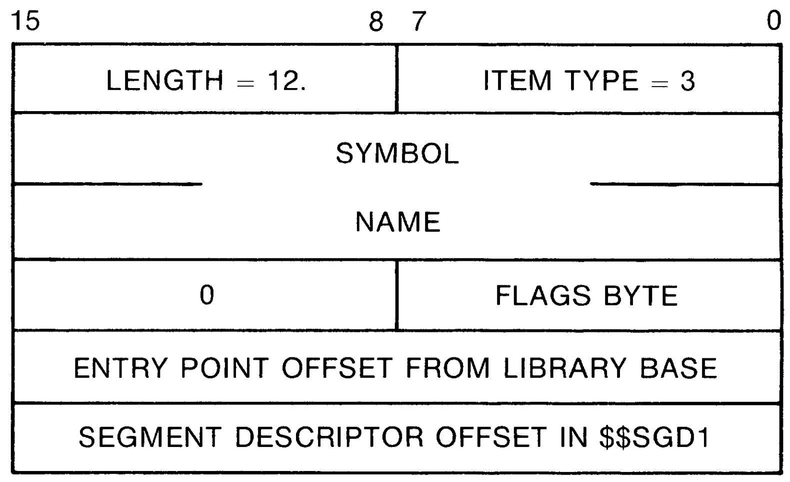

15
8 7
0
LENGTH = 22.
ITEM TYPE = 2
8-WORD TIME-STAMP1
2-WORD NUMBER2
1. Its form is that which is returned by the GTIM$ directive.
2. TKB generates this number as an additional check on correspondence. Currently always zero.
ZK-1061-82

A.5.2.3 Autoloadable Library Entry Point Item (3)

TKB outputs the autoloadable library entry point item into an STB file when building overlaid resident libraries. The ISD record contains the needed information for TKB to dynamically generate autoload vectors for entry points in the library. Autoload vectors appear only for those entry points that are referenced by the task. Unlike the other items, the autoloadable library entry point item is not for use by debuggers.

The format of the autoloadable entry point item is shown in Figure A-34.

Figure A-34: Format of an Autoloadable Library Entry Point Item (3)


15
8 7
0
LENGTH = 12.
ITEM TYPE = 3
SYMBOL
NAME
0
FLAGS BYTE
ENTRY POINT OFFSET FROM LIBRARY BASE
SEGMENT DESCRIPTOR OFFSET IN $$SGD1

ZK-1062-82

A.5.3 Relocatable/Relocated Records (Type 2)

These records are the central part of TKB's involvement in debugger communication. Every item in these records must be standardized, and only standard items can appear. The general format of relocatable/relocated records is the same as that shown in Figure A-30.

A language processor outputs these record types as type 2. When TKB processes them, it changes the type to type 3. It also fills in or modifies some fields. In the descriptions of the following items, fields that are filled in by TKB are marked with an asterisk (*). They should be left as 0 in language processor output.

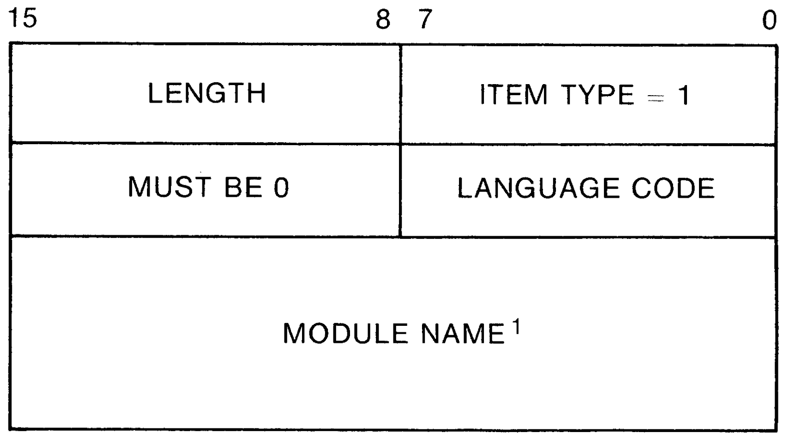
A.5.3.1 Module Name Item (1)

A module name item should be the first ISD entry of each object module. A debugger can assume that all following ISD information up to the next module name item relates to this module.

The language code is included so that a debugger for a specific language can determine whether to ignore a module if it is written for another language. The language code has the same range of values as that of a language-dependent ISD record (128 to 255) and has the same meaning.

The format of the module name item is shown in Figure A-35.

Figure A–35: Format of a Module Name Item (1)



[1]. A counted ASCII string of the required length. A counted ASCII string is a byte string in which the first byte indicates the number of bytes to follow.

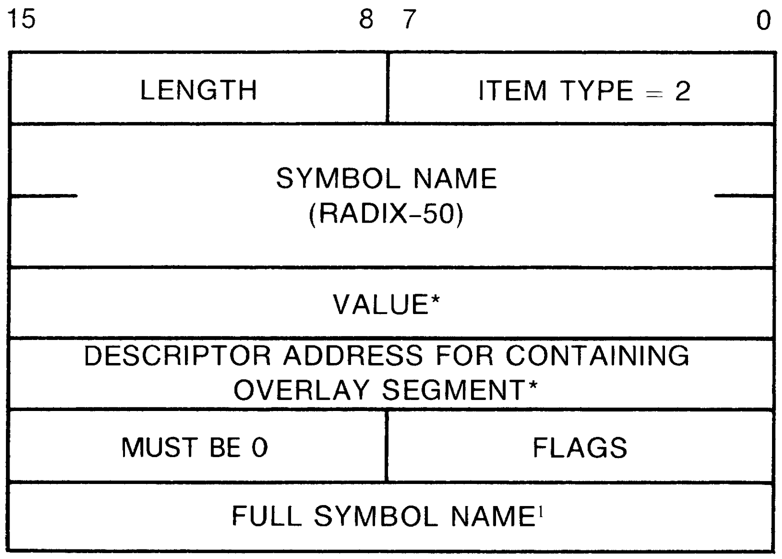
ZK-1063-82

A.5.3.2 Global Symbol Item (2)

One global symbol item must appear for each global symbol definition that the language processor wants the debugger to understand. It need not, however, include definitions generated for the language processor run-time system.

The format of the global symbol item is shown in Figure A-36.

Figure A-36: Format of a Global Symbol Item (2)



[1]. A counted ASCII string of the required length. A counted ASCII string is a byte string in which the first byte indicates the number of bytes to follow.

ZK-1053-82

A.5.3.3 Program Section Item (3)

A concatenated program section has two base addresses: one for the whole program section, and another for the part of it that belongs to this module. It is the base address for the part that belongs to this module that may be used by a debugger to convert local symbol values to absolute addresses.

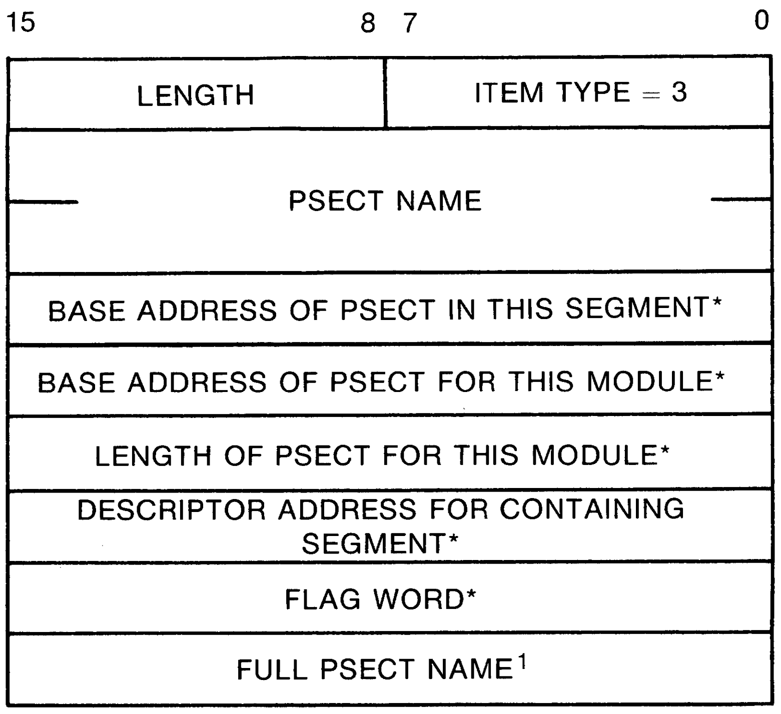
The segment descriptor address is necessary because program sections may move to segments other than the one in which it was placed. This address is relevant to languages that provide semi-automatic overlay generation, such as COBOL-11. This word may be 0 if the program section has not moved to another segment.

The flag word is a copy of the flag word built by TKB. It allows for the identification of virtual program sections.

The full program section name may be needed for some languages.

The format of a program section item is shown in Figure A-37.

Figure A-37: Format of a Program Section Item (3)


15
8 7
0
LENGTH
ITEM TYPE = 3
PSECT NAME
BASE ADDRESS OF PSECT IN THIS SEGMENT*
BASE ADDRESS OF PSECT FOR THIS MODULE*
LENGTH OF PSECT FOR THIS MODULE*
DESCRIPTOR ADDRESS FOR CONTAINING SEGMENT*
FLAG WORD*
FULL PSECT NAME1

1. A counted ASCII string of the required length. A counted ASCII string is a byte string in which the first byte indicates the number of bytes to follow.

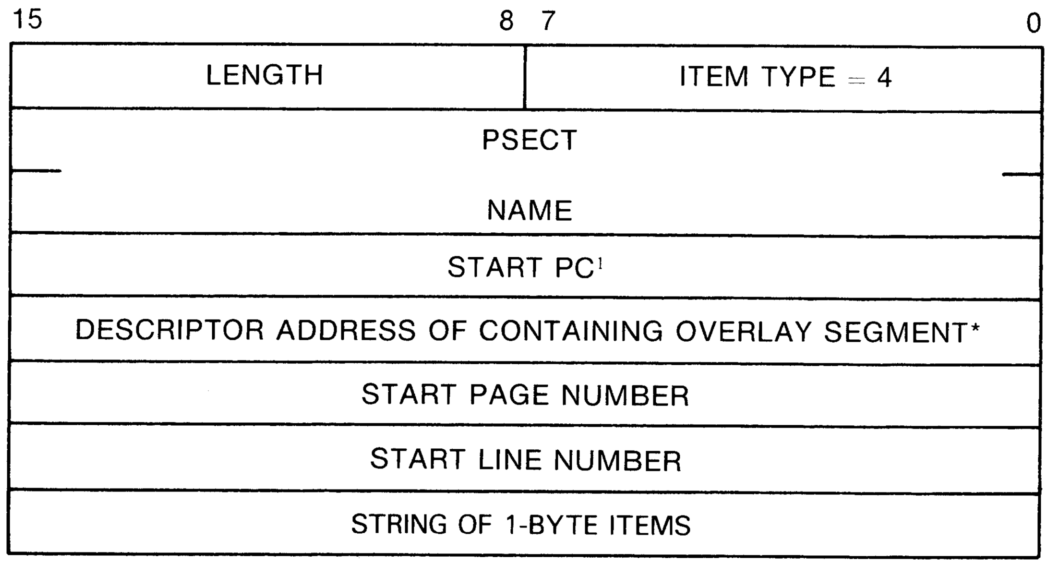
ZK-1054-82

A.5.3.4 Line-Number or Program Counter (PC) Correlation Item (4)

This item provides the information needed to translate a source line number into a task image address, or a task image address into a source line number.

The format of a line-number or PC correlation item is shown in Figure A-38.

Figure A-38: Format of a Line-Number or PC Correlation Item (4)



1. Offset into PSECT in type 2 records; absolute address in type 3 records.

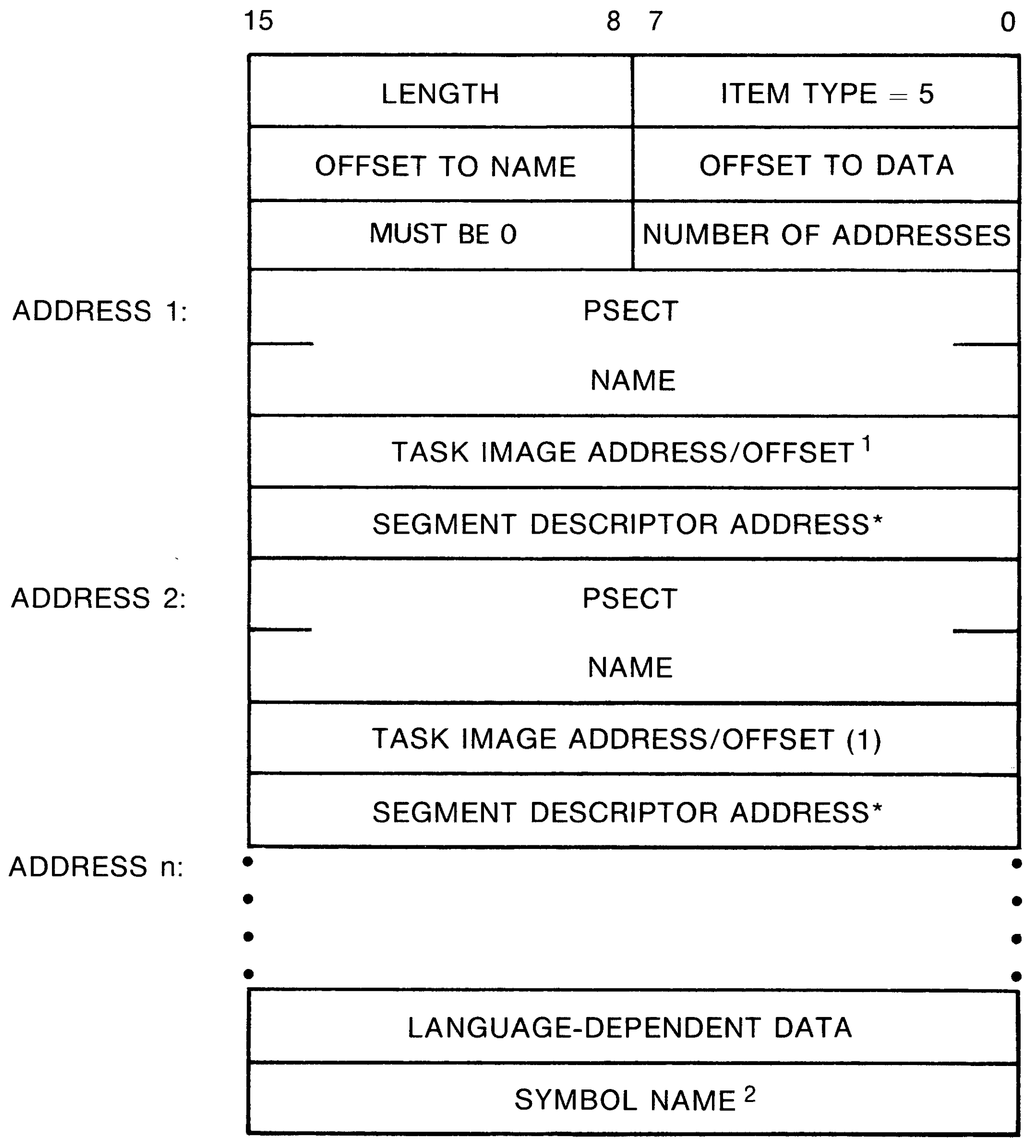
ZK-1055-82

A.5.3.5 Internal Symbol Name Item (5)

The internal symbol name item allows for the fact that a name may have more than one associated address. For example, a COBOL variable may have three associated addresses: the address of the area that contains the data, the address of a CIS descriptor, and the address of a picture string.

The internal symbol name item is shown in Figure A-39.

Figure A-39: Format of an Internal Symbol Name Item (5)



1. Modified by TKB.

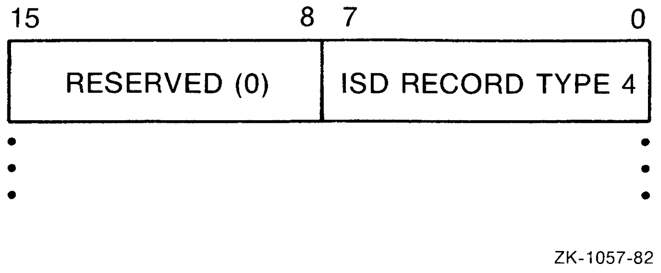
2. A counted ASCII string of the required length. A counted ASCII string is a byte string in which the first byte indicates the number of bytes to follow.

ZK-1056-82

A.5.4 Literal Records (Type 4)

Literal records may take any form, but the 2-byte header shown in Figure A-40 must be present.

