RSTS/E Task Builder Reference Manual

Order Number: AA-5072D-TC

digital
software

RSTS/E Task Builder Reference Manual

August 1990

This document describes the RSTS/E Task Builder (TKB), and tells how you use it to link programs.

August 1990

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation. The following are trademarks of Digital Equipment Corporation:

ALL–IN–1	DEUNA	RSX
DEC/CMS	DIBOL	RT
DECdx	EDT	RT-11
DEC/FMS-11	IAS	TOPS-10
DECmail	LA	TOPS-21
DECnet	LN01	ULTRIX
DECnet/E	Micro/RSX	UNIBUS
DECSA	OS/8	VAX
DECserver	PDP	VAXmate
DECsystem-10	PDP-11	VMS
DECSYSTEM-20	PDT	VT
DECUS	Q-BUS	WPS-PLUS
DECworld	RMS-11	Rainbow
DELUA	RSTS	digital™
DEQNA		

IBM is a registered trademark of International Business Machines Corporation.
RMS is a trademark of American Management Systems, Inc.

Contents

Part II Overlays

Part III   System Aspects

Chapter 8   User-Mode I- and D-Space

Chapter 9   Supervisor-Mode Library

Part IV   Reference Section

Chapter 10   Task Builder Command Line Format

Chapter 11 Task Builder Switches

Chapter 12   Task Builder Options

Chapter 13   Overlay Description Language (ODL)

Appendixes

Appendix A   Error Messages

Appendix B   Task Builder Input Data Formats

Appendix C   Executable File Structure

Appendix D   Reserved Symbols

Appendix E   Improving Task Builder Performance

Index

Examples

Figures

Tables

Preface

Objectives

This manual describes how to use the RSTS/E Task Builder to link your compiled or assembled programs and subprograms into an executable program file to run on RSTS/E.

On RSTS/E systems, your programs must be linked by the Task Builder if they were written in languages for the compilers listed below. Note that this manual is current for the versions shown in parentheses. Information about using the Task Builder may change for subsequent versions.

Compiler (Version)

BASIC-PLUS-2 (V2.6)
FORTRAN-77 (V5.3)
PDP-11 C (V1.0)
COBOL-81 (V3.0)
DIBOL (V6.1)

This manual also applies to the MAC assembler (V5.5) for MACRO programs.

Audience

Although you do not need to be a computer expert to use this manual, you should have a general understanding of computer languages and be familiar with using programs and subprograms.

Document Structure

This manual contains four parts, as indicated by the divider sheets preceding each section, and five appendixes:

Part I   Tells all you need to know to get a program built with the right libraries to run on a RSTS/E system. The two types of libraries (disk libraries and memory-resident libraries) are explained, along with details on how to link them with your program.

Part II	Discusses overlays. Chapter 3 describes how to specify an overlay structure for programs that are too large to fit in the available space. The key statements of the Overlay Description Language (ODL) are described and examples are given. Chapter 4 extends the discussion of overlays by describing a special overlay structure, called co-trees. Chapter 5 explains the autoload indicator, an ODL symbol, and tells how you can use this symbol to save some space in your program. Chapter 6 describes overlays from another point of view: working with units called "program sections." Special ODL commands are available to deal with these units; they are described in this section, and more examples are given.
Part III	Describes system aspects of Task Building. Chapter 7 describes how to build your own memory-resident library, and how to build your own cluster library. Chapter 8 describes techniques for revectoring cluster libraries, and Chapter 9 describes the use of instruction and data (I- & D-) space.
Part IV	Is a reference source. Chapters 10, 11, and 12 describe the full Task Builder command format, switches, and options, respectively. Chapter 13 describes the Overlay Description Language in detail.
	Appendix A describes error messages provided by the Task Builder. Appendixes B and C describe internal data formats used by the Task Builder and the format of the executable file produced by the Task Builder. Appendix D lists and describes global symbols and program section names reserved for use by the Task Builder. Appendix E describes how to improve Task Builder performance.
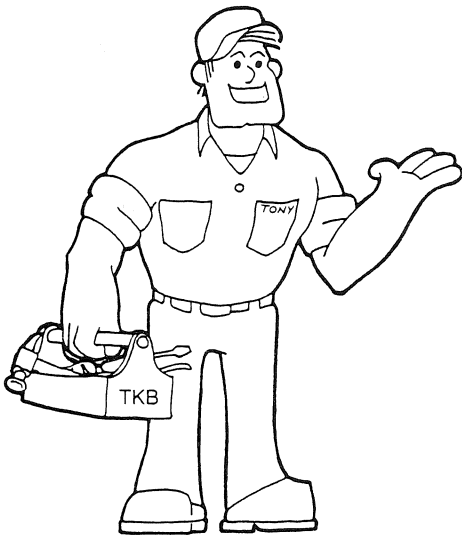
MK–00565–00

Conventions

UPPERCASE COMMANDS	In general command format descriptions, UPPERCASE indicates commands that you must type as shown.
lowercase commands	Indicate variables that you supply. For example:
	`.ROOT structure`
Red Print	In the Task Builder command examples, user input is shown in red. The Task Builder's responses and prompts are printed in black. For example:
	`ENTER OPTIONS:` `TKB> UNITS=8`
$	The dollar sign is the default DCL prompt.

Summary of Technical Changes

RSTS/E V10.0 is a major release of the RSTS/E PDP-11 operating system. This manual describes the following technical changes to the Task Builder function:

- There are two new switches: the /FM and /FO switches.

- Four new options are available: RESSUP, SUPLIB, VARRAY AND VSECT.

- Prior to RSTS/E V9.7, when a library was included in a task build, whichever APRs were assigned to the RESLIB or LIBR libraries included both I- and D-spaces. Beginning with V9.7, both the I- and D-spaces are initially assigned to the libraries; however, the COMMON and RESCOM options can be used to reallocate the D-space APRs to data-only regions. This feature, called "concurrent libraries", allows for more efficient use of memory when I-only code libraries (such as RMS) are used. Note that this feature is available only on CPUs that support separate I- and D- spaces.

Introduction

You need to use the Task Builder (TKB) if you write programs on RSTS/E systems in BASIC-PLUS-2, FORTRAN-77, PDP-11 C, COBOL-81, DIBOL, or the MACRO assembly language using the MAC assembler.

