July 1978

This document describes the use of the Linker on the TRAX system.

TRAX
Linker Reference Manual

Order No. AA-D342A-TC

OPERATING SYSTEM AND VERSIONS: TRAX Version 1.0

digital equipment corporation · maynard. massachusetts

First Printing, August 1978

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

TABLE OF CONTENTS

FIGURES

TABLES

CHAPTER 1

INTRODUCTION

1.1 MANUAL OBJECTIVES AND READER ASSUMPTIONS

This manual is a tutorial, intended to introduce the user to the concepts and capabilities of the TRAX Linker.

Examples are used to introduce and describe features of the Linker. These examples proceed from the simplest case to the most complex.

The user should be familiar with the basic concepts of the TRAX system described in the Introduction to TRAX, and with basic operating procedures described in the TRAX Support Environment User's Guide. (See Section 1.3.)

1.2 STRUCTURE OF THE DOCUMENT

The manual has seven chapters. Chapters 1 through 4 describe the basic capabilities of the Linker, and Chapters 5 and 6 describe its advanced capabilities. Chapter 7 discusses memory dumps. The appendixes include error messages, data formats, and a glossary.

Chapter 1 outlines the capabilities of the Linker.

Chapter 2 describes the command sequences used to interact with the Linker.

Chapter 3 lists the command qualifiers.

Chapter 4 discusses options that you can specify in a LINK command line.

Chapter 5 describes the overlay capability and the language used to define an overlay structure.

Chapter 6 gives the two methods that can be used for loading overlay segments.

Chapter 7 describes two memory dumps--Post-mortem and Snapshot.

1.3 ASSOCIATED DOCUMENTS

Other manuals closely allied with the purpose of this document are the TRAX:

- Support Environment User's Guide
- BASIC-PLUS-2 User's Guide and the
- COBOL User's Guide.

1.4 INTRODUCTION TO THE LINKER

The fundamental executable unit in the TRAX support environment is the task. A routine becomes an executable task image, as follows:

1. The routine is written in a supported source language.
2. It is entered as a text file, through the editor.
3. It is translated to an object module, using the appropriate language translator.
4. The object module is converted to a task image by the Linker program.
5. The task is run.

If errors are found in the routine as a result of executing the task, the user makes corrections to the text file using the editor, and then repeats steps 3 through 5.

If a single routine is to be executed, the use of the Linker is appropriately simple. The user specifies as input only the name of the file containing the object module produced from the translation of the program, and specifies as output a name for the task image.

Typically, however, a collection of routines is run rather than a single program. In this case the user names each of the object module files, and the Linker links the object modules, resolves any references to the system library, and produces a single task image, ready to be installed and executed.

The Linker makes a set of assumptions (defaults) about the task image based on typical usage and storage requirements. These assumptions can be overridden by including switches and options in the task-building terminal sequence. This allows the user to build a task that is tailored to its own input/output and storage requirements.

The Linker also produces, upon request, a memory allocation file (map) that contains information describing the allocation of storage, the modules that make up the task image, and the value of all global symbols. The user can also request that a list of global symbols, accompanied by the name of each referencing module, be appended to the file (global cross-reference).

The Linker provides the user with an overlay capability as a means of reducing the memory requirements of a task. A task can be divided into several overlay segments that reside on disk. Disk-resident overlays are loaded into memory when they are needed.

If the task is configured as an overlay structure (that is, as a multi-segment task), the user becomes responsible for loading segments into memory as they are needed. The method provided for loading overlay segments is called autoload.

With the autoload method, no special calls are required to load the task's overlay segments. The segments are loaded automatically by the Overlay Runtime System according to the structure defined by the user at the time the task was built.

The user can become familiar with the capabilities of the Linker by degrees. Chapter 3 gives sufficient basic information about Linker commands to handle many applications. The remaining chapters deal with special features and capabilities for handling advanced applications and tailoring the task image to suit the application.

This manual describes the development of an example application, BILL. In the first treatment of BILL, the user builds a task using all the default assumptions. Successive treatments illustrate the main points of each chapter in a realistic manner. Qualifiers and options are added as they are required, an overlay structure is defined when the task increases in size, the loading of overlays is illustrated, and finally the task is moved from a development system to a system with a different configuration.

The memory allocation files for the various stages of task development are included. The effect of a change can be observed by examining the map for the previous example and the map for the example in which the change is made.

1.5 RMS FILE SPECIFICATION INFORMATION

A TRAX file specification conforms to standard RMS conventions. It has the following form:

```
device:[ufd]filename.filetype;version
```

where:

device	is 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 a 1- or 2-digit octal unit number and a colon(:); for example, DM0: or DB1:. A logical device name may also be used.
[ufd]	is the user file directory specification consisting of two octal numbers in the range of 1 through 377 (octal). These numbers **must** be enclosed in brackets and separated by a comma and must be in the following format: [group,member] For example, member 225 of group 300 would use the following entry: [300,225]
file name	is the name of the desired file. The file name can be from one to nine alphanumeric characters; for example, BILLRTN. You must always specify the file name. There is no default specification for this component. Failure to specify the file name causes an error to be generated. The lone exception is when the wild card (*) specification is used. The wild card (*) specification causes the system to process **all** file names on the specified user file directory.

file type	is the 3-character file type identification. Separate the file name and file type with a period(.). Files with the same name but different functions are distinguished from one another by the file type; for example, BILL.TSK and BILL.OBJ might be the task file and the object file, respectively, for the program BILL. You may omit the file type, but you should avoid this practice when dealing with system programs which generally assume a default file type for various operations. The wild card (*) specifier may be used in place of a file type. The action specified will be applied to all file types associated with a given file name or wild card file name. (E.G. *.* specifies the current version of **all** files in the user file directory.)
version	is the octal version number of the file in the range 1 through 77777 (octal). Separate the extension and version by a semicolon(;). Various versions of the same file are distinguished from each other by the version number; for example, BILL.OBJ;1 and BILL.OBJ;2 are successive versions of the same file. The version number may be omitted, in which case the current (highest-numbered) version is assumed. To act upon all versions of a file, you can use the wild card (*) specification.

The device, user file directory code, file type, and the version specification are all optional.

Table 1-1 lists the default assumptions applied to missing components of a file specification. Table 1-2 lists the file types assumed by PDP-11 system software.

Table 1-1

Item	Default
device	user's current default device
[ufd]	user's current default [ufd]
version	for an input file, the default version number is the existing version with the highest (octal) number. For an output file, the default is calculated as one greater than the highest existing version number for that file.

Table 1-2

File Contents Description	Default File type
task image file	.TSK
memory allocation file	.MAP
symbol definition file	.STB
object module	.OBJ
object module library file	.OLB
overlay description file	.ODL
indirect command file	.CMD
COBOL source text file	.CBL
BASIC-PLUS-2 source file	.B2S
MACRO source file	.MAC

CHAPTER 2

PROGRAM DEVELOPMENT AND THE LINK COMMAND

This chapter is divided into two parts. The first part describes the program development process as it applies to the TRAX Support Environment. Illustrations of the steps in program development are made using examples similar to those found in TRAX Support Environment applications.

The second part presents the concept of linking a task image, and describes the LINK command in terms of syntax rules and available features.

2.1 AN OVERVIEW OF THE PROGRAM DEVELOPMENT PROCESS

The program development process can be divided into several distinct parts. Consider the following outline in developing a program to be run on the PDP-11.

1. Define the operations to be performed. (Flow-charting)
2. Code the defined procedure into source language statements. (Coding)
3. Create a source statement file in machine-readable form. (Editing)
4. Compile the source language statements into an object module. (Compiling)
5. Correct errors and recompile until your program is clean. (Debugging)
6. Link the compiled object modules and required system libraries to form an executable task image file. (Linking)
7. Run your program.

This manual assumes that you can define a procedure and code it into appropriate source language statements. In the case of TRAX, COBOL and BASIC-PLUS-2 are supported languages. For details about syntax and coding of these languages, refer to the TRAX Language Reference Manual and User's Guide for the language you are using.

The remaining parts of this section describe the last five steps of program development as listed above, and how they are implemented in the TRAX Support Environment.

2.1.1 Creating Source Language Statement Files

After you have defined your procedure and translated the definition into appropriate source language statements, the next step is to enter the source statements into the computer. In the TRAX Support Environment, this is done from a "support" terminal using the DEC Editor. Consult your system manager for the location of your installation's support terminals, and the procedure required to log in to the Support Environment.

The DEC Editor is a utility program which will allow you to create and maintain text files from a video or hard-copy terminal. If you are unfamiliar with the Editor's operations, consult the DEC Editor Reference Manual or the Beginner's Guide to the DEC Editor.

The Editor is entered by typing the command string:

```
     >EDIT [file-specification]
```

The file-specification is to be supplied by you according to the rules discussed in Section 1.5 for RMS files. When you are creating a new file, the file specification should be a new file name, and the file type should follow the style suggested in Section 1.5 for the particular source language you are using. (E.G. COBOL source files should have a .CBL file type; BASIC-PLUS-2 files should have a .B2S file type etc.).

2.1.2 Compiling the COBOL Programs

After you have created source language input files with the help of the DEC Editor, the next step in the program development process is compilation. The compiler is a system program which translates your high-level source language statements into object modules which consist of machine language instructions coded as octal numbers. If you were to compile the example programs that were entered into a source file in Section 2.1.1, you would invoke the COBOL compiler with the following command string:

```
     >COBOL/LIST/SWITCHES:(/KE:ST)
     FILE? STATE
     >COBOL/LIST/SWITCHES:(/KE:LA)
     FILE? LABELS
     >COBOL/LIST/SWITCHES:(/KE:CR)
     FILE? CREDLM
     >COBOL/LIST/SWITCHES:(/KE:EX)
     FILE? EXCEPT
```

The first command invokes the COBOL compiler and directs it to take source input from STATE.CBL and place the relocatable object code in INRTN.OBJ. The remaining commands perform similar actions for the source files LABELS.CBL, CREDLM.CBL, and EXCEPT.CBL. The /LIST and /SWITCHES:(/KE:) keywords inform the COBOL compiler that a listing is to be spooled to the line printer, and that the four routines are to be compiled with the kernel names specified by the /KE switch. Further information regarding language compilers and the features available can be obtained by consulting the appropriate TRAX User's Guide for the source language you are using.

The listing output from your compilation will indicate errors in your source language text, and will provide information regarding the cause of the error. You then can use the DEC Editor to make the required corrections to your source statements, and recompile. Several iterations of the compile and editing process are usually needed to obtain an error-free compilation. Once the compiler has reported that your compilation is error-free or "clean", you may then proceed to the next program development step, linking the object modules to form a task. In the higher-level languages, such as COBOL and BASIC-PLUS-2, you must first run a language utility to create the appropriate overlay description file for your program.

2.1.3 Linking the Task

The Linker is a system program that takes object modules and system library modules as input, and merges this information to form a task image file. The task image file can be copied into memory and run by the operating system. Linking is the final step in the program development process.

The example programs that have been entered and compiled in the previous sections can be linked by issuing the following command string:

```
>LINK/TASK:STATEMENT/MAP:STATE/OVERLAY:STATE/OPTIONS
OPTIONS? UNITS=7
OPTIONS? ASG=TI:1
OPTIONS? ASG=SY:2:3:4:5:6:7
```

The LINK command specifies the name of the task image file (BILL.TSK;1), the name of the memory allocation file (BILL.MAP;1), and the name of the input ODL file. Section 2.2 of this manual presents the Link command, and describes the syntax required to use it. Chapter 3 discusses the Qualifiers that may be used with the Link command, and Chapter 4 explains the Options that may be selected.

2.1.4 Running Your Program

After all steps of program development (editing, compiling and linking) have been successfully completed, you may run your program by entering the run command followed by the file name of the task image file that was created by the Linker. In our example programs, the command string:

```
>RUN STATEMENT
```

will tell the TRAX operating system to copy the task image file BILL.TSK into memory and execute the program. A number of Qualifiers are available to be used with the RUN command. They are described in the TRAX Support Environment User's Guide.

2.2 THE TRAX LINKER

The TRAX Linker combines the input files and resolves references to the system libraries to create a single executable task image. The Linker produces output files according to the command qualifier settings. A task image file is produced by default, although the qualifier /TASK:[filespec] can be used to give the task image file a different name from the input file. Generation of the task image file can be suppressed by using the /NOTASK command qualifier. A memory allocation map, which identifies the size and location of components within the task, is produced on the line printer by explicit use of the /MAP qualifier. The /MAP:[filespec] qualifier produces a memory allocation file which is stored on the user's system device. The /SYMBOLS qualifier must be specified to produce a symbol definition file that contains the global symbol definitions in the task and their virtual or relocatable addresses, in a format suitable for reprocessing by the Linker.

Output task image files assume the file name of the **first** input file unless the command qualifier specifies a particular file specification as part of the qualifier.

2.2.1 LINK Command Formats

The standard LINK command format is:

You must separate the command qualifiers from the input file specification by inserting at least one space between them. To use the Linker in an interactive prompting mode, first invoke the Linker by typing:

```
     > LINK[/command qualifiers]
```

The system responds with a file? prompt. You then enter the input file description arguments as shown:

```
     > FILE?[file-specification[/file-qualifiers]]
```

You must respond to a FILE? prompt with at least one file specification when you are in the interactive mode. You may specify more than one input file specification, but must separate them with a comma, a space or a tab character as a delimiter.

2.2.2 Command Qualifiers

Command Qualifiers are keywords which are used to specify output if files, and to tell the Linker to search for and include certain system library modules in the executable task image file. They always begin with a slash (/) and may be abbreviated to the fewest number of digits that causes the qualifiers to remain unique. Inserting NO between the / and the first letter of the keyword results in the negation of that qualifier. Command Qualifiers are presented in table form and discussed in detail in Chapter 3.

2.2.3 File Specifications

File specifications conform to the RMS standard format shown in Section 1.5. Output file specifications are appended to their corresponding keywords after a colon (:). Some special types of input files (DEBUG,OVERLAY) are also specified following a command qualifier. Input object modules and system library files are specified following the command qualifiers in the input string. A space must appear between the last command qualifier and the first input file specification. Multiple input files may be specified, and must be delimited by a space, comma, or tab character. Input and library files may be qualified through the use of file-qualifiers which are described in Section 3.2 of this manual.

2.2.4 OPTIONS

Options are used to specify the characteristics of the task being built. If you type the command qualifier /OPTIONS as part of the LINK command, the Linker prompts for additional input by displaying OPTIONS?: on the line following the last line of the input file specification. You then enter one of the Linker options and terminate the line by entering a carriage return. Prompting continues on successive lines until you type a slash (/) followed by a carriage return in response to an OPTIONS?: prompt. This sequence of characters causes the prompting to cease and activates the Linker to process your command string. A second form of option specification is /OPTIONS:[filespec] where option input keywords and arguments are contained in the file specified by the /OPTIONS command qualifier. The second form suppresses interactive prompting for option input.

The example in Section 2.1.3 illustrates interactive prompting for options:

An example of the second form of /OPTIONS qualifier is given in Section 2.2.6. The syntax and interpretation of each TRAX Linker option are described in Chapter 4.

The form of an option is a keyword followed by an equal sign (=) and an argument list. The arguments in the list are separated from one another by colons (:). In the preceding example, the first option consists of a keyword UNITS and a single argument 6 indicating that the task being linked is to be assigned 6 logical units. The second option consists of the keyword ASG and an argument list DB2:5,DB1:6 indicating that disk unit 2 has been assigned to logical unit 5, and disk unit 1 to logical unit 6. This is a demonstration of the manner in which several arguments may be presented within the same option argument list. In the absence of the /OPTIONS qualifier, the user task is linked using the default option settings.

2.2.5 Multiple Line Input

LINK command lines are often complex, requiring command qualifiers and file specifications that cause the command string to exceed the number of characters allowed on a single input line. To enter a LINK command line over more than one line, type a hyphen (-) as the last printing character on the line, then continue the command on the next line.

```
>LINK/TASK:STATEMENT/MAP:STATE-
DCL>/OVERLAY:STATE/DEBUG/OPTIONS
OPTIONS?
```

2.2.6 Indirect Command File Facility

The LINK command string can also be entered as a text file and later invoked through the indirect command file facility. To use the indirect command file facility, you prepare a file that contains the desired command string input to link object modules into a task-image file. The contents of the indirect command file are invoked by typing @ followed by the file name of the indirect command file.

```
> @AFIL
```

When the symbol "@" is encountered, search for commands is directed to the file specified following the "@" symbol. While accepting input from an indirect file, the Linker does not display prompting messages on the terminal. If the input file specifications are not in the indirect command file, prompting for the input file follows the last item in the command file. If the /OPTIONS command qualifier appears in the indirect command file **without** an accompanying file specification, interactive prompting for option input will occur.

The single line command string which references the indirect command file AFIL.CMD is equivalent to the following keyboard sequence:

```
     LINK/TASK:STATEMENT/MAP:STATE/OVERLAY:STATE/DEBUG-
     DCL>/OPTIONS:BFIL
```

When you create the indirect command file, you must follow the syntax rules for command qualifiers, file specifications, and options which are associated with the LINK command.

Suppose the file BFIL.CMD contains a set of standard options that are requested by a number of users at an installation. That is, every programmer in the group uses the options in BFIL.CMD. These standard options can be included in a link command file by modifying AFIL.CMD to include an input file reference to BFIL.CMD as the file specification following the /OPTIONS: command qualifier.

The contents of BFIL.CMD are:

```
     UNITS=7
     ASG=TI:1
     ASG=SY:2:3:4:5:6:7
```

You include this file specification in the file AFIL.CMD

If the command:

```
     > @AFIL
```

is issued, it then becomes the equivalent of the following sequence:

```
     LINK/TASK:STATEMENT/MAP:STATE/OVERLAY:STATE/DEBUG-
     DCL>/OPTIONS

     OPTIONS?  UNITS=7
     OPTIONS?  ASG=TI:1
     OPTIONS?  ASG=SY:2:3:4:5:6
     OPTIONS?  //
     >
```

The /OPTIONS command qualifier is described in Section 2.2.4, and detailed examples of its use are given in Chapter 4. A complete discussion of indirect command files is included in the TRAX Support Environment User's Guide. The discussion includes examples of how several different types of TRAX commands can be used in the same indirect command file.

In the case of BASIC-PLUS-2 tasks, the BUILD command is issued to the BASIC-PLUS-2 compiler which then creates an indirect command file containing the command qualifiers, file specifiers, and standard options required to create the desired task-image file from the BASIC object module. For example, consider a BASIC-PLUS-2 source program and object modules called SORT02. The BUILD command produces a file called SORT02.CMD which may be input to the Linker by typing:

```
     >LINK/BASIC SORT02
```

You should not modify the generated command file produced by the BUILD command in BASIC-PLUS-2, because unpredictable and possibly fatal results may occur.

2.2.6.1 **Comments** - You can document the purpose and status of a task by adding comments to the Link command file. Comments can be placed at any point in the command file. Begin a comment with an exclamation point (!), and terminate it with a carriage return. The Linker interprets the text between the exclamation point and the carriage return as a comment and does not process it.

Consider the annotation of the following LINK command string which is to be executed as an indirect command file. Comments have been added to the lines of the command string to document the functions performed by the Linker, as well as a brief description of the contents of the input object modules. A note concerning the current status of the task has been inserted at the end of the file.

```
! TASK STATEMENT
! COBOL TASK USING COBOL MERGE ODL FILE STATE.ODL
! FOUR INPUT MODULES
! STATE - MAINLINE STATEMENT PROGRAM
! LABELS - SUBROUTINE TO PRINT LABELS
! CREDLM - SUBROUTINE TO WRITE CREDIT LETTERS
! EXCEPT - EXCEPTION PROCESSING SUBROUTINE

LINK/TASK:STATEMENT/MAP:STATE/OVERLAY:STATE/DEBUG-
/OPTIONS:BFIL
! 7 UNITS USED, 1 TO TI:, 2-7 for SY:
```

This feature is extremely useful for installations where maintenance of existing programs and tasks is not generally performed by the original developer. The comment capability allows you to explain your logic in building a task in the same way as you would place comments in a source program file.