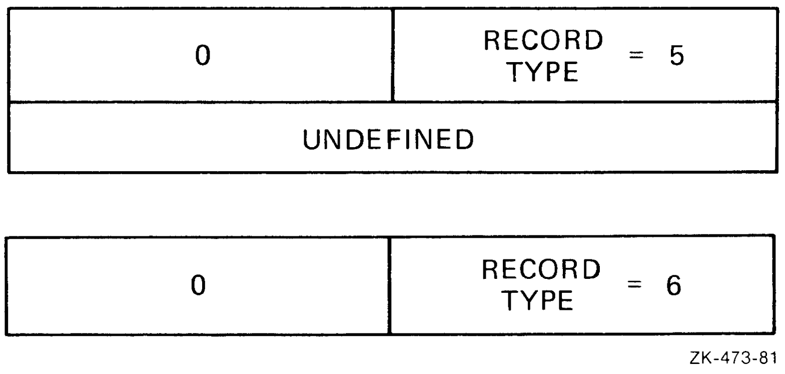
Figure A–40: Format of a Literal Record Type



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-41, this record is one word in length.

Figure A–41: End-of-Module Record Format



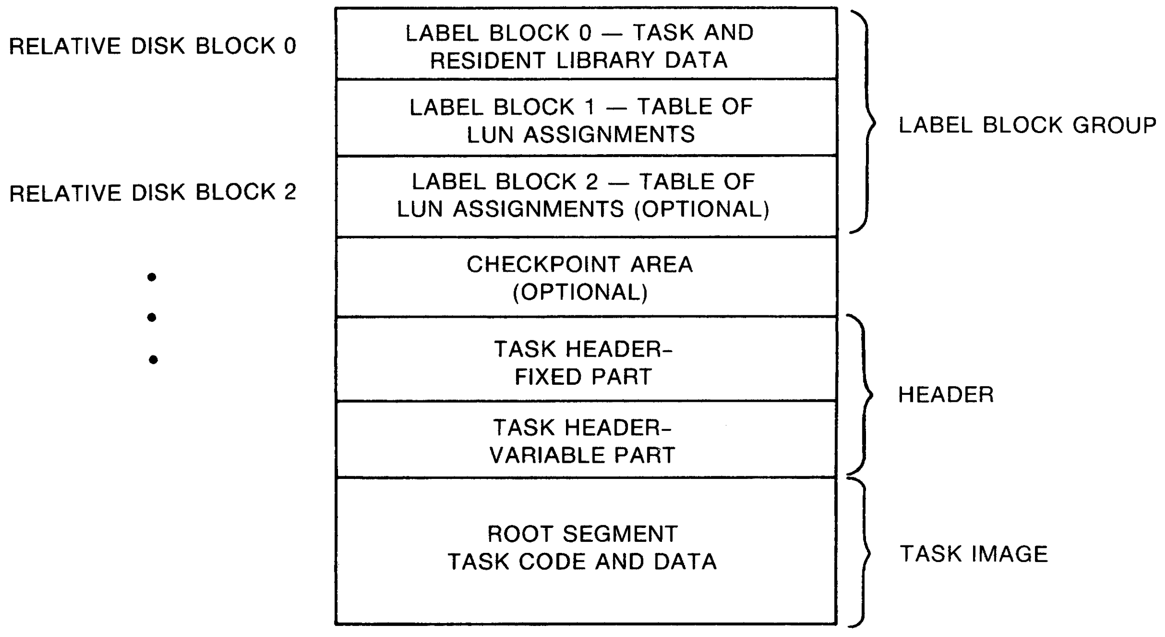
Appendix B

Detailed Task Image File Structure

Figures B-1 through B-4 illustrate how the Task Builder (TKB) records a task image on disk. As noted in the following sections, parts of the task disk image shown in these figures 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:

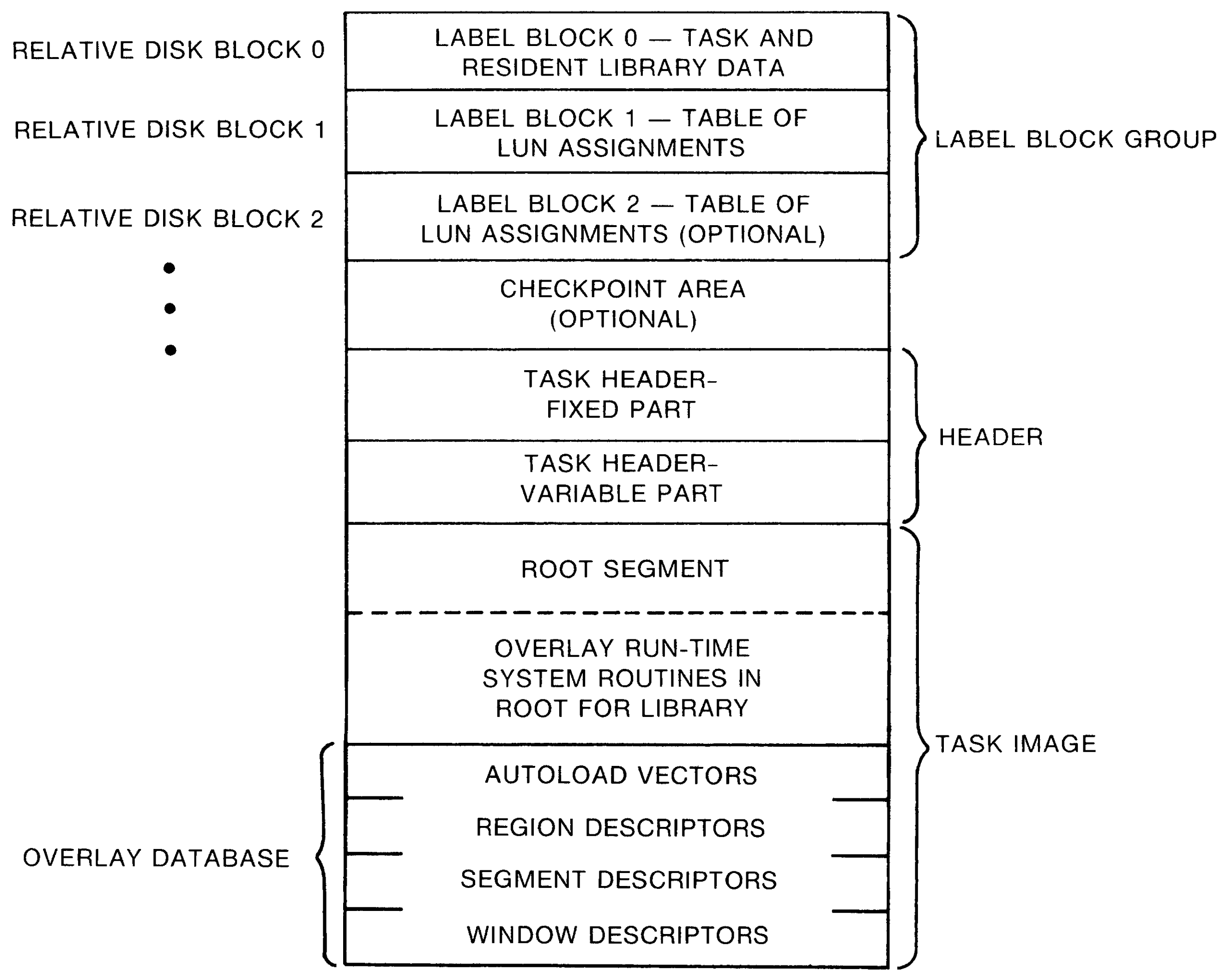
Figure B–1: Image on Disk of Nonoverlaid Conventional Task



RELATIVE DISK BLOCK 0
RELATIVE DISK BLOCK 2
LABEL BLOCK 0 — TASK AND RESIDENT LIBRARY DATA
LABEL BLOCK 1 — TABLE OF LUN ASSIGNMENTS
LABEL BLOCK 2 — TABLE OF LUN ASSIGNMENTS (OPTIONAL)
CHECKPOINT AREA (OPTIONAL)
TASK HEADER- FIXED PART
TASK HEADER- VARIABLE PART
ROOT SEGMENT TASK CODE AND DATA
LABEL BLOCK GROUP
HEADER
TASK IMAGE

ZK-1064-82

Figure B–2: Image on Disk of Conventional Nonoverlaid Task Linked to Overlaid Library


RELATIVE DISK BLOCK 0
LABEL BLOCK 0 — TASK AND RESIDENT LIBRARY DATA
RELATIVE DISK BLOCK 1
LABEL BLOCK 1 — TABLE OF LUN ASSIGNMENTS
RELATIVE DISK BLOCK 2
LABEL BLOCK 2 — TABLE OF LUN ASSIGNMENTS (OPTIONAL)
LABEL BLOCK GROUP
CHECKPOINT AREA (OPTIONAL)
TASK HEADER- FIXED PART
TASK HEADER- VARIABLE PART
HEADER
ROOT SEGMENT
OVERLAY RUN-TIME SYSTEM ROUTINES IN ROOT FOR LIBRARY
TASK IMAGE
AUTOLOAD VECTORS
REGION DESCRIPTORS
SEGMENT DESCRIPTORS
WINDOW DESCRIPTORS
OVERLAY DATABASE

ZK-1065-82

Figure B-3: Image on Disk of Conventional Overlaid Task

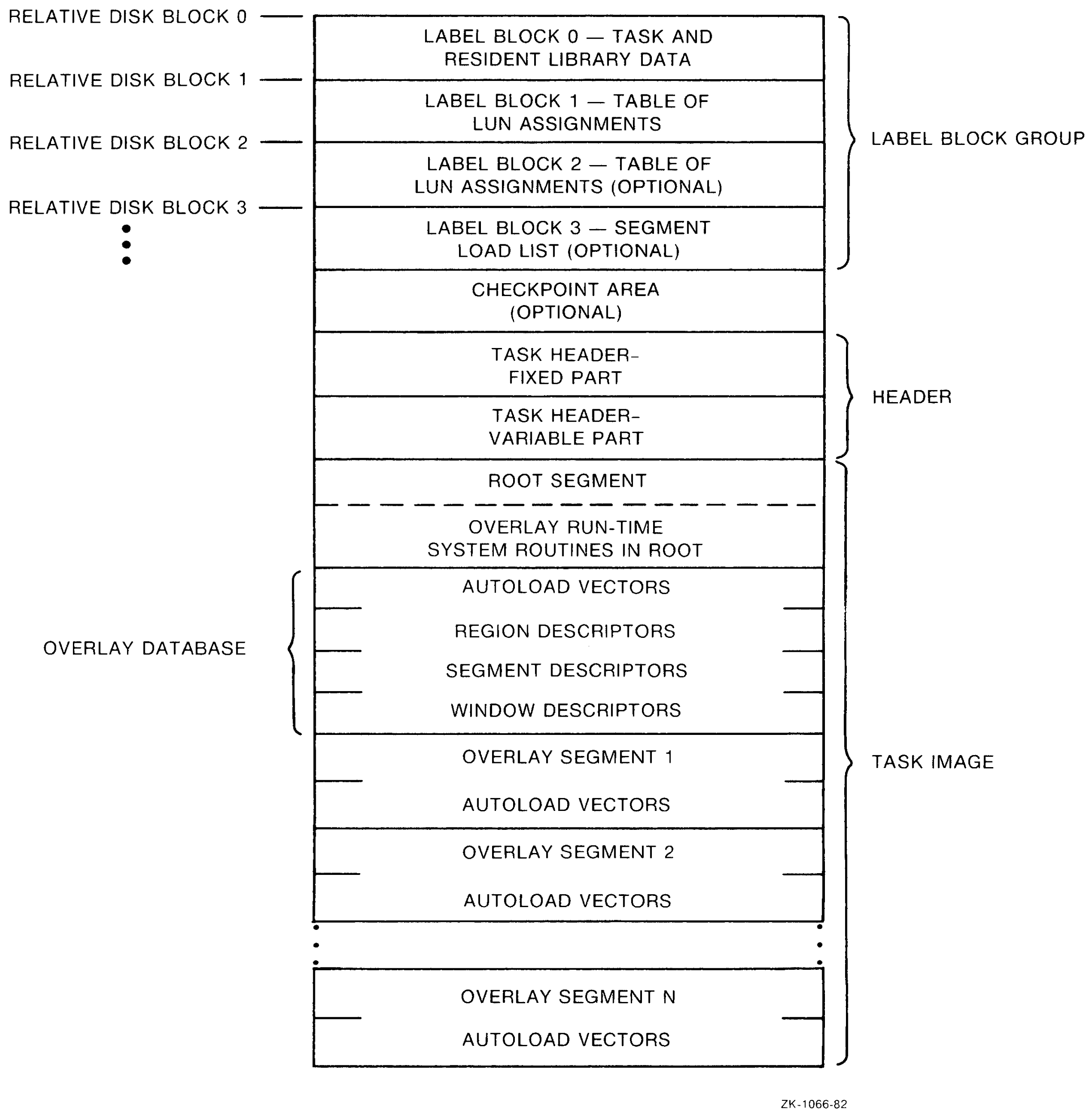
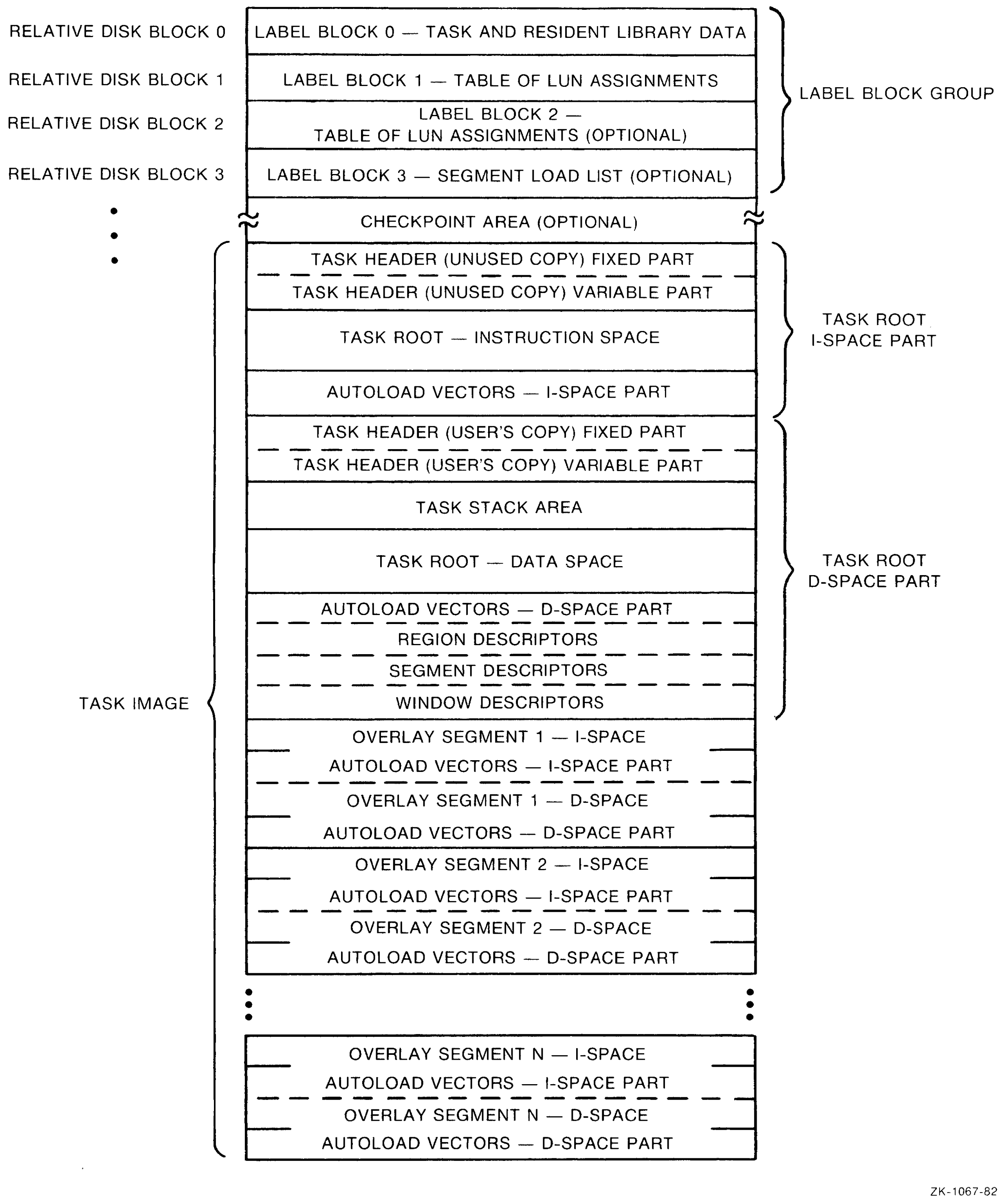


Figure B–4: Image on Disk of Overlaid I- and D-Space Task


RELATIVE DISK BLOCK 0
LABEL BLOCK 0 — TASK AND RESIDENT LIBRARY DATA
RELATIVE DISK BLOCK 1
LABEL BLOCK 1 — TABLE OF LUN ASSIGNMENTS
RELATIVE DISK BLOCK 2
LABEL BLOCK 2 — TABLE OF LUN ASSIGNMENTS (OPTIONAL)
RELATIVE DISK BLOCK 3
LABEL BLOCK 3 — SEGMENT LOAD LIST (OPTIONAL)
LABEL BLOCK GROUP
CHECKPOINT AREA (OPTIONAL)
TASK HEADER (UNUSED COPY) FIXED PART
TASK HEADER (UNUSED COPY) VARIABLE PART
TASK ROOT — INSTRUCTION SPACE
AUTOLOAD VECTORS — I-SPACE PART
TASK ROOT I-SPACE PART
TASK HEADER (USER'S COPY) FIXED PART
TASK HEADER (USER'S COPY) VARIABLE PART
TASK STACK AREA
TASK ROOT — DATA SPACE
AUTOLOAD VECTORS — D-SPACE PART
REGION DESCRIPTORS
SEGMENT DESCRIPTORS
WINDOW DESCRIPTORS
TASK ROOT D-SPACE PART
TASK IMAGE
OVERLAY SEGMENT 1 — I-SPACE
AUTOLOAD VECTORS — I-SPACE PART
OVERLAY SEGMENT 1 — D-SPACE
AUTOLOAD VECTORS — D-SPACE PART
OVERLAY SEGMENT 2 — I-SPACE
AUTOLOAD VECTORS — I-SPACE PART
OVERLAY SEGMENT 2 — D-SPACE
AUTOLOAD VECTORS — D-SPACE PART
OVERLAY SEGMENT N — I-SPACE
AUTOLOAD VECTORS — I-SPACE PART
OVERLAY SEGMENT N — D-SPACE
AUTOLOAD VECTORS — D-SPACE PART

ZK-1067-82

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, as follows:

- 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-5 illustrates how TKB organizes this data in label block 0. The INSTALL task 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-5 by calling the LBLDF$ macro that resides in macro library LB:[1,1] EXEMC.MLB. The LBLDF$ macro on your system will have the correct offsets.