The compilers and assemblers associated with these languages translate your programs and subprograms (called source code) into machine instructions (object code). The Task Builder applies the final touches, converting the object code produced by the compilers to code that can be executed by the computer. Figure 1–1 shows the steps involved in creating a program.

Figure 1–1: Steps in Creating a Program
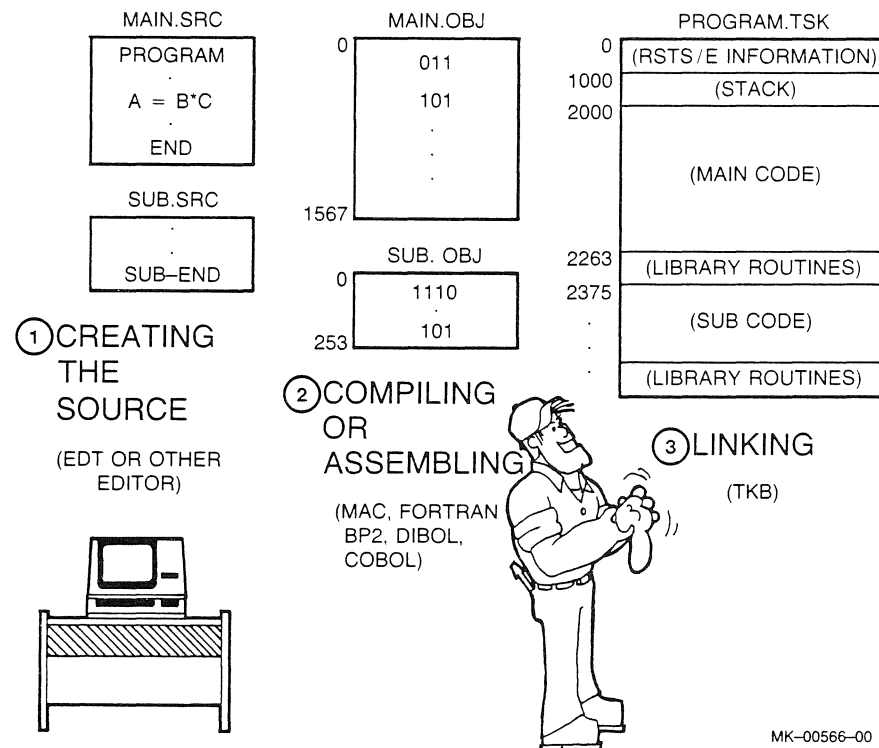
1.1 What the Task Builder Does

The Task Builder handles two basic functions: linking and producing overlays.

1.1.1 Linking

Linking is necessary because you seldom write programs as one unit. It is easier to work with programs that are written as modules – programs and subprograms—that you can separately design, code, debug, and maintain.

Even if you code your program as one main program, with no separately assembled or compiled subprograms, every compiler translates some source statements into calls to subroutines kept in libraries. For example, all the compilers generate calls to library subroutines to perform I/O or do mathematical calculations. Libraries are provided with the system and with the compilers available with RSTS/E systems.

The Task Builder links these separate modules—your main program, subprograms, and library routines—together in the order you specify, resolving any references that cross module boundaries. For example, Figure 1–2 shows a call to SUB1 from the program MAIN.

Figure 1–2: The Task Builder Resolves Global References
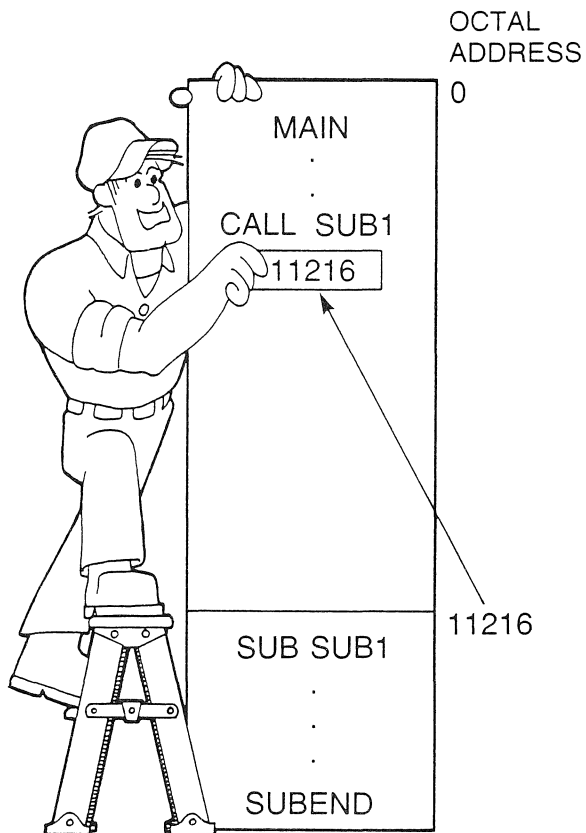

```
RUN $TKB
TKB>MAIN = MAIN, SUB1, LB:F4POTS/LB
TKB>//
```

MK–00567–00

The command to the Task Builder (the line after RUN $TKB) says that these two modules are to be linked together. In addition, any routines necessary from the FORTRAN library are to be linked with these two modules. To simplify, the figure shows only the linking of MAIN and SUB1. Part of the linking process involves generating the proper succession of addresses. As Figure 1–1 showed, the compilers and assemblers generate what are called "relative addresses"; the first address of each module (MAIN and SUB1) is numbered 0 at the compilation stage. When the Task Builder links modules, it changes the addresses of the second and following modules to begin where the addresses of the previous module left off. So, the final addresses for the linked program, as assigned by the Task Builder, range upward from 0 in succession.

The second aspect of linking is resolving references to what are called "global symbols." At compile time, for example, MAIN's reference to SUB1 cannot be resolved. SUB1 is flagged as a global reference (somewhere in the "world outside of MAIN") when MAIN is compiled. Likewise, when SUB1 is compiled, it is again flagged as a global symbol; it will serve as an entry point from the "outside world."

The Task Builder, as shown in Figure 1–2, keeps track of the addresses assigned to global symbols and substitutes the address for the entry point of SUB1 into the call in MAIN. Then, when the program is run, and the call is executed, control transfers to address 11216, the entry point for SUB1.