CHAPTER 3

COMMAND AND FILE QUALIFIERS

Command qualifiers provide information to the TRAX Linker. This information is used by the Linker to determine how it will process your compiler-generated object modules into executable task image files. Command qualifiers allow you to provide the Linker with four general types of information:

1. You may specify the types of output files to be created by the Linker. You are given the option of specifying file names for the output files.

2. You may tell the Linker to include predefined or user-defined object modules in the task image.

3. You may specify how the Linker is to search for object modules in system libraries.

4. You may specify input files of a specialized type. The Linker recognizes the qualifier and links the task based upon the contents of the specialized input file.

Table 3-1 lists the command qualifiers in alphabetical order. A short description of their function is also included. Section 3.1 gives a more detailed explanation of each command qualifier.

Input File Qualifiers allow you to instruct the Linker to perform specialized processing with certain types of input files. The Input File Qualifiers are shown in Table 3-2, and described in detail in Section 3.2.

Table 3-1
Link Command Qualifiers

Keyword	Function
/BASIC	Tells the Linker that the input file is a command file created by the BASIC-PLUS-2 compiler.
/CHECKPOINT[:keyword]	The Linker should include checkpoint capability in the task image file. The optional keyword specifies TASK or SYSTEM checkpoint space allocation.

(Continued on next page)

Table 3-1 (Cont.)
Link Command Qualifiers

Keyword	Function
/CROSS_REFERENCE	Tells the Linker to include a global symbol cross-reference listing in the memory allocation file.
/DEBUG[:filespec]	Includes a debugging aid in the task image file. Optional file name contains a user-written debugging module.
/DUMP	The task image is linked with modules that provide a post-mortem dump in the event of abnormal task termination.
/FULL_SEARCH	Controls symbol table searching in overlaid tasks with co-trees.
/MAP[:filespec(/file-qualifier)	Tells the Linker to produce a memory allocation file. file-qualifiers: /FULL Include all modules in map. /NARROW Format map for 72-col. output. /SHORT Produce only a summary map. /WIDE Format map for 132-col. output.
/OPTIONS	Apply LINK command options specified after command string.
/OPTIONS[:filespec]	Apply Link Command options contained in the specified file.
/OVERLAY[:filespec]	The Linker does its processing according to the specified overlay description file.
/SEQUENTIAL	Task object modules are allocated memory sequentially.
/SYMBOLS[:filespec]	Instructs the Linker to produce a symbol table file.
/TASK[:filespec]	The Linker is to produce a task image file.

3.1 COMMAND QUALIFIERS

A detailed description of each command qualifier is presented in this section. The meaning and effect of each qualifier are described and the default condition is identified.

3.1.1 The BASIC Command Qualifier

Syntax: LINK/BASIC [Command File Specification]

The input file is a command file created when you issued the BUILD command to the BASIC-PLUS-2 compiler. The Linker decodes the command file and links the task image file according to the information supplied in the command file. No prompting for files or options occurs.

The /BASIC command qualifier should only be specified in conjunction with BASIC-PLUS-2 compiler-generated command files. For further information, see the discussion of the BUILD command in the TRAX BASIC-PLUS-2 User's Guide.

CAUTION

Do not attempt to modify the command file after it has been created. Unpredictable or fatal results may occur when user-edited BASIC-PLUS-2 command files are supplied to the Linker.

3.1.2 The CROSS-REFERENCE Command Qualifier

Syntax: /CROSS_REFERENCE

A global symbol cross-reference listing is produced. The cross reference listing is appended to the memory allocation (MAP) file. An example of this listing is provided in Appendix A.

The Linker will not produce a global symbol cross reference listing unless this qualifier is specified.

3.1.3 The DEBUG Command Qualifier

Syntax: /DEBUG[:file specification]

This qualifier instructs the Linker to include a debugging aid in the task image file. If the file specification is omitted, the system's debugging aid (ODT) is assumed to be the default module. If a file specification is present, the debugging aid contained in the specified file will be linked into the task image. The user-generated debugging aid must be in object module format. See Appendix D for additional information on including a debugging aid.

3.1.4 The DUMP Command Qualifier

Syntax: /DUMP

This qualifier instructs the Linker to include system modules in the task image file that will provide a post-mortem dump in the event that your task is abnormally terminated.

Memory dumps are discussed in detail in Chapter 7.

The default assumption is /NODUMP

3.1.5 The FULL-SEARCH Command Qualifier

Syntax: /FULL_SEARCH

When processing modules from the default object module library, the presence of this qualifier causes the Linker to search all co-tree overlay segments for a matching definition or reference.

If this switch is negated, unintended global references between co-tree segments are eliminated. Definitions of global symbols from the default library are restricted in scope to references in the main root and the current tree. Use of this qualifier is discussed in detail in Chapter 6.

/NOFULL_SEARCH is the default setting assumed by the Linker.

3.1.6 The MAP Command Qualifier

Syntax: /MAP[:filespec] or /MAP[:filespec/filequalifier]

This qualifier instructs the Linker to produce a memory allocation (.MAP) file as it links the task image file.

If the file specification is present, the file type field may be omitted. The Linker assumes the .MAP file type.

If this qualifier is present and no file is specified, the memory allocation file is spooled directly to the line printer. The memory allocation file (is deleted after printing) (remains on your user file directory, taking the file name of the task image file, and the file type .MAP).

The Linker assumes a default of /NOMAP. The following qualifiers may be applied to the file specification:

/FULL	The Linker will include all modules in the memory allocation file, even those which explicitly or by default have the /NOMAP input file qualifier (see Section 3.2.4).
/NARROW	The Linker produces a map listing 72 characters wide, suitable for printing on an output terminal.
/SHORT	Tells the Linker to include only the segment headings in the memory allocation file.
/WIDE	Produce a map 132 characters wide, suitable for printing on a line printer. When /MAP is specified, this is the default file qualifier.

3.1.7 The OPTIONS Command Qualifier

Two forms of the /OPTIONS command qualifier are available. The first form prompts you for option input. The second form allows you to specify a file which contains option input strings. The syntax for each form appears before the text explaining its usage. Chapter 4 contains detailed information on TRAX Linker options.

3.1.7.1 **Interactive Format** - Syntax: /OPTIONS

The Linker interactively prompts for option input lines after you have supplied the command qualifiers and input file specifications to the Linker. Prompting will continue until you enter a slash (/) followed by a carriage return in response to an OPTIONS?: prompt. The slash and carriage return sequence signals the Linker that all option input has been supplied. The Linker then begins processing the input files.

3.1.7.2 **Command File Format** - Syntax: /OPTIONS:[file specification]

When the filespec is supplied with the /OPTIONS qualifier, the Linker treats that file as a series of option input-lines. Interactive prompting for options does not occur. The default file type for the input file is .CMD.

3.1.8 The OVERLAY Command Qualifier

Syntax: /OVERLAY:[ODL File Specification]

The input file specified with the /OVERLAY command qualifier is assumed to be an Overlay Description Language (ODL) file. The Linker creates the task image file according to the overlay structure defined in the specified input file.

An overlay description file must be supplied with this qualifier. The user cannot supply another input file. The Linker will not accept any input files other than those described in the supplied ODL file.

Overlay descriptions are discussed in Chapter 6. ODL files are also discussed in the TRAX COBOL and TRAX BASIC-PLUS-2 User's Guides.

3.1.9 The SEQUENTIAL Command Qualifier

Syntax: /SEQUENTIAL

The task image is constructed from the specified object files in the order stated in the LINK command string. Chapter 5 describes the allocation of storage within the task image, and gives an example of the allocation performed under the default assumption and the allocation that results when the /SEQUENTIAL qualifier is specified.

The Linker does not reorder the program files alphabetically. This qualifier must not be used for modules that rely upon alphabetical program section allocation; in TRAX such modules include RMS modules from RMSLIB.

The default condition is non-sequential storage allocation.

3.1.10 The SYMBOLS Command Qualifier

Syntax: /SYMBOLS[file specification]

This qualifier tells the Linker to produce a symbol definition file. If the file specification is present, the file type field is optional. The Linker assumes the .STB file type.

If the filespec is absent, the first input file name becomes the symbol table definition file name, and .STB the assumed file type.

/NOSYMBOLS is the default setting for this qualifier.

3.1.11 The TASK Command Qualifier

Syntax: /TASK[:file specification]

This qualifier instructs the Linker to create a task image file. It is set by default, with the file name being taken from the first input file, and the file type assumed to be .TSK.

If a file name is specified, the file type is optional; in that case the default assumption file type is .TSK.

Use of the /NOTASK qualifier causes the Linker to process the input files for unresolved symbol references, but suppresses creation of a task image file.

Table 3-2
Input File Qualifiers

Keyword	Function
/CONCATENATED	The input file consists of concatenated object modules. The /NO prefix with this qualifier instructs the Linker to take only the first object module from a series of concatenated files.
/DEFAULT_LIBRARY[:filespec]	Directs the Linker to use the specified file as the system default library. If this qualifier is absent, the default is LB0:[1,1]SYSLIB.OLB.
/LIBRARY[:modl:...:mod n]	Identifies the input file as an object module library file. Module specifiers direct the Linker to read only those modules from the library.
/MAP	This qualifier tells the Linker to include the modules in the associated file in the memory allocation (.MAP) file. /NOMAP results in the modules not being listed in the map.
/SELECT_SYMBOLS	The input file is to be selectively searched for unresolved global symbol references. Only those modules which resolve global symbols will be included in the task image file.

3.2 INPUT FILE QUALIFIERS

Input File Qualifiers tell the Linker that specialized processing is to be performed on the associated input file.

3.2.1 The CONCATENATED Input File Qualifier

Syntax: Input File Specification/[NO]CONCATENATED

The Linker normally processes all modules in the input file to form the task image. When /NOCONCATENATED is present, the Linker processes only the first module in the task image, regardless of the number present. Do not use this qualifier with the /LIBRARY qualifier, as it will be overridden.

/CONCATENATED is the default setting for this qualifier.

3.2.2 The DEFAULT LIBRARY Input File Qualifier

Syntax: /DEFAULT_LIBRARY:file specification

The Linker searches the specified library file when it is resolving undefined global symbol references. This qualifier overrides the default system library LB0:[1,1]SYSLIB.OLB.

If the specified library is empty (no modules have been inserted into it), the default library reverts to the system library.

The Linker assumes a default system library ([1,1]SYSLIB.OLB). Any other default library name must be specified by the use of this file qualifier.

3.2.3 The LIBRARY Input File Qualifier

There are two forms of this qualifier. The first form allows you to provide a Library (.OLB) file as input to the Linker. The Library is used to resolve global symbol references. The second form allows you to specify certain modules from an existing Library file as input to the Linker. The named modules are included in the task image file being created by the Linker.

3.2.3.1 **Resolve all Global Symbols** - Syntax: Input File Specification/LIBRARY

The Linker searches the specified input library file to resolve undefined global symbol references. The Linker extracts any modules which resolve global references, and includes them in the task image file.

You must append /LIBRARY to any input library file.

3.2.3.2 **Include Selected Library Modules** - Syntax: Input file specification/LIBRARY:[(]mod-1[,...,mod-n)]

The input module is assumed to be a library (.OLB) file of relocatable object modules from which the modules named in the argument list are to be copied for inclusion in the task image. The module names are those defined at assembly time by the .TITLE directive (or if no .TITLE directive, the file name (first six characters) when inserted by the Librarian). Up to eight modules can be specified. The Linker includes only the specified object modules in the task image file.

The /LIBRARY file qualifier must be appended to the input file specification. It is never assumed as a default.

NOTE

To direct the Linker to search a library file for both global symbol references and selected modules that are needed in the task image, the You must name the library file twice. First, specify the /LIBRARY qualifier and no other arguments. Second, specify the desired modules, directing the Linker to include those modules in the task image file that is being created. See Section 3.2.3.2 to see how you may specify named modules with the /LIBRARY file qualifier.

3.2.4 **The MAP Input File Qualifier**

Syntax: Input File Specification/MAP

This qualifier instructs the Linker to include the input file when it creates the memory allocation file.

If /NOMAP is specified, no details of modules contained in the file will appear in the memory allocation map or cross-reference listing.

User supplied input object module files are assumed to have the input file qualifier /MAP as a default.

For system library files, resident libraries, and common area, /NOMAP is assumed as the default file qualifier.

NOTE

The /NOMAP qualifier, when it is qualifying an input file, is overridden by /FULL_SEARCH. (See Table 3-1 and Section 3.1.6).

3.2.5 The SELECT SYMBOLS Input File Qualifier

Syntax: Input File Specification/SELECT_SYMBOLS

This qualifier tells the Linker to selectively search the input file. The search is made for only those global symbols for which an undefined reference exists. The Linker includes only the required symbol definitions from the specified file as distinct from all global symbols of that file. This qualifier is useful when an input file is the symbol table (.STB) output of another Link command, because it reduces the size of symbol table searches.

If this file qualifier is absent, all global symbols from the input file will be included in the task image file. This is the default condition.

If the input file specified with this qualifier is a Library or concatenated file, the qualifier is active for each module in the input file.

CHAPTER 4

COMMAND OPTIONS

LINK Command Options are keywords that allow you to supply the Linker with information about task memory requirements and references to other global symbols, libraries, and logical units.

Most of these options interest all system users. A few are of interest only to the MACRO Programmer. These options have been identified by the word (MACRO) in Table 4-1.

Options may be divided into four general classes. The identifying mnemonics and a brief description of each category are listed below:

1. Allocation options allow you to modify the task's memory allocation at execution time. (Alloc)
2. Storage-sharing options provide your task with access to shareable global areas. (Share)
3. Device-specifying options let you specify the number of units required by the task and allow you to assign physical devices to logical unit numbers. (Device)
4. Content-altering options permit you to define a global symbol and value. You can also use them to introduce patches in the task-image. (Alter)

Table 4-1 briefly describes each LINK command option, and also provides the interest range and option class for each option.

NOTE

TRAX restricts the use of MACRO to subroutines which do not require RMS file-handling facilities.

Unless noted in the table, all options can be used for high-level and MACRO tasks. The category to which the option belongs is also indicated in the table.

The options are then described in detailed fashion, by category, in Section 4.1.

Table 4-1
TRAX LINK Command Options

Option	Category	Meaning
ABSPAT	Alter	Allows you to declare absolute patch values. (MACRO)
ASG	Device	Allows you to assign physical devices to logical units.
COMMON	Share	Allows you to declare a task's intention to access a shared region of executable code.
EXTTSK	Alloc	Allows you to extend task memory allocation at task installation time.
GBLDEF	Alter	Allows you to declare a global symbol definition. (MACRO)
LIBR	Share	Allows you to declare a task's intention to access a shared library region.
UNITS	Device	Allows you to specify the maximum number of logical units required by the task.

4.1 ALLOCATION OPTIONS

This option directs the Linker to change the allocation of task memory.

4.1.1 EXTTSK (Extend Task Memory)

The EXTTSK option declares the amount of additional memory to be allocated to the task when RUN in a system-controlled partition.

The amount of memory available to the task is the sum of the task size plus the increment specified in the EXTTSK keyword (rounded up to the nearest 32-word boundary). If the task is Linked for a user-controlled partition, the allocation of task memory reverts to the partition size.

Syntax:

```
    EXTTSK = length
```

where:

length	is a decimal number specifying the increase in task memory allocation (in words).

4.2 STORAGE-SHARING OPTIONS

When you wish to access a shared region of memory, such as a global common area or a shared library or Object Time System, you can use two options: COMMON and LIBR. These options are of interest to all users of the system.

The COMMON option indicates that the shared region contains only data, while the LIBR option indicates a shared global region that contains only object code.

4.2.1 COMMON (System-Owned Common Block) LIBR (System-Owned Resident Library)

The identical options. The COMMON and LIBR options declare that the task is to access a system-owned shared global region. There is no default setting for either of these options; they must be specified by you.

Syntax:

```
COMMON = name:access-code

or

LIBR = name:access-code
```

where:

name	is the 1- to 6-character alphanumeric name specifying the library.
access-code	is the code RW (read-write) or the code RO (read-only) indicating the type of access the task requires.

4.3 DEVICE SPECIFICATION OPTIONS

The two device specification options are of interest to all users of the system. The UNITS option declares the maximum number of logical input-output units that the task can use. All units from 1 to the number specified are made available to the task. The ASG option declares the devices that are assigned to these units.

Using a logical unit number greater than this option will cause an error at task execution time.

To increase the number of units and assign devices to these units, enter the UNITS option first and then the ASG option. Because Linker processes the options as they are encountered, entering the options in the reverse order can produce an error message.

4.3.1 UNITS (Logical Unit Usage)

The UNITS option declares the maximum number of logical units that the task can use.

Syntax:

```
UNITS = max-units
```

where:

max-units	is a decimal integer in the range 0-250 specifying the maximum number of logical units. A device may be assigned up to a maximum of eight logical unit numbers.

The Linker assumes a default value of 6 UNITS.

4.3.2 ASG (Device Assignment)

The ASG option assigns the physical devices to their corresponding logical units.

Syntax:

```
     ASG = device-name:unit-num-1:unit-num-2:...:unit-num-8
```

where:

```
     device-name     is a 2-character alphabetic device  name  followed
                     by a 1- or 2-decimal unit number.

     unit-num-1      are  decimal integers indicating  the logical unit
     unit-num-2      numbers.
          .
          .
          .
     unit-num-8
```

The default logical units assignments are:

```
     ASG = SY0:1:2:3:4,TI0:5,CL0:6
```

4.4 STORAGE-ALTERING OPTIONS

Storage-altering options cause the Linker to modify the task image and are of interest only to the experienced MACRO programmer. The GBLDEF option declares a global symbol and value. The option ABSPAT allows you to insert a patch into the task image.

4.4.1 ABSPAT (Absolute Patch)

The ABSPAT option specifies a series of patches starting at the specified base address. A maximum of eight patch values may be supplied.

Enter the ABSPAT option in the following format:

Syntax:

```
     ABSPAT = seg-name:address:val-1:val-2:...:val-8
```

where:

seg-name	is the 1- to 6-character alphanumeric name of the segment.
address	is the octal address of the first patch. The address may be on a byte boundary; however, two bytes are always modified for each patch.
val-1	is an octal number in the range of 0 to 177777 to be assigned to address.
val-2	is an octal number in the range of 0 to 177777 to be assigned to address+2.
...	...
val-8	is an octal number in the range of 0 to 177777 to be assigned to address+16(octal).

default: none

NOTE

All ABSPAT patches must be within the segment memory limits or a fatal error is generated.

4.4.2 GBLDEF (Global Symbol Definition)

The GBLDEF option defines a global symbol. The symbol definition is considered absolute.

Syntax:

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

where:

symbol-name	is a 1- to 6-character alphanumeric name of the defined symbol.
symbol-value	is an octal number in the range of 0 to 177777 which is assigned to the defined symbol.

CHAPTER 5

OVERLAY CAPABILITY

The Linker provides the user with a means of reducing the memory and/or virtual address space requirements of a task -- tree-like overlay structures created with the aid of the Overlay Description Language (ODL). Overlay segments are specified to reside on disk.

5.1 OVERLAY DESCRIPTION

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

- a single root segment, which is always in memory, and
- any number of overlay segments, which reside on disk and share virtual address space and memory with one another.

A segment consists of a set of modules and p-sections. Segments that overlay each other must be logically independent; that is, the components of one segment cannot reference the components of a segment with which it shares virtual address space. In addition to the logical independence of the overlay segments, you must consider the general flow of control within the task.

The user must also consider the kind of overlay segment to have at a given position in the structure, and how to construct it. Dividing a task into disk-resident overlays saves physical space, but introduces the overhead activity of loading these segments each time they are needed, but not present in memory.

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

5.1.1 Disk-Resident Overlay Structures

Disk-resident overlays conserve memory by sharing it. 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 whenever either needs to be used.