One of the symbols that LBLDF$ defines is $LBXL (label block-extra length). $LBXL is defined as 8 times R$LSIZ, or 340_8 bytes. To get to L$BPRI on your system, you must add 340_8 to the offset L$BPRI. For a task built with options, there are fifteen 14-word entries in L$BLIB, and L$BPRI is found at 706.

Table B-1: Task and Resident Library Data

Parameter	Definition
L$BTSK	Task name consisting of two words in Radix-50 format. This parameter is set by the TASK option.
L$BPAR	Partition name consisting of two words in Radix-50 format. This parameter is set by the PAR option.
L$BSA	Starting address of task. Marks the lowest task virtual address. This parameter is set by the PAR option.
L$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$BSYS	System ID—High byte (4=RSX-11M-PLUS).
L$BWND	Number of task windows (excluding resident libraries).

Table B–1 (Cont.): Task and Resident Library Data

Parameter	Definition
L$BSEG	Size of overlay segment descriptors (in bytes).
L$BFLG	Task flag word. The following flags are defined:

Mask	Bit	Flag	Meaning When Bit = 1
100000	15	TS$PIC	Task contains position-independent code (PIC).
40000	14	TS$NHD	Task has no header.
20000	13	TS$ACP	Task is Ancillary Control Processor.
10000	12	TS$PMD	Task generates Postmortem Dump.
4000	11	TS$SLV	Task can be slaved.
2000	10	TS$NSD	No Send can be directed to task.
1000	9		(Not used.)
400	8	TS$PRV	Task is privileged.
200	7	TS$CMP	Task is built in compatibility mode.
100	6	TS$CHK	Task is not checkpointable.
40	5	TS$RES	Task has memory-resident overlays.
20	4	TS$IOP	Privileged task does not map I/O page.
10	3	TS$SUP	Image linked as supervisor-mode library.
4	2	TS$XHR	Task was built with external header.
2	1	TS$NXH	Task was built with pool-resident header (nonexternal).
1	0	TS$NEW	Task label block uses new format (means L$BLRL describes revision level).

Parameter	Definition
L$BDAT	Three words containing the task creation date as 2-digit integer values as follows: • Year since 1900 • Month of year • Day of month
L$BLIB	Resident library entries.
L$BPRI	Task priority set by the PRI option.
L$BXFR	Task transfer address. Used to initiate a bootable core image; for example, the resident Executive.

Table B-1 (Cont.): Task and Resident Library Data

Parameter	Definition
L$BEXT	Task extension size in units of 32-word blocks. This parameter is set by the EXTTSK option.
L$BSGL	Relative block number of segment load list. Set to 0 if no list is allocated.
L$BHRB	Relative block number of header.
L$BBLK	Number of blocks in label block group.
L$BLUN	Number of logical units.
L$BROB	Relative block number of RO image.
L$BROL	RO load size in 32-word blocks.
L$BRDL	Size of RO data in 32-word blocks.
L$BHDB	Relative block number of data header.
L$BDHV	High virtual address of data window 1.
L$BDMV	High virtual address of data.
L$BDLZ	Load size of data.
L$BDMZ	Maximum size of data.
L$BFL2	Second task flag word. The following flags are defined:

Mask	Bit	Flag	Meaning When Bit = 1
2	1	T2$FMP	Task uses fast-mapping facility.
1	0	T2$CLI	Task is a CLI.

Parameter	Definition
L$BLRL	Label block revision level.
L$AME	Always null for AME compatibility. The last word in label block 0.

Figure B–5: Label Block 0—Task and Resident Library Data

Label	Offset		Library entry 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	0
	42	Name	2 R$LNAM
	44	Base address of library	4 R$LSA
	46	Highest address in first library window	6 R$LHGV
	50	Highest address in library	10 R$LMXV
	52	Library load size (64-byte blocks)	12 R$LLDZ
	54	Library maximum size (64-byte blocks)	14 R$LMXZ
	56	Library offset into region	16 R$LOFF
	60	Number of library window blocks	20 R$LWND
	62	Size of library segment descriptors	22 R$LSEG
	64	Library flag word	24 R$LFLG
	66	Library creation date – Year	26 R$LDAT
	70	– Month	30
	72	– Day	32
	⋮	⋮	
	344	0	
L$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	
L$BRDL	370	R/O data size in 32-word blocks	
L$BHDB	372	Relative block number of data header	
L$BDHV	374	High virtual address of data window 1	
L$BDMV	376	High virtual address of data	
L$BDLZ	400	Load size of data	
L$BDMZ	402	Maximum size of data	
	⋮	⋮	
L$BFL2	772	Second task flags word	
L$BLRL	774	Label block revision number	
	776	AME (must be 0)	

Library entry offsets 0–32: Library Request (maximum of 7 14-word entries in RSX-11M systems and maximum of 15 14-word entries in RSX-11M-PLUS systems)

*Less library window blocks.

ZK-475-81

Table B-2 describes the contents of the resident shared region name block. TKB 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

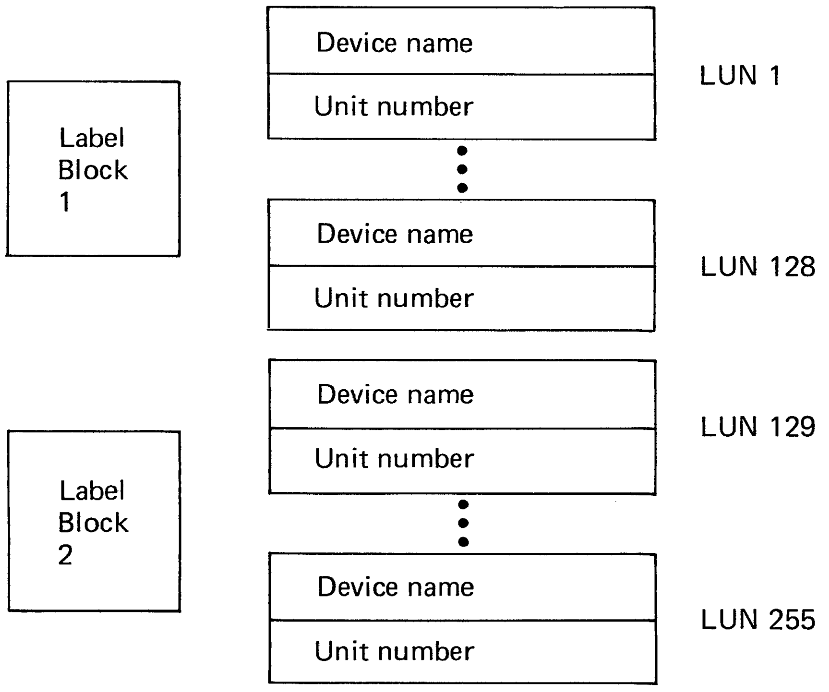
Parameter	Definition
R$LNAM	Shared region name consisting of two words in Radix–50 format.
R$LSA	Base virtual address of library or common.
R$LHGV	Highest address mapped by first library window.
R$LMXV	Highest virtual address in library or common.
R$LLDZ	Shared region load size in 64-byte blocks.
R$LMXZ	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 flag word. The following flags are defined:

Mask	Bit	Flag	Meaning
100000	15	LD$ACC	Access intention (1=read/write, 0=read-only)
40000	14	LD$RSV	APR was reserved
20000	13	LD$CLS	Library is part of a cluster
40	5	LD$RES	Library has memory-resident overlays
10	3	LD$SUP	Supervisor-mode library (1=yes)
4	2	LD$REL	Position-independent code (PIC) flag (1=PIC)
2	1	LD$TYP	Shared region type (1 = common, 0 = library)

Parameter	Definition
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-6, 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.

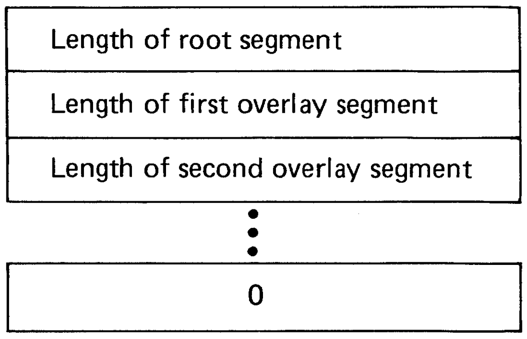
Figure B–6: Label Blocks 1 and 2—Table of LUN Assignments



ZK-476-81

TKB creates the segment load list if the image contains only memory-resident overlays. The segment load list is used only in RSX–11S systems for loading tasks that have resident overlays. Figure B-7 illustrates the segment load list. Each entry in the list gives the length, in bytes, of a memory-resident overlay segment.

Figure B–7: Label Block 3—Segment Load List


Length of root segment
Length of first overlay segment
Length of second overlay segment
0

ZK-477-81

B.2 Checkpoint Area

The checkpoint area is created by the /AL switch (refer to Chapter 10) or the /CHECKPOINT:TASK qualifier (refer to Chapter 11). The checkpoint area is as large as the task image plus any areas created by the EXTTSK, PAR, or VSECT options. The checkpoint area does not include space for the external header if the /XH switch or /EXTERNAL qualifier was specified.

B.3 Header

As shown previously in Figures B-1 through B-4, 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.

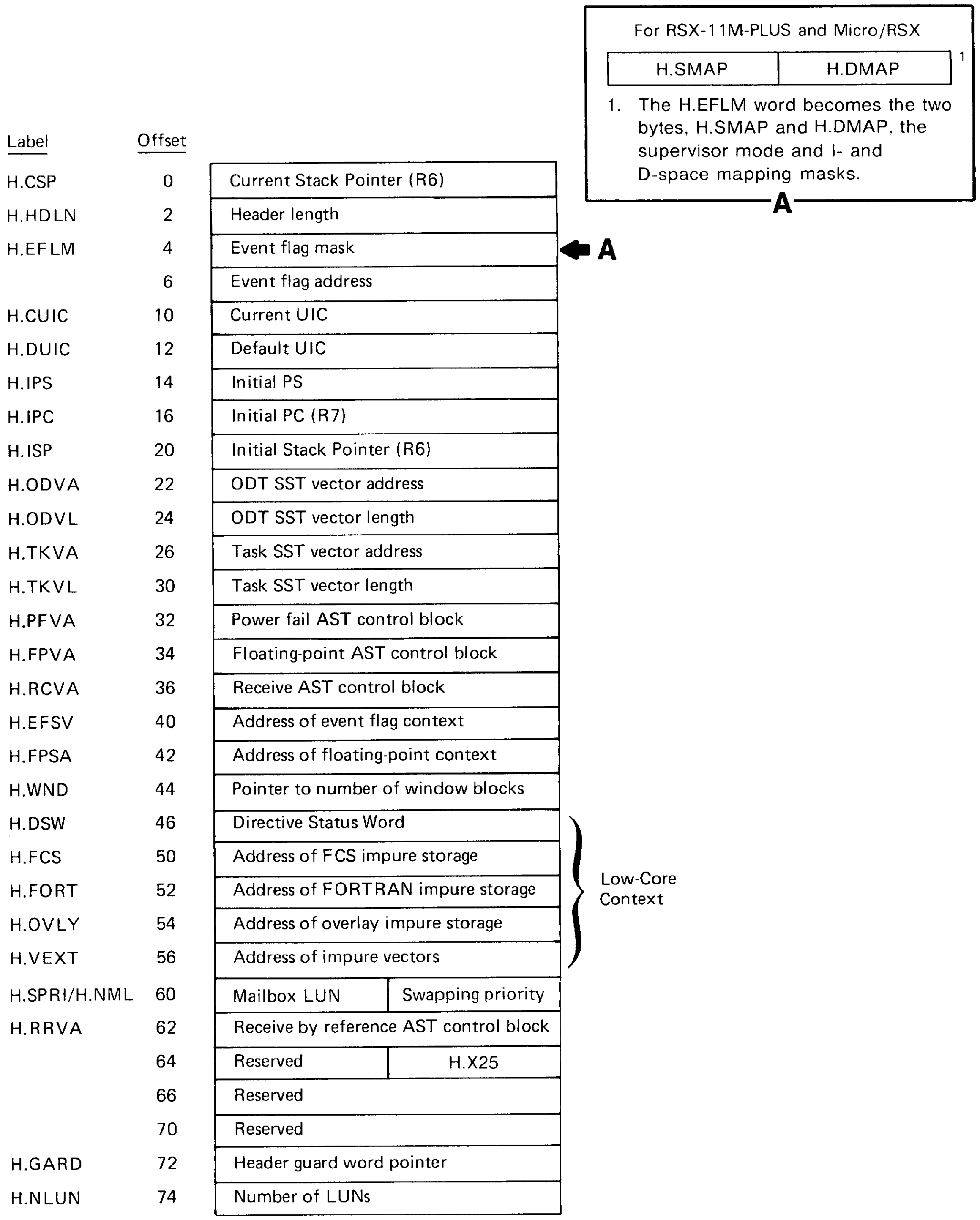
The header is divided into two parts: a fixed part as shown in Figure B-8 and a variable part as shown in Figure B-9. The offsets for the fixed part are defined by macro HDRDF$ residing in LB:[1,1]EXEMC.MLB.

The variable part of the header contains window blocks that describe the following information:

- 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. 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.

Figure B–8: Task Header, Fixed Part



Label	Offset	Contents
H.CSP	0	Current Stack Pointer (R6)
H.HDLN	2	Header length
H.EFLM	4	Event flag mask ◀ A
	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 (Low-Core Context)
H.FORT	52	Address of FORTRAN impure storage (Low-Core Context)
H.OVLY	54	Address of overlay impure storage (Low-Core Context)
H.VEXT	56	Address of impure vectors (Low-Core Context)
H.SPRI/H.NML	60	Mailbox LUN | Swapping priority
H.RRVA	62	Receive by reference AST control block
	64	Reserved | H.X25
	66	Reserved
	70	Reserved
H.GARD	72	Header guard word pointer
H.NLUN	74	Number of LUNs

For RSX-11M-PLUS and Micro/RSX

H.SMAP	H.DMAP

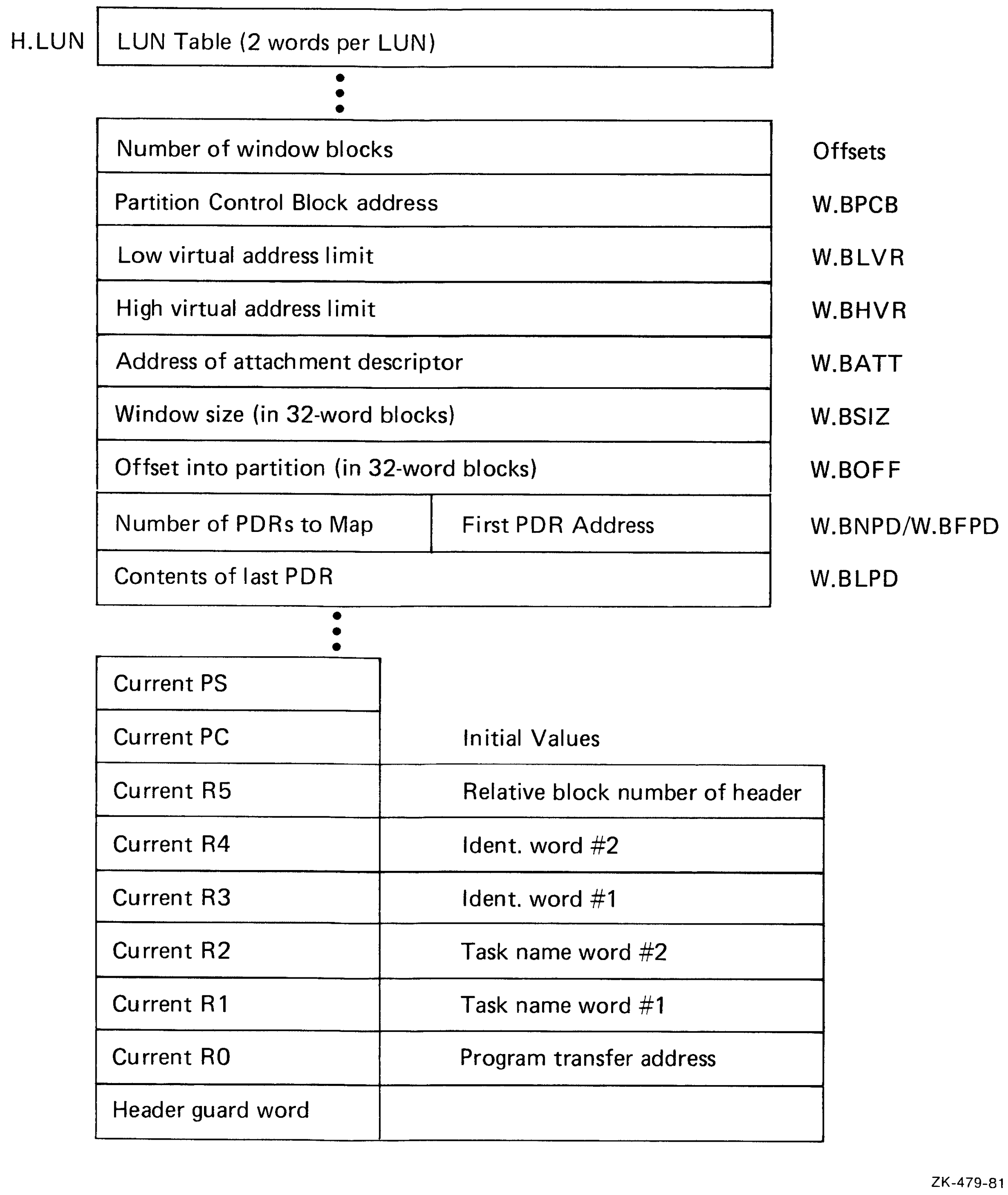
[1]

1. The H.EFLM word becomes the two bytes, H.SMAP and H.DMAP, the supervisor mode and I- and D-space mapping masks.

A

ZK-478-81

Figure B–9: Task Header, Variable Part



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-9).