1.1.2 Overlays

The second necessary service that the Task Builder provides is a means to construct overlays. The amount of memory from which programs can be executed is limited on PDP-11 computers to 32,000 words. On RSTS/E systems, for reasons described in Chapter 2, there are further limitations. If your program is too large to fit in the space available, you must specify how you want it overlaid—such that sections of code and data can be called into memory at different times (the new sections "overlaying" the old).

Figure 1–3 shows the concept behind overlays. The Task Builder links both the modules SUB1 and SUB2 to start at address 15,726. The Task Builder then inserts code into MAIN such that, when MAIN's call to SUB2 is executed, SUB2 will replace SUB1, called and executed previously. SUB1 does not have to be the same length as SUB2, but both will be linked to start at the same address.

Figure 1-3 also shows something called the "high segment" in high address space. This code is the main reason your program does not have the full 32,000 words available on PDP-11 systems. For further information, see Chapter 2.

Figure 1–3: The Task Builder Constructs the Overlays You Specify

```
RUN $TKB
TKB> PROG = OVR/MP
TKB>//
```

(THE 'MAP FILE'
OVR.ODL CONTAINS
AN OVERLAY
DESCRIPTION)
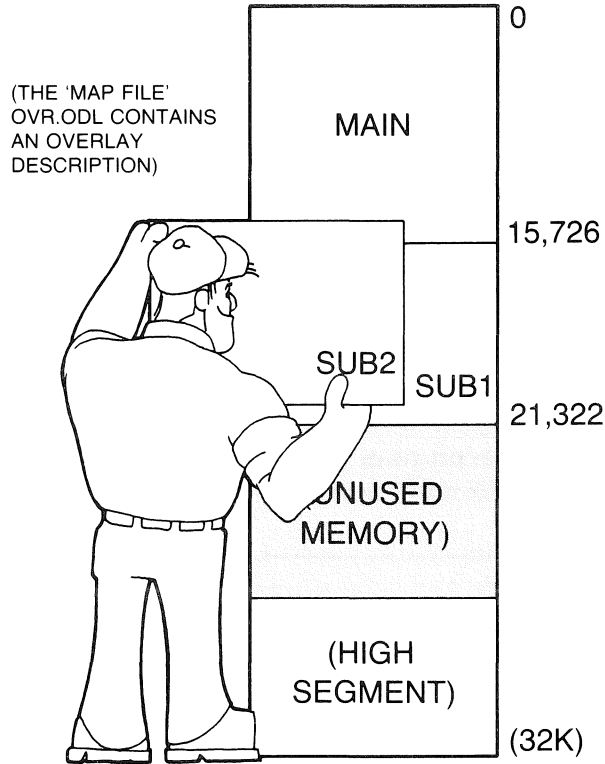
```
                              0
        MAIN

                              15,726

     SUB2   SUB1
                              21,322

     (UNUSED
     MEMORY)

      (HIGH
    SEGMENT)
                              (32K)      MK-00568-00
```

1.2 Relationship to the DCL LINK Command

You can use the DCL LINK command to link your programs, as described in the
RSTS/E DCL User's Guide. Like all DCL commands, the LINK command is
somewhat simpler to use, compared to typing a RUN command to execute TKB.
However, the LINK command does not offer all the features and flexibility of the
Task Builder. Note that the DCL LINK command does not work any faster than
running TKB; LINK also runs the Task Builder to perform the requested action.

Part I
Getting Started

Building Programs

This chapter tells how to build nonoverlaid programs. How large can a program be before it must be overlaid? The answer depends on the language you used to write your program; Section 2.1 discusses some specifics. The library routines built into your executable program also affect its size (Figure 2–1). Section 2.2 names and describes the disk libraries currently provided by Digital for the various languages. Section 2.3 discusses the Task Builder command line in general, and Section 2.4 gives specific examples for building programs written in each of the various languages.

Figure 2–1:   You Tell the Task Builder Which Libraries to Include


```
RUN $TKB
TKB>MAIN, SUB1, LB: F4POTS/LB
TKB> //
```
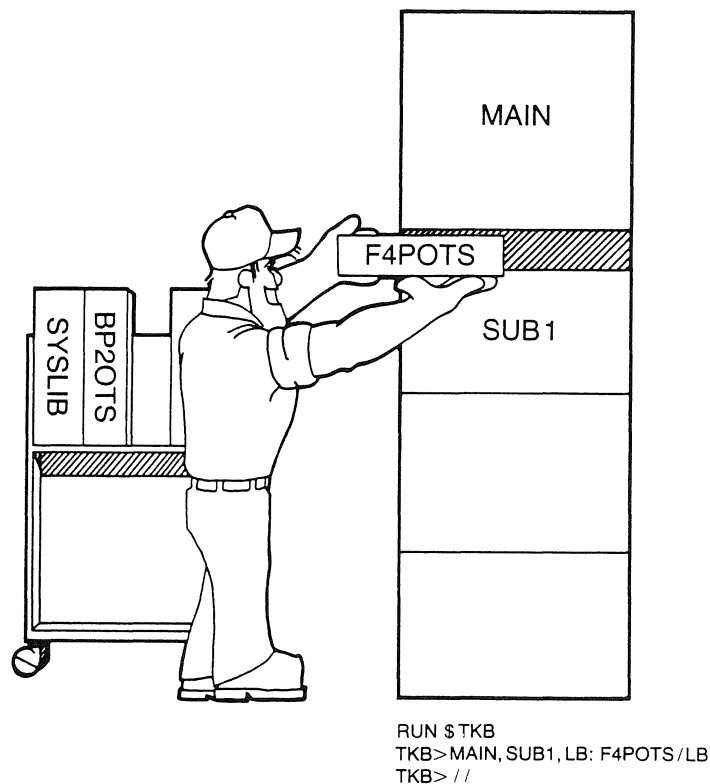
MK–00569–00

2.1 Job Area

As Chapter 1 mentions, the hardware imposes a limit on your program's size. The PDP-11 computer handles instruction and data in terms of a "16-bit word." A 16-bit word can reference 2^{16} ($65,536_{10}$) bytes, or 32,768 words. Thus, unless your program uses user-mode I- and D-space, 32K words is the maximum area of computer memory you can work with at one time.

If you have a PDP-11/44, 11/45, 11/50, 11/53, 11/55, 11/70, 11/73, 11/83, 11/84, 11/93 or 11/94 system, you can use user-mode I- and D-space. This feature lets you extend your task to 64K words of virtual address space (32K-word maximum of instruction space, and 32K-word maximum of data space). See Chapter 7 for more information on user-mode I- and D-space.