OVERLAY CAPABILITY

The use of disk-resident overlays is shown in this section by an example -- task statement, which consists of four overlaid input files. Each input file consists of a single module having the same name as the file. The task is built by the command string shown in Chapter 2.

The user knows that the modules A, B, and C are logically independent. In this example:

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.

It is possible to define a disk-resident overlay structure in which A, B, and C are overlay segments that occupy the same storage area in 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.

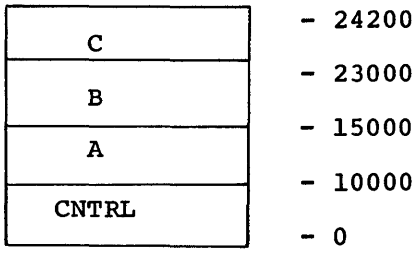
In this example, the loading of overlays occurs only four times during the execution of the task. Therefore, the user can reduce the memory requirements of the task without unduly increasing the overhead activity.

The effect of an overlay structure on the allocation of memory for the task is shown in the following paragraphs.

The lengths of the modules (expressed in octal) are:

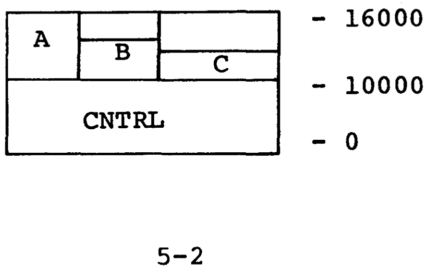
CNTRL	10000 bytes
A	6000 bytes
B	5000 bytes
C	1200 bytes

The memory allocation produced as a result of building the task as a single segment on a system with memory-mapping hardware is as follows:



The memory allocation for a single-segment task requires 24200 (octal) bytes.

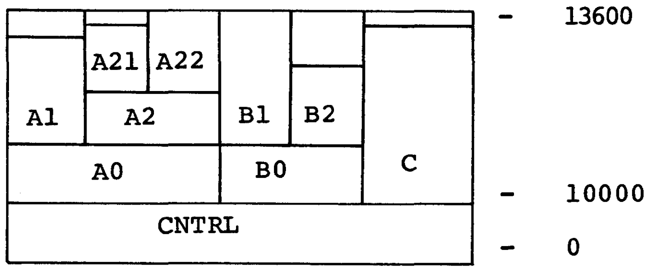
The memory allocation produced as a result of using the overlay capability and building a multi-segment task is as follows:



The multi-segment task requires 16000 (octal) bytes. In addition to the module storage, storage is required for overhead in handling the overlay structure. This overhead is described further on and illustrated in the example STATEMENT.

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

The memory allocation for the task produced by the additional overlays defined for A and B is shown in the following figure:



A vertical line can be drawn through the memory diagram to indicate a state of memory. In this diagram, the leftmost vertical line shows memory when CNTRL, A0, and A1 are loaded. The next vertical line shows memory when CNTRL, A0, and A1 are loaded. The next vertical line shows memory when CNTRL, A0, A2, and A21 are loaded, and so on.

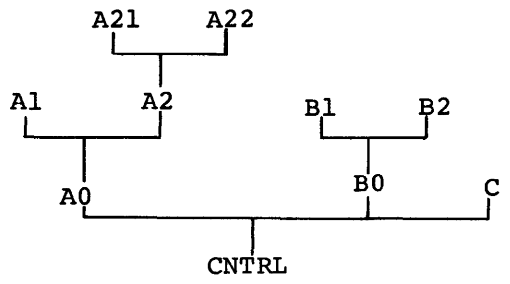
A horizontal line can be drawn through the memory diagram to indicate segments that share the same storage. The uppermost horizontal line shows A1, A21, A22, B1, B2, and C, all of which can use the same memory. The next horizontal line shows A1, A2, B1, B2, and C, and so on.

5.1.2 Overlay Tree

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

The Linker provides a language for representing an overlay structure consisting of one or more trees (described in Section 5.1.3).

The memory allocation for the previous example (in Section 5.1.1) can be represented by the single overlay tree shown below:



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 in the understanding of the overlay loading mechanism and the resolution of global symbols.

5.1.2.1 **Loading Mechanism** - Modules can call other modules that exist on the same path. The module CNTRL 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 method for loading overlays is described in Chapter 6.

5.1.2.2 **Resolution of Global Symbols in a Multi-segment Task** - In resolving global symbols for a multi-segment task, the Linker performs the same activities as it does for a single-segment task.

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

The following points, illustrated in the tree figure below, describe the two distinct cases of multiply-defined symbols, and ambiguously defined symbols.

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

In a multi-segment task:

- Two global symbols with the same name can be defined if they are on separate paths, and not referenced from a segment that is common to both.
- If a global symbol is defined more than once on separate paths, but referenced from a segment that is common to both, the symbol is ambiguously defined.
- If a global symbol is defined more than once on a single path, it is multiply-defined.

The procedure for resolving global symbols can be summarized as follows:

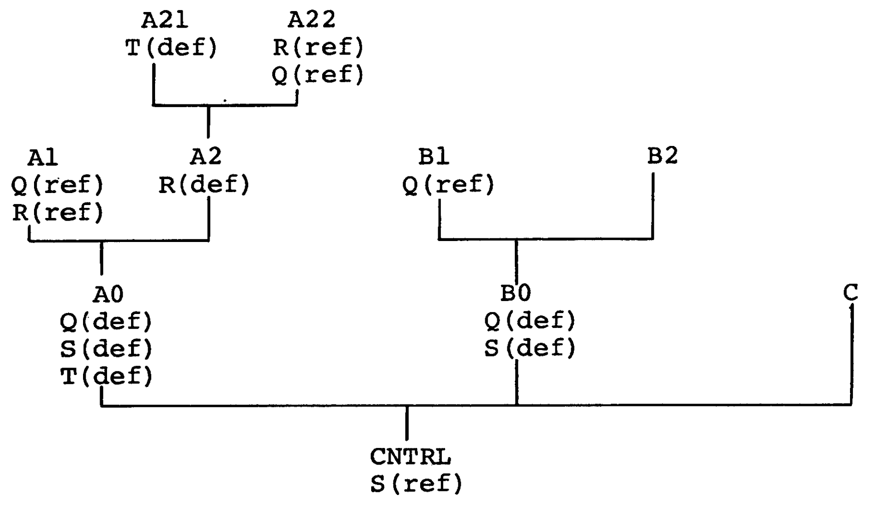
1. The Linker selects an overlay segment for processing.
2. Each module in the segment is scanned for global definitions and references.
3. If the symbol is a definition, the Linker searches all segments on paths that pass through the segment being processed, and looks for references that must be resolved.
4. If the symbol is a reference, the Linker performs the tree search as described in step 3, looking for an existing definition.
5. If the symbol is new, it is entered in a list of global symbols associated with the segment.

Overlay segments are selected for processing in an order corresponding to their distance from the root. That is, the Linker considers a segment farther away from the root, before processing an adjoining segment.

When a segment is being processed, the search for global symbols proceeds in the following order:

- the segment being processed
- all segments toward the root
- all segments away from the root
- all co-trees (see Section 6.1.4.1)

Example:



The following remarks apply to the use of the symbols Q, R, S, and T, shown in the diagram above:

Q The global symbol Q is defined in the segment A0 and in the segment B0. The reference to Q in segment A22 and the reference to Q 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.

R The global symbol R is defined in the 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.

S The global symbol S is defined in A0 and B0. References to S from 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. S is ambiguously defined.

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

5.1.2.3 **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 are scanned by the Linker in the first pass. If any undefined symbols remain, the Linker initiates a second pass over the structure in an attempt to resolve such symbols by searching the default object module library (normally SY0:[1,1]SYSLIB.OLB). Any undefined symbols remaining after the second pass are reported to the user.

When multiple tree structures (co-trees) are defined, as described in Section 5.1.4.1, 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 occur. To eliminate these conditions, the tree search on the second pass is restricted to:

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

When the current segment is the main root, the tree search is extended to all segments. The user can unconditionally extend the tree search to all segments by including the /FU (full) switch in the task image file specification.

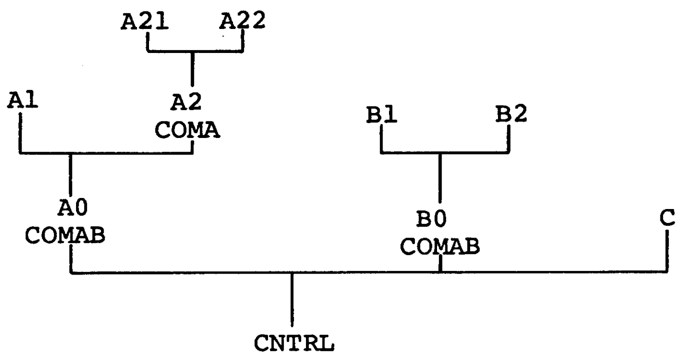
5.1.2.4 **Resolution of P-sections in a Multi-segment Task** - A p-section has an attribute that indicates whether the p-section is local (LCL) to the segment in which it is defined or is of global (GBL) extent.

Local p-sections with the same name can appear in any number of segments. Storage is allocated for each local p-section in the segment in which it is declared. Global p-sections that have the same name, however, must be resolved by the Linker.

When a global p-section is defined in several overlay segments along a common path, the Linker allocates all storage for the p-section in the overlay segment closest to the root.

If the programs A0 and B0 use a common block COMAB, however, the Linker allocates the storage 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.

The tree for the task TK1, including the allocation of the common blocks COMA and COMAB, is:



The allocation of p-sections can be specified by the user. If A0 and B0 need to share the contents of COMAB, the user can force the allocation of this p-section into the root segment by the use of the .PSECT directive, described in Section 5.1.3.4.

5.1.3 Overlay Description Language (ODL)

The Linker provides a language that allows the user to describe the overlay structure of a task. An overlay description is a text file consisting of a series of ODL directives, one directive per line. This file is entered in a Linker command line, and is identified as an ODL file by the presence of the /OVERLAY: switch (see Section 3.1.9) after the filename. If an overlay description text file is entered, it must be the only input file specified.

The format for an ODL line is:

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

A label is a necessary part of the .FCTR directive only (see Section 5.1.3.2).

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

If the name belongs to a file, a complete file specification can be given. Defaults for omitted parts of the file specification are as described in Chapters 2 and 3, except that the default device is always SY0, and the default UFD is always taken from the terminal UIC.

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

- The dot character (.) can only be used in a filename.
- Comments cannot appear on a line ending with a filename.

5.1.3.1 **.ROOT and .END Directives** - There must be one .ROOT directive and one .END directive. The .ROOT directive tells the Linker where to start building the tree, and the .END directive tells the Linker where the input ends.

The arguments of the .ROOT directive make use of three operators to express concatenation, overlaying, and memory residency. A pair of parentheses delimits a group of segments that start at the same virtual address. The number of nested parenthetical groups cannot exceed 16.

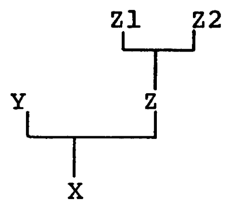
- The hyphen operator (-) indicates the concatenation of storage. For example, X-Y means that sufficient memory will be allocated to contain X and Y simultaneously. X and Y are allocated in sequence.
- The comma operator (,) appearing within parentheses indicates the overlaying of virtual memory. For example, Y,Z means that virtual memory can contain either Y or Z. If no exclamation point (!) precedes the left parenthesis, Y and Z also share physical memory.

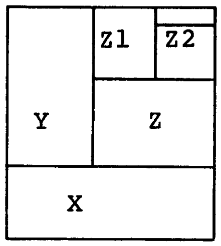
  The comma operator (,) is also used to define multiple tree structures (as described in Section 5.1.5).

Example:

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

These directives describe the following tree and its corresponding memory diagram:





To create the overlay description for the task described in Section 5.1.1, the user creates a file that contains the directives:

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

To build the task with that overlay structure, the user types:

```
>LINK/TASK/OVERLAY:STATEM
```

The switch /OVERLAY tells the Linker that there is only one input file, .ODL, and that this file contains an overlay description for the task.

5.1.3.2 **.FCTR Directive** - Because the tree that represents the overlay structure can be complex, the Overlay Description Language includes another directive, .FCTR, that allows the user to build large trees and represent them more clearly.

The .FCTR directive has a label to its left at the beginning of the line, that is pointed to by a reference in a .ROOT or another .FCTR statement. The .FCTR directive allows the user to extend the tree description beyond a single line. (There can be only one .ROOT directive.)

The decision to use the .FCTR directive is based on considerations of clarity. To simplify the tree given in the file TFIL, the user can introduce the .FCTR directive into the overlay description as follows:

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

The label BFCTR, is used in the .ROOT directive to designate the argument of the .FCTR directive, B0-(B1,B2). The resulting overlay description is easier to interpret than the original description. The tree consists of a root, CNTRL, and three main branches. Two of the main branches have sub-branches.

The .FCTR directive can be nested (to a level of 16). The user can 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
```

5.1.3.3 **.NAME Directive** - The .NAME directive allows the user to specify a name for a segment, and in so doing, to attach attributes to the segment. The name must be unique with respect to filenames, p-section names, .FCTR labels, and other segment names that are used in the overlay description.

The chief uses of this directive are:

1. to name uniquely a segment that is to be loaded through the manual load facility, and

2. to permit a segment that does not contain executable code, to be loaded through the autoload mechanism.

(Loading mechanisms are discussed in Chapter 6.)

The format of the .NAME directive is

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

where:

segname = a 1- to 6-character name composed from the Radix-50 character set, exclusive of the period (.); i.e., A-Z, 0-9, and $

[ ] denote optional attributes

attr = one of the following:

GBL	The name is entered in the segment's global symbol table. The GBL attribute makes it possible to load non-executable overlay segments by means of the autoload mechanism (see Chapter 6).
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 task image on disk need be reserved. If a NODSK attribute has been specified, an attempt to initialize a segment with data at task-build time results in a fatal error.
NOGBL	The name is not entered in the segment's global symbol table. If the GBL attribute is not present, NOGBL is assumed.
DSK	Disk storage is allocated to the named segment. If the NODSK attribute is not present, DSK is assumed.

The attributes described are not attached to a segment until the name is used in a .ROOT or .FCTR statement that defines an overlay segment. When multiple segment names are applied to a segment, the attributes of the last name given are in effect.

In the following modified tree for TK1, the user gives names to the three main branches, A0, B0, and C, by specifying them in the .NAME directive, and using them in the .ROOT directive. The default attributes NOGBL and DSK are in effect for BRNCH1 and BRNCH3, but BRNCH2 has the complementary attributes (GBL and NODSK) that will cause the name BRNCH2 to be entered into its segment's global symbol table, and the allocation of disk space for the segment to be suppressed. 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
```

(* is the autoload indicator; it is discussed in Chapter 6.)

The data overlay segment BRNCH2 is loaded by including the following statement in the user's program.

```
     CALL BRNCH2
```

This action is immediately followed by an automatic return to the next instruction in the program.

NOTE

In the absence of a unique name specification, the Linker establishes a segment name, using the first .PSECT, file, or library module name occurring in the segment.

5.1.3.4 **.PSECT Directive** - The .PSECT directive allows the placement of a global p-section in an overlay structure, to be specified directly. The name of the p-section (a 1- to 6-character name composed from the set A-Z, 0-9, and $) and its attributes are given in the .PSECT directive. This allows use of the name to indicate which segment the p-section will be allocated to.

5.1.3.5 **Indirect Files** - The Overlay Description Language processor can accept ODL text indirectly, if the text is included in a file specified in the proper format. If an @ is the first character in an ODL line, it instructs the processor to read text from the file specified immediately after the @. It accepts the ODL text from the file as input, at the point in the overlay description where the file is specified.

For example, if the file BIND.ODL contains

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

then this text can be replaced by a line beginning with @BIND, at the position where the text would have appeared:

```
                 Indirect                       Direct

                 .ROOT A-(B,C)                  .ROOT A-(B,C)
     C:          .FCTR C1-(C2,C3)    =     C:   .FCTR C1-(C2,C3)
     @BIND                                 B:   .FCTR B1-(B2,B3)
                 .END                           .END
```

Two levels of indirection are allowed.

5.1.4 Multiple-Tree Structures

The Linker allows the definition of more than one tree within the 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 Runtime System.

Except for this distinction, all overlay trees have identical characteristics -- a root segment that resides in memory, and usually 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 referenced 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.

The following paragraphs describe the procedure for specifying multiple trees in the Overlay Description Language, and illustrate the use of co-trees to produce the memory allocation best suited to the needs of the task.

5.1.4.1 **Defining a Multiple-Tree Structure** - Multiple-tree structures are specified within the Overlay Description Language by extending the function of the comma operator. As previously discussed, this operator, when included within parentheses, defines a pair of segments that share storage. The inclusion of the comma operator outside all parentheses delimits overlay trees. The first overlay tree thus defined is the main tree. Subsequent trees are co-trees.

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

In the example above, two overlay trees are specified: a main tree containing the root segment X0 and three overlay segments, and 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 Runtime System.

A co-tree must have a root segment to establish linkage with its own overlay segments. Co-tree root segments need not contain code or data. A segment of this type, called a null segment, can be created by means of the .NAME directive. The previous example is modified, as shown below, to move file Y0.OBJ to the root, and include a null segment.

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

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

5.1.4.2 **Multiple-Tree Example** - The following example illustrates the use of multiple trees to reduce the size of the task.

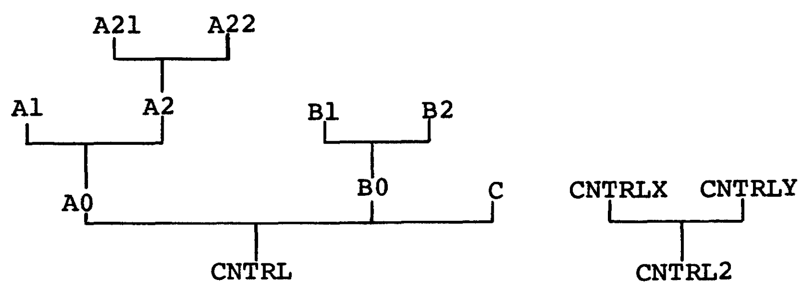
In the example, the root segment CNTRL consists of a small dispatching routine and two modules, CNTRLX and CNTRLY. CNTRLX and CNTRLY are logically independent of each other, approximately equal in length, and must be accessed from modules on all the paths of the main tree.

The user can define a co-tree for CNTRLX and CNTRLY and reduce the amount of storage required by the task. The overlay description in TFIL is modified as follows:

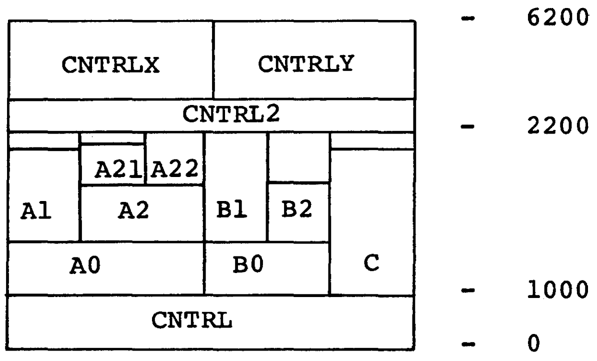
```
     .NAME CNTRL2
     .ROOT CNTRL-(AFCTR,BFCTR,C),CNTRL2-(CNTRLX,CNTRLY)
        .
        .
        .

     .END
```

The co-tree is defined in the .ROOT directive, by the placement of the comma operator outside all parentheses (immediately before CNTRL2). A co-tree must have a root segment to establish linkage with the overlay segments within the co-tree. When no code or data logically belongs in the root, the .NAME directive can be used to create a null root segment.

The tree for the task TK1 now is:



The corresponding memory diagram is:



The specification of the co-tree decreases the storage allocation by 4000 bytes. CNTRLX and CNTRLY can still be accessed by all modules in the main tree. The only requirement imposed by the introduction of the co-tree is the logical independence of CNTRLX and CNTRLY.