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 set only when you include a debugging aid in the task image.

B.3.1 Low-Memory Context

The low-memory context for a task consists of the Directive Status Word (DSW) and the impure area vectors. TKB recognizes the following global names:

Name	Meaning
.FSRPT	File Control Services (FCS) 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

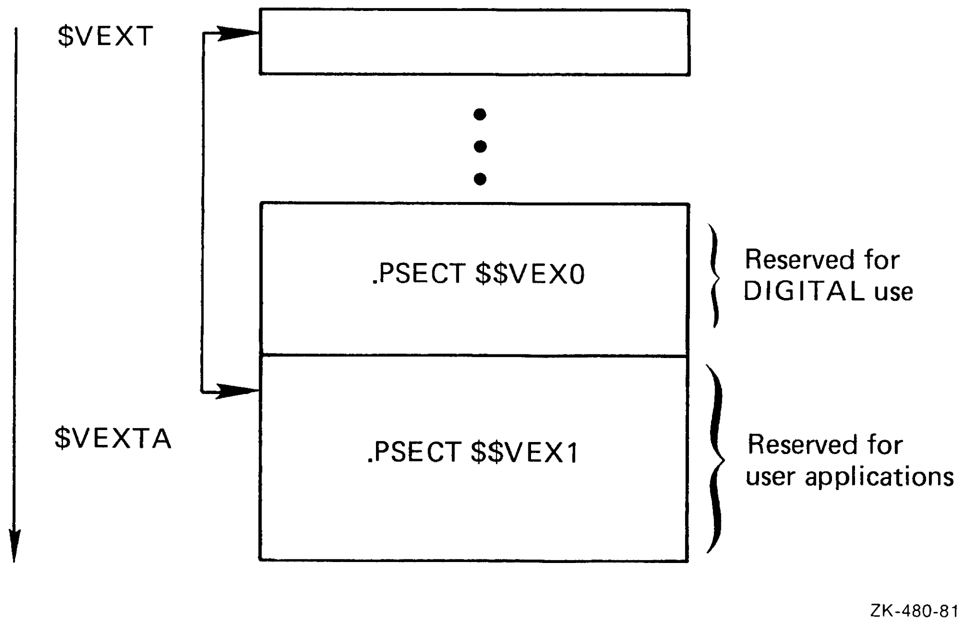
The only proper reference to these pointers is by symbolic name. The pointers are read-only. If you write into them, the result will be lost on the next context switch.

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.

Figure B-10 illustrates the format of the vector extension area. Each location within this area contains the address of an impure storage area that can be referred to by subroutines within a resident library; these subroutines must be reentrant. The address of this area (location $VEXTA) is contained at absolute address $VEXT in the task header. 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–10: Vector Extension Area Format



The program sections $$VEX0 and $$VEX1 have the attributes D, GBL, RW, REL, and OVR.

The program section attribute OVR facilitates defining the offset to the vector and initializing the vector location at link time. For example:

```
                .GLOBL  $VEXTA    ; MAKE SURE VECTOR AREA IS LINKED
                .PSECT  $$VEX1,D,GBL,REL,OVR

$$$=.                             ; POINT TO BASE OF POINTER TABLE
                .BLKW   N         ; OFFSET TO CORRECT LOCATION
                                  ; IN VECTOR AREA

LABEL:          .WORD   IMPURE    ; SET IMPURE AREA ADDRESS
OFFSET==LABEL-$$$                 ; DEFINE OFFSET
                .PSECT

IMPURE:
                  .
                  .
                  .
```

You should centralize all offset definitions within a single module from which the actual vector space allocation is made. Also, you should write the source code with conditional statements to create two object modules: one that reserves the vector storage, and one that defines the global offsets that 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 reports an error if this value differs from the centralized definition.