2.1.1 Your Program Within the Job Area

Except in special applications such as BASIC-PLUS and RT-11, the monitor loads programs. Monitor-loaded programs include BASIC-PLUS-2, PDP-11 C, PDP-11 COBOL, COBOL-81, DIBOL, FORTRAN-77, and MACRO programs assembled with the MAC assembler.

This section describes how your program fits within the job area if your program uses a run-time system.

The Task Builder constructs your executable program so that it fits within the job area in the low address space, beneath the run-time system (see Figure 2–2). Note the way your job area is constructed of various regions in physical memory.

For example, Figure 2–2 shows physical memory addresses for user program 2 that are actually higher than the so-called "hiseg" or run-time system. Yet the Task Builder, when it builds a program, constructs addresses for the program as though it operated within one 32K-word job area in memory. The RSTS/E monitor resolves this difference by using active page registers (APRs).

The job area is sometimes called "virtual address space," because it appears to you that your program and its associated run-time system reside in a contiguous 32K-word area. As Figure 2–2 shows, this is not actually the case in physical memory.

Figure 2–2: Job Area: Two User Programs

PHYSICAL MEMORY

VIRTUAL ADDRESS
SPACE

JOB1
- 0
- 4K
- 8K — USER PROGRAM
- 12K
- 16K
- 20K
- 24K — USER PROGRAM
- 28K — RSX RUN-TIME SYSTEM
- 32K

RSX RUN-TIME SYSTEM

JOB2
- 0
- 4K — USER PROGRAM
- 8K
- 12K
- 16K — USER PROGRAM
- 20K
- 24K
- 28K — RSX RUN-TIME SYSTEM
- 32K
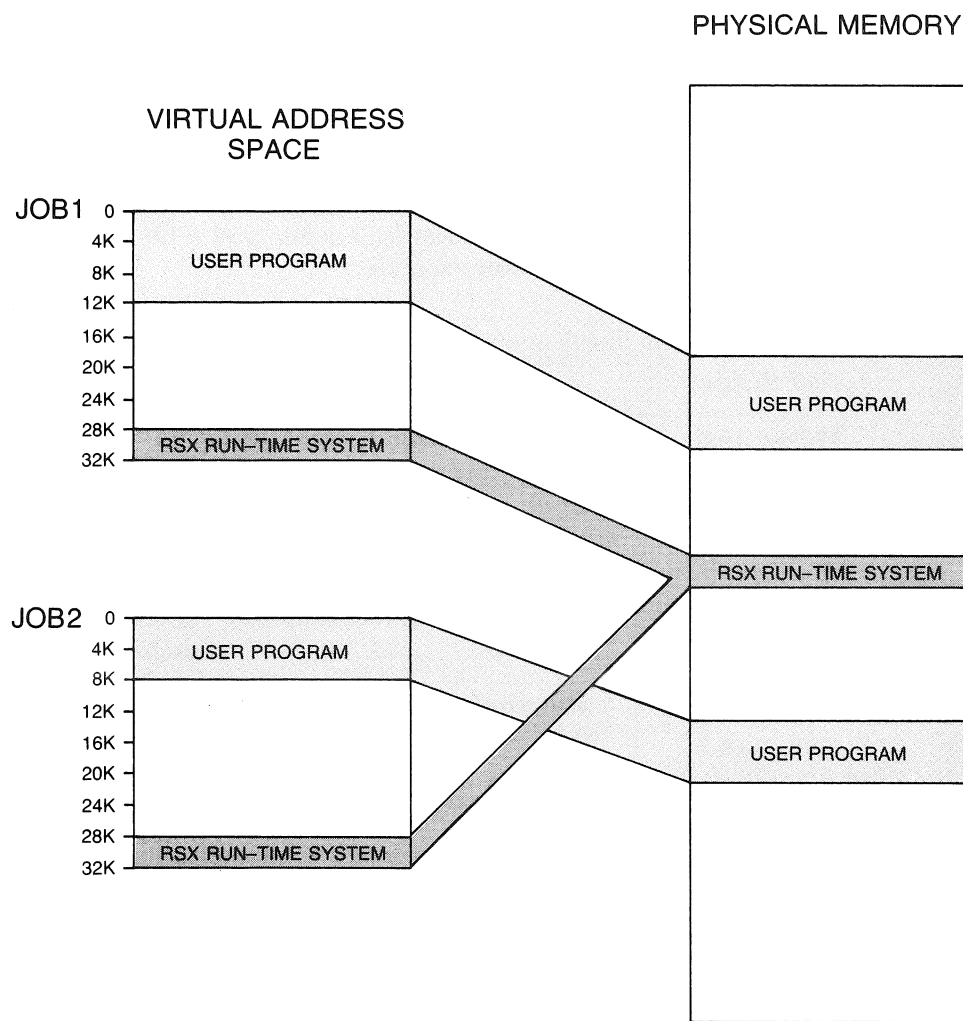
MK–00570–00

2.2 Libraries

As mentioned in Chapter 1, every compiler translates some source statements into calls to subroutines. These subroutines are kept in what are called "libraries." Digital supplies libraries of subroutines used with each language. Because the Task Builder has no way of knowing the source language you used, you must tell it what libraries contain routines that are referenced by your program. Two general types of libraries may be available on your system: disk libraries and resident libraries.

2.2.1 Disk Libraries

The libraries listed in Table 2–1 are currently shipped with RSTS/E and its associated languages. Note that the table is current for the versions of the software mentioned in the Preface. As new versions of languages are released, library names and contents may change. In addition, other products available with RSTS/E can have associated libraries, and your own installation may have generated its own libraries.

One way to find out what libraries are available is to get a directory of the system library device (LB:) with a wildcard file name and a file type of .OLB. (OLB stands for object library.) For example:

```
DIR LB:*.OLB

Name .Typ    Size  Prot       DR3:[1,1]
SYSLIB.OLB   220   < 40>
RMSLIB.OLB   300   < 40>
BP2OTS.OLB   225   < 40>
COBLIB.OLB   178   < 40>
```

Table 2–1 describes some of the libraries in this account that your program may use.