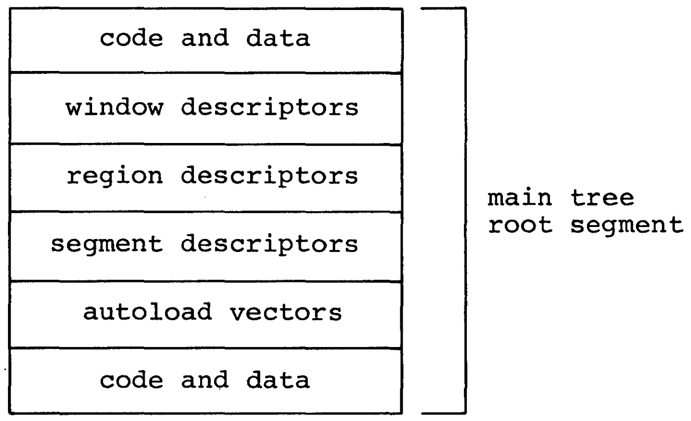
Any number of co-trees can be defined. Additional co-trees can access all modules in the main tree and other co-trees.

5.1.5 **Overlay Core Image**

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

The root segment of the main tree contains modules that are resident in memory throughout the execution of the task, along with the following data required by the overlay loading routines:

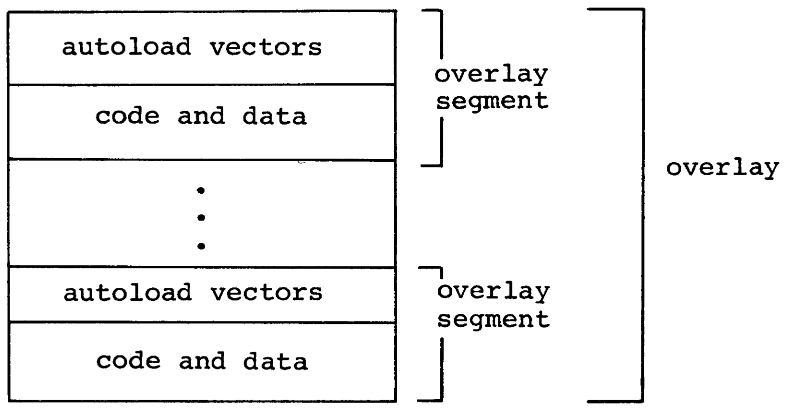
- Segment tables
- Autoload vectors



The segment table contains 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.

Autoload vectors appear in every segment that calls modules in another segment located farther away from the root of the tree. Autoload vectors are described in the discussion of loading mechanisms in Chapter 6.

The main tree overlay region consists of memory allocated for the overlay segments of the main tree. The overlays are read into this area of memory as they are needed.



The co-tree overlay region consists of memory allocated for co-tree overlay segments.

The co-tree root segment contains modules that, once loaded, must remain resident in memory.

5.1.6 Overlaying Programs Written in a Higher-level Language

Programs that are written in a higher-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:

- Linker throughput may be drastically reduced because of the number of library references in each overlay segment.
- Library references from the default object module library, which 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:

1. Linker throughput can be increased by linking commonly used library routines into the main root segment.
2. Ambiguous and multiple definitions, and cross-tree references can be eliminated by using the /NOFULL switch (the default) to restrict the scope of the default library search.

The user 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.

The appropriate user's guide for the language should be consulted for other ways to reduce task size.

5.1.7 Defining the ODL File

The user constructs a file, STATE, of ODL directives to represent the tree for STATE, using the COBOL Merge Utility of the BASIC-PLUS-2 Build Command.

(The * in the ODL description is the autoload indicator; it is described in Chapter 6.)

5.1.8 Building the Task

The names of the input files are specified by a single filename that designates the file containing the overlay description:

The reader should note that the ODL file specification automatically terminates command input, and the Linker automatically prompts for options.

5.2 SUMMARY OF THE OVERLAY DESCRIPTION LANGUAGE

1. An overlay structure consists of one or more trees. Each tree contains at least one segment. A segment is a set of modules and p-sections that can be loaded by a single disk access.

   A tree can have only one root segment, but it can have any number of overlay segments.

2. An overlay description is a text file consisting of a series of ODL directives, one directive per line. This file is entered in a Linker command line, and is identified as an ODL file by the presence of the /OVERLAY switch after the filename. If an overlay description text file is entered, it must be the only input file specified.

3. The overlay description language provides five directives for specifying the tree representation of the overlay structure, namely:

   .ROOT
   .END
   .PSECT
   .FCTR
   .NAME

These directives can appear in any order in the overlay description, subject to the following restrictions:

a. There can be only one .ROOT and one .END directive.

b. The .END directive must be the last directive because it terminates input.

4. The tree structure is defined by the operator's hyphen (-), comma (,), and by the use of parentheses.

   - The hyphen operator (-) indicates that its arguments are to be concatenated and thus 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 parentheses group segments that begin at the same point in memory.

     For example,

     ```
     .ROOT A-B-(C,D-(E,F))
     ```

     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 location in memory; and similarly, the inner parentheses indicate that E and F start at their own shared address.

5. The .ROOT directive defines the overlay structure. The arguments of the .ROOT directive are one or more of the following:

   - File specifications as described in Section 2.9

   - Factor labels

   - Segment names

   - P-section names

6. The .END directive terminates input.

7. The .FCTR directive provides a means for replacing text by a symbolic reference (the factor label). This replacement is useful for two reasons:

   a. The .FCTR directive extends the text of the .ROOT directive to more than one line and thus allows complex trees to be represented.

   b. The .FCTR directive allows the overlay description to be written in a form that makes the structure of the tree more apparent.

For example:

```
          .ROOT A-(B-(C,D),E-(F,G),H)
          .END
```

can be expressed, using the .FCTR directive, as follows:

```
          .ROOT A-(F1,F2,H)
 F1:      .FCTR B-(C,D)
 F2:      .FCTR E-(F,G)
          .END
```

The second representation makes it clear that the tree has three main branches.

8. The .PSECT directive provides a means for directly specifying the segment in which a p-section is placed. It accepts the name of the p-section and its attributes. For example:

   ```
   .PSECT ALPHA,CON,GBL,RW,I,REL
   ```

   ALPHA is the p-section name and the remaining arguments are attributes. P-section attributes are described in Chapter 5.

   The p-section name (composed from the characters A-Z, 0-9, and $) must appear first in the .PSECT directive, but the attributes can appear in any order, or be omitted. If an attribute is omitted, a default condition is assumed. The defaults for p-section attributes are RW, I, LCL, REL, and CON.

   In the example above, therefore, it is necessary to specify only those attributes that do not correspond to the defaults: .PSECT ALPHA,GBL

9. The .NAME directive provides a means for designating a segment name for use in the overlay description, and for specifying segment attributes. This directive is useful for creating a null segment, naming a segment that is to be loaded manually, or naming a non-executable segment that is to be autoloadable. If the .NAME directive is not used, the name of the first file, p-section, or library module in the segment is used to identify the segment.

   The .NAME directive creates a segment name as follows:

   ```
   .NAME segname,attr,attr
   ```

   where segname is the designated name (composed from the character set A-Z, 0-9, and $), and attr is an optional attribute taken from the following: GBL, NODSK, NOGBL, DSK. The defaults are NOGBL and DSK. The defined name must be unique with respect to the names of p-sections, segments, files, and factor labels.

10. A co-tree can be defined 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.

5.3 OVERLAYING BASIC-PLUS-2 PROGRAMS

Programs can be logically broken into sections (subprograms) that are compiled and input to the task builder as object modules. These sections can then be overlaid, which allows you to create programs that would otherwise exceed the maximum in-core program limits.

A program requires overlays when in-core program needs exceed the system default maximum job size.

5.3.1 Overlays

When you use the Linker's overlay facility, you can specify only one input file in the command line. This input file describes the overlay structure, the location of program sections, and the loading procedures.

Overlay structure is defined by means of the Overlay Description Language (ODL). This structure is analogous to a tree, with the main program being the root and the program sections representing the branches. The ODL directives are contained in a user-created file that is specified in the command string. The OVERLAY qualifier must be included in the LINK command line to identify the file as an ODL file.

Note that you can use the output produced by the BUILD command to create overlaid program segments. That is, the BUILD command produces a command file that contains all of the command input required by the Linker and an ODL file. You can examine the ODL file and use the Editor to modify its content before the command file is input to the Linker.

At a minimum, the overlay description must contain a .ROOT and an .END directive. The .ROOT directive declares the overlay tree structure and the .END directive signifies the end of input. Note that an overlay description can contain only one .ROOT directive, which limits the tree structure declaration to a single line of input. The Linker truncates an input line that exceeds 80 characters, but this limitation should not affect the majority of tree structure declarations because you can use the .FCTR directive to build large trees and extend the description beyond a single line. For a description of the .FCTR directive, refer to Section 5.2, item 7.

Suppose, for example, you have a program consisting of a main program and calls to three external subprograms. One subprogram does pre-processing of data, the second does primary processing, and the third does post-processing. The main program and three subprograms are compiled as object modules named MAIN.OBJ, PRE.OBJ, PROC.OBJ, and POST.OBJ, respectively.

You can build an overlay structure that causes the main program to be resident in memory and the three subprograms to share the same area of memory. The ODL directive that creates this structure has the form:

```
.ROOT MAIN-*(PRE,PROC,POST)
.END
```

In this example:

.ROOT MAIN	defines the root of the overlay structure as the object module named MAIN.OBJ.

-	the hyphen indicates that the following modules are concatenated to the preceding module.
*	the asterisk indicates that modules are loaded automatically (autoload). The asterisk must precede every module. If all modules within parentheses are to be autoloaded, a single asterisk preceding the parentheses is used.
( )	parentheses group the descriptions of overlay sections.
PRE,PROC,POST	commas separating object modules contained in parentheses indicate that the named modules occupy the same virtual memory area.

Figure 5-1 is a graphic illustration of the overlay structure specified above as it would appear in memory.

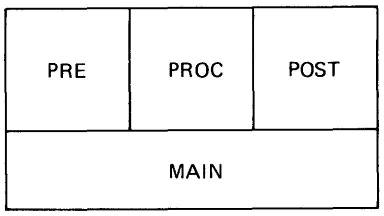


Figure 5-1 Overlay Structure

To create an overlaid program by means of the BUILD command, you edit the ODL file that is generated. That is, a BUILD command produces a command file (file type .CMD) and an overlay description language file (file type .ODL). The ODL file must be edited to reflect the desired overlay structure prior to input to the Linker.

For example, if the object modules described in Figure 5-1 (i.e., MAIN, PRE, PROC, and POST) are used as arguments in the BUILD command:

```
BUILD MAIN,PRE,PROC,POST
```

the result is a command file (MAIN.CMD) that is invoked by the command line:

```
LINK/BASIC MAIN
```

MAIN.CMD corresponds to the following LINK command string:

```
LINK/TASK:MAIN/CHECKPOINT/MAP:MAIN/OVERLAY:MAIN/OPTIONS:BASOPT
```

where the options command file BASOPT consists of the following option lines:

```
LIBR=BASIC2:RO
UNITS = 14
ASG=SY:5:6:7:8:9:10:11:12
ASG=TI:13
```

The BUILD command also generates an overlay description file (MAIN.ODL) that appears as follows:

```
                .ROOT  USER
     USER:      .FCTR  MAIN-PRE-PROC-POST-LIBR
     LIBR:      .FCTR  [1,1]BASIC2/LB
                .END
```

You can edit this ODL file to create an overlay as follows:

```
                .ROOT  USER
     USER:      .FCTR  MAIN-LIBR-*(PRE-LIBR,PROC-LIBR,POST-LIBR)
     LIBR:      .FCTR  [1,1]BASIC2/LB
                .END
```

The overlay structure used in this example duplicates that shown in Figure 5-1. Note that each branch of the structure must be associated with the library. This procedure ensures that the correct routines are linked at run time.

The path of an overlay structure is from the root of the structure, along a series of branches, to the outermost section. The root section can call any overlay section. However, a subprogram in an overlay section can call another overlay section only if they share a common path. Therefore, in the previous example, MAIN can call PRE, PROC, and POST, but the three subprograms cannot call each other.

The concept of paths is better illustrated with a tree diagram. For example:

```
     .ROOT A-B-*(C,D-(E,F,G))
     .END
```

where A and B are two object modules representing the root section. C and D are the branches of A and B. E, F, and G are branches of D. A tree diagram of this structure appears in Figure 5-2.

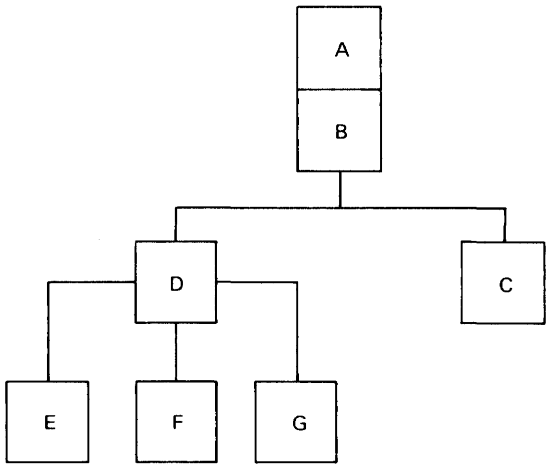


Figure 5-2 Overlay Path

The paths of this structure are: A-B-C, A-B-D-E, A-B-D-F, and A-B-D-G. Within this structure:

1. A and B can call all the sections.
2. D can call E, F, and G.
3. C and D cannot directly call each other.
4. C cannot call E, F, and G.
5. E, F, and G cannot call each other.

Note that if A calls C, C in turn can call B. However, if B simultaneously calls D and then attempts to return to C, an error occurs. The error is due to B returning to an overlaid segment, i.e., D overlays C.

```
    OLD NONAME

    Basic 2

    COM

    Basic 2

    BUILD NONAME/IND

    Basic2

    EXIT

    LINK/BASIC NONAME.CMD
```

In this command series, BUILD is used to create a command file (NONAME.CMD) composed of a previously compiled object module. The command file contains all the libraries and options that are required input to the Linker as well as the BASIC switch (/IND) that enables the use of RMS indexed I/O. The command file is used as input to the Linker and results in a map file and an executable task (NONAME.MAP and NONAME.TSK).

Note that no additional qualifiers or options can be associated with the command file input specification. For example:

```
    LINK/NOMAP/BASIC NONAME.CMD
```

is illegal.

The use of an RMS switch (/VIR, /SEQ, /REL, or /IND) causes the BUILD command to change the generated. .ODL file as required for RMS I/O. These changes are made automatically when the appropriate switch is appended to the BUILD command. Consider the following example of NONAME.ODL:

```
               .ROOT  BIROT4-USER,RMS
    USER:      .FCTR  NONAME-LIBR
    LIBR:      .FCTR  [1,1]BASIC2/LB
    RMS:       .FCTR  BIO047
    @SY:[1,1]         BASIC4
               .END
```

5.4 USING OVERLAYS WITH TRAX COBOL

5.4.1 Standard ODL File

The standard ODL file generated by the TRAX COBOL compiler consists of a header and a body. The header contains information that is required to merge one or more ODL files. The body contains ODL directives that describe the object program.

5.4.2 ODL File Header

The ODL file header consists of a sequence of comment lines. Two are required in every ODL file, and others are supplied as needed. The required comment lines are:

```
;COBOBJ=XXXXXX.OBJ
;COBKER=KK
```

Where:

XXXXXX.OBJ	is the name of the object module being described
KK	is the kernel that was used to generate the PSECT names for the COBOL program.

The following comment lines are supplied as needed:

;COBMAIN	This comment line is supplied if the program being described is a main program. The absence of this line means that the ODL file was generated for a COBOL subprogram.
;RMSSEQ=CIOOXY	This comment line is specified if the program requires RMS I/O support. One or more lines may be supplied. X and Y represent integer codes that respectively specify the file organization and operational support required for that organization. File organization is specified by the following codes:

CODE	ORGANIZATION
1	sequential
2	relative
3	indexed

5.4.3 ODL File Body

The ODL file body describes the overlay structure of the COBOL program. The body contains the following ODL directive types:

1. .PSECT defines the name of the code PSECT and makes it known to the TRAX Linker.

2. .NAME defines the name to be assigned to the overlay segment by the Trax Linker.

3.	.FCTR	describes the contents of the segments.
4.	.ROOT	defines the root.
5.	.END	informs the TRAX Linker that the end of the ODL file has been reached.
6.	;comments	contains comment entries.

The .ROOT and .END directives are not supplied by the COBOL compiler. They are inserted into the ODL file generated by the Merge Utility. If you are generating a stand alone ODL file, these directives must be supplied by you. If the ODL file you are generating is to be used as input to the Merge Utility, leave these directives out.

Within a compiler-generated ODL file, the directives .PSECT, .NAME, and .FCTR are generated around the PSECT kernel. If the PSECT name kernel for a given program is KK, the format of the names generated in the ODL file is:

Entity	Format of Name
.PSECT	$KKMMM
.NAME	KK$MMM
.FCTR	KKMMM$

Each .PSECT defined in the ODL file begins with a $ followed by the two-character kernel ($KK). Each .NAME directive begins with the two-character kernel followed by $ (KK$). Finally, each .FCTR directive begins with the two-character kernel and ends with a $ (KKMMM$).

COMPILER-GENERATED ODL FOR COBOL PSECTS

The following sections discuss the ODL directives generated for different types of overlay requirements. The characters NNN, when used in examples, refer to the three character suffix generated by the compiler for each PSECT. The characters KK refer to the kernel characters that make the PSECT unique to a particular compilation.

5.4.4 ODL Generated for Overlays Containing Only One PSECT

For overlays containing only one PSECT, the following lines are generated:

```
        .PSECT    $KKNNN,GBL,RW,CON,I

        .NAME     KK$NNN,GBL

KKNNN$  .FCTR     *KK$NNN-$KKNNN
```

5.4.5 ODL Generated for Overlays Containing More Than One PSECT

For each overlay that contains more than one PSECT, a .PSECT directive is generated for each PSECT in the overlay. These .PSECT directives are followed by a .NAME and .FCTR directive. Consider the following example.

Example

Two PSECTs, $AA001 and $AA002, are to be placed in the same overlay. The segment-number assigned to the PSECTs is 20. The following ODL directives are generated:

```
                ;DEFINE      PSECT $AA001

                .PSECT       $AA001,GBL,RW,CON,I

                ;DEFINE      PSECT $AA002

                .PSECT       $AA002,GBL,RW,CON,I

                ;DEFINE THE OVERLAY NAME

                .NAME        AA$020,GBL

                ;DEFINE OVERLAY CONTENTS

AA020$:         .FCTR        *AA$020-$AA001-$AA002
```

ODL Generated for All Overlayable PSECTS

All .FCTR directives that describe the overlayable PSECTs must be collapsed into one final .FCTR directive. This directive describes the entire overlayable portion of the object code. The name associated with this .FCTR directive is derived from the two-character kernel assigned to the PSECTs. If the kernel is KK, then the name of the .FCTR directive that describes the entire overlayable part of the object code is KKOVR$.