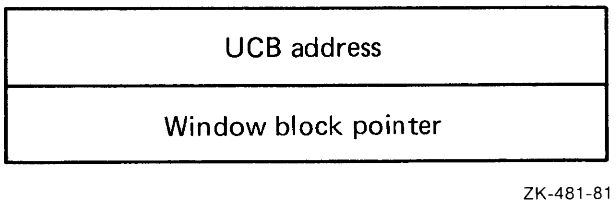
You can locate your vector through a sequence of instructions similar to the following:

```
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-11 illustrates the format of each entry in the Logical Unit Table.

Figure B-11: Logical Unit Table Entry



The first word contains the address of the device Unit Control Block (UCB) 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 (ALUN$) 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 Figures B-1 through B-4). The root segment of a conventional task image is a set of contiguous disk blocks; it is therefore loaded with a single disk access. However, an I- and D-space task root contains two sets of contiguous blocks; therefore, it requires two disk accesses, one for the D-space part and one for the I-space part. The D-space part is loaded first. Additionally, each segment of an overlaid I- and D-space task requires two disk accesses if it contains both I- and D-space.

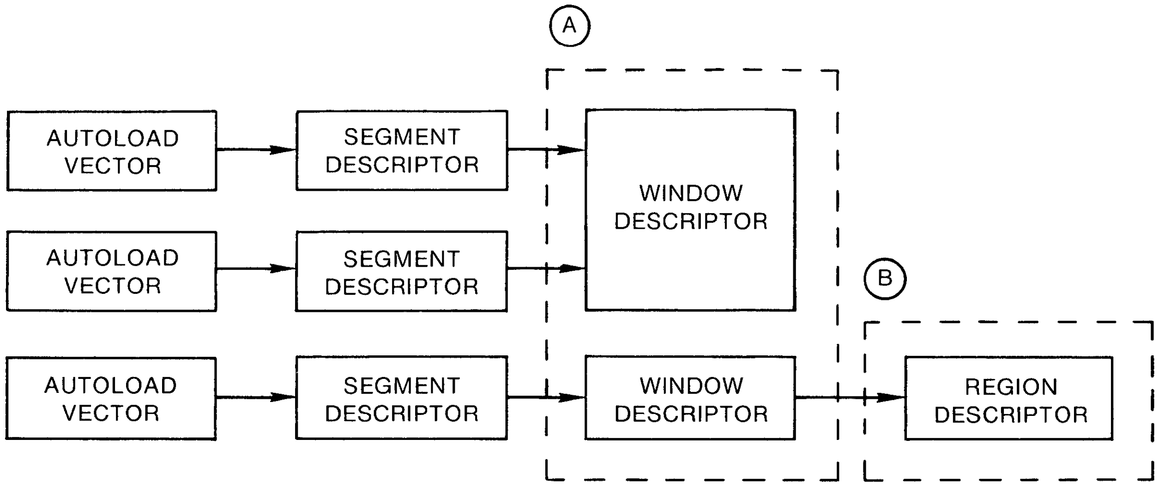
Each overlay segment of the task image begins on a block boundary (see Figure B-3). Note that a given overlay segment occupies as many contiguous disk blocks as it needs to supply its space request. The maximum size for any segment, including the root, is 32K minus 32 words.

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.

Figures B-12 and B-13 illustrate the structure and principal components of the task-resident overlay database.

Figure B–12: Task-Resident Overlay Database for a Conventional Overlaid Task

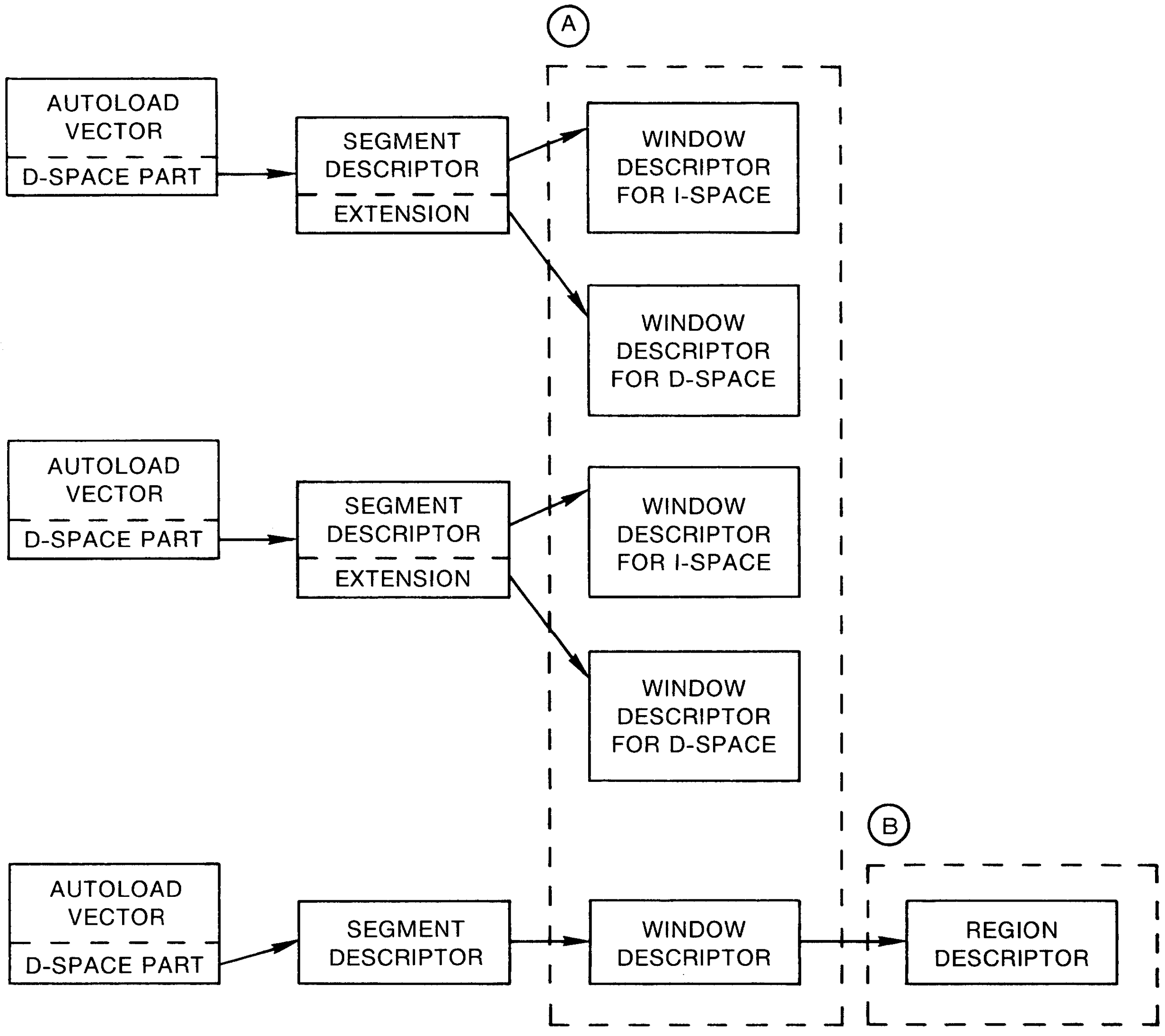


(A) Window descriptors are necessary for the windows that the overlay run-time system uses to map memory resident overlays. The overlay run-time system also needs window descriptors to map disk-resident overlays that are up-tree from memory-resident overlay segments.

(B) The overlay run-time system uses region descriptors to map overlaid libraries.

ZK-1068-82

Figure B–13: Task-Resident Overlay Database for an I- and D-Space Overlaid Task



(A) Window descriptors are necessary for the windows that the overlay run-time system uses to map memory resident overlays. The overlay run-time system also needs window descriptors to map disk-resident overlays that are up-tree from memory-resident overlay segments.

(B) The overlay run-time system uses region descriptors to map overlaid libraries.

ZK-1069-82

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.

Segment descriptors for I- and D-space tasks have an extension for the D-space part that contains the disk block address, virtual load address, segment length in bytes, and window pointer.

The window descriptor contains information required to issue the mapping directives. TKB allocates one window descriptor for each memory-resident overlay in the structure.

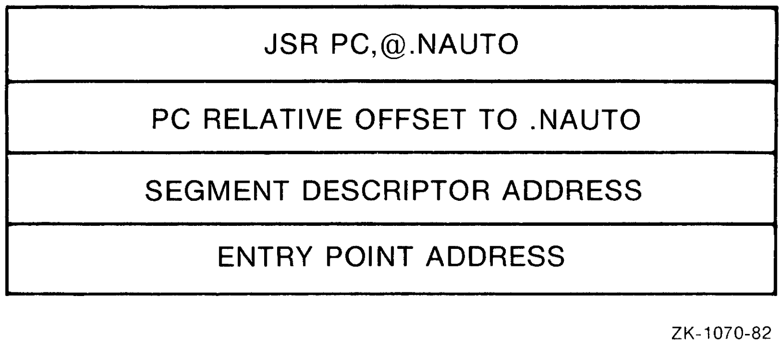
The region descriptor contains information required to attach a resident library or common block. There is one region descriptor for each shared region containing memory-resident overlays.

The following sections describe each database component in greater detail.

B.4.1 Autoload Vectors for Conventional Tasks

The autoload vector table consists of one entry (put into the task image for each autoload entry point) in the form shown in Figure B-14.

Figure B-14: Autoload Vector Entry for Conventional Tasks



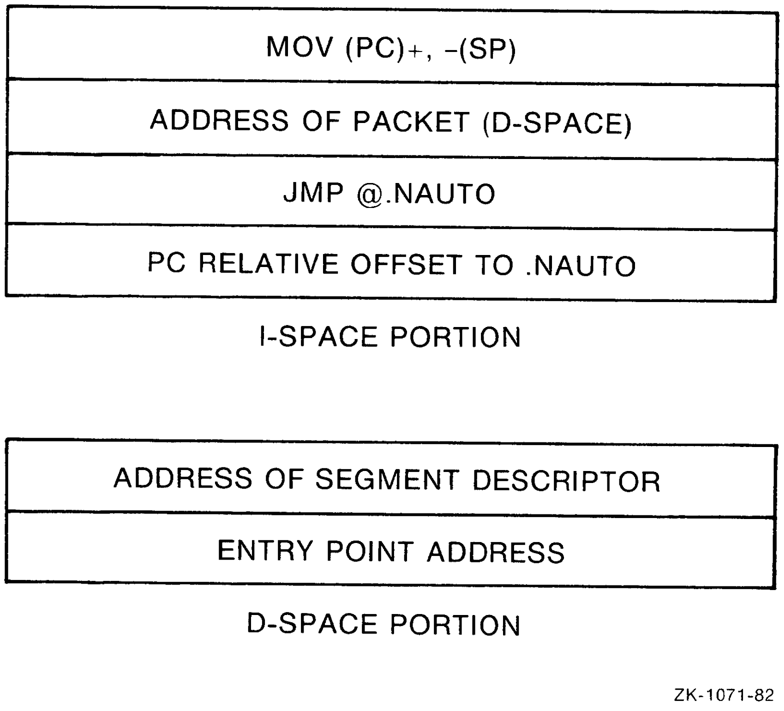
The autoload vector executes an indirect JSR (Jump to Subroutine) instruction to $AUTO through .NAUTO. Following the JSR instruction is a pointer to the descriptor for the segment to be loaded. Following the descriptor is the real address of the required entry point.

B.4.2 Autoload Vectors for I- and D-Space Tasks

The autoload vector table consists of two entries (put into the task image for each autoload entry point) in the form shown in Figure B-15.

The I-space part of the autoload vector contains a MOV (Move) instruction that places the address of the D-space part of the vector on the stack. The vector then executes an indirect JMP (Jump) to $AUTO through .NAUTO. The D-space part of the vector contains the segment descriptor address and the entry point address of the required routine.

Figure B–15: Autoload Vector Entry for I- and D-Space Tasks



B.4.3 Segment Descriptor

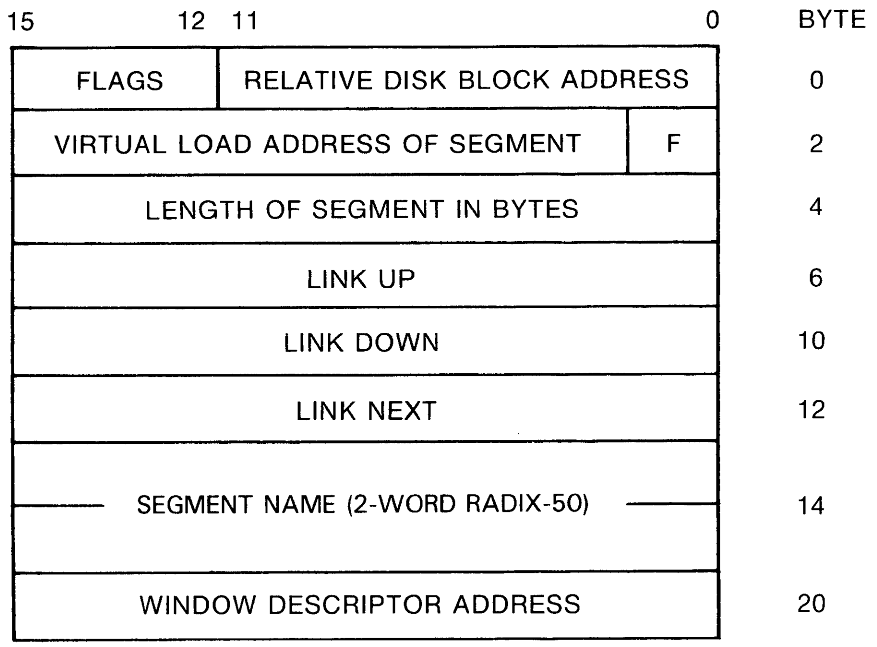
The segment descriptor for a conventional task consists of a fixed part and two optional parts. The fixed part is six words in length. If the manual-load feature is used ($LOAD), two words are added containing the segment name. When a memory-resident overlay structure is included, a ninth word is appended that points to the window descriptor.

The segment descriptor for an I- and D-space task consists of a fixed part that is nine words long and an optional part that is four words long. This optional part is always present for task segments and never present for library segments.

Figure B-16 illustrates the contents of the segment descriptor.

Figure B–16: Segment Descriptor

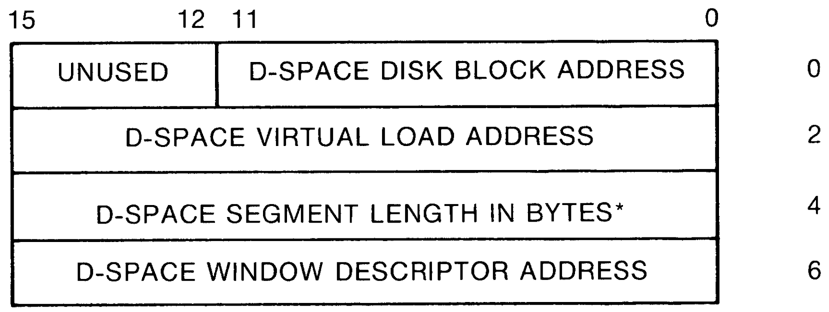
TASK-RESIDENT SEGMENT DESCRIPTOR OFFSETS


15
12
11
0
BYTE
FLAGS
RELATIVE DISK BLOCK ADDRESS
0
VIRTUAL LOAD ADDRESS OF SEGMENT
F
2
LENGTH OF SEGMENT IN BYTES
4
LINK UP
6
LINK DOWN
10
LINK NEXT
12
SEGMENT NAME (2-WORD RADIX-50)
14
WINDOW DESCRIPTOR ADDRESS
20

FLAGS: 15–TASK RESIDENT FLAG (ALWAYS 1)
14–SEGMENT HAS DISK ALLOCATION (1=NO)
13–SEGMENT IS LOADED FROM DISK (1=YES)
12–SEGMENT IS LOADED AND MAPPED (0=YES)

F: 0–SEGMENT FOR I- AND D-SPACE TASK (1=YES)

TASK-RESIDENT SEGMENT DESCRIPTOR EXTENSION
OFFSETS FOR I- AND D-SPACE TASKS ONLY


15
12
11
0
UNUSED
D-SPACE DISK BLOCK ADDRESS
0
D-SPACE VIRTUAL LOAD ADDRESS
2
D-SPACE SEGMENT LENGTH IN BYTES*
4
D-SPACE WINDOW DESCRIPTOR ADDRESS
6

*0 IF ONLY I-SPACE SEGMENT

ZK-1072-82

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 status 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 is loaded from disk.
12	0 = Segment is loaded and mapped. 1 = Segment is either not loaded or not mapped.

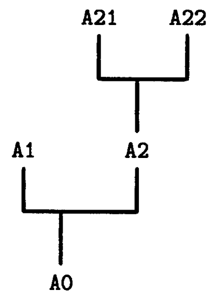
Word 1 contains the load address of the segment. This address is the first virtual address of the area where the segment will be loaded.

Word 2 specifies the length of the segment in bytes.

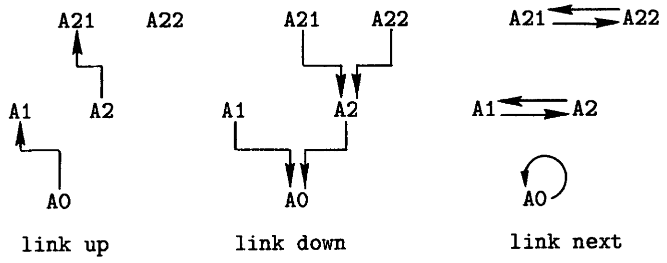
Words 3, 4, and 5 point to the following segment descriptors:

- 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:



If there is a co-tree, the link-next pointer 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).

Notes

There will be some changes in the above information when tasks are autoloaded under the following conditions:

- When you build autoloaded tasks that use memory-resident overlays, word 1 of the segment name field contains a pointer to the root for that co-tree. Word 2 of the segment name field contains information used for the OTS Fast Map routine.
- When you build autoloaded tasks that have only disk-resident overlays, the task-resident flag (bit 15 in word 1) is set to 0 if it is a root segment descriptor, and to 1 if it is an overlay. If the task resident-flag is set to 0, word 5 (link down) will contain a pointer to the current highest-loaded overlay. When the task is run, word 5 will contain a pointer to the current highest-loaded overlay.

B.4.4 Window Descriptor

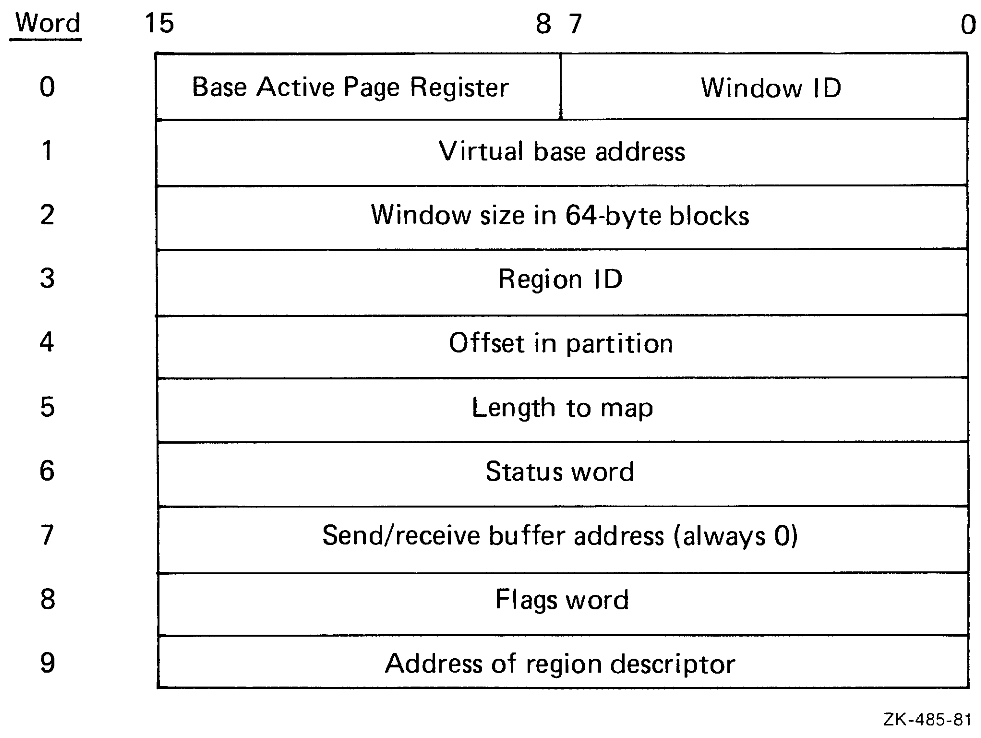
TKB allocates window descriptors only if you define a structure containing memory-resident overlays. Figure B-17 illustrates the format of a window descriptor.

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 0. If the memory-resident overlay is part of a shared region, the overlay loading routine fills in the ID at run time.

Words 8 and 9 contain additional data that is referred to by the overlay routines. Bit 15 of the flag 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 0.

Figure B-17: Window Descriptor

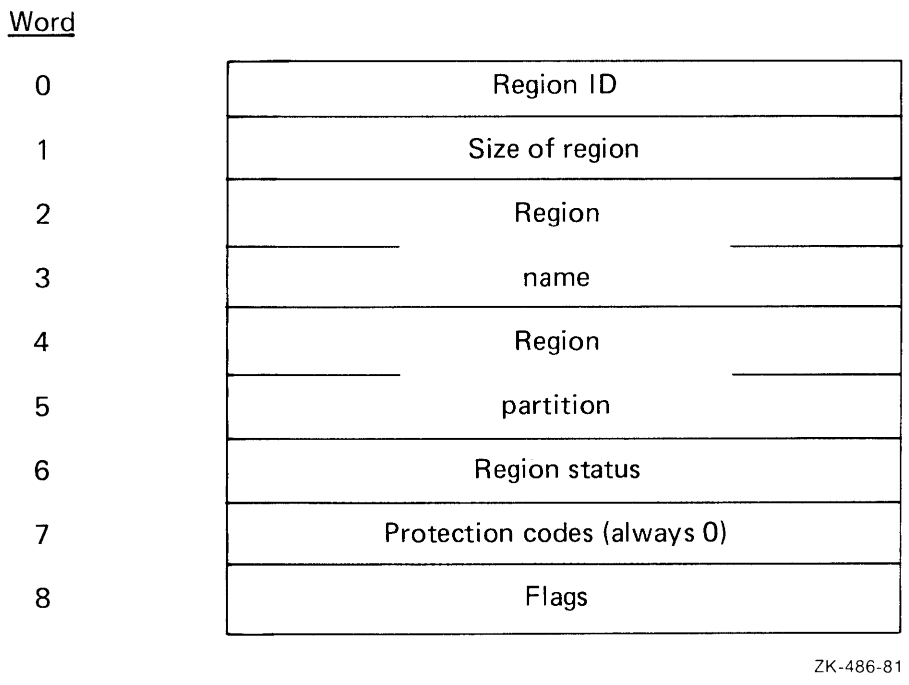

Word
15
8 7
0
0
Base Active Page Register
Window ID
1
Virtual base address
2
Window size in 64-byte blocks
3
Region ID
4
Offset in partition
5
Length to map
6
Status word
7
Send/receive buffer address (always 0)
8
Flags word
9
Address of region descriptor
ZK-485-81

B.4.5 Region Descriptor

The region descriptor is allocated only when the memory-resident overlay structure is part of a shared region. Figure B-18 illustrates the format of a region descriptor.

Words 0 through 7 constitute a region descriptor in the format required by the mapping directives. The flag word is referred to by the overlay load routine. Bit 15 of the flag 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 (ATRG$) directive (with protection code set to 0) to obtain the identification.

Figure B–18: Region Descriptor



Appendix C
Host and Target Systems

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 TKB command could be as follows:

```
>TKB TK3,TK3=SQ1,SQ2 RET
```

Or, the equivalent LINK command could be as follows:

```
$ LINK/TAS:TK3/NOMEN/MAP:TK3/OPT SQ1,SQ2 RET
Option? PAR=PART1:100000:40000 RET
Option? RET
$
```

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. You can do this with a TKB command similar to the following:

```
>TKB RET
TKB>TK3/-MM,TK3=SQ1,SQ2 RET
TKB>/ RET
Enter Option:
TKB>PAR=PART1:100000:40000 RET
TKB>// RET
>
```

Or, with a LINK command similar to the following:

```
$ LINK/TAS:TK3/NOMEN/MAP:TK3/OPT SQ1,SQ2 RET
Option? PAR=PART1:100000:40000 RET
Option? RET
$
```

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 using the following 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, use the Memory Management switch (/MM or /-MM) or the /[NO]MEMORY_MANAGEMENT qualifier to reflect the mapping status of the target system.

The task code must not use any hardware options (FPP, EIS, EAE, and so forth) 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 *VAX FORTRAN Installation Guide/Release Notes* and the *VAX FORTRAN User's Guide* for more information on FORTRAN requirements).

C.1 Example C-1: Transferring a Task from a Host System to a Target System

In this section, the resident library LIB and the task that refers to it, MAIN (from Example 4, Chapter 5), 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, you need to make only the following two changes to the original command sequence that builds the library:

1. You must attach the negated memory management switch (/-MM) to the image file specification.
2. You must specify the partition base and length.

The modified TKB command sequence is as follows:

```
TKB>LIB/-HD/PI/-MM,LIB/-WI,LIB=LIB RET
TKB>/ RET
Enter Options:
TKB>STACK=0 RET
TKB>PAR=LIB:136000:20000 RET
TKB>// RET
>
```

The equivalent LINK command sequence is as follows:

```
$ LINK/TAS/NOHEA/COD:PIC/NOMEM/MAP/NOWID/SYM/OPT LIB RET
Option? STACK=0 RET
Option? PAR=LIB:136000:20000 RET
Option? RET
$
```

If the target system does not contain a partition of the same name as that of 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, you must make the following three changes to the command sequence that builds the task MAIN:

1. You must attach the negated memory management switch (/-MM) to the task image file specification.
2. You must eliminate the APR parameter of the RESLIB option.
3. You must explicitly specify the base address and length of the partition in which the task is to reside.

The modified TKB command sequence is as follows:

```
TKB>MAIN/-MM,MAIN/MA/-WI=MAIN  RET
TKB>/  RET
Enter Options:
TKB>RESLIB=LIB/RO  RET
TKB>PAR=GEN:30100:40000  RET
TKB>//  RET
>
```

The equivalent modified LINK command is as follows:

```
$ LINK/TAS/NOMEM/MAP/NOWID/SYS/OPT MAIN  RET
Option? RESLIB=LIB/RO  RET
Option? PAR=GEN:30100:40000  RET
Option?  RET
$
```

Example C-1, Part 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 option 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 TKB will place the program section names in the library's symbol definition file.

Example C-1: Part 1, Task Builder Map for LIB.TSK

```
LIB.TSK;1    Memory allocation map  TKB M43.00      Page 1
                      10-DEC-87   11:50

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: 000000 000163
R-W disk blk limits: 000002 000002 000001 00001.

*** 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;1
DIVV  :(RO,I,GBL,REL,CON) 000024 000026 00022.

                          000024 000026 00022.  LIB    01     LIB.OBJ;1
MULL  :(RO,I,GBL,REL,CON) 000052 000024 00020.
                          000052 000024 00020.  LIB    01     LIB.OBJ;1
SAVAL :(RO,I,GBL,REL,CON) 000076 000042 00034.
                          000076 000042 00034.  LIB    01     LIB.OBJ;1
SUBB  :(RO,I,GBL,REL,CON) 000140 000024 00020.
                          000140 000024 00020.  LIB    01     LIB.OBJ;1


Global symbols:

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


*** Task builder statistics:

    Total work file references: 368.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 7086. words (27. PAGES)
    Size of word file: 768. words (3. PAGES)

    Elapsed time:00:00:03
```

Example C-1, Part 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.

Example C-1: Part 2, Task Builder Map for MAIN.TSK

```
MAIN.TSK;1     Memory allocation map  TKB M43.00      Page 1
                        11-DEC-87   13:41


Partition name : GEN
Identification : 01
Task  UIC      : [303,3]
Stack    limits: 000274 001273 001000 00512.
PRG xfr address: 001634
Total address windows: 2.