Table 2–1: Disk Libraries Used with RSTS/E

Disk Library Name	Description
SYSLIB.OLB	The system library. Contains many routines used by programs written in MACRO (for the MAC assembler) and the higher-level languages. The Task Builder always searches this library to resolve undefined symbols. You do not need to specify it in a Task Builder command line.
RMSLIB.OLB	Contains routines needed if you use RMS (Record Management Services) on RSTS/E systems.
RMSDAP.OLB	Contains routines needed for network record access through RMS on RSTS/E systems.
BP2OTS.OLB	Contains routines needed to run your BASIC-PLUS-2 program under the RSX run-time system.
DBLLIB.OLB	Contains routines needed to run your DIBOL program if it uses the DIBOL Management System (DMS) for I/O. Note that you must also declare a resident library (DBLRES) if you use this disk library. See Section 2.3.4 for information on how to specify resident libraries.
DBRLIB.OLB	Contains routines needed to run your DIBOL program if you use the Record Management System (RMS) for I/O. Note that you must also declare a resident library (DBRRES) if you use this disk library. See Section 2.3.4 for information on how to specify resident libraries.
COBLIB.OLB	Contains routines needed to run your PDP-11 COBOL program. If you use this library rather than COBOVR.OLB, your program will take more memory but will run faster.

Table 2–1 (Cont.):   Disk Libraries Used with RSTS/E

Disk Library Name	Description
COBOVR.OLB	Contains routines needed to run your PDP-11 COBOL program if it is overlaid. You use this library if you use the PDP-11 COBOL segmentation facility. However, if you use this library rather than COBLIB.OLB, your program will run slower, as the routines are called in as needed and overlay each other.
C81CIS.OLB	Contains routines needed to run your COBOL-81 program if the program was compiled with the /CIS switch. This is the normal default if your computer has the Commercial Instruction Set (CIS) option.
C81LIB.OLB	Contains routines needed to run your COBOL-81 program if the program was compiled with the /-CIS switch. This is the normal default if your computer does not have the Commercial Instruction Set (CIS) option.
FDVDBG.OLB	Contains routines needed if you use the FMS form driver with debug mode support.
FDVLIB.OLB	Contains routines needed if you use the FMS form driver without debug mode support.
F4POTS.OLB	Contains routines needed to run your FORTRAN-77 program.
F4PRMS.OLB	Contains routines for FORTRAN-77 programs using RMS (Record Management Services) for I/O.

2.2.2   Resident Libraries

In addition to disk libraries, you may also have to work with resident libraries on RSTS/E systems. "Resident" means residing in computer memory. The system manager defines libraries as resident so that they can be shared by more than one user. Instead of building routines into your program (as is done with disk libraries), you use a copy of the library. The copy is resident in memory as long as you or someone else is using it.

The Task Builder links your program to appropriate routines in the resident library by a technique called "mapping." Mapping is the process of accessing different logical areas of memory. With the mapping technique, many programs can use routines from the same space in computer memory. The system manager usually defines a library to be resident when it is heavily used. In such cases, less overall computer memory is taken by a resident library than by having each program include its own copy of routines from the library.

Figure 2–3 shows the difference between disk and resident libraries. For disk libraries, the Task Builder takes a copy of each routine that you reference in your program and builds it into your program. Note that a copy of RTNA has been built into both PROG1 and PROG2 in this figure. However, both programs can reference a resident library from the same area of physical memory.

Figure 2-3: Disk and Resident Libraries

PROG1

RTN E
RTN F
RTN A

PROG2

RTN A
RTN E
RTN C
RTN B

RTN A
RTN B
RTN C
RTN D
RTN E
RTN F
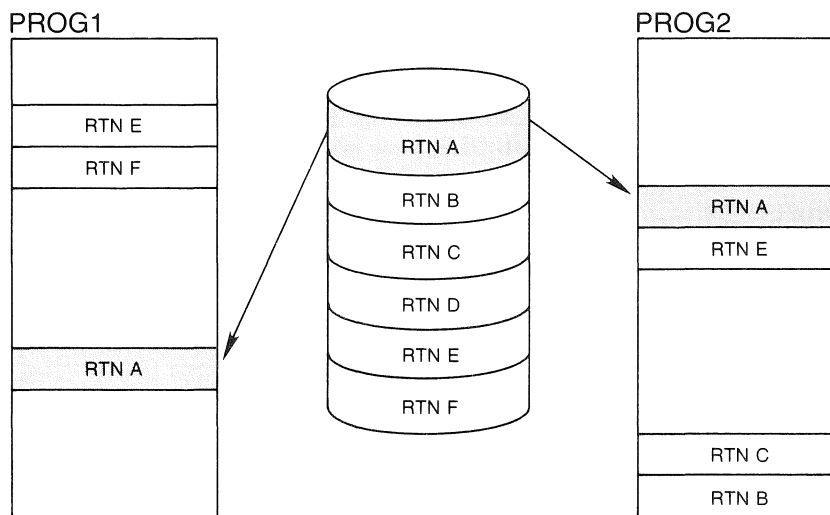
DISK LIBRARY: COPIES OF ROUTINES ARE BUILT INTO EACH PROGRAM.

PROG1   PHYSICAL   PROG2
                          MEMORY

RESIDENT
LIBRARY

RESIDENT LIBRARY   RESIDENT LIBRARY
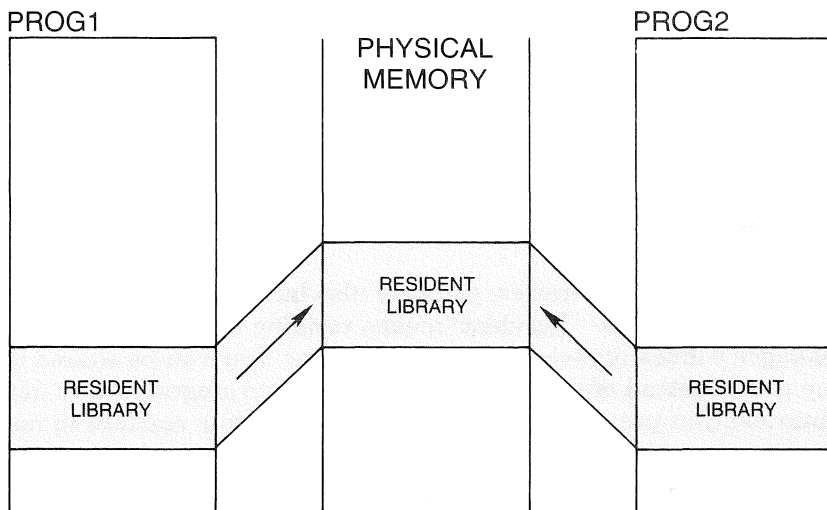
RESIDENT LIBRARY: MANY PROGRAMS CAN USE ONE COPY OF THE LIBRARY IN MEMORY.