The following example shows how the KKOVR$ factor is developed for various overlay configurations:

Example 1: All Code Psects Overlay One Another

```
                .PSECT       $AA001,GBL,RW,CON,I
                .NAME        AA$001,GBL
AA001:          .FCTR        *AA$001-$AA001
                ;
                .PSECT       $AA002,GBL,RW,CON,I
                .NAME        AA$002,GBL
AA002$:         .FCTR        *AA$002-$AA002
                ;
                .PSECT       $AA003,GBL,RW,CON,I
                .NAME        AA$003,GBL
AA003$:         .FCTR        *AA$003-$AA003
                ;
                .PSECT       $AA004,GBL,RW,CON,I
                .NAME        AA$004,GBL
AA004$:         .FCTR        *AA$004-$AA004
                ;
                .PSECT       $AA005,GBL,RW,CON,I
                .NAME        AA$005,GBL
AA005$:         .FCTR        *AA$005-$AA005
                ;IN THIS EXAMPLE, ALL PSECTS OVERLAY
                :ONE ANOTHER.
AAOVR$:         .FCTR        (AA001$,AA002$,AA003$,AA004$,AA004$,AA005$)
```

Example 2: Two Code Psects Are in the Same Overlay

```
            .PSECT      $AA001,GBL,RW,CON,I
            ;
            .PSECT      $AA002,GBL,RW,CON,I
            ;
            .NAME       AA$001,GBL
AA001$:     .FCTR       *AA$001-$AA001-$AA002
            ;
            .PSECT      $AA003,GBL,RW,CON,I
            .NAME       AA$003,GBL
AA003$:     .FCTR       *AA$003-$AA003
            ;
            .PSECT      $AA004,GBL,RW,CON,I
            .NAME       AA$004,GBL
AA004$:     .FCTR       *AA$004-$AA004
            ;
            .PSECT      $AA005,GBL,RW,CON,I
            .NAME       AA$005,GBL
AA005$:     .FCTR       *AA$005-$AA005
            ;
AAOVR$:     .FCTR       AA001$,AA003$,AA004$,AA005$
```

Example 3: Two Occurrences of Two Psects in the Same Overlay

```
            ;IN THIS EXAMPLE, PSECTS $AA001 AND $AA002
            ;ARE IN THE SAME OVERLAY.  PSECTS $AA003
            ;AND $AA004 ARE IN THE SAME OVERLAY.
            ;PSECT $AA005 IS IN AN OVERLAY ALL BY ITSELF
            ;
            ;PSECT      $AA001,GBL,RW,CON,I
            ;
            ;PSECT      $AA002,GBL,RW,CON,I
            ;
            .NAME       AA$001,GBL
AA001$:     .FCTR       *AA$001-$AA001-$AA002
            ;
            ;PSECT      $AA003,GBL,RW,CON,I
            ;
            .PSECT      $AA004,GBL,RW,CON,I
            ;
            .NAME       AA$003,GBL
AA003$:     .FCTR       *AA$003-$AA003-$AA004
            ;
            .PSECT      $AA005,GBL,RW,CON,I
            .NAME       AA$005,GBL
AA005$:     .FCTR       *AA$005-$AA005
            ;
AAOVR$:     .FCTR       AA001$,AA003$,AA005$
```

5.4.6 **Merging Standard ODL Files**

To develop an ODL file for a task composed of more than one COBOL object program, it is necessary to merge the ODL files for each individual object program into a single ODL file that describes the overlay requirements for the task.

All of the ODL files to be merged are partial ODL files. That is, none of these ODL files can be submitted directly to the Linker to link a task, because none of the compiler-generated ODL files contain a .ROOT directive. The .ROOT directive that describes the task is supplied by the Merge Utility.

Merging COBOL compiler-generated ODL files is accomplished by executing the ODL merge utility. (See TRAX COBOL User's Guide Section 2.6, Using the ODL Merge Utility.)

5.4.7 Including Non-COBOL Programs in a Task

To use the Merge Utility to include a non-COBOL object module in a task image, you must:

1. Create a standard COBOL ODL file (use the DEC editor)
2. Specify this ODL file as input to the ODL Merge Utility.

5.4.8 Creating a Standard COBOL ODL File

A standard COBOL ODL file for a non-COBOL object module contains one or two directive lines:

1. Object Program ID Line - This line is required. It identifies the object module to be included in the task image. The format of this line is:

   ```
   ;COBOBJ=XXXXXX.OBJ
   ```

   Where XXXXXX.OBJ is the name of the object module to be included in the task image.

2. Main Program ID Line - This line is present only for non-COBOL object modules that are main programs as opposed to being subprograms. The format of the line is:

   ```
   ;COBMAIN
   ```

For each invocation of the COBOL ODL Merge Utility, one and only one main program ODL file can be specified. If no main program ODL file is specified, the Merge Utility continues to request more input until a main program ODL file is specified. If more than one main program ODL file is specified, all but the first is rejected, and appropriate diagnostic error messages are issued. Consider the following examples.

Example 1

MACRO program START.OBJ is a main program in a task consisting of a main program and several subprograms. The ODL file to be hand-generated is:

```
;COBOBJ=START.OBJ
;COBMAIN
```

Example 2

Macro subprogram SUBX.OBJ is to be part of a task image that consists of several COBOL subprograms and a COBOL main program. The ODL file to be hand-generated is:

```
;COBOBJ=SUBX.OBJ
```

5.4.9 Rearranging a Compiler-generated ODL File

The ODL file generated by the compiler can be rearranged to modify the overlay structure of a task. If the ODL file describes a task that has overlayable segments, one or more of these segments can be converted into non-overlayable segments by:

1. Modifying the compiler-generated ODL file.
2. Specifying a one-line Linker option at Link time for each segment made non-overlayable.

5.4.10 Modifying the Compiler-generated ODL File

Modifying the compiler-generated ODL file requires the following steps:

1. Each overlayable segment is named in the ODL file by an ODL.NAME directive. This .NAME directive must be removed.
2. Each name appearing in a .NAME directive is marked with an * and placed as the first element of a .FCTR directive. For each .NAME directive removed by step 1, this .FCTR directive must be removed.
3. All references to the name of the .FCTR directive removed in step 2 must be removed from the ODL file.
4. All PSECTs referenced in the .FCTR directive that was removed in step 3, must be removed from the ODL file.

Example

The task image contains three overlayable segments, C$$010, C$$015, and C$$020. Segment C$$020 is to be forced into the root. Figure 5-3 contains a listing of the merged ODL file.

```
;MERGED ODL FILE CREATED ON 26-JAN-77 AT 10:50:00
;COBOL STANDARD ODL FILE GENERATED ON: 26-JAN-77  10:48:37
;COBOBJ=TEST1.OBJ
;COBKER=C$
;COBMAIN
         .NAME C$$010,GBL
         .PSECT $C$003,GBL,I,RW,CON
C$010$:  .FCTR *C$$010-$C$003
         .NAME C$$015,GBL
         .PSECT $C$004,GBL,I,RW,CON
C$015$:  .FCTR *C$$015-$C$004
         .NAME C$$020,GBL
         .PSECT $C$005,GBL,I,RW,CON
C$020$:  .FCTR *C$$020-$C$005
C$OVR$:  .FCTR         C$010$,C$015$,C$020$
CBOBJ$:  .FCTR TEST1.OBJ
CBOVR$:  .FCTR C$OVR$
CBOTS$:  .FCTR [320,13]COBLIB/LB
RMS$:    .FCTR  [1,1]RMSLIB/LB
OBJRT$:  .FCTR CBOBJ$-CBOTS$-RMS$
         .ROOT OBJRT$-(CBOVR$)
         .END
```

Figure 5-3 Merged ODL File Listing

To force segment C$$020 into the root, the merged ODL file must be modified as follows:

1. The .NAME directive referencing C$$020 must be removed.
2. The .FCTR directive containing *C$$020 must be removed.
3. All references to the PSECTs in the removed .FCTR directive must be removed.

Figure 5-4 contains the ODL listing after the modifications have been made.

```
;MERGED ODL FILE CREATED ON 26-JAN-77 AT 10:55:22
;COBOL STANDARD ODL FILE GENERATED ON: 26-JAN-77  10:48:37
;COBOBJ=TEST1.OBJ
;COBKER=C$
;COBMAIN
        .NAME C$$010,GBL
        .PSECT $C$003,GBL,I,RW,CON
C$010$:  .FCTR *C$$010-$C$003
        .NAME C$$015,GBL
        .PSECT $C$004,GBL,I,RW,CON
C$015$:  .FCTR *C$$015-$C$004
C$OVR$:  .FCTR         C$010$,C$015$
CBOBJ$:  .FCTR TEST1.OBJ
CBOVR$:  .FCTR C$OVR$
CBOTS$:  .FCTR [1,1]COBLIB/LB
RMS$:    .FCTR  [1,1]RMSLIB/LB
OBJRT$:  .FCTR CBOBJ$-CBOTS$-RMS$
        .ROOT OBJRT$-(CBOVR$)
        .END
```

Figure 5-4 Modified ODL File

CHAPTER 6

LOADING MECHANISMS

The method for loading disk-resident overlays is called:

Autoload	in which the Overlay Runtime System is automatically called upon to load those segments that are marked by the user.

In the autoload method, loading and error recovery are handled by the Overlay Runtime System. Overlays are automatically loaded by being referenced through a transfer-of-control instruction (CALL, JMP, or JSR). No explicit calls to the Overlay Runtime System are needed.

Provision must be made for loading the overlay segments of the main tree, and the root segments and overlay segments of the co-trees. Once loaded, the root segment of a co-tree remains in memory.

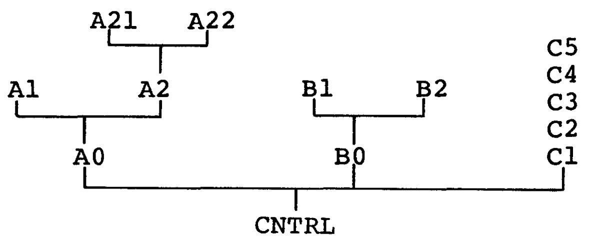
6.1 AUTOLOAD

To use the autoload method, the user places the autoload indicator, *, in the ODL description of the task, at the points indicating where loading must take place. The execution of a transfer-of-control instruction to an autoloadable segment up-tree (farther away from the root) automatically initiates the autoload process.

6.1.1 Autoload Indicator

The autoload indicator, *, marks as autoloadable the segment or other task element (as defined below). If the autoload indicator is applied to an ODL statement enclosed in parentheses, then every task element named within the parentheses is marked as autoloadable. Applying the autoload indicator at the outermost parenthesis level of the ODL tree description marks every module in the overlay segments as autoloadable.

If, in the TKl example of Chapter 6, segment C consisted of a set of modules Cl, C2, C3, C4, and C5, the tree diagram would be:



Placing the autoload indicator at the outermost parenthesis level, 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 the task with this provision is:

```
        .ROOT CNTRL-*(AFCTR,BCTR,CFCTR)
AFCTR:  .FCTR A0-(Al,A2-(A21,A22))
BFCTR:  .FCTR B0-(Bl,B2)
CFCTR:  .FCTR Cl-C2-C3-C4-C5
        .END
```

To ensure that all modules of a co-tree are properly loaded, the user must mark its root segment (CNTRL2) as well as its outermost parenthesis level as follows:

```
.ROOT CNTRL-*(AFCTR,BFTCR,CFCTR),*CNTRL2-*(CNTRLX,CNTRLY)
   .
   .
   .
```

The example above assumes that one or more modules containing executable code reside in CNTRL2.

The autoload indicator can be applied to the following elements:

- Filenames - 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. This includes elements within any nested parentheses.
- P-section names - to make the p-section autoloadable. The p-section must have the I (instruction) 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 of the .FCTR are autoloadable if they are enclosed in parentheses.

In the following example, the user introduces two .PSECT directives and a .NAME directive into the ODL description for TK1, and then applies autoload indicators in the following way:

```
               .ROOT CNTRL-(*AFCTR,*BFCTR,*CFCTR)
     AFCTR:    .FCTR A0-*ASUB1-ASUB2-*(A1,A2-(A21,A22))
     BFCTR:    .FCTR (B0-(B1,B2))
     CFCTR:    .FCTR CNAM-C1-C2-C3-C4-C5
               .NAME CNAM,GBL
               .PSECT ASUB1,I,GBL,OVR
               .PSECT ASUB2,I,GBL,OVR
               .END
```

The interpretation for each autoload indicator in the overlay description is as follows:

(*AFCTR,*BFCTR,*CFCTR)

The autoload indicator is applied to each factor name.

- *AFCTR=*A0
- *BFCTR=*(B0-(B1-B2))
- *CFCTR=*CNAM

CNAM, however, is an element defined by a .NAME directive; therefore, all the components of the segment to which the name applies are made autoloadable; that is, C1, C2, C3, C4, and C5.

*ASUB1 The autoload indicator is applied to the name of a p-section having the I attribute, so the p-section ASUB1 is made autoloadable.

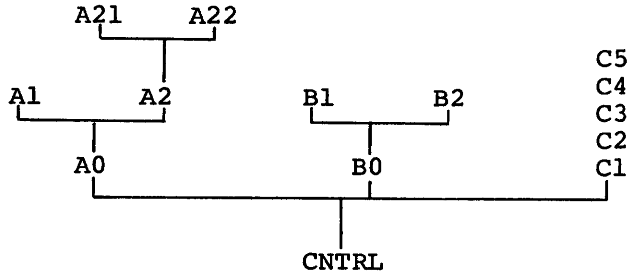
*(A1,A2-(A21,A22)) The autoload indicator is applied to a portion of the ODL description enclosed in parentheses, so every element within the parentheses is made autoloadable (that is, files A1, A2, A21, and A22).

The net effect of this ODL description is to make every element except file ASUB2 autoloadable.

6.1.2 Path-Loading

Autoload uses the technique of 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 be resident in physical memory and will 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.

Example:



In the example above, if CNTRL calls A2, then all the modules between the calling module CNTRL and the called module A2 are loaded. In this case, modules A0 and A2 are loaded.

The Overlay Runtime System keeps a record of the segments that are loaded and mapped, and issues disk-load requests only for those segments 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.

6.1.3 Autoload Vectors

When the Linker sees a reference to a global symbol in an autoloadable segment up-tree, it generates an autoload vector for the referenced global symbol. The reference is changed to a definition that points to an autoload vector entry. The autoload vector has the following format:

JSR PC,sub
$AUTO
SEGMENT DESCRIPTOR ADDR.
ENTRY POINT ADDRESS

A transfer-of-control instruction to the global symbol executes the call to the autoload routine, $AUTO.

An exception to the procedure for generating autoload vectors is made in the case of a p-section with the D (data) attribute. References from a segment to a global symbol up-tree in a p-section with the D attribute are resolved directly.

Because the Linker can obtain no information about the flow of control within the task, it often generates more autoload vectors than are necessary. The user, however, can apply knowledge of the flow of task control and knowledge of path-loading to determine the placement of autoload indicators. By placing the autoload indicators only at the points where loading is actually required, the user 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 the autoload indicator is placed at the outermost parenthesis level, thirteen autoload vectors are generated for this task; however, since A2 and A0 are loaded by path-loading to A21, the autoload vectors for A2 and A0 are unnecessary. Moreover, the call to C1 loads the segment that contains C2, C3, C4, and C5; therefore, autoload vectors for C2 through C5 are unnecessary.

The user eliminates the unnecessary autoload vectors by placing the autoload indicator only at the points where explicit loading is required, as follows:

```
        .ROOT CNTRL-(AFCTR,*BFCTR,CFCTR)
AFCTR:  .FCTR A0-(*A1,A2-*(A21,A22))
BFCTR:  .FCTR (B0-(B1,B2))
CFCTR:  .FCTR *C1-C2-C3-C4-C5
        .END
```

With this ODL description, the Linker generates seven autoload vectors -- those for A1, A21, A22, B0, B1, B2, and C1.

6.1.4 Autoloadable Data Segments

Overlay segments containing no executable code can make use of the autoload facility in the following way. The user must first include a .NAME directive with the GBL attribute, as described in Section 6.1.3.4.

For example:

```
        .ROOT A-*(B,C)
        .NAME BNAME,GBL
    B:  .FCTR BNAME-BFIL
        .END
```

The global symbol BNAME is created and entered into the symbol table of segment BNAME. Since this segment is marked to be autoloaded, root segment A calls segment BNAME as follows:

```
     CALL BNAME
```

The segment is autoloaded and an immediate return to in-line code occurs.

The user should place the data of BFIL in a .PSECT with the D attribute to suppress the creation of autoload vectors.

6.2 GLOBAL CROSS-REFERENCE OF AN OVERLAID TASK

Chapter 6 introduced the global cross-reference feature. This section illustrates a global cross-reference that has been created for an overlaid task. The task consists of a root segment containing the module ROOT.OBJ, and two overlay segments composed of modules OVR1 and OVR2. The overlay description of the file is as follows:

```
     .ROOT  ROOT-(OVR1,*OVR2)
```

Only segment OVR2 is autoloadable.

The resulting cross-reference listing is shown in Figure 6-1. By consulting the cross-reference listing, the user can make the following observations.

The global symbol OVR1 is defined in the module OVR1, and a single non-autoloadable, up-tree reference is made to this symbol by the module ROOT, as indicated by the circumflex (up-arrow on some printers). Note that there is no way to load segment OVR1 because of the restriction against mixing manual load and autoload in the same task.

The asterisk preceding the module OVR2 indicates that the global symbol OVR2 is an autoload symbol, and is referenced from the module ROOT through an autoload vector, as shown by the @ character.

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

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

```
OVRTST     CREATED BY  TKB    ON 1-OCT-76 AT 12:04       PAGE 1

GLOBAL CROSS REFERENCE                                   CREF   V01

SYMBOL  VALUE      REFERENCES...

N.ALER  000010       AUTO     # OVRES
N.IOST  000004       OVCTL    # OVRES
N.MRKS  000016     # OVRES
N.OVLY  000000       OVCTL    # OVRES
N.OVPT  000054       AUTO       OVCTL    # VCTDF
N.RDSG  000014     # OVRES
N.STBL  000002     # OVRES
N.SZSG  000012     # OVRES
OVR1    002014-R   # OVR1     ^ ROOT
OVR2    002014-R   * OVR2     @ ROOT
ROOT    001176-R   # ROOT
$ALBP1  001320-R   # AUTO
$ALBP2  001416-R   # AUTO
$ALERR  001246-R   # ALERR      OVDAT
$AUTO   001302-R   # AUTO
$DSW    000046       ALERR    # VCTDF
$MARKS  001546-R   # OVCTL
$OTSV   000052     # VCTDF
$SAVRG  001452-R     AUTO     # SAVRG
$VEXT   000056     # VCTDF
.FSRPT  000050     # VCTDF
.NALER  001442-R   # OVDAT
.NIOST  001436-R   # OVDAT
.NMRKS  001450-R   # OVDAT
.NOVLY  001432-R   # OVDAT
.NOVPT  000042     # OVDAT
.NRDSG  001446-R   # OVDAT
.NSTBL  001434-R   # OVDAT
.NSZSG  001444-R   # OVDAT
```

```
OVRTST     CREATED BY  TKB    ON 1-OCT-76 AT 12:04       PAGE 2

SEGMENT CROSS REFERENCE                                  CREF   V01

SEGMENT NAME    RESIDENT MODULES

OVR1            OVR1
OVR2            OVR2
ROOT            ALERR    AUTO     OVCTL    OVDAT    OVRES    ROOT     SAVRG
                VCTDF
```

Figure 6-1 Sample Overlaid Cross-Reference Listing

CHAPTER 7

MEMORY DUMPS

7.1 POST-MORTEM DUMPS

The task PMD... generates a post-mortem dump of a task that is abnormally terminated. A task can be made eligible for a post-mortem dump in any of three ways:

1. At task-build time by specifying the /DUMP switch for the task file. /NODUMP disables dumps; it is the default condition.

2. When using the ABORT command (described in the TRAX Support Environment User's Guide), a dump can be specified by including the switch /DUMP in the command line.

The Post-mortem Dump task PMD... is automatically installed by the system in a 4K partition in which all other tasks are checkpointable. This allows the dump to be generated in a timely manner and prevents the system from being locked up while the dump is being generated. The dump task is capable of dumping from memory or from the checkpoint image of the user's task. The Post-mortem Dump task is sensitive to the location of the aborted task; therefore, if the aborted task is checkpointed during the dump, the dump task switches to reading the checkpoint image. Once the task is checkpointed, PMD locks it out of memory until it has completed formatting the dump.

Dumps are always generated on the system disk under UFD [1,4]. When the dump task finishes generating the dump, it attempts to queue it to the print spooler for subsequent printing. If no spooler is installed, the dump file is left on the disk and can be printed later using the PRINT Command.