Task  image  size  : 1152. WORDS
Task address limits: 00000 004327
R-W disk blk limits: 000002 000006 000005 00005.

*** Root segment: MAIN


R/W mem  limits: 000000 004327 004330 02264
Disk blk limits: 000002 000006 000005 00005.


Memory allocation synopsis:

Section                                        Title  Ident  File
-------                                        -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON) 001274 002620 01424.
                          001274 000530 00344. MAIN   01     MAIN.OBJ;1
                          .
                          .
                          .


Global symbols:

AADD    160000-R  SAVAL  060000-R  $CBDSG 003110-R  $CBTMG 003132-R
DIVV    160000-R  SUBB   060000-R  $CBOMG 003116-R  $CBVER 003116-R
IO.WVB  011000    $CBDAT 003074-R  $CBOSG 003124-R  $CDDMG 003656-R
MULL    060000-R  $CBDMG 003102-R  $CBTA  003154-R  $CDTB  003312-R
.
.
.


*** Task builder statistics:

    Total work file references: 1889.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 7086. WORDS (27. PAGES)
    Size of work file: 1024. WORDS (4. PAGES)

    Elapsed time:00:00:07
```

Appendix D
Memory Dumps

The RSX-11M-PLUS and Micro/RSX Postmortem Dump task (PMD) generates postmortem memory dumps of tasks that are terminated abnormally. In addition, PMD can produce edited dumps, called snapshot dumps, for tasks that are running. Section D.1 describes Postmortem Dumps in general; Section D.2 discusses the specific case of snapshot dumps. Both types of dumps are useful debugging aids.

D.1 Postmortem Dumps

You can make a task eligible for a Postmortem Dump in any of the following ways:

- By using, at task-build time, the /PM switch for the task file or the /POSTMORTEM output qualifier in the LINK command. Using the /-PM switch or not using the /POSTMORTEM qualifier disables dumps; it is the default condition.
- By installing a task with the /PMD switch in MCR or the /[NO]POSTMORTEM qualifier in DCL to override the task-build /PM switch or /POSTMORTEM qualifier. /PMD=YES and /POSTMORTEM enable dumping; /PMD=NO and /NOPOSTMORTEM disable dumping.
- By using the ABORT command in MCR (described in the *RSX-11M-PLUS MCR Operations Manual*) and including the /PMD switch in the command line to specify a dump; or by using the ABORT command in DCL (described in the *Micro/RSX User's Guide* or the *RSX-11M-PLUS Command Language Manual*) and including the /POSTMORTEM qualifier to specify a dump.

You should install the PMD task in a 4K-word 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. 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 in directory [1,4]; therefore, to avoid errors from PMD, you must create a directory 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 the *RSX-11M-PLUS Utilities Manual*).

Note

> Dump files tend to be somewhat large. The dump of an 8K-word 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.

Example D-1 shows the contents of a Postmortem Dump and snapshot dump. The notes that follow the figure are keyed to the figure and provide a description of the dump's contents. Snapshot dumps are explained more fully in Section D.2.

Example D-1: Sample Postmortem Dump (Truncated)

```
                         POSTMORTEM DUMP ❶

TASK: TT6  ❷                                              TIME: 5-OCT-87 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
                                                                  ⓬
STARTING RELATIVE BLOCK: 000002   BASE: 000000   LENGTH: 001454
STARTING RELATIVE BLOCK: 000004   BASE: 001454   LENGTH: 000264

TASK STACK

   ADDRESS CONTENTS ASCII RAD50  ⓭
    000304  000045    %      7
```

(Continued on next page)

Example D-1 (Cont.): Sample Postmortem Dump (Truncated)

```
                    TASK IMAGE

      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 SIO   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!               14
       000 000  150 170  004 000  246 123                  !  hx     &S!
000100  000000   051646   000000   051646  !    MON     MON!
       000 000  246 123  000 000  246123                   !  &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 03.!
       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/!
       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  ***
```

(Continued on next page)

Example D-1 (Cont.): Sample Postmortem Dump (Truncated)

```
000240  000000   000000   001110   000000  !         NX    !
        000 000  000 000  110 002  000 000                 !     H     !
000250  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    !
000270  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 !   14
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                 !      .  & !
```

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 (PC) 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 the ATL or TAL command in the *RSX-11M-PLUS MCR Operations Manual* for the meaning of the flags. Also, for DCL, see the description of the SHOW TASK/FULL and the SHOW TASK /ACTIVE/FULL commands in the *Micro/RSX User's Guide* or the *RSX-11M-PLUS Command Language Manual.*

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 D.2.2).

❼ The task UIC and the current value of the Directive Status Word.

❽ The task's priority and default priority, the 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 you did not specify the /FP switch or the /CODE:FPP qualifier, or if it uses the EAE unit and you did not specify the /EA switch or the /CODE:EAE qualifier, 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, you can determine the name of the segment in memory. Zero-length segments are usually co-tree roots.

⓭ 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.

⓮ 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.

D.2 Snapshot Dumps

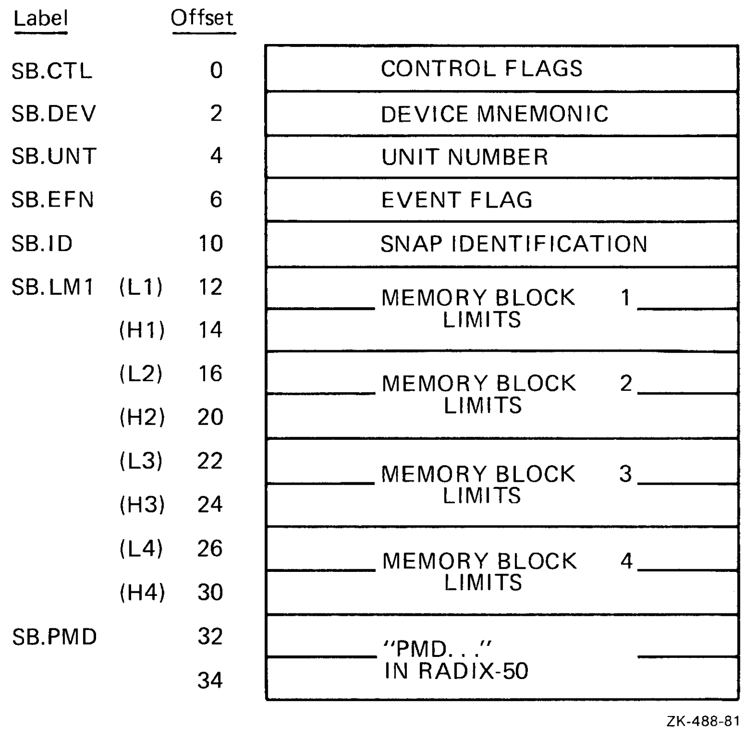
Snapshot dumps are edited dumps produced for running tasks. You can request a snapshot dump any number of times during the execution of a task. The information generated is under the control of the programmer.

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 Figure D-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 D.2.1 and D.2.2 describe the SNPBK$ macro and the SNAP$ macro, respectively.

Figure D-1: Snapshot Dump Control Block Format



D.2.1 Format of the SNPBK$ Macro

The format of the SNPBK$ macro call is as follows:

```
SNPBK$ dev,unit,ctl,efn,id,L1,H1,L2,H2,L3,H3,L4,H4
```

Parameters

dev

The 2-character ASCII name of the device to which the dump is directed. If it is a directory device, the directory [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 as follows:

SC.HDR	Print the dump header (items 3 to 10 in Figure D-1). Items 1 and 2 are always printed.
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.

D.2.2 Format of the SNAP$ Macro

The format of the SNAP$ macro is as follows:

```
SNAP$ ctl,efn,id,L1,H1,L2,H2,L3,H3,L4,H4
```

Parameters

ctl

The set of flags that control the format of the dump and the data to be printed. The flags are as follows:

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 (WTSE$) 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 the memory blocks to be dumped.

H1,H2,H3,H4

The ending addresses of the memory blocks to be dumped.

Notes

1. If no memory blocks are to be dumped, each limit (L1,L2,L3,L4; H1,H2,H3,H4) should be 0.
2. You can set the control flags 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.

D.2.3 Example of a Snapshot Dump

The sample program shown in Example D-2 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_8). Examples D-3 and D-4 show the dumps generated by the sample program.

Example D–2: Sample Program That Calls for Snapshot Dumps

```
SNPTST - TEST SNAP DUMP AND PMD MACRO M1010  03-SEP-87 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-87 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
```

Example D-3: Sample Snapshot Dump (in Word Octal and Radix-50)

```
                         SNAPSHOT DUMP  ID: 1

TASK: TT6                                         TIME: 5-OCT-87 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!
```

(Continued on next page)

Example D-3 (Cont.): Sample Snapshot Dump (in Word Octal and Radix-50)

```
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 FB0  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!
```

Example D-4: Sample Snapshot Dump (in Byte Octal and ASCII)

```
                         SNAPSHOT DUMP  ID: 64

TASK: TT6                                        TIME: 5-OCT-87 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          !
```

(Continued on next page)

Example D-4 (Cont.): Sample Snapshot Dump (in Byte Octal and ASCII)

```
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 !
```

Appendix E
Reserved Symbols

Several global symbols and program section names[1] are reserved for use by the Task Builder.[2] Special handling occurs when TKB encounters the 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 TKB. The relocated value of the symbol is taken as the modification address.

The following global symbols are reserved by TKB:

Global Symbol	Modification Value
$ALERR	OTS address of overlay load error handler.
$AUTO	OTS address of autoload routine.
.FSRPT	Address of file storage region work area (.FSRCB).
$DBTS	Debugger time stamp.
$FSTIN	OTS address of Fast Map overlay routine.
$MARKS	OTS address of MARK segment routine.
.MBLUN	Mailbox logical unit number.
.MOLUN	Error message output device.
.NALER	OTS entry point to overlay load error handler.
.NAUTO	OTS entry point to $AUTO or $LOAD.

1 Absolute sections (ASECTs) and both blank and named control sections (CSECTs) are supplanted by program sections. 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.

Global Symbol	Modification Value
.NDTDS	OTS highest displaced segment.
.NFAST	OTS AST suppression control flags.
.NFMAP	OTS entry point to Fast Map initialization routine.
.NIOST	OTS common I/O status doubleword.
.NLUNS	The number of logical units used by the task, not including the message output and overlay units.
.NMRKS	OTS entry point to MARK segments.
.NOVLY	The overlay logical unit number.
N.OVPT	Address of overlay run-time system work area (.NOVLY).
.NRDSG	OTS entry point to READ segments.
.NSTBL	The address of the segment description tables. This location is modified only when the number of segments is greater than one.
.NSZSG	OTS size of resident segment descriptors.
.ODTL1	A word containing the logical unit number for the ODT terminal device TI.
.ODTL2	A word containing the logical unit number for the ODT line printer device CL.
$OTSV	Address of Object Time System work area ($OTSVA).
.PTLUN	Logical unit number for plotter/graphics software.
$RDSEG	OTS address of READ segment routine.
.SUML1	POS standard utility module LUN.
.TRLUN	The trace subroutine output logical unit number.
.USLU1	Logical unit number for special-purpose user software.
.USLU2	Logical unit number for special-purpose user software.
$VEXT	Address of vector extension area ($VEXTA).

Global Symbol	Modification Value
The following global symbols are reserved by TKB for tasks using disk-resident overlays:	
$MARDS	OTS entry point to I/D MARK segment routine.
$MAFKS	OTS entry point to optimized MARK segment routine.
$MAFDS	OTS entry point to optimized I/D MARK segment routine.
The following global symbols are reserved by TKB for tasks using memory-resident overlays:	
$MARKR	OTS entry point to MARK segment routine.
$MARDR	OTS entry point to I/D MARK segment routine.
$MAFKR	OTS entry point to optimized MARK segment routine.
$MAFDR	OTS entry point to optimized I/D MARK segment routine.
The following global symbols are reserved by TKB for tasks using cluster libraries:	
$MARKC	OTS entry point to MARK segment routine.
$MARDC	OTS entry point to I/D MARK segment routine.
$MAFKC	OTS entry point to optimized MARK segment routine.
$MAFDC	OTS entry point to optimized I/D MARK segment routine.

TKB 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.

Source Location	Section Name	Description
TKB	$$ALER	Contains code to process or trap overlay run-time system segment load errors. Provides named areas in the task for the FORTRAN IV Object Time System and the overlay run-time system.
TKB	$$ALVC	Contains the segment autoload vectors for tasks without I- and D-space.
TKB	$$ALVD	Contains the D-space portions of the segment autoload vectors in an I- and D-space task.
TKB	$$ALVI	Contains the I-space portions of the segment autoload vectors in an I- and D-space task.
TKB	$$AUTO	Contains code to determine if a called subroutine in an overlay segment is already in memory or if that overlay segment should be read into memory before control is passed to the subroutine that is called.

Source Location	Section Name	Description

Input Module	$$DBTS	This symbol should appear in the debugger input module with the symbol $DBTS as follows:
		`        .PSECT  $$DBTS`
		`$DBTS::`
		`        .PSECT`
		The Task Builder extends $$DBTS and fills it with time-stamp information followed by the filename information of the STB file.

SYSLIB	$$DEVT	The extension length (in bytes) is calculated from the following formula: `EXT = <S.FDB+52>*UNITS` The definition of S.FDB is obtained from the root segment symbol table, and UNITS is the number of logical units used by the task, excluding the message output, overlay, and ODT units.
SYSLIB	$$FSR1	The extension of this section is specified by the ACTFIL option.
SYSLIB	$$FSTM	Contains the code to map memory-resident overlays using the Fast Map facility instead of the standard Executive mapping directive CRAW$.
SYSLIB	$$IOB1	The extension of this option is specified by the MAXBUF option.
TKB	$$IOB2	A zero-length .PSECT containing a label, IOBFND, that is stored in the work area offset, W.BEND, representing the upper bound of the I/O buffer, $$IOB1. TKB uses $$IOB2 as a boundary value to determine whether the I/O buffer has overflowed.
TKB	$$LOAD	Overlay manual-load routine.
TKB	$$MRKS	Contains code to properly mark those segments that are not needed any longer or have been overlaid by another segment as being out of memory. This ensures that a fresh copy of the overlay segment will be read in the next time the overlay segment is needed.
SYSLIB	$$OBF1	FORTRAN OTS uses this area to parse array-type format specifications. This section can be extended by the FMTBUF option.
TKB	$$OBF2	A zero-length .PSECT containing a label, OBFH, that is stored in the work area offset, W.OBFH, which represents the upper bound of the run-time format buffer, $$OBF1. TKB uses $$OBF2 to determine whether the run-time format buffer has overflowed.
TKB	$$OVDT	The overlay run-time system impure data area. The symbol .NOVPT in low memory points to this area. This area defines the operational parameters with which the overlay run-time system operates on disk-resident and memory-resident overlay structures.

Source Location	Section Name	Description
TKB	$$OVRS	The .ABS. program section that redefines the overlay run-time system impure data area with different symbols, defined as offsets and relative to 0. These offsets are necessary for proper linkages between the subroutines in the overlay run-time system. This program section is never included in the memory allocation of the task because of its absolute program section attribute.
TKB	$$PDLS	Cluster library service routine.
TKB	$$RDSG	Contains the code that reads into memory the overlay segment selected by the code contained in the programs section $$AUTO.
TKB	$$RGDS	Contains the region descriptors for resident libraries referred to by the task.
TKB	$$RTQ	Defines the program section used for selective enabling of AST recognition in the overlay run-time system. $$RTQ is of zero length if $AUTOT is not included.
TKB	$$RTR	Defines the program section used for selective disabling of AST recognition in the overlay run-time system. $$RTR is of zero length if $AUTOT is not included.
TKB	$$RTS	Contains the return instruction.
TKB	$$SLVC	Supervisor-mode library transfer vectors.
TKB	$$SGDO	Contains the program section adjoining the task-segment descriptors.
TKB	$$SGD1	Contains the task-segment descriptors.
TKB	$$SGD2	Contains a .WORD 0 following the task-segment descriptors.
FORTRAN	$$TSKP	TKB fills in the following words in the program section: • APR bit map in word $APRMP • Total contribution of FORTRAN virtual arrays • Maximum physical read/write I-space memory needed for task in word $MXLGH • Maximum physical read-only I-space memory needed for task in word $MXLGH+2 • Task extension in 32-word blocks in word $LBEXT • Task offset into region in word $LBOFF • Maximum physical read-only D-space memory needed for task in word $MXLGH+4 • Maximum physical read/write D-space memory needed for task in word $MXLGH+6
TKB	$$WNDS	Contains task-window descriptors

Appendix F
Improving Task Builder Performance

This appendix contains procedures to assist you in maximizing Task Builder (TKB) performance. These procedures include doing the following:

- Evaluating and improving TKB throughput
- Modifying command switch defaults to provide a more efficient user interface
- Using the slow mode of the 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 G.

Using the procedures described in this appendix may require relinking TKB. You can do this only in a system that has, as a minimum, a 14K-word 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–11M–PLUS Utilities Manual*).

Modifications to the TKB build file imply one or more of the following files located in directory [1,24]:

TKBBLD.CMD
STKBLD.CMD

These files reside on the disk containing the utility object files.

F.1 Evaluating and Improving Task Builder Throughput

Task Builder throughput is determined by the following factors:

- The amount of disk latency incurred because of overlays
- The amount of memory available for table storage
- The amount of disk latency due to input file processing

The following sections outline methods for improving throughput in the last two cases.

F.1.1 Table Storage

The principal factor governing TKB 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 bytes. 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 TKB 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 0. 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 the 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 TKB in a larger partition or, if TKB is running in a system-controlled partition, by using the INSTALL/INC command in MCR or the INSTALL/EXTENSION command in DCL to allocate more space.

In a system that includes support for the Extend Task (EXTK$) directive, TKB 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 or the DCL command SET SYSTEM/EXTENSION_LIMIT.

Increasing the proportion of resident dynamic memory reduces the amount of I/O necessary for access to TKB 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 work-file 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, for example a fixed-head disk.

Displacement of pages to the work file is done on a least recently used basis. The work file extends automatically as necessary to hold all pages displaced. The parameter W$KEXT is provided in the build command file of TKB and defines the file extension properties. A negative value indicates that the extend is noncontiguous; a positive value indicates that the extend is contiguous. If a contiguous extend fails, a noncontiguous request is attempted; if a noncontiguous extend fails, a fatal work-file I/O error is reported. As long as the work file remains contiguous, a higher access rate can be obtained.

It is not possible to state exactly how many symbols TKB 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 the characteristics of the task being built and may vary from release to release).

Structure Name	**Description**	**Approximate Size (in Words)**
Segment Descriptor	Contains listhead sizes, the pointers defining the overlay tree, and the segment name. Part of this structure becomes the segment descriptor in the resulting task image.	80_{10}
Program Section Descriptor	Contains the name, address size, and attributes of a program section.	10_{10}
Symbol Descriptor	Contains symbol name, value, flags, and pointers to defining segment and program section descriptors.	8_{10} (nonoverlaid task) 15_{10} (overlaid task)
Element Descriptor	Contains module name, ident, file name, count of program section, and some flags.	8_{10}-18_{10}
Control Section Mapping Table	Table of program section size and program section descriptor addresses.	2 words for each program section in each module

The maximum usage of virtual memory occurs during phase three of TKB, 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 file name encountered during the parsing of the tree description. In addition, a program 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 can 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 the following variables:

- The number of directives
- The length of .FCTR lines
- The number of operators (that is, commas, dashes, and parentheses)
- The number of file names encountered

There are a number of ways to reduce the amount of virtual memory required during the build of a specific task. Reducing the data structures in virtual memory also increases the speed of searching the tables and reduces the amount of paging to the work file. The ways to reduce required memory are as follows:

- Extract object modules by name from relocatable object libraries (for example., 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.
- Use concatenated object modules for the same reasons as above.
- 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 program sections to be defined only once, rather than on multiple branches of the tree.
- Place modules that occur on parallel branches of the tree in a common segment (for example, closer to the root) for the same reasons as in the previous method.
- Use the /SS switch for TKB or the /SELECTIVE_SEARCH qualifier for LINK on symbol table files (STB) that describe absolute symbol definitions so that only those symbols referenced are extracted from the module.
- 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 creating one segment in parallel would be longer.

In addition to the above methods, a version of TKB can be built that has less throughput but requires less virtual memory for each element than TKB.

There are four error messages associated with the virtual memory system, as follows:

- No dynamic storage available

  This error occurs when there is insufficient resident storage for creating some data structures. As much as possible of the data already allocated (all unlocked pages) has been paged to the work file, 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. Reducing the ODL and extending the Task Builder memory allocation (see above) are the recommended recovery procedures.

- Unable to open work file

  The probable causes of this error are as follows:

  - The 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 off line, not ready, write-locked, or faulty.
  - There is no such device.

  The MCR command LUN ...TKB may be used to determine which device the Task Builder is attempting to use.

- Work file I/O error

  The probable causes of this error are as follows:

  - Hardware error (for example, parity error on the disk).
  - Device is not ready, or is write-locked.
  - An extend failure has occurred (for example, the disk is full).

- 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 TKB was installed. You can change the device by reassigning logical unit 8 through MCR or by editing the build file and relinking TKB.

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, TKB requests disk space in contiguous increments. The size of each increment is equal to the value of symbol W$KEXT defined in the TKB 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 TKB resorts to noncontiguous allocation whenever contiguous allocation fails.

You can reduce the size of the work file by doing the following:

- Linking your task to a core-resident library containing commonly used routines (for example, the 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.

F.1.2 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, as follows:

- 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.

F.1.3 Summary

In summary, you can use the following procedures to improve TKB throughput:

- Use the MCR command INSTALL/INC or the Executive directive EXTK$ to allocate more table space
- Use the DCL command INSTALL/EXTENSION 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 in 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

F.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 switch setting /-EA, or being forced to use the /CODE:EAE qualifier in LINK, would be unsatisfactory at an installation that made frequent use of the KE11-A Extended Arithmetic Element hardware.

Thus, you are allowed to tailor the switch defaults by altering the contents of the words that contain initial switch states. Modifying TKB in this way is a 3-step process, as follows:

1. Consult Tables F-1 through F-4 to determine the switch word and bit to be altered.
2. Edit the appropriate TKB command file to include the switch word modification through a GBLPAT option referring to the global switch word name.

3. Relink TKB using the modified command file.

However, be aware that if you use the DCL LINK command and you change TKB switch defaults in a TKB build command file, you also alter, in effect, the LINK command defaults. For example, the following LINK command line:

```
$ LINK/TAS:CALC/MAP:CALC/SYM:CALC MOD1,MOD2,MOD3/LB [RET]
```

is translated by DCL into the following TKB command line:

```
CALC,CALC,CALC=MOD1,MOD2,MOD3/LB
```

If you had changed the Task Builder /-EA switch default to /EA for your installation, the same TKB command line as shown above would still be constructed by DCL, but the hidden default (hidden from the point of view of LINK) is now assumed by TKB to be /EA instead of /-EA.

The command files for system tasks, as provided with the released system, require the standard set of TKB defaults. Therefore, you must retain and use an unmodified copy of TKB whenever such tasks are relinked.

You use Tables F-1 through F-4 to alter the defaults as follows:

1. You identify the switch and the file to which it applies.
2. You consult the switch category entry in each table to locate the applicable switch words.
3. You look at the switch settings to find the switch and associated bit.
4. You specify the revised value and switch word as arguments in a GBLPAT option.
5. You relink TKB to produce a version containing the appropriate defaults.

For example, to change the TKB extended arithmetic element default to /EA, perform the steps described below.

By consulting Table F-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 or /CODE:EAE qualifier in bit 13. Setting this bit to 1 changes the initial switch setting to /EA. This new value is combined with the initial contents to yield the revised setting 120002. The required option input is as follows:

```
TKB>GBLPAT=TASKB:$DFTSK:120002 [RET]
```

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 (/ANCILLARY_PROCESSOR and /PRIVILEGED in LINK). In these cases, the value is the number of the initial APR used to map the task. You can alter the default by changing the value of the GBLDEF option for the symbol D$FAPR in the TKB build file. Only the value 4 or 5 can be used.

Table F-1: Task File Switch Defaults

Switch Category: Task file

Switch Word: $DFSWT

Initial Contents: 0

Switch Settings:

			Initial Condition	
Bit	**Initial State**	**TKB Switch**	**LINK Qualifier**	**Meaning**
15	0	/-XT	Not /ERR	Not abort after n diagnostics
11	0	/-SQ	Not /SEQ	Not sequential program section allocation
4	0	/-FU	Not /FUL	Not full overlay tree search
3	0	/RO	/RES	Recognize memory-resident overlay operator
1	0	/-ID	Not /COD:DAT	Not user D-space

Switch Category: Task file

Switch Word: $DFTSK

Initial Contents: 100002

Switch Settings:

			Initial Condition	
Bit	**Initial State**	**TKB Switch**	**LINK Qualifier**	**Meaning**
15	1	/-CP /-AL	/NOCHECK:SYS /NOCHECK:TAS	Not checkpointable[1]
14	0	/-FP	Not /COD:FPP	Not Floating Point Processor
13	0	/-EA	Not /COD:EAE	Not extended arithmetic element
12	0	/HD	/HEA	Header
11	0	/-CM	Not /COM	Not compatibility mode
10	0	/-DA	/NODEB	No debugging aid
9	0	/-PI	Not /COD:PIC	Not position independent
8	0	/-PR	Not /PRI	Not privileged
7	0	/-TR	Not /TRA	No trace

[1]The combination of not checkpointable with checkpoint allocation (100000) is illogical and should not be used.

Table F-1 (Cont.): Task File Switch Defaults

Bit	Initial State	TKB Switch	LINK Qualifier	Meaning
6	0	/-PM	Not /POS	No Postmortem Dump
5	0	/-SL	Not /SLA	Not slave task
4	0	/SE	/REC	Send to task allowed
2	0	/-AC	Not /ANC	Not Ancillary Control Processor
1	1	/-AL	/NOCHECK:TAS	No checkpoint allocation
0	0	/XH	/EXT	External header

Switch Category: Task File

Switch Word: $DFTSO

Initial Contents: 000010

Switch Settings:

Initial Condition

Bit	Initial State	TKB Switch	LINK Qualifier	Meaning
8	0	/-XH	/NOEXT	No external header
3	1	/-SG	/NOSEG	RO and RW program sections

Table F-2: Map File Switch Defaults

Switch Category: Map file

Switch Word: $DFLBS

Initial Contents: 120000

Switch Settings:

Initial Condition

Bit	Initial State	TKB Switch	LINK Qualifier	Meaning
15	1	/-MA	/NOSYS	Do not include system library and STB files in map

Switch Category: Map file

Switch Word: $DFMAP

Initial Contents: 2040

Table F-2 (Cont.): Map File Switch Defaults

Switch Settings:

Initial Condition

Bit	Initial State	TKB Switch	LINK Qualifier	Meaning
10	1	/SH	/MAP and not /LONG	Short map
8	0	/SP	/PRINT	Spool
6	0	/-CR	/NOCRO	No CREF
5	1	/WI	/WID	Wide format

Table F-3: Symbol Table File Switch Defaults

Switch Category: Symbol table file

Switch Word: $DFSTB

Initial Contents: 0

Switch Settings:

Initial Condition

Bit	Initial State	TKB Switch	LINK Qualifier	Meaning
12	0	/HD	/HEA	Build task with header
9	0	/-PI	Not /COD:PIC	Task is not position independent

Table F-4: Input File Switch Defaults

Switch Category: Input file

Switch Word: $DFINP

Initial Contents: 000100

Table F-4 (Cont.): Input File Switch Defaults

Switch Settings:

		Initial Condition		
Bit	Initial State	TKB Switch	LINK Qualifier	Meaning
15	0	/MA	/SYS	Include file contents in map
6	1	/CC	/CON	File contains two or more concatenated object modules

F.3 The Slow Mode of the Task Builder

The default Task Builder and the Fast Task Builder use a symbol table structure that can be searched quickly, but which requires more work-file space than that of 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 F.1. If these procedures do not sufficiently reduce the work-file size, you can select the slow mode of the Task Builder by using the /SB switch. The slow mode is combined in one task image (TKB.TSK) with the default mode of TKB. The slow mode requires less storage, but runs considerably slower than either the Fast Task Builder or the default Task Builder. The build file is TKBBLD.CMD, the same build file as for the default Task Builder. The default name of the slow mode is ...TKB.

You can also create a Task Builder that has /SB as its default. On the RL02 pregenerated distribution kit, there is a command file named LB:[1,2]MAKESTK.CMD. This command procedure creates a version of the slow mode in LB: <LIBUIC> . The command procedure accepts a parameter that determines whether the FCSRES library or the FCSFSL library is used. You can specify RES, FSL, or no parameter.

If you specify RES, MAKESTK.CMD copies the TKBRES.TSK file (the latest version of TKB on your system) into STKRES.TSK. This TKB uses the FCS resident library to resolve the symbols in the FCS routines used by TKB.

If you specify FSL, MAKESTK.CMD copies the TKBFSL.TSK file (the latest version of TKB on your system) into STKFSL.TSK. This TKB uses the supervisor-mode version of the FCS resident library to resolve the symbols in the FCS routines used by TKB. Only the RL02 kit for I- and D-space systems contains FSL.

If you specify no parameter, MAKESTK.CMD copies the TKB.TSK file into STK.TSK. This TKB contains the FCS routines used by TKB.

MAKESTK.CMD places the requested version of TKB in LB: <LIBUIC> . Then, using a ZAP command, MAKESTK.CMD changes the TKB default from /-SB or not enabling /SLOW, to /SB or /SLOW.

You can also invoke a command procedure called STKBLD.BLD during system generation for an RSX-11M-PLUS system. STKBLD.BLD creates the Task Builder CMD and ODL files. The distributed TKB on the system uses these files to build one of the following files with /SB as the default:

```
STK.TSK
STKRES.TSK
STKFSL.TSK
```

Appendix G
The Fast Task Builder

The Fast Task Builder (FTB) allows you to build simple tasks about four times faster than the Task Builder (TKB). However, FTB has limited functionality. It can only link single-segment, nonprivileged tasks, and supports a limited number of switches and options.

If you use DCL, you can invoke the Fast Task Builder by using the following DCL command:

```
$ RUN $FTB [RET]
```

However, if you invoke the Fast Task Builder, you will have to use the TKB switches and options (as described in Chapters 1, 10, and 12) in the standard TKB-format command line, because the LINK command invokes only the standard Task Builder (TKB).

FTB is intended for use as a load-and-go type of linker. It contains very few options and does not support the following:

- New map format
- Overlaid programs
- FORTRAN virtual arrays
- Production of symbol table files
- Creation of resident libraries
- Privileged tasks
- Cluster libraries

The only supported switches are as follows:

- /SP on map file (default = /SP)
- /CP on task file (default = /CP)[1]
- /EA on task file (default = /-EA)
- /MM on task file (default = /MM)
- /FP on task file (default = /FP)

[1] No checkpoint space is allocated in the task image file.

- /DA on input or task image (default = /-DA)
- /LB on an input file in the form