MK-00571-00

You need to be aware of the distinction between disk and resident libraries
because the Task Builder commands that cause a link to resident libraries differ
from those for disk libraries. You can tell what resident libraries are on your
system by running the SYSTAT program. One section of the system status report
is headed "Resident Libraries:". You can request just this section of the report by
using the SHOW LIBRARIES DCL command.

The following example shows resident libraries. The RMS libraries are sup-
plied with all RSTS/E systems. They contain routines providing RMS (Record
Management Services) for input/output. The two BASIC resident libraries,
BP2RES and BP2SML are components of the layered product BASIC-PLUS-2 and
are discussed further in Section 2.2.3.

```
$ SHOW LIBRARY
Resident Libraries:
 Name   Prot      Acct        Size  Users  Comments
 RMSRES < 42>  DR1:[  0,1  ]    4K    1    Temp, Addr:733
 RMSLBB < 42>  DR1:[  0,1  ]    4K    1    Temp, Addr:737
 RMSLBA < 42>  DR1:[  0,1  ]    4K    0    Temp, Addr:741
 RMSLBC < 42>  DR1:[  0,1  ]    3K    0    Non-Res, Addr:745
 RMSLBD < 42>  DR1:[  0,1  ]    2K    0    Temp, Addr:748
 RMSLBE < 42>  DR1:[  0,1  ]    4K    0    Temp, Addr:750
 RMSLBF < 42>  DR1:[  0,1  ]    4K    0    Temp, Addr:754
 BP2RES < 42>  DR1:[  0,1  ]   19K    0    Non-Res, Addr:760
 BP2SML < 42>  DR1:[  0,1  ]    8K    0    Temp, Addr:779
```

The Task Builder allows your program to access up to seven resident libraries on
RSTS/E systems.

2.2.3 Comparison of Disk and Resident Libraries

Resident libraries require a large amount of physical memory. However, if many
tasks run at the same time, resident libraries reduce the total amount of physical
memory required by these tasks.

For example, BP2RES contains most of the BASIC Object Time System (OTS),
that is, most of the library routines supplied with BASIC-PLUS-2. It occupies
19K words of physical memory and takes 8K words of virtual address space in
your program. BP2SML contains a subset of the most commonly used BASIC
routines. It uses 8K words of physical memory and 8K words of virtual address
space. Even though BP2RES takes up 19K words of physical memory, that would
be less than, say five running copies of a program each using 4K words of BP2
routines built into each copy from a disk library (20K words total).

Therefore, the main advantage of using resident libraries is that their code can
be shared by many programs. In addition, task building is much faster when
using resident libraries because the Task Builder does not have to access the
library on disk as often. If you program in BASIC-PLUS-2, note that the resident
libraries (BP2SML and BP2RES) do not contain the entire OTS, therefore, most
BASIC-PLUS-2 programs will reference some entry points within the disk library
BP2OTS.OLB.

2.3 How to Run the Task Builder

To run the Task Builder, type:

RUN $TKB

Or, if the system manager has installed TKB as a concise command language
(CCL) command, you can simply type:

TKB

The Task Builder responds with the prompt TKB> and you type a command. If
TKB has been installed as a CCL command, you can type TKB and the command
on the same line:

TKB command

We describe the format of Task Builder commands below. Note that the Task
Builder allows much flexibility in the way you can specify commands. The
following sections show only the simplest and most direct way. For a detailed
description of all the features available, including command file input to the Task
Builder, see Chapter 10.

2.3.1 Command Line

The Task Builder produces up to three files as output from its analysis of the object files you specify as input. The general form of the command is shown below in lowercase letters:

```
RUN $TKB
TKB>task-file,map-file,symbol-file=object,....,object
TKB>//
```

where:

task-file	is the file specification you give to name the executable program file produced by the Task Builder. If you do not want this file produced, simply type the comma. If you leave off the file type from the file specification, the Task Builder supplies a default type of .TSK.
map-file	is the file specification you give to name the memory map file produced by the Task Builder. This map can be very useful if you are doing overlays; it is not particularly helpful otherwise. See Chapters 3, 4, and 6, where overlays are discussed, for a description of the map file.
	If you do not want this file, simply type the comma delimiter. If you leave off the file type from the file specification, the Task Builder supplies a default type of .MAP.
symbol-file	is the file specification you give to name the symbol-table file produced by the Task Builder. This file is necessary if you want to build your own resident library. It is also used by the COBOL-81 symbolic debugger. It is not useful otherwise. See Chapter 7 for a description of the symbol file.
	If you do not want this file, simply leave out the file specification. If you leave off the file type from the file specification, the Task Builder supplies a default type of .STB.
object,...	are the object files produced from the assembly or compilation of your program and subroutines, plus disk library files containing subroutines needed to complete the program. These files are input to the Task Builder. The Task Builder combines these object files in the order you specify, and resolves cross-references to produce the task file.
	You signify disk library files by appending the switch /LB to the file specification. This notifies the Task Builder that the file named is a library to be searched. The library is searched for routines that resolve references to undefined global symbols in all files to the left of the library file in the input list. So, be sure to put the library to the right of all object files that may contain references to routines in the library. (Usually, you put the library or libraries at the end of the input list.)
	If you do not specify file types, the Task Builder assumes a default type of .OBJ for object files and a default type of .OLB for object libraries.
	If you give a device or project-programmer number in a file specification in the input list (to the right of the equal sign), it applies to all file specifications to the right in the list.

Consider a build using MACRO object programs, for example. Assuming that TKB has been installed as a concise command language (CCL) command, a suitable command line is:

```
TKB EXE1,EXE1,EXE1=OBJ1,OBJ2,LB:RMSLIB/LB
```

The Task Builder constructs the executable file EXE1.TSK, the map file EXE1.MAP, and the symbol table file EXE1.STB from the files OBJ1.OBJ, OBJ2.OBJ, and relevant modules from the library LB:RMSLIB.OLB. (The relevant modules are those referenced in your program. You may have referred to them in source statements, or the MAC assembler may have translated source statements into calls referring to this library.)