NOTE

Dump files tend to be somewhat large. The dump of an 8K partition averages about 340 blocks. Therefore, if there is little space on the disk, it is important to print and delete the dump file without delay. The print spooler automatically deletes all files with the type .PMD after printing them.

The following description of the contents of Post-mortem and Snapshot dumps is keyed to Figure 7-1. Snapshot dumps are explained more fully in Section 7.2.

Item	Description
1	Type of dump - Post-mortem or Snapshot. If it is a Snapshot dump, the dump ID is printed.
2	The name of the task being dumped, and the date and time the dump was generated.
3	The program counter at the time of the dump; and if it is a Post-mortem dump, the reason the task was aborted.
4	The general registers, stack pointer, and processor status at the time of the dump.
5	The task status flags at the time of the dump. See the description of ATL or TAL in the TRAX Support Environment User's Guide for the meaning of the flags.
6	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.
7	The task UFD and the current value of the directive status word.
8	The task's priority and default priority, number of outstanding I/O requests, and the terminal from which the task was initiated (TI:).
9	The task load device and the logical block number for the start of the task image on the device.
10	The floating point unit (FPU) registers or the extended arithmetic element (EAE) registers if the task is using one of these hardware features. If the task is not using the FPU or EAE, these registers are not printed. If the task uses the FPU and does not specify /FP on the task image file, or if it uses the EAE unit and has not specified the /EA switch, the registers are not printed. If the machine being used has both an FPU and an EAE, PMD assumes the user is using the FPU because it is the unit of choice for arithmetic computations.
11	The logical unit assignments at the time of the dump. UNIT is the logical unit number, and DEVICE is the device the logical unit is assigned to. For Snapshot dumps, file status displays the file name of any open files. Post-mortem Dumps will not display this information because all the files will have been closed as a result of the I/O rundown on the aborted task.
12	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 loads, 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 Linker. The starting block number for each segment is the relative block number of the segment. By obtaining a match, the name of the segment in memory can be determined. Zero length segments are usually co-tree roots.

Item	Description
13	The task stack at the time of the dump. The address is displayed, along with the contents, in octal, ASCII, and RAD50. Each word on the stack is dumped. If the stack pointer is above the initial value of the stack (H.ISP), only one word is dumped. The rest is dumped as part of the task image.
14	The task image itself. The partition being dumped and the limits of interest are displayed. For Post-Mortem Dumps, all address windows in use are dumped. For Snapshot Dumps, these are the virtual task limits requested by the user. The dump routine rounds the requested low limit down to the nearest multiple of 8 bytes and rounds the requested high limit up to the nearest multiple of 8 bytes. The dump image displays the virtual starting address of a four-word block of memory, the data in both octal and RAD50 on the first line, and byte octal and ASCII on the second line. A four-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.

7.2 SNAPSHOT DUMP

The task PMD... is also capable of producing edited dumps for running tasks. These dumps are called Snapshot Dumps, and they are useful as debugging aids. A Snapshot Dump can be requested 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 control bits, which control the format of the dump.
SNPBK$	allocates the Snapshot Dump Control Block (see Figure 7-2).
SNAP$	causes a Snapshot Dump to be generated.

SNPBK$ and SNAP$ issue calls to SNPDF$, so, in most cases, the programmer does not have to issue the SNPDF$ macro call explicitly.

```
                          POST-MORTEM DUMP  ①

TASK: TT6   ②                                    TIME: 5-OCT-76 15:06

PC: 000724   ③    IOT EXECUTION   ③

REGS:      R0 - 000345   R1 - 074400   R2 - 000120   R3 - 140130  }
                                                                  }  ④
           R4 - 000000   R5 - 000000   SP - 000304   PS - 170000  }

TASK STATUS:   MSG AST DST -CHK HLT STP REM MCR   ⑤

EVENT FLAG MASK FOR <1-16> 000001   ⑥

CURRENT UIC: [007,001]   DSW: 1.   ⑦

PRIORITY: DEFAULT - 50.  RUNNING - 50.   I/O COUNT: 0.   TI DEVICE - TT6:   ⑧

LOAD DEVICE - DB0:     LBN: 1,160034   ⑨

FLOATING POINT UNIT                                       }
                                                          }
    STATUS - 000000                                       }
                                                          }
    R0 - 000000   000000   000000   000000                }  ⑩
    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                }
  LOGICAL UNITS                                 }
                                                }
   UNIT  DEVICE    FILE STATUS                  }
                                                }  ⑪
    1     DB0:                                  }
    2     DB0:                                  }
    3     DB0:                                  }
    4     DB0:                                  }
OVERLAY SEGMENTS LOADED AND RESIDENT LIBRARIES MAPPED                        }
                                                                             }  ⑫
STARTING RELATIVE BLOCK: 000002   BASE: 000000   LENGTH: 001454              }
STARTING RELATIVE BLOCK: 000004   BASE: 001454   LENGTH: 000264              }


  TASK STACK                                    }
                                                }  ⑬
    ADDRESS CONTENTS ASCII RAD50                }
     000304  000045    %       7                }
```

Figure 7-1 Sample Post-Mortem Dump (Truncated)

```
                         TASK IMAGE


      PARTITION: GEN        VIRTUAL LIMITS: 000000 - 001777

000000  000304   000162   000001   067426  ! D6  B4   A Q0B!
        304 000  162 000  001 000  026 157                 !D  r      o!
000010  003401   003401   170017   000352  !AD3 AD3 8PO  E4!
        001 007  001 007  017 360  352 000                 !        p ] !
000020  000304   000000   000000   000000  ! D6             !
        304 000  000 000  000 000  000 000                 !D          !
000030  000000   000000   000000   000000  !                !
        000 000  000 000  000 000  000 000                 !           !
000040  000000   140162   074106   000001  !    01Z SI0   A!
        000 002  162 300  106 170  001 000                 !   r* Fx   !
000050  000000   000000   001104   000000  !          NT    !
        000 000  000 000  104 002  000 000                 !      D    !
000060  000373   000000   000000   000000  ! FK             !
        373 000  000 000  000 000  000 000                 !(          !
000070  000000   074150   000004   051646  !    SJX   D MON!
        000 000  150 170  204 000  246 123                 !   hx    &S!
000100  000000   051646   000000   051646  !    MON      MON!
        000 000  246 123  000 000  246 123                 !   &S    &S!
000110  000000   051646   000000   000001  !    MON        A!
        000 000  246 123  000 000  001 000                 !   &S      !
000120  067420   000000   001777   061404  !QXP        YW 03.!
        020 156  000 000  377 003  004 143                 ! n        c!
000130  000020   000000   030600   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    v e !
000160  000000   000000   000000   000000  !                !
        000 000  000 000  000 000  000 000                 !           !

***  DUPLICATE THROUGH  000236  ***

000240  000000   000000   001110   000000  !          NX    !
        000 000  000 000  110 002  000 000                 !      H    !
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 13      !
000300  001051   000001   000045   050114  ! M3   A    7 L36!
        051 002  001 000  045 000  114 120                 !)      %  LP!
000310  000000   000001   000100   000304  !       A   AX  D6!
        000 000  001 000  100 000  304 000                 !      *  D !
000320  000524   000000   000000   000000  ! HT             !
        124 001  000 000  000 000  000 000                 !T          !
000330  000000   000000   000000   063014  !             PMD!
        000 000  000 000  000 200  014 146                 !          f!
000340  131574   047123   052120   052123  !... LUK MSX MS$!
        174 263  123 116  120 124  123 124                 !13 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                 !      .  & !
```

14

Figure 7-1 (Cont.) Sample Post-Mortem Dump (Truncated)

Symbol		Offset	Description
SB.CTL		0	Control Flags
SB.DEV		2	Device Mnemonic
SB.UNT		4	Unit Number
SB.EFN		6	Event Flag
SB.ID		10	Snap Identification
SB.LM1	(L1)	12	Memory Block 1 Limits
	(H1)	14	
	(L2)	16	Memory Block 2 Limits
	(H2)	20	
	(L3)	22	Memory Block 3 Limits
	(H3)	24	
	(L4)	26	Memory Block 4 Limits
	(H4)	30	
SB.PMD		32	"PMD..." in RAD50
		34	

Figure 7-2 Format of Snapshot Dump Control Block

7.2.1 Format of the SNPBK$ Macro

The format of the SNPBK$ macro call is:

```
SNPBK$     dev,unit,ctl,efn,id,L1,H1,L2,H2,L3,H3,L4,H4
```

where:

dev	is the 2-character ASCII name of the device the dump is directed to. If it is a directory device, the UFD [1,4] must be on the volume. The dump is written to the disk and then spooled to the line printer. If there is no print spooler, the file is left on the disk. If the device is not a directory device, the dump goes directly to the device.
unit	is the unit number of the device the dump is directed to.
ctl	is the set of flags that control the format of the dump and the data to be printed. The flags are:

SC.HDR	Print the dump header (items 1-10 in Figure 9-1).

SC.LUN	Print information on all assigned LUNs (item 11).
SC.OVL	Print information about all loaded overlay segments (item 12).
SC.STK	Print the user stack (item 13).
SC.WRD	Print the requested memory in octal words and RAD50 (item 14).
SC.BYT	Print the requested memory in octal bytes and ASCII (item 14).

efn is the event flag to be used to synchronize the user program and the task PMD... .

id is 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 are the starting addresses of the memory blocks to be dumped.

H1,H2, H3,H4 are 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 zero.

Only one snap block is allowed. It generates the global label ..SPBK.

NOTE

Because SNPBK$ is used to allocate storage for the snap block, all arguments except dev must be valid arguments for .WORD or .BYTE directives.

7.2.2 **Format of the SNAP$ Macro**

The format of the SNAP$ macro is:

```
SNAP$ ctl,efn,id,L1,H1,L2,H2,L3,H3,L4,H4
```

where:

ctl is the set of flags that control the format of the dump and the data to be printed. The flags are:

SC.HDR	Print the dump header.
SC.LUN	Print information on all assigned LUNs.
SC.STK	Print the user stack.
SC.OVL	Print information about all loaded overlay segments.
SC.WRD	Print the requested memory in octal words and RAD50.
SC.BYT	Print the requested memory in octal bytes and ASCII.

efn	is the event flag to be used to synchronize the user program and the task PMD... . A Wait-For-Single-Event-Flag directive is always generated to perform synchronization.
id	is 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	are the starting addresses of memory blocks to be dumped.
H1,H2, H3,H4	are the ending addresses of memory blocks to be dumped.

NOTE

1. If no memory is to be dumped, each limit (L1,L2,L3,L4,H1,H2,H3,H4) should be zero.

2. The control flags can be set in any combination. They are not mutually exclusive. Thus, any number of options can be obtained; e.g., SC.HDR!SC.LUN!SC.WRD prints the header, LUNs, and the requested memory in word octal and RAD50 mode.

3. Arguments should be specified only to override the information already in the snap control block.

4. Because SNAP$ generates instructions to move data into the snap block, its arguments must be valid source operands for MOV instructions.

7.2.3 Example of a Snapshot Dump

The sample program shown in Figure 7-3 causes two Snapshot dumps to be printed directly on LP0:. The first dump uses the parameters defined in the Snap Control Block. The header is generated, and the data in relative locations BLK to BLK+220 is displayed, in word octal and RAD50. The identification on the dump is 1.

The second dump causes the data in the locations BLK to BLK+220 to be displayed in byte octal and ASCII. A header is also generated. The dump identification is 64 (100 octal). Figures 7-4 and 7-5 show the dumps generated by the sample program.

```
SNPTST - TEST SNAP DUMP AND PMD MACRO M1010  03-Jun-78 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-Jun-78 15:57  PAGE 1-1
SYMBOL TABLE

BLK      000000R         SB.EFN= 000006          SC.BYT= 000040          SC.STK= 000010          $DSW   = ****** GX
BUF      000036R         SB.ID  = 000010         SC.HDR= 000001          SC.WRD= 000020          $$$T2  = 000027
IE.ACT= ****** GX        SB.LM1= 000012          SC.LUN= 000002          START    000046R         ..SPBK   000000RG
SB.CTL= 000000           SB.PMD= 000032          SC.OVL= 000004          $CAT5  = ****** GX      ...SNP= 000032
SB.DEV= 000002           SB.UNT= 000004

  ABS.   000000     000
         000414     001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  1335 WORDS  ( 6 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  30 PAGES
ASSEMBLY TIME (ELAPSED):  00:00:14
SNPTST,SNPTST=SNPTST
```

Figure 7-3 Sample Program that Calls for Snapshot Dumps

```
                          SNAPSHOT DUMP  ID: 1

TASK: TT6                                              TIME: 05-Jun-78 15:06

PC: 000522

REGS:     R0 = 000000   R1 = 100104   R2 = 000000   R3 = 140130

          R4 = 000000   R5 = 000000   SP = 000304   PS = 170000

TASK STATUS:   MSG -CHK STP WFR REM MCR

EVENT FLAG MASK FOR <1-16> 000001

CURRENT UIC: [007,001]   DSW: 1.

PRIORITY: DEFAULT = 50.  RUNNING = 50.   I/O COUNT: 0.   TI DEVICE = TT6:

LOAD DEVICE = DB0:     LBN: 1,160034


FLOATING POINT UNIT

     STATUS = 000000

     R0 = 000000  000000  000000  000000
     R1 = 000000  000000  000000  000000
     R2 = 000000  000000  000000  000000
     R3 = 000000  000000  000000  000000
     R4 = 000000  000000  000000  000000
     R5 = 000000  000000  000000  000000

OVERLAY SEGMENTS LOADED AND RESIDENT LIBRARIES MAPPED

STARTING RELATIVE BLOCK: 000002   BASE: 000000   LENGTH: 001454

                       TASK IMAGE


      PARTITION: GEN       VIRTUAL LIMITS: 000304 - 000524

000300  001051  000001  000025  050114  ! M3   A   U L36!
000310  000000  000001  000001  000304  !      A   A  D6!
000320  000524  000000  000000  000000  ! HT            !
000330  000000  000000  000000  063014  !            PMD!
000340  131574  047123  052120  052123  !... LUK MSX MS$!
000350  000000  016746  177734  012746  !    D1N  7T CTF!
000360  001037  104377  103456  005046  ! MW U61 UYF AX8!
000370  012746  000304  012746  000336  !CTF  D6 CTF  EV!
000400  017646  000000  062766  000002  !EBV     PLV   B!
000410  000002  017666  000002  000002  !  B EB8   B   B!
000420  012746  002507  104377  103435  !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!
```

Figure 7-4 Sample Snapshot Dump (Words Octal and RAD50)

```
                              SNAPSHOT DUMP  ID: 64

TASK: TT6                                              TIME:05-Jun-78 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: [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
STARTING RELATIVE BLOCK: 000004   BASE: 001454   LENGTH: 000264

                         TASK IMAGE


     PARTITION: GEN       VIRTUAL LIMITS: 000304 - 000524

000300  051 002  001 000  045 000  114 120     !)      %  LP!
000310  000 000  001 000  100 000  304 000     !       •  D !
000320  124 001  000 000  000 000  000 000     !T            !
000330  000 000  000 000  000 000  014 146     !            f!
000340  174 263  123 116  120 124  123 124     !|3 SN PT  ST!
000350  000 000  346 035  334 377  346 025     !   f   \   f !
000360  037 002  377 210  056 207  046 012     !        .  & !
000370  346 025  304 000  346 025  336 000     !f  D   f   - !
000400  246 037  000 000  366 145  002 000     !&      ve    !
000410  002 000  266 037  002 000  002 000     !   6         !
000420  346 025  107 005  377 210  035 207     !f  G         !
000430  046 012  046 012  046 012  046 012     !&  &   &   & !
000440  346 025  336 000  246 037  000 000     !f  -   &     !
000450  366 145  002 000  002 000  266 037     !ve         6 !
000460  002 000  002 000  346 025  013 007     !       f     !
000470  377 210  006 206  337 045  371 377     !      _%   y !
000500  046 000  002 003  261 000  005 001     !&      1     !
000510  346 035  176 377  346 025  051 002     !f  ~   f   ) !
000520  377 210  300 025  342 000  367 011     !   •   b   w !
```

Figure 7-5 Sample Snapshot Dump (Bytes Octal and ASCII)

APPENDIX A

ERROR MESSAGES

The Linker 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. If the same error is detected in an indirect file by the Linker, a correction cannot be made and the link operation is aborted.

Some diagnostic error messages merely advise the user of an unusual condition. If the user considers the condition normal to his task, he can install and run the task image.

This appendix tabulates the error messages produced by the LINKER. Most of the messages are self-explanatory. In some cases, the line in which the error occurred is printed.

A Software Performance Report (SPR) should be submitted to DIGITAL in cases where the explanation accompanying a message refers to a system error.

ALLOCATION FAILURE ON FILE file-name

> The Linker could not acquire sufficient disk space to store the task image file, or did not have write-access to the UFD or volume that was to contain the file.

BLANK P-SECTION NAME IS ILLEGAL
overlay-description-line

> The overlay-description-line printed contains a .PSECT directive that does not have a p-section name.

COMMAND I/O ERROR

> I/O error on command input device. (Device may not be online, or possible hardware error.)

COMMAND SYNTAX ERROR
command-line

The command-line printed has incorrect syntax.

COMPLEX RELOCATION ERROR - DIVIDE BY ZERO: MODULE
module-name

A divisor having the value zero was detected in a complex expression. The result of the divide was set to zero. (Probable cause - division by a global symbol whose value is undefined.)

FILE file-name ATTEMPTED TO STORE DATA IN VIRTUAL SECTION

The file contains a module that has attempted to initialize a virtual section with data.

FILE file-name HAS ILLEGAL FORMAT

The file file-name contains an object module whose format is not valid.

ILLEGAL APR RESERVATION

An APR specified in a COMMON, LIBR, RESCOM, or RESLIB keyword is outside the range 0-7.

ILLEGAL DEFAULT PRIORITY SPECIFIED
option-line

The option-line printed contains a priority greater than 250.

ILLEGAL ERROR-SEVERITY CODE octal-list

System error (no recovery). An SPR should be submitted with a copy of the message containing the octal-list as printed.

ILLEGAL FILENAME
invalid-line

The invalid-line printed contains a wild card (*) in a file specification. The use of wild cards is prohibited.

ILLEGAL GET COMMAND LINE ERROR CODE

System error (no recovery).

ILLEGAL LOGICAL UNIT NUMBER
invalid-line

The invalid-line printed contains a device assignment to a unit number larger than the number of logical units specified by the UNITS keyword, or assumed by default if the UNITS keyword is not used.

ILLEGAL MULTIPLE PARAMETER SETS
invalid-line

The invalid-line printed contains multiple sets of parameters for a keyword that allows only a single parameter set.

ILLEGAL NUMBER OF LOGICAL UNITS
invalid-line

The invalid-line printed contains a logical unit number greater than 250.

ILLEGAL ODT OR TASK VECTOR SIZE

ODT or SST vector size specified greater than 32 words.

ILLEGAL OVERLAY DESCRIPTION OPERATOR
invalid-line

The invalid-line printed contains an unrecognizable operator in an overlay description. This error occurs if the first character in a p-section or segment name is a dot (.).

ILLEGAL OVERLAY DIRECTIVE
invalid-line

The invalid-line printed contains an unrecognizable overlay directive.

ILLEGAL PARTITION/COMMON BLOCK SPECIFIED
invalid-line

User-defined base or length not on 32-word boundary.

ILLEGAL P-SECTION/SEGMENT ATTRIBUTE
invalid-line

The invalid-line printed contains a p-section or segment attribute that is not recognized.

ILLEGAL REFERENCE TO LIBRARY P-SECTION p-sect-name

A task has attempted to reference a p-sect-name existing in a resident library (shared region), but has not named the library in a keyword.

ILLEGAL SWITCH
file-specification

The file-specification printed contains an illegal switch or switch value.