  ```
  >TKB TASK=PROG.OBJ,LIBRARY/LB [RET]
  ```

  but not in the form

  ```
  >TKB TASK=PROG.OBJ,LIBRARY/LB:MODULE [RET]
  ```

· The supported option inputs are as follows:

- ASG (same defaults as TKB)
- STACK (same default as TKB)
- UNITS
- TASK (same default as TKB)
- EXTSCT
- ACTFIL (same default as TKB)
- MAXBUF (same default as TKB)
- LIBR
- COMMON
- RESLIB (same defaults as TKB)
- RESCOM (same defaults as TKB)
- SUPLIB
- RESSUP (same defaults as TKB)

FTB supports shared regions and linking to shared regions, but not building a shared region. FTB cannot link to clustered libraries. Although FTB can link to supervisor-mode libraries, it cannot link to overlaid supervisor-mode libraries.

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 extension (or equivalent) of 8K words 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.

FTB uses asynchronous system traps (ASTs) and therefore requires AST support in the Executive.

Appendix H
Error Messages

The Task Builder (TKB) 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 two statuses: it returns an ERROR status when it encounters a diagnostic error, and a SEVERE ERROR when it encounters a fatal error. (For more information about the Exit with Status directive, see the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual.*)

This appendix lists the error messages produced by TKB. 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

Explanation: TKB could not acquire sufficient disk space to store the task image file, or did not have write access to the directory or volume that was to contain the file.

Blank P-section name is illegal
overlay-description-line

Explanation: The overlay-description-line displayed contains a .PSECT directive that does not have a program section name.

Cluster library element, element-name, is not resident overlaid

Explanation: The listed cluster element has been built without memory-resident overlays. This kind of element cannot be used as a cluster library element. Cluster libraries 2 through 6 must be memory-resident and overlaid.

Cluster library element library-name does not have null root

Explanation: This is a fatal error. All libraries except the first must be memory-resident-overlaid and have a null root. The first library in the group can be nonoverlaid or overlaid with a null or nonnull root.

Command I/O error

Explanation: An I/O error occurs on a command input device. (The device may not be on line, or possible hardware error.)

Command syntax error
command-line

Explanation: The command-line displayed has incorrect syntax.

Complex relocation error - divide by zero: module
module-name

Explanation: A divisor having the value 0 was detected in a complex expression. The result of the divide was set to 0. (Probable cause: division by a global symbol whose value is undefined.)

Conflicting base addresses in cluster library

Explanation: This conflict arises when you specify APRs for both PIC and non-PIC libraries that are included in the cluster. See the APR parameter as described in the CLSTR option. This is a fatal error.

Disk image core allocation too large
invalid-line

Explanation: The minimum disk allocation specified in the invalid line is greater than 128.

File file-name attempted to store data in virtual section

Explanation: The file contains a module that has attempted to initialize a virtual section with data.

File file-name has illegal format

Explanation: The file file-name contains an object module whose format is not valid.

Illegal APR reservation

Explanation: An APR specified in a COMMON, LIBR, RESCOM, or RESLIB option is outside the range 0 to 7.

Illegal cluster configuration

Explanation: If the cluster contains a nonoverlaid library, that library must be the first library in the cluster. Check the configuration of the libraries in the cluster. This is a fatal error.

Illegal default priority specified
option-line

Explanation: The option-line displayed contains a priority greater than 250.

Illegal device/volume
invalid-line

Explanation: The invalid-line displayed contains an illegal device specification.

Illegal directory
invalid-line

Explanation: The invalid-line displayed contains an illegal directory name.

Illegal error-severity code octal-list

Explanation: This is a 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

Explanation: The invalid-line displayed contains a wildcard (*) in a file specification. Using wildcards is prohibited.

Illegal get command line error code

Explanation: This is a system error (no recovery).

Illegal logical unit number
invalid-line

Explanation: The invalid-line displayed contains a device assignment to a unit number larger than the number of logical units specified by the UNITS option, or assumed by default if the UNITS option is not used.

Illegal multiple parameter sets
invalid-line

Explanation: The invalid-line displayed contains multiple sets of parameters for an option that allows only a single parameter set.

Illegal number of logical units
invalid-line

Explanation: The invalid-line displayed contains a logical unit number greater than 250.

Illegal ODT or task vector size

Explanation: The ODT or SST vector size specified is greater than 32 words.

Illegal overlay description operator
invalid-line

Explanation: The invalid-line displayed contains an unrecognizable operator in an overlay description. This error occurs if the first character in a program section or segment name is a period (.).

Illegal overlay directive
invalid-line

Explanation: The invalid-line displayed contains an unrecognizable overlay directive.

Illegal partition/common block specified
invalid-line

Explanation: The user-defined base or length is not on a 32-word boundary.

Illegal P-section/segment attribute
invalid-line

Explanation: The invalid-line displayed contains a program section or segment attribute that is not recognized.

Illegal reference to library P-section p-sect-name

Explanation: A task has attempted to reference a program section name existing in a shared region but has not named the shared region in an option. This error occurs when you explicitly specify an STB file as an input file, but you have not specified in an option the library to which the STB file belongs.

Illegal switch
file-specification

Explanation: The file-specification contains an invalid switch or switch value.

Incompatible OTS module

Explanation: The OTS (overlay run-time system) module requested by TKB has not been found. The OTS modules are part of the system library. This error occurs if you are using an incompatible version of the system library (SYSLIB.OLB).

Incompatible reference to library P-section p-sect-name

Explanation: 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

Explanation: The invalid-line displayed contains a module name with a non-Radix–50 character.

Indirect command syntax error
invalid-line

Explanation: The invalid-line displayed contains a syntactically incorrect indirect command file specification.

Indirect file depth exceeded
invalid-line

Explanation: The invalid-line displayed gives the file reference that exceeded the permissible indirect command file depth (2).

Indirect file open failure
invalid-line

Explanation: The invalid-line displayed contains a reference to a command input file that could not be located.

Insufficient APRs available to map read-only root

Explanation: TKB could not find enough free APRs to map the read-only portion of a multiuser task.

Insufficient parameters
invalid-line

Explanation: The invalid-line displayed contains a keyword with an insufficient number of parameters to complete its meaning.

Invalid APR reservation
invalid-line

Explanation: APR is specified on a keyword for an absolute library.

Invalid keyword identifier
invalid-line

Explanation: The invalid-line displayed contains an unrecognizable keyword.

Invalid partition/common block specified
invalid-line

Explanation: A partition is invalid for one of the following reasons:

- TKB cannot find the partition name in the host system in order to get the base and length.
- The system is mapped, but the base address of the partition is not on a 4K boundary (for a nonrunnable task) or is not 0 (for a runnable task).
- The memory bounds for the partition overlap a shared region.
- The partition name is identical to the name of a previously defined COMMON or LIBR shared region.
- The top address of the partition for a runnable task exceeds 32K minus 32 words for a mapped system.

Invalid reference to mapped array by module module-name

Explanation: The module has attempted to initialize the mapped array with data. An SPR should be submitted if DIGITAL-supplied software caused this problem.

Invalid window block specification
invalid-line

Explanation: The number of extra address windows specified exceeds the number permitted. On RSX-11M-PLUS and Micro/RSX systems, you can specify as many as 15 extra window blocks.

I/O error library image file

Explanation: 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

Explanation: This error occurs when TKB cannot read an input file specification (for example, when the command line is greater than 80 characters). The error may also occur if the variable-length records in an object file are longer than 128_{10} bytes.

I/O error on output file file-name

Explanation: This error occurs when TKB cannot write an ouput file (for example, when the specified device is off line). The error may also occur if the specified directory does not exist or if there is not enough space on the device for TKB to write the output file.

Label or name is multiply defined
invalid-line

Explanation: The invalid-line displayed defines a name that has already appeared as a .FCTR, .NAME, or .PSECT directive.

Library file file-name has incorrect format

Explanation: A module has been requested from a library file that has an empty module name table.

Library not built as a supervisor mode library

Explanation: The library referred to in a RESSUP or SUPLIB option was built without a completion (CMPRT=X) routine and is not a supervisor-mode library.

Library library-name not found in any cluster

Explanation: All task image and symbol table files to be included as cluster elements must reside in LB:[1,1].

Library references overlaid library
invalid-line

Explanation: 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

Explanation: 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 conditions:

- An attempt to initialize a program section contained in a shared region
- An attempt to initialize an absolute location outside the limits of the segment or in the task header
- A patch outside the limits of the segment to which it applies
- An attempt to initialize a segment having the NODSK attribute

Lookup failure on file file-name
invalid-line

Explanation: The invalid-line displayed contains a file name that cannot be located in the directory.

Lookup failure on system library file

Explanation: TKB cannot find the system library (SY0:[1,1]SYSLIB.OLB) file to resolve undefined symbols.

Lookup failure resident library file - filename.typ

Explanation: No symbol table or task image file can be found for the shared region "filename.typ." If the shared region was linked to another shared region, ensure that the task image of both regions and the symbol table files exist on the same device and in the same UIC as the UIC referenced by the option RESLIB, RESCOM, LIBR, COMMON, RESSUP, or SUPLIB.

Module module-name ambiguously defines P-section p-sect-name

Explanation: The program 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

Explanation: 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 contains incompatible autoload vectors

Explanation: You are trying to build an I- and D-space task and link it to an older existing library that contains an old-style vector format. Rebuild the library with the RSX–11M–PLUS Version 2.1 or later version Task Builder to create new-style vectors for the library. Then, rebuild your task.

Module module-name illegally defines xfr address p-sect-name addr

Explanation: This message occurs under any one of the following conditions:

- The start address printed is odd.
- 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.
- The address is in a program section that has not yet been defined. An SPR should be submitted if DIGITAL-supplied software caused this problem.

Module module-name multiply defines P-section p-sect-name

Explanation: This error occurs for the following reasons:

- The program section p-sect-name has been defined more than once in the same segment with different attributes.
- A global program 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

Explanation: 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

Explanation: This error occurs when more than one module making up the root has a start address.

Module module-name not in library

Explanation: TKB could not find the module named on the /LB switch in the library.

No dynamic storage available

Explanation: TKB needs additional symbol table storage and cannot obtain it. If possible, install TKB in a larger partition.

No memory available for library library-name

Explanation: TKB could not find enough free virtual memory to map the specified shared region.

No root segment specified

Explanation: The overlay description did not contain a .ROOT directive.

No virtual memory storage available

Explanation: The maximum permissible size of the work file is exceeded. You should consult Appendix F for suggestions on reducing the size of the work file.

Open failure on file file-name

Explanation: This error occurs when TKB cannot open a specified file (for example, when the specified device is off line). The error may occur if the specified directory does not exist or if the specified device is write-locked.

Option syntax error
invalid-line

Explanation: The invalid-line displayed contains unrecognizable syntax.

Overlay directive has no operands
invalid-line

Explanation: All overlay directives except .END require operands.

Overlay directive syntax error
invalid-line

Explanation: The invalid-line displayed contains a syntax error or references a line that contains an error.

Partition partition-name has illegal memory limits

Explanation: This messages occurs for the following reasons:

- The partition-name defined in the host system has a base address alignment that is not compatible with the target system.
- You attempted to build a privileged task in a partition whose length exceeds the task's available address space (8K or 12K).

Pass control stack overflow at segment segment-name

Explanation: This is a 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

Explanation: You have attempted to build a position-independent shared region that references another shared region.

P-section p-sect-name has overflowed

Explanation: A program section greater than 32K words has been created.

Required input file missing

Explanation: At least one input file is required for a task build.

Required partition not specified

Explanation: The PAR option was not used when running TKB on an RSX–11D host system. The option must contain explicit base address and length specifications.

Resident library has incorrect address alignment
invalid-line

Explanation: The invalid-line displayed specifies a shared region that has one of the following problems:

- The library references another library with invalid address bounds (that is, not on a 4K boundary in a mapped system).
- The library has invalid address bounds.

Resident library mapped array allocation too large
invalid-line

Explanation: The invalid-line displayed 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

Explanation: One of the following problems has occurred:

- More than seven shared regions have been specified.
- A shared region has been specified more than once.
- Non-position-independent shared regions whose memory allocations overlap have been specified.

Root segment is multiply defined
invalid-line

Explanation: The invalid-line displayed contains the second .ROOT directive encountered. Only one .ROOT directive is allowed.

Segment seg-name has addr overflow: allocation deleted

Explanation: Within a segment, the program has attempted to allocate more than 32,767 words. A map file is produced, but no task image file is produced.

Segment seg-name not found for patch

Explanation: The Task Builder could not locate the named segment for a global patch. The option used was GBLPAT=X:Y:0.

Supervisor mode completion routine is undefined

Explanation: TKB could not locate the symbol X, which was specified in the CMPRT=X option.

Symbol sym-name not found for patch

Explanation: TKB could not locate symbol Y for a global patch. The option used was GBLPAT=X:Y:0.

Task has illegal memory limits

Explanation: 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

Explanation: 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 file-name is noncontiguous

Explanation: Insufficient contiguous disk space was available to contain the task image. A noncontiguous file was created. After deleting unnecessary files, the /CO switch in PIP for MCR or the COPY/CONTIGUOUS command in COPY for DCL should be used to create a contiguous copy.

Task requires too many window blocks

Explanation: The number of address windows required by the task and any shared regions exceeds 16 for RSX–11M–PLUS and Micro/RSX tasks.

Task-build aborted via request
option-line

Explanation: The option-line displayed contains a request from the user to abort the task build.

Too many nested .ROOT/.FCTR directives
invalid-line

Explanation: The invalid-line displayed contains a .FCTR directive that exceeds the maximum nesting level (16).

Too many parameters
invalid-line

Explanation: The invalid-line displayed contains a keyword with more parameters than required.

Too many parentheses levels
invalid-line

Explanation: The invalid-line displayed contains a parenthesis that exceeds the maximum nesting level (16).

Truncation error in module module-name

Explanation: 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

Explanation: 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

Explanation: The invalid-line displayed contains unbalanced parentheses.

n Undefined symbols segment seg-name

Explanation: 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

Explanation: The option-line displayed contains a VSECT option whose base address plus window size exceeds 177777.

Work file I/O error

Explanation: An I/O error occurred during an attempt to reference data stored by TKB in its work file.

Glossary

absolute shared region
A shared region that has the same virtual addresses in all tasks that refer to it.

Autoload
The method of loading overlay segments, in which the overlay run-time system routines automatically load overlay segments when they are needed and handles any unsuccessful load requests.

Autoload vector
A transfer-of-control instruction generated by the Task Builder to resolve an up-tree reference to a global symbol.

co-tree
One of one or more secondary tree structures within a multiple-tree overlay structure. When a co-tree's root segment contains code or data, the root segment of the co-tree is made resident in physical memory through calls to the overlay run-time system routines.

common block
Another name for resident common.

disk-resident
That which resides on disk storage until needed.

disk-resident overlay segment
An overlay segment that shares the same physical memory and virtual address space with other segments. The segment is read in from disk each time it is loaded (compare "memory-resident overlay segment").

global cross-reference
A list of global symbols, in alphabetical order, accompanied by the name of each referencing module.

global symbol
A symbol whose definition is known outside the defining module.

header
That portion of a task image that contains the task's characteristics and status. Shared regions, although built like a task, do not have a header.

host system
The system on which a task is built.

logical addresses
The actual physical addresses that the task can access.

logical address space
The total amount of physical memory to which the task has access rights.

main tree
An overlay tree whose root segment is loaded by the Executive when the task is made active.

manual load
The method of loading overlay segments in which you include explicit calls in your routines to load overlays and handle unsuccessful load requests.

mapped array area
An area of the task's physical memory, preceding the task image, that is used for storing large arrays. Space in the area is reserved by means of the VSECT option or through a Mapped Array Declaration contained in an object module. Access is through the mapping directives issued at run time.

memory allocation file (map)
The output file created by the Task Builder that lists information about the size and location of components within a task.

memory-resident
In general, that which resides in memory all the time. The entity, as in the case of memory-resident overlays, may initially reside on disk.

memory-resident overlay segment
An overlay segment that shares virtual address space with other segments, but which resides in its own physical memory. The segment is loaded from disk only the first time it is referenced; thereafter, mapping directives are issued in place of disk load requests.

multiuser tasks
A task whose read-only region is shared among several copies of the same task.

Overlay Description Language
A language that allows you to describe the overlay structure of a task.

overlay run-time routines
A set of system library subroutines linked as part of an overlaid task that are called to load segments into memory.

overlay segment
A segment that shares virtual address space with other segments, and is loaded when needed.

overlay tree
A tree structure consisting of a root segment and, optionally, one or more overlay segments.

path
A route that is traced from one segment in the overlay tree to another segment in that tree.

path-down
A path toward the root of the tree.

path-loading
The technique used by the Autoload method to load all segments on the path between a calling segment and a called segment.

path-up
A path away from the root of the tree.

physical address
The assigned byte location in physical memory, which is usually located in the processing unit.

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

privileged task
A task that has privileged access rights to memory. A privileged task can access the Executive and the I/O page in addition to its own partition and referenced shared regions.

program section
A section of memory that is a unit of the total allocation. A source program is translated into object modules that consist of program sections with attributes describing access, allocation, relocatability, and so forth.

region
A contiguous block of physical addresses in which a driver, a task, a resident common, or a library resides.

resident common
A shared region in which resides data that can be shared by two or more tasks.

resident library
A shared region in which reside single copies of commonly used subroutines that can be shared by two or more tasks.

root segment
The segment of an overlay tree that, once loaded, remains in memory during the execution of the task.

runnable task
A task that has a header and stack and that can be installed and executed.

shared region
A shared region is a block of data or code that resides in physical memory and can be used by any number of tasks. A shared region is built and installed separately from the task.

supervisor-mode library
A library of routines that uses the supervisor-mode memory management APRs to map to both the task and its own routines.

symbol definition file
The output object file created by the Task Builder that contains the global symbol definitions and values and sometimes program section names, attributes, and allocations in a format suitable for reprocessing by the Task Builder. Symbol definition files contain linkage information about shared regions.

target system
The system on which a task executes.

task image file
The output file created by the Task Builder that contains the executable portion of the task.

virtual addresses
The addresses within the task. Task addresses can range from 0 to 177777_8, depending on the length of the task.

virtual address space
That space encompassed by the range of virtual addresses that the task uses.

virtual program section
A program section that has virtual memory allocated to it, but not physical memory. Virtual address space is mapped into physical memory at run time by means of the mapping directives.

window
A continuous virtual address space that can be moved to allow the task to examine different parts of a region or different regions.

window block
A structure defined by the Task Builder that describes a range of continuous virtual addresses.

Index

RSX-11M-PLUS and Micro/RSX
Task Builder Manual
AA-JS08A-TC

READER'S COMMENTS

Your comments and suggestions are welcome and will help us in our continuous effort to improve the quality and usefulness of our documentation and software.

Remember, the system includes information that you read on your terminal: help files, error messages, prompts, and so on. Please let us know if you have comments about this information, too.

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.

____

____

____

____

____

____

What kind of user are you? ___ Programmer ___ Nonprogrammer

Years of experience as a computer programmer/user: ______________

Name ______________________ Date__________

Organization ______________________

Street ______________________

City ______________________ State or Country __________ Zip Code __________

—— **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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

—— **Do Not Tear - Fold Here** ————————————————————

RSX-11M-PLUS and Micro/RSX
Task Builder Manual
AA-JS08A-TC

READER'S COMMENTS

Your comments and suggestions are welcome and will help us in our continuous effort to improve the quality and usefulness of our documentation and software.

Remember, the system includes information that you read on your terminal: help files, error messages, prompts, and so on. Please let us know if you have comments about this information, too.

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.

____
____
____
____
____
____

What kind of user are you? ____ Programmer ____ Nonprogrammer

Years of experience as a computer programmer/user: ______________

Name ______________________ Date ______________

Organization ______________________

Street ______________________

City ______________________ State or Country __________ Zip Code __________

--- 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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---