To omit the map file, type:

```
TKB EXE1,,EXE1=OBJ1,OBJ2,LB:RMSLIB/LB
```

To produce only the executable file, type:

```
TKB EXE1=OBJ1,OBJ2,LB:RMSLIB/LB
```

To produce no output files, type:

```
TKB=OBJ1,OBJ2,LB:RMSLIB/LB
```

The example above is useful if you are running the Task Builder only to see error messages; that is, a diagnostic run.

Note how project-programmer numbers and device designators work when they are given for a file specification in the input list:

```
TKB=OBJ1,[2,243]OBJ2,OBJ3,LB:RMSLIB/LB,MYLIB/LB
```

For this command, the Task Builder would search for the file OBJ1.OBJ in the user's account and for the files OBJ2.OBJ and OBJ3.OBJ in the account [2,243]. The project-programmer number also applies to the library; that is, the Task Builder would look on the system library disk for a file RMSLIB.OLB under the account [2,243]. Likewise, since the device name LB: also applies to MYLIB, the Task Builder looks on the system library disk under account [2,243] for the library file MYLIB.OLB.

If you do not want this to happen, respecify the project-programmer number and device that you want to apply to remaining files. The simplest way to accomplish this is to assign a logical name to the account [2,243] and use the system-wide logical SY: to "get back to" your account on the public disk structure. For example:

```
ASSIGN SY:[2,243] JOHN
Ready
TKB=OBJ1,JOHN:OBJ2,SY:OBJ3,LB:RMSLIB/LB,SY:MYLIB/LB
```

This can also be accomplished using multiline commands, as shown in the following section.

2.3.2 Multiline Command

Because you can specify any number of input files to the Task Builder, you sometimes need to use more than one line to enter a command.

If you type RUN $TKB or just TKB, so that the Task Builder prompts with TKB>, it continues prompting for input until it receives a line consisting only of two slash characters (//). For example:

```
RUN $TKB
TKB> IMG1,IMG1,IMG1=SY:[2,243]FILE1
TKB> FILE2,FILE3,LB:RMSLIB/LB
TKB> MYLIB/LB
TKB> //
```

The above sequence produces the same result as the single-line command:

```
TKB IMG1,IMG1,IMG1=JOHN:FILE1,SY:FILE2,FILE3,LB:RMSLIB/LB,SY:MYLIB/LB
```

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

2.3.3 Options

You may need to specify options to build a particular program. An option modifies the action taking place during the build. To include options, you must use the multiline format as shown below. When you type a line consisting of a single slash (/), the Task Builder assumes that the last input file has been entered and prompts for options by displaying "ENTER OPTIONS:" and another "TKB>" prompt.

```
RUN $TKB
TKB>command
TKB>continued-command
TKB>/
ENTER OPTIONS:
TKB>option=value:value
TKB>//
```

The format for options is shown here because some languages require certain options for a Task Build. If your language manual set includes a user's guide, you will probably find helpful pointers about necessary or particularly useful options for your language. Table 12–1 in the Reference Section of this manual (Part IV) gives an overview of all the options available for the Task Builder. The options are then described in detail in the remainder of Chapter 12.

The options you will probably find most useful regardless of source language are RESLIB and LIBR. You need to use these options if you need to link to one or more resident libraries. Since resident libraries are commonly used, these options are discussed in the following section. Some examples of these and other options are shown in Section 2.4.

2.3.4 The LIBR and RESLIB Options

You can link to a maximum of seven user mode resident libraries using the Task Builder on RSTS/E systems. Supervisor mode resident libraries are explained in Chapter 9. With either the LIBR or RESLIB option, you specify that you want to link your program to one resident library. The choice between LIBR or RESLIB depends on whether the library is "system-owned" or "user-owned."

The LIBR option declares that your program intends to access a "system-owned" resident library. "System-owned" simply means that the file containing the library is located in the library account (LB:). This can be any account on any disk, as assigned by the system manager.

"User-owned" means that the library can be on some disk or account other than LB:. With the RESLIB option, you specify the disk containing the resident library files.

The formats for the options are:

LIBR=name:access-code[:apr]

RESLIB=file-specification/access-code[:apr]

Note that with the LIBR option, you name only the resident library. The Task Builder looks for the appropriate files (name.STB and name.TSK) on the system library disk (LB:) when it is building the code necessary to load the resident library. With the RESLIB option, you specify a complete file specification. This names the device, account, and file name of the executable file to be loaded. You do not specify the file type. The Task Builder uses the executable file and the symbol table file for the library, and requires that they have file types of .TSK and .STB.

The access-code is either RW (read/write) or RO (read-only), indicating how your program intends to access the library. (It will be RO for Digital-provided resident libraries such as RMSRES.)

The Active Page Register (APR) parameter is an integer in the range of 1 to 7 that specifies the first APR reserved for the library. If you leave this parameter off, the Task Builder assigns the highest APR it can to the resident library.

It is not really necessary to understand Active Page Registers to understand or use the APR modifier. Think of your 32K word user job area as divided into eight parts of 4K words each, numbered from 0 through 7. Your program occupies one or more of the lowest-numbered segments.

You can "map" resident libraries into the area between the top of the program and the highest address of virtual memory. The map must begin on a 4K-word boundary. For example, suppose your program takes 6K words and the run-time system takes 4K words of memory. You can map up to 20K words of resident library into your job, beginning with APR 2.

NOTE

With the use of advanced programming techniques, it is possible to use resident libraries with certain run-time systems other than RSX. However, Digital supports the use of resident libraries only under the RSX run-time system.

2.3.5 The CLSTR Option

You can use the CLSTR option if you need to use more than one resident library. CLSTR lets multiple resident libraries share the same virtual address space in your program. However, not all resident libraries available with RSTS/E can take advantage of this feature. Table 2-2 lists those libraries to which CLSTR can apply.