INCOMPATIBLE REFERENCE TO LIBRARY P-SECTION p-sect-name

A task has attempted to reference more storage in a shared region than exists in the shared region definition.

INCORRECT LIBRARY MODULE SPECIFICATION
invalid-line

The invalid-line contains a module name with a non-Radix-50 character.

INDIRECT COMMAND SYNTAX ERROR
invalid-line

The invalid-line printed contains a syntactically incorrect indirect file specification.

INDIRECT FILE OPEN FAILURE
invalid-line

The invalid-line contains a reference to a command input file which could not be located.

INSUFFICIENT PARAMETERS
invalid-line

The invalid-line contains a keyword with an insufficient number of parameters to complete its meaning.

INVALID APR RESERVATION
invalid-line

APR specified on a keyword for an absolute library.

INVALID KEYWORD IDENTIFIER
invalid-line

The invalid-line printed contains an unrecognizable keyword.

INVALID PARTITION/COMMON BLOCK SPECIFIED
invalid-line

Partition is invalid for one of the following reasons:

1. The Linker cannot find the partition name in the host system in order to get the base and length.

2. The system is mapped, but the base address of the partition is not on a 4K boundary for a non-runnable task or is not 0 for a runnable task.

3. The memory bounds for the partition overlap a shared region.

4. The partition name is identical to the name of a previously defined COMMON or LIBR shared region.

5. The top address of the partition for a runnable task exceeds 32K minus 32 words for a mapped system, or exceeds 28K minus 1 for an unmapped system.

6. A system-controlled partition was specified for an unmapped system.

INVALID REFERENCE TO MAPPED ARRAY BY MODULE module-name

The module has attempted to initialize the mapped array with data. An SPR should be submitted if this problem is caused by DIGITAL-supplied software.

INVALID WINDOW BLOCK SPECIFICATION
invalid-line

The number of extra address windows specified exceeds 7.

I/O ERROR LIBRARY IMAGE FILE

An I/O error has occurred during an attempt to open or read the Task Image File of a shared region.

I/O ERROR ON INPUT FILE file-name

I/O ERROR ON OUTPUT FILE file-name

LABEL OR NAME IS MULTIPLY DEFINED
invalid-line

The invalid-line printed defines a name that has already appeared as a .FCTR, .NAME, or .PSECT directive.

LIBRARY FILE filename HAS INCORRECT FORMAT

A module has been requested from a library file that has an empty module name table.

LIBRARY REFERENCES OVERLAID LIBRARY
invalid-line

An attempt was made to link the resident library being built to a shared region that has memory-resident overlays.

LOAD ADDR OUT OF RANGE IN MODULE module-name

An attempt has been made to store data in the task image outside the address limits of the segment. This problem is usually caused by one of the following:

1. an attempt to initialize a p-section contained in a shared region

2. an attempt to initialize an absolute location outside the limits of the segment or in the task header

3. a patch outside the limits of the segment it applies to

4. an attempt to initialize a segment having the NODSK attribute

LOOKUP FAILURE ON FILE filename
invalid-line

The invalid-line printed contains a filename that cannot be located in the directory.

LOOKUP FAILURE ON SYSTEM LIBRARY FILE

The Linker cannot find the system Library (SY0:[1,1]SYSLIB.OLB) file to resolve undefined symbols.

LOOKUP FAILURE RESIDENT LIBRARY FILE
invalid-line

No symbol table or task image file can be found for the shared region.

MAXIMUM INDIRECT FILE DEPTH EXCEEDED
invalid-line

The invalid-line printed gives the file reference that exceeded the permissible indirect file depth (2).

MODULE module-name AMBIGUOUSLY DEFINES P-SECTION p-sect-name

The p-section p-sect-name has been defined in two modules not on a common path, and has been referenced from a segment common to both paths.

MODULE module-name AMBIGUOUSLY DEFINES SYMBOL sym-name

Module module-name references or defines a symbol sym-name whose definition exists on two different paths, but is referenced from a segment that is common to both paths.

MODULE module-name ILLEGALLY DEFINES XFR ADDRESS p-sect-name addr

1. The start address printed is odd.

2. The module module-name is in an overlay segment and has a start address. The start address must be in the root segment of the main tree.

3. The address is in a p-section that has not yet been defined. An SPR should be submitted if this is caused by DIGITAL-supplied software.

MODULE module-name MULTIPLY DEFINES P-SECTION p-sect-name

1. The p-section p-sect-name has been defined more than once in the same segment with different attributes.

2. A global p-section has been defined more than once with different attributes in more than one segment along a common path.

MODULE module-name MULTIPLY DEFINES SYMBOL sym-name

Two definitions for the relocatable symbol sym-name have occurred on a common path. Or two definitions for an absolute symbol with the same name but different values have occurred.

MODULE module-name MULTIPLY DEFINES XFR ADDR IN SEG
segment-name

This error occurs when more than one module making up the root has a start address.

MODULE module-name NOT IN LIBRARY

The Linker could not find the module named on the /LIB switch in the library.

NO DYNAMIC STORAGE AVAILABLE

The Linker needs additional symbol table storage and cannot obtain it. (If possible, install the Linker in a larger partition.)

NO MEMORY AVAILABLE FOR LIBRARY library-name

The Linker could not find enough free virtual memory to map the specified shared region.

NO ROOT SEGMENT SPECIFIED

The overlay description did not contain a .ROOT directive.

NO VIRTUAL MEMORY STORAGE AVAILABLE

Maximum permissible size of the work file exceeded. The user should consult Appendix D for suggestions on reducing the size of the work file.

OPEN FAILURE ON FILE file-name

OPTION SYNTAX ERROR
invalid-line

The invalid-line printed contains unrecognizable syntax.

OVERLAY DIRECTIVE HAS NO OPERANDS
invalid-line

All overlay directives except .END require operands.

OVERLAY DIRECTIVE SYNTAX ERROR
invalid-line

The invalid-line printed contains a syntax error.

PARTITION partition-name HAS ILLEGAL MEMORY LIMITS

1. The partition-name defined in the host system has a base address alignment that is not compatible with the target system.

2. The user has attempted to build a privileged task in a partition whose length exceeds the task's available address space (8K or 12K).

PASS CONTROL OVERFLOW AT SEGMENT segment-name

System error. An SPR should be submitted with a copy of the ODL file associated with the error.

PIC LIBRARIES MAY NOT REFERENCE OTHER LIBRARIES
invalid-line

The user has attempted to build a position-independent shared region that references another shared region.

P-SECTION p-sect-name HAS OVERFLOWED

A section greater than 32K has been created.

REQUIRED INPUT FILE MISSING

At least one input file is required for a task-build.

ROOT SEGMENT IS MULTIPLY DEFINED
invalid-line

The invalid-line printed contains the second .ROOT directive encountered. Only one .ROOT directive is allowed.

SEGMENT seg-name HAS ADDR OVERFLOW: ALLOCATION DELETED

Within a segment, the program has attempted to allocate more than 32K. A map file is produced, but no task image file is produced.

TASK HAS ILLEGAL MEMORY LIMITS

An attempt has been made to build a task whose size exceeds the partition boundary. If a task image file was produced, it should be deleted.

TASK HAS ILLEGAL PHYSICAL MEMORY LIMITS
mapped-array   task-image   task extension

The sum of the parameters displayed -- mapped array size, task image size, and task extension -- exceeds 2.2 million bytes. The quantities are shown as octal numbers in units of 64-byte blocks. Any resulting task image file should be deleted.

TASK IMAGE FILE filename IS NON-CONTIGUOUS

Insufficient contiguous disk space was available to contain the task image. A non-contiguous file was created. After deleting unnecessary files, the /CONTIGUOUS switch in PIP should be used to create a contiguous copy.

TASK REQUIRES MORE THAN 8 WINDOW BLOCKS

The number of address windows required by the task and any shared regions exceeds 8.

TASK-BUILD ABORTED VIA REQUEST
option-line

The option-line contains a request from the user to abort the task-build.

TOO MANY NESTED .ROOT/.FCTR DIRECTIVES
invalid-line

The invalid-line printed contains a .FCTR directive that exceeds the maximum nesting level (16).

TOO MANY PARAMETERS
invalid-line

The invalid-line printed contains a keyword with more parameters than required.

TOO MANY PARENTHESES LEVELS
invalid-line

The invalid-line printed contains a parenthesis that exceeds the maximum nesting level (16).

TRUNCATION ERROR IN MODULE module-name

An attempt has been made to load a global value greater than +127 or less than -128 into a byte. The low-order eight bits are loaded.

UNABLE TO OPEN WORK FILE

The work file device is not mounted. (The work file is located on the same device as the Linker.)

UNBALANCED PARENTHESES
invalid-line

The invalid-line printed contains unbalanced parentheses.

n UNDEFINED SYMBOLS SEGMENT seg-name

The segment named contains n undefined symbols. If no memory allocation is requested, the symbols are printed on the terminal.

VIRTUAL SECTION HAS ILLEGAL ADDRESS LIMITS
option-line

The option-line printed contains a VSECT keyword whose base address plus window size exceeds 177777.

WORK FILE I/O ERROR

I/O error during an attempt to reference data stored by the Linker in its work file.

APPENDIX B

MEMORY ALLOCATION

The Linker is responsible for allocating the physical memory and virtual address space required by a task. This allocation can consist of two parts -- a region containing the task itself, and memory not physically a part of the task image, containing subroutines or data that are shared by several tasks.

B.1 TASK MEMORY STRUCTURE

Task memory (see Figure B-1) is divided into two physically contiguous areas containing:

1. The task image, and

2. Additional memory allocated while the task is running, by means of the Extend Task system directive; or before the task is running, by means of the Linker EXTTSK option.

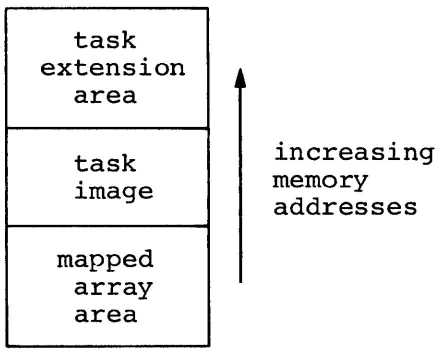


Figure B-1 Task Memory Structure

TRAX supports only mapped systems with memory management hardware.

In a mapped system, the task is usually bound to virtual zero and relocated by the mapping hardware, and therefore, the task can be installed in any partition large enough to contain it.

In a mapped system, the task can access only memory specifically owned by the task.

In a mapped system, the largest task size is normally 32K words minus 32 words.

B.2 TASK IMAGE MEMORY

The area of memory allocated for task image storage contains a header, a stack, and a set of named areas called program sections (p-sections).

The header contains task parameters and data required by the Executive and provides a storage area for saving the task's context.

The stack is an area that can be used for temporary storage and subroutine linkage, and is referenced by general register 6, the stack pointer (SP).

A p-section is an area of task memory, containing code or data, that can be referenced by name. Associated with each p-section is a set of attributes that control the allocation and placement of the section within the task image.

B.2.1 P-sections

A p-section, the basic unit of memory for a task, is composed of the following elements:

- a name by which it is referenced
- a set of attributes that define its contents, mode of access, allocation, and placement in memory
- a length that determines how much storage will be reserved for the p-section

P-sections can be created or referenced in either of the following ways:

- The language processors automatically include p-sections in the object module to reserve storage for code or data.
- The user can explicitly create p-sections by using facilities present in the language processors or Linker.

P-sections are created through the Linkage Section and segmentation facilities in COBOL or the COMMON statement in BASIC-PLUS-2, or the .PSECT directive in MACRO. The .PSECT directive allows the MACRO programmer to attach attributes to the section. A p-section of the specified name is conveyed to the Linker via the object module, whether it was created through COMMON or .PSECT.

The Linker's overlay processor allows p-sections to be created and inserted at specific points in the overlay structure. This facility is described in Chapter 5.

As noted above, each reference to a p-section is accompanied by a length and set of attributes that define how memory is to be allocated to the p-section. The Linker collects scattered references to the p-section in a single area of task memory. The attributes, listed in Table B-1, control the way the Linker collects and places this storage.

Table B-1
P-section Attributes

ATTRIBUTE	VALUE	MEANING
access-code	RW	Read/Write - Data can be read from, and written into, the p-section.
	RO	Read Only - Data can be read from, but cannot be written into, the p-section.
type-code *	D	Data - The p-section contains data.
	I	Instruction - The p-section contains either instructions, or data and instructions.
scope-code	GBL	Global - The p-section name is recognized across overlay segment boundaries. The Linker allocates storage for the p-section from references outside the defining overlay segment.
	LCL	Local - The p-section name is recognized only within the defining overlay segment. The Linker allocates storage for the p-section from references within the defining overlay segment only.
alloc-code	CON	Concatenate - All references to a given p-section name are concatenated. The total allocation is the sum of the individual allocations.
	OVR	Overlay - All references to a given p-section name overlay each other. The total allocation is the length of the longest individual allocation.
reloc-code	REL	Relocatable - The base address of the p-section is relocated relative to the virtual base address of the task.
	ABS	Absolute - The base address of the p-section is not relocated. It is always zero.
memory-code **	HIGH	High - The p-section is to be loaded into high-speed memory.
	LOW	Low - The p-section is to be loaded into low-speed memory.

The scope-code and type-code are meaningful only when an overlay structure is defined for the task. The scope-code is described in Chapter 6, in the context of p-section resolution. The memory-code is not used by the Linker.

----

* These codes should not be confused with the I and D space hardware on PDP-11 systems.

** Not used by the Linker.

The access-code and alloc-code are used by the Linker to determine the placement and the size of the p-section in task memory.

The Linker divides storage into read/write and read-only memory, and places the p-sections in the appropriate area according to access-code. Memory allocated to read-only p-sections is not hardware protected.

The alloc-code is used to determine the starting address and length of memory allocated by modules that reference a common p-section. If the alloc-code indicates that such a p-section is to be overlaid, the Linker places the allocation from each module at the same location in task memory, and determines the total size from the length of the longest reference to the p-section. If the alloc-code indicates that a p-section is to be concatenated, the Linker places the allocation from the modules one after the other in task memory, and determines the total allocation from the sum of the lengths of each reference.

The allocation of memory for a p-section always begins on a word boundary. If the p-section has the D (data) and CON (concatenate) attributes, all storage contributed by subsequent modules is appended to the last byte of the previous allocation. This occurs regardless of whether that byte is on a word or nonword boundary. For a p-section with the I (instruction) and CON attributes, however, all storage contributed by subsequent modules begins at the nearest following word boundary.

B.3 TASK IMAGE FILE

The Task Image file contains a copy of the task that can be read into memory and initiated with little system overhead. All binding, memory allocation, and address resolution are performed by the Linker; therefore, the only function performed by the system is the loading of the task image and the transfer of control to it.

In addition to the core image, the task image file contains a label block group and possibly a checkpoint area. The label block group contains data that is used by the Install processor to create an entry for the task in the system task directory.

The checkpoint area is allocated only if the user specifies that the task is checkpointable, and requests checkpoint space by using the /CHECKPOINT:filespec. The /CHECKPOINT:filespec switch indicates that the task is checkpointable, and causes the Linker to allocate checkpoint space within the task image file. /CHECKPOINT can be used instead, if the system incorporates dynamic allocation of checkpoint space. This makes the task checkpointable without the allocation of extra disk space in the task image file.

When the task is checkpointable and the /CHECKPOINT:filespec switch is used, the Linker must reserve space in the task image file large enough to save all of the memory owned by the task.

When the task is to reside in a system-controlled partition, the size of the required area is determined by the sum of:

- Amount of memory allocated for mapped array storage
- Size of the task image
- Size of the task extension

B.4 MEMORY ALLOCATION FILE

The memory allocation (.MAP) file lists information about the allocation of task memory and the resolution of global symbols. A global cross-reference list can be appended to the file by means of the /CROSS_REFERENCE switch.

B.4.1 Contents of the Memory Allocation File

The memory allocation file consists of the following items:

- Page Header
- Task Attributes
- Overlay Description (if applicable)
- Segment Description
- Memory Allocation Synopsis
- Global Symbols
- File Contents
- Summary of Undefined Global Symbols
- Linker Statistics

A sample of the memory allocation file produced by the command is shown in Figure B-2, where each item is identified. The overlay description does not apply to this task, and is therefore not shown.

If the /CROSS_REFERENCE switch is used to request a global cross-reference, then the following items are also included:

o Cross-Reference Page Header

o Global Cross-Reference

o Segment Cross-Reference

Figure B-3 illustrates a global cross-reference obtained by appending the /CROSS_REFERENCE switch to the memory allocation file specification of the previous example.

The paragraphs following Figure B-3 discuss the map items in greater detail.

```
IMG1.TSK;1   MEMORY ALLOCATION MAP  TKB M26          PAGE 1 ]- Page
                  01 Jun-78 11:59                              Header

PARTITION NAME : GEN                            ]
IDENTIFICATION : 00                             ]
TASK  UIC      : [301,303]                      ]   Task
STACK    LIMITS: 000172 001171 001000 00512.    ]-  attributes
TOTAL ADDRESS WINDOWS: 1.                       ]   section
TASK  IMAGE  SIZE  : 576. WORDS                 ]
TASK ADDRESS LIMITS: 000000 002153              ]

*** ROOT SEGMENT: IN1                           ]
                                                ]_  Segment
R/W MEM  LIMITS: 000000 002153 002154 01132.    ]   description
DISK BLK LIMITS: 000002 000004 000003 00003.    ]

MEMORY ALLOCATION SYNOPSIS:                                             ]
                                                                        ]
SECTION                                          TITLE  IDENT  FILE     ]
-------                                          -----  -----  ----     ]
. BLK.:(RW,I,LCL,REL,CON) 001172 000000 00000.                          ]
A     :(RW,I,LCL,REL,OVR) 001172 000300 00192.                          ]   Memory
                          001172 000300 00192. IN1    00     IN1.OBJ;1  ]
                          001172 000250 00168. IN2    00     IN2.OBJ;1  ]-  allocation
B     :(RW,I,LCL,REL,CON) 001472 000220 00144.                          ]   synopsis
                          001472 000100 00064. IN1    00     IN1.OBJ;1  ]
                          001572 000120 00080. IN2    00     IN2.OBJ;1  ]
C     :(RO,I,LCL,REL,CON) 001732 000220 00144.                          ]
                          001732 000150 00104. IN1    00     IN1.OBJ;1  ]
                          002102 000050 00040. IN3    00     IN3.OBJ;1  ]
$$$   :(RW,I,LCL,REL,CON) 001712 000020 00016.                          ]
                          001712 000020 00016. XXX    00     SYSLIB.OLB;1]

GLOBAL SYMBOLS:                                 ]
                                                ]_ Global
A     001176-R  B2     001472-R  XXX  001712-R  ]  symbols
B1    001472-R  L1     001172-R                 ]
```

Figure B-2 Memory Allocation File for IMG1.TSK on an Unmapped System

```
FILE: IN1.OBJ;1  TITLE: IN1      IDENT: 00
    <. ABS.>: 000000 000000 000000 00000.
 >>>>>>>>>>>> UNDEFINED REFERENCE: C1

    <B      >: 001472 001571 000100 00064.
       B1      001472-R  B2      001472-R
    <A      >: 001172 001471 000300 00192.
    <C      >: 001732 002101 000150 00104.
FILE: IN2.OBJ;1  TITLE: IN2      IDENT: 00
    <A      >: 001172 001441 000250 00168.
       A       001176-R
    <B      >: 001572 001711 000120 00080.
       B1      001472-R
```

File contents section

```
IMG1.TSK;1    MEMORY ALLOCATION MAP  TKB M26          PAGE 2
IN1                 01- Jun-78 11:59
```

File contents section (cont.)

```
FILE: IN3.OBJ;1  TITLE: IN3      IDENT: 00
    <C      >: 002102 002151 000050 00040.


FILE: LBR1.OLB;1  TITLE: L1       IDENT: 08
    <. BLK.>: 001172 001172 000000 00000.
       L1      001172-R


FILE: SYSLIB.OLB;1  TITLE: XXX      IDENT: 00
    <$$$    >: 001712 001731 000020 00016.
       XXX     001712-R
```

```
************

UNDEFINED REFERENCES:

    C1
```

Undefined references

```
*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 669.
    WORK  FILE  READS: 0.
    WORK  FILE  WRITES: 0.
    SIZE OF CORE POOL: 2066. WORDS (8. PAGES)
    SIZE OF WORK FILE: 512. WORDS (2. PAGES)

    ELAPSED TIME:00:00:11
```

Task Builder statistics

Figure B-2 (Cont.) Memory Allocation File for IMG1.TSK on an Unmapped System

```
SYMBOL     VALUE      REFERENCES...

$OTSV      000052       $CLOSE     $EOL       $ERRPT     $FIO       $INITI
                        $ISNLS     $OTI       $RETS      $STPPA     $VTRAN
$OTSVA     016206-R   # $OTV       .MAIN.
$PUTRE     011662-R     $IFW     # $PUTRE
$RLCB      027034-R   # RQLCB      RSTFDB
$RQCB      027136-R     OPFNB    # RQLCB
$R50       011714-R     $ERRPT   # $R50
$SAVRG     027274-R     RQLCB    # SAVRG
$SBR       001516-R   # $FADD
$SEQC      016206-R   # $OTV
$SST       016734-R   # $OTV
$SST0      004310-R   # $ERRPT     $OTV
$SST1      004316-R   # $ERRPT
$SST2      004330-R   # $ERRPT     $OTV
$SST3      004336-R   # $ERRPT     $OTV
$SST4      004344-R   # $ERRPT     $OTV
$SST5      004352-R   # $ERRPT     $OTV
$SST6      004436-R   # $ERRPT     $OTV
$SST7      004362-R   # $ERRPT     $OTV
$SVTK$     006002-R   # $ERRPT
.ASLUN     030436-R   # ASSLUN
.CLOSE     024702-R   # CLOSE      $CLOSE
.FATAL     031436-R   # COMMON     WAITI
.FINIT     025140-R   # FINIT      $OTI
.FSRCB     024264-R   # FCSFSR     $OTV
.FSRPT     000050       ASSLUN     CREATE     FCSFSR     FINIT      OPFNB
                        PARDI      RSTFDB     WAITI      XQIOI
.GTDID     025244-R   # GETDID     $OPEN
.MBFCT     024364-R   # FCSFSR
.MOLUN     016214-R   # $OTV
.NLUNS     016212-R   # $OTV
.OPFNB     025262-R   # OPFNB      $OPEN
.PPASC     032604-R     PARDI    # PPNASC
.PPR50     032132-R     DIDFND   # PPNR50
.PUTSQ     027320-R   # PUTSQ      $PUTRE
.SAVR1     030402-R     ASSLUN     CKALOC     CLOSE      FINIT      GETDID
                        OPFNB      PUTSQ    # SAVR1      $ERRPT     $OPEN
..ALC1     034012-R   # CKALOC
..ALOC     033742-R   # CKALOC     WTWAIT
..ALUN     030442-R   # ASSLUN     GETDI      OPFNB
..BDRC     030632-R   # BDBREC     OPFNB      RDWAIT     WTWAIT
..BKRG     030714-R   # BKRG       CLOSE      RDWAIT     WTWAIT
..CREA     030734-R   # CREATE     OPFNB
..DEL1     031122-R   # DEL        OPFNB
..DID      032216-R     DIDFND   # DIFND
..DIDF     032076-R   # DIDFND     PARDID
..EFCK     031162-R   # EOFCHK     PUTSQ
..EFC1     031170-R   # EOFCHK
..ENTR     031336-R     DEL      # DIRECT     OPFNB
..EXTD     034042-R   # CKALOC
..EXT1     034116-R   # CKALOC
..FCSX     031424-R     CLOSE    # COMMON     CREATE     OPFNB      PUTSQ
                        WAITI      WATSET     WTWAIT
..FIND     031352-R     DIFND    # DIRECT     OPFNB
..FINI     025150-R   # FINIT      OPFNB
..GTDI     031440-R   # GETDI      GETDID
```

Figure B-3 Cross-Reference Listing for MP1.MAP

The map items are described in the following paragraphs.

1. The page header shows the name of the task image file and the overlay segment name, along with the date, time, and version of the Linker that was used.

2. The task attribute section contains the following information:

   a. Task name

   b. Task partition

   c. Identification (task version)

   d. Task UFD

   e. Task priority

   f. Stack limits -- consisting of the low and high addresses, followed by the length in octal and decimal bytes

   g. ODT transfer address -- starting address of the debugging aid

   h. Program transfer address

   i. Task attributes -- shown only if they differ from the defaults. One or more of the following may be displayed:

AL	Task is checkpointable, and task image file contains checkpoint space allocation
CP	Task is checkpointable
DA	Task contains debugging aid
FP	Task uses floating-point processor
-HD	Task image does not contain header
PI	Task contains position-independent code and data
PM	Post-mortem dump requested in the event of abnormal task termination
SL	Task can be slaved
TR	Task initial PS word has T-bit enabled

   j. Total address windows -- the number of address windows allocated to the task

   l. Task extension -- the increment of physical memory allocated through the EXTTSK keyword

   m. Task image -- the amount of memory required to contain task code

   n. Total task size -- the amount of memory allocated to mapped arrays, task extension, and task image listed above

   o. Task address limits -- the lowest and highest virtual addresses allocated to the task.

3. The overlay description shows the address limits, length, and name of each overlay segment. Indenting is used to illustrate the overlay structure. The overlay description is printed only when a multi-segment task is created. An example of overlay description output is shown in Figure 5-1.

4. The segment description gives the name of the segment, along with the segment address and disk space limits.

5. The memory allocation synopsis gives information about the p-sections that make up the memory allocated to each overlay segment. The information shown consists of the p-section name, attributes, starting address, and length in bytes, followed by a list of modules that contributed storage to the section. The entry for each module shows the starting address and length of the allocation, the module name, module identification, and file name.

   If the /SEQUENTIAL switch is applied, the p-sections are listed in the order of input; otherwise they appear in alphabetical order.

   The following p-section information is omitted:

   a. The absolute section, . ABS. is not shown because it appears in every module and always has a length of 0.

   b. The unnamed relocatable section, shown as . BLK., is not displayed if its length is 0, because it appears in every module.

6. Global symbols that are defined in the segment are listed along with their octal values. An -R is appended to the value if the symbol is relocatable. The list is alphabetized in columns.

7. The file contents section lists the module name, file name, p-sections, and global definitions occurring in the module. Any undefined global references made by the module are also displayed.

8. A summary of undefined global references is printed after the listing of file contents.

9. The display of Linker statistics lists the following information, which may be used to evaluate Linker performance.

   - Work File References -- The number of times that the Linker accessed data stored in its work file.

   - Work File Reads -- The number of times that the work file device was accessed to read work file data.

   - Work File Writes -- The number of times that the work file device was accessed to write work file data.

- Size of Core Pool -- The amount of memory that was available for work file data and table storage.

- Size of Work File -- The amount of device storage that was required to contain the work file.

- Elapsed Time -- The amount of wall-clock time required to construct the task image and produce the memory allocation 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).

  Appendix D should be consulted for a more detailed discussion of the work file.

10. 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, in the following format:

```
GLOBAL  CROSS REFERENCE                          PAGE

map file name  CREATED BY TKB ON date AT time CREF Vn

SYMBOL          VALUE           REFERENCES...
```

11. The cross-reference list contains an alphabetic listing of each global symbol along with its value and the name of each referencing module. When a symbol is defined in several segments within an overlay structure, the last defined value is printed. Similarly, if a module is loaded in several segments within the structure, the module name will be displayed more than once within each entry.

    The suffix -R is appended to the value if the symbol is relocatable.

    Prefix symbols accompanying each module name define the type of reference as follows:

Prefix Symbol	Reference Type
blank	Module contains a reference that is resolved in the same segment or in a segment toward the root.
^	Module contains a reference that is resolved directly in a segment away from the root or in a co-tree.
@	Module contains a reference that is resolved through an autoload vector.
#	Module contains a non-autoloadable definition.
*	Module contains an autoloadable definition.

12. The segment cross-reference lists the name of each overlay segment and the modules that compose it.

NOTE

The reader should consult the glossary and Chapter 6 for a discussion of unfamiliar terms.

B.4.2 Control of Memory Allocation File Contents and Format

By using the memory allocation and input file switches described below, the user can eliminate nonessential information from the output, improve Linker throughput, and obtain output in a format that is more compatible with the hard-copy device.

The amount of information presented in the memory allocation file is controlled by the /SHORT and /MAP switches. When the /SHORT switch is included in the map file specification, the Linker eliminates the file contents section of the allocation listing. The list of global definitions by module, and the list of unresolved global references within a module are not produced. All other information can be found elsewhere in the output.

In general, the short format provides sufficient information for debugging while reducing task-build time considerably. Listings that contain a full description of file contents can be obtained at less frequent intervals and kept for later reference.

The contents of individual input files can be excluded from the listing by negating the /MAP switch (/NOMAP). Suppressing such output eliminates the following information from the allocation and cross-reference output of the excluded file:

- P-section contributions as shown in the memory allocation synopsis
- Global symbol definitions
- File contents
- Global definitions or references, and module names as shown in the cross-reference listing.

To disable map output for individual files, the user includes /NOMAP in the appropriate input file specification. To disable such output for the default system object module library and all memory-resident library files, the user includes /NOMAP in the memory allocation file specification.

The width of the listing is controlled by the /WIDE switch. This switch is included in the map file specification to increase the listing format to 132 columns. The global symbols, overlay description, and cross-reference output are expanded to fill the additional space.

MEMORY ALLOCATION

```
PARTITION NAME : GEN
IDENTIFICATION : 214097
TASK  UIC      : [0,202]
STACK    LIMITS: 000216 001215 001000 00512.
PRG XFR ADDRESS: 141260
TOTAL ADDRESS WINDOWS: 1.
TASK  IMAGE  SIZE  : 32448. WORDS
TASK ADDRESS LIMITS: 000000 176573

. OVERLAY DESCRIPTION:

BASE    TOP      LENGTH
----    ---      ------
000000  174337  174340  63712.       EXCEPT
174340  174423  000064  00052.              EX$025
174340  174743  000404  00260.              EX$049
174340  174433  000074  00060.              DD$025
174340  174443  000104  00068.              CR$025
174340  174673  000334  00220.              ST$047
174340  175337  001000  00512.              ST$048
174340  176573  002234  01180.              ST$049

 *** ROOT SEGMENT: EXCEPT


 R/W MEM  LIMITS: 000000 174337 174340 63712.


 MEMORY ALLOCATION SYNOPSIS:

 SECTION                                   TITLE  IDENT  FILE
 -------                                   -----  -----  ----
 . BLK.:(RW,I,LCL,REL,CON) 001216 001420 00784.
                           001216 000026 00022. ROCLOS 0005CM RMSLIB.OLB;16
                           001244 000026 00022. ROCONN 0003CM RMSLIB.OLB;16
                           001272 000104 00068. ROCREA 0012CM RMSLIB.OLB;16
                           001376 000026 00022. ROFIND 0004CM RMSLIB.OLB;16
                           001424 000026 00022. ROGET  0010CM RMSLIB.OLB;16
                           001452 000212 00138. ROINIT 0011CM RMSLIB.OLB;16
                           001664 000114 00076. ROFBDB 0007CM RMSLIB.OLB;16
                           002000 000124 00084. ROOPEN 0013CM RMSLIB.OLB;16
                           002124 000026 00022. ROPUT  0004CM RMSLIB.OLB;16
 ACDDAT:(RW,D,GBL,REL,OVR) 002636 000244 00164.
                           002636 000244 00164. ACDQIO 1A.18  COBLIB.OLB;1

                          152054 000130 00088. ROIMPA 0017CM RMSLIB.OLB;16
$$RTS :(RW,I,GBL,REL,OVR) 152204 000002 00002.
$$SGD0:(RW,D,LCL,REL,OVR) 152206 000000 00000.
$$SGD2:(RW,D,LCL,REL,OVR) 152346 000002 00002.
$$VEX0:(RW,D,GBL,REL,OVR) 152350 000004 00004.
$$VEX1:(RW,D,GBL,REL,OVR) 152354 000000 00000.
$$WNDS:(RW,D,LCL,REL,CON) 152354 000000 00000.
.CSID :(RW,I,LCL,REL,CON) 152354 000142 00098.
.CSII :(RW,I,LCL,REL,CON) 152516 002464 01332.


GLOBAL SYMBOLS:

ACCBUF 002712-R  AWFB3  013754-R  COVALU 111070-R  DISQIO 004232-R  EMCE   013354-R  FCP2   013606-R  INTEG  161226-R

ACCQIO 004112-R  AWFB4  013766-R  CREDLM 122516-R  DOCATS 127750-R  EMDD   017364-R  FCP3   013610-R  IOFLGS 111060-R

$ALBDB 023100-R  $COPS  166224-R  $GETBK 034766-R  $RLSBK 041200-R  $SVFRQ 044510-R  $XINIT 173270-R

$ALBST 023144-R  $CPOOL 023036-R  $GETTV 152004-R  $RMCLO 001216-R  $UNLK  047122-R  $XINRD 105334-R


*** TASK BUILDER STATISTICS:

    TOTAL WORK FILE REFERENCES: 279702.
    WORK  FILE  READS: 0.
    WORK  FILE WRITES: 0.
    SIZE OF CORE POOL: 17028. WORDS (66. PAGES)
    SIZE OF WORK FILE: 15360. WORDS (60. PAGES)

    ELAPSED TIME:00:00:46

*** SEGMENT: EX$025


R/W MEM  LIMITS: 174340 174423 000064 00052.


MEMORY ALLOCATION SYNOPSIS:

SECTION                                   TITLE  IDENT  FILE
-------                                   -----  -----  ----
$EX006:(RW,I,GBL,REL,CON) 174340 000064 00052.
$$ALVC:(RW,D,LCL,REL,CON) 174424 000000 00000.
$$RTS :(RW,I,GBL,REL,OVR) 152204 000002 00002.

GLOBAL SYMBOLS:

EX$025 152204-R
```

Figure B-4 Memory Allocation File Sample Program

APPENDIX C

RESERVED SYMBOLS

Several global symbols and p-section* names are reserved for use by the Linker.** Special handling occurs when a definition of one of these names is encountered in a task image.

The definition of a reserved global symbol in the root segment causes a word in the Task Image to be modified with a value calculated by the Linker. The relocated value of the symbol is taken as the modification address.

The following global symbols are reserved by the Linker:

Global Symbol	Modification Value
.FSRPT	Address of File Storage Region work area (.FSRCB)
.MOLUN	Error message output device
.NLUNS	The number of logical units used by the task, not including the Message Output and Overlay units
.NOVLY	The overlay logical unit number
N.OVPT	Address of Overlay Runtime System work area (.NOVLY)
.NSTBL	The address of the segment description tables. Note that this location is modified only when the number of segments is greater than one.
.ODTL1	Logical unit number for the ODT terminal device TI:
.ODTL2	Logical unit number for the ODT line printer device CL:
$OTSV	Address of Object Time System work area ($OTSVA)
.TRLUN	The trace subroutine output logical unit number
$VEXT	Address of vector extension area ($VEXTA)

----

* P-sections are created by .ASECT, .CSECT, or .PSECT directives. The .PSECT directive obviates the need for either the .ASECT or .CSECT directive, these being retained only for compatibility with other systems. In this document all sections are referred to as p-sections unless the specific characteristics of .ASECT or .CSECT apply.

** In addition, all symbols and p-section names containing a . or $ are reserved for DIGITAL-supplied software.

The following p-section names are reserved by the Linker. In some cases, the definition of a reserved p-section causes the p-section to be extended if the appropriate option input is specified (see Section 3.2.3.4).

Section Name	Description
$$ALVC	Contains segment autoload vectors
$$DEVT	The extension length (in bytes) is calculated from the formula EXT = <S.FDB+52>*UNITS Where the definition of S.FDB is obtained from the root segment symbol table and UNITS is the number of logical units used by the task, excluding the Message Output, Overlay, and ODT units.
$$FSR1	The extension of this section is specified by the ACTFIL option input.
$$IOB1	The extension of this section is specified by the MAXBUF option input.
$$OBF1	FORTRAN OTS uses this area to parse array type format specifications. May be extended by FMTBUF keyword.
$$RGDS	Contains region descriptors for resident libraries referenced by the task
$$RTS	Contains return instruction
$$SGD0	P-section adjoining task segment descriptors
$$SGD1	Contains task segment descriptors
$$SGD2	P-section following task segment descriptors
$$WNDS	Contains task window descriptors

APPENDIX D

INCLUDING A DEBUGGING AID

The user can include a program that controls the execution of a task, by naming the appropriate object module as an input file, and applying the /DEBUG command qualifier.

When such a program is input, the Linker causes control to be passed to the program when the task execution is initiated.

Such control programs might trace a task, printing out relevant debugging information, or monitor the task's performance for analysis.

The switch has the following effect:

1. The transfer address in the debugging aid overrides the task transfer address.

2. On initial task load, the following registers have the indicated value:

   R0 - Transfer address of task
   R1 - Task name in Radix-50 format (word #1)
   R2 - Task name (word #2)

APPENDIX E

TRAX LINKER GLOSSARY

AUTOLOAD	The method of loading overlay segments, in which the Overlay Runtime System automatically loads overlay segments when they are needed and handles any unsuccessful load requests.
CO-TREE	An overlay tree whose segments, including the root segment, are made resident in memory through calls to the Overlay Runtime System.
DISK-RESIDENT OVERLAY SEGMENT	An overlay segment that shares 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.
HOST SYSTEM	The system on which the task is built.
MAIN PARTITION	A partition whose memory may be subdivided into fixed-length sub-partitions, or dynamically allocated to each task by the Executive (system-controlled partitions).
MAIN TREE	An overlay tree whose root segment is loaded by the Monitor when the task is made active.
MAPPED ARRAY AREA	An area of the task's physical memory, preceding the task image, that is used for storage of large arrays. Space in the area is reserved by means of the VSECT keyword or through a Mapped Array Declaration contained in an object module. Access is through the mapping directives issued at run-time.
MEMORY ALLOCATION FILE	The output file created by the Linker that describes the allocation of task memory.
OVERLAY DESCRIPTION LANGUAGE	A language that describes the overlay structure of a task.

OVERLAY RUNTIME SYSTEM	A set of subroutines linked as part of an overlaid task that are called to load segments into memory.
OVERLAY SEGMENT	A segment that shares physical memory and/or 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.
PARTITION	An area of memory reserved for the execution of tasks.
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.
P-SECTION	A section of memory that is a unit of the total allocation. A source program is translated into object modules that consist of p-sections with attributes describing access, allocation, relocatability, etc.
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.
SEGMENT	A group of modules and/or p-sections that occupy memory simultaneously and that can be loaded by a single disk access.
SUB-PARTITION	A partition that resides within a main partition.
SYMBOL DEFINITION FILE	The output file created by the Task Builder that contains the global symbol definitions and values in a format suitable for reprocessing by the Linker. Symbol definition files are used to link tasks to shared regions.
SYSTEM-CONTROLLED PARTITION	A partition whose memory may be dynamically allocated by the Executive to several concurrently active, resident tasks.
TARGET SYSTEM	The system on which the task executes.
TASK IMAGE FILE	The output file created by the Task Builder that contains the executable portion of the task.

USER-CONTROLLED PARTITION	A partition that can accommodate only one active, resident task.
VIRTUAL ADDRESS SPACE	The set of addresses ranging from 0 to 177777 octal that are contained in a 16-bit word and referenced directly by a user's program.

INDEX

TRAX
Linker Reference Manual
AA-D342A-TC

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)

Name Date

Organization

Street

City State Zip Code or

Please cut along this line.

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML5-5/E39
Maynard, Massachusetts 01754