Table 2-2: Applicable Libraries for the CLSTR Option

Disk Library Name	Description
BP2RES	Clusterable resident library for BASIC-PLUS-2 programs.
BP2SML	Clusterable resident library (a subset of BP2RES) for BASIC-PLUS-2 programs.
C81CIS	Clusterable resident library for COBOL-81 programs compiled with the /CIS switch (normal default if your computer has the Commercial Instruction Set [CIS] option).
C81LIB	Clusterable resident library for COBOL-81 programs compiled with /-CIS switch (normal default if your computer does not have the CIS option).
DIBOLR	Clusterable resident library for RMS DIBOL programs.
F4PCLS	Clusterable resident library for RMS FORTRAN-77 programs.
FDVRDB	Clusterable resident library for the FMS form driver with debug mode support.
FDVRES	Clusterable resident library for the FMS form driver without debug mode support.
RMSRES	Clusterable resident library for RMS-11 that supports sequential, relative, and indexed file operations.
DAPRES	Clusterable resident library for network record access through RMS.
SMRES	Clusterable resident library for SORT/MERGE.

Refer to the documentation for your specific languages to see whether their libraries can cluster.

Figure 2-4 illustrates the concept of cluster libraries. In the figure, three libraries form a cluster for the user program: LIB1, LIB2, and LIB3. LIB1 is the "default library"; that is, it is mapped into the high end of the user program's address space before any calls have been made to any library at execution time.

Figure 2-4 also illustrates that at "time 2" a call is executed to a routine in LIB3. LIB1 is unmapped from the high-address space, and LIB3 is mapped, so the routine can be executed. When control passes from the library routine back to the user program (time 3), LIB3 is unmapped, and LIB1 (the default library) is mapped again. At time 4, a call is executed to a routine in LIB2; again, LIB1 is unmapped and LIB2 is mapped to the high-address space.

This process of mapping and unmapping proceeds throughout execution of the user program. The resident libraries forming a cluster share the same high-address space in the job area (virtual address space). They take much less space from the user program than they would if all three libraries were mapped to the virtual address space at the same time.

Figure 2–4:   Clustered Resident Libraries

USER
VIRTUAL
ADDRESS SPACE

PHYSICAL
MEMORY

USER
VIRTUAL
ADDRESS SPACE

PHYSICAL
MEMORY

USER PROGRAM		LIB1 (default library)
LIB1		LIB3
		LIB2

Time 1

USER PROGRAM		LIB1 (default library)
LIB3		LIB3
		LIB2

Time 2

USER
VIRTUAL
ADDRESS SPACE

PHYSICAL
MEMORY

USER
VIRTUAL
ADDRESS SPACE

PHYSICAL
MEMORY

USER PROGRAM		LIB1 (default library)
LIB1		LIB3
		LIB2

Time 3

USER PROGRAM		LIB1 (default library)
LIB2		LIB3
		LIB2

Time 4

MK–01048–00

To use cluster libraries, use the CLSTR option. The format is:

CLSTR=default-library,library-2,...,library-5:access-code[:apr]

The first library listed in the CLSTR option is the default library. Because of the way clustering works, only certain libraries can be default libraries. If you want to build libraries to be clusterable, the techniques are described in Chapter 7. If you simply want to use libraries in a resident library cluster, the Digital-supplied libraries are designed so the language library can always serve as the default library.

Thus, for the resident libraries listed previously, you can use either BP2SML or BP2RES for BASIC-PLUS-2 programs, or C81CIS or C81LIB for COBOL-81 programs. As a secondary library in the cluster, you can use either FDVRES or RMSRES or both.

Up to five resident libraries can form a cluster. A cluster for Digital-supplied libraries must occupy the upper 8K words of your address space. If your site builds its own clusterable libraries, however, these libraries can occupy their own separate cluster, as long as the limit of five resident libraries for each task build is not exceeded. (You can have no more than five libraries involved in clusters.)

Thus, you can cluster either of two variations of the COBOL-81 library (C81CIS or C81LIB) with the FMS library (FDVRES) and/or the RMS library (RMSRES), and any two of your own clusterable libraries either in the same cluster or in a separate cluster in lower virtual address space.

Likewise, you can cluster BP2RES or BP2SML with the RMS (RMSRES) or FMS (FDVRES) libraries or both, together with any two of your own clusterable libraries.

The access-code is either RW (read/write) or RO (read-only). This code is an attribute of the library itself. That is, you could not select RW (indicating your program can read from or write to the library) if the library has been built RO. The access-code is RO for Digital-provided resident libraries such as BP2RES, FDVRES, C81CIS, and C81LIB. For example:

```
TKB>CLSTR=C81CIS,FDVRES,RMSRES:RO
```

The Active Page Register (APR) parameter is an integer in the range of 1 to 7 that specifies the first APR reserved for the clustered libraries. If you omit this parameter, the Task Builder assigns the highest APR it can to the cluster (APRs 6 and 7 for the command line above).

Currently, Digital-supplied libraries are built to use the top two APRs available to the cluster. APRs are assigned according to the following guidelines:

1. If the language library is part of a cluster, the cluster will occupy APRs 6 and 7. (You need not specify an APR parameter.)

2. If the language library is not part of a cluster and occupies the top two APRs, such as the BP2SML resident library, the cluster will occupy APRs 4 and 5. (You specify an APR parameter of 4.) This description applies mainly to users who are building their own cluster libraries.

3. If a run-time system occupies the top APR (7), the cluster will occupy APRs 5 and 6. (You specify an APR parameter of 5.)

2.4 Examples of Simple Builds

The examples in this section illustrate building programs in various languages and with various kinds of libraries. Note that in all the examples, an executable program file is requested. You might want to request the other files once to see what they look like. For these simple builds, however, neither the map file nor symbol table file are particularly useful. Map files become useful when you are working with overlays; they are described in Chapters 3, 4, and 6. Symbol table files are chiefly useful when you are constructing your own resident libraries (Chapter 7), or when you are using the COBOL-81 symbolic debugger.

2.4.1 BASIC-PLUS-2 Examples Including Disk, Resident, and Cluster Libraries

Note that RSX directive emulation code must be installed on your system in order to use BASIC-PLUS-2 V2.0.

To build a BASIC-PLUS-2 program using disk and resident libraries, you can type:

```
RUN $TKB
TKB> PROG=OBJ1,OBJ2,OBJ3,LB:BP2OTS/LB
TKB> /
ENTER OPTIONS:
TKB> LIBR=BP2SML:RO
TKB> LIBR=RMSRES:RO
TKB> UNITS=12
TKB> ASG=SY:5:6:7:8:9:10:11:12
TKB> EXTTSK=512
TKB> //
```

The first line tells the Task Builder to create the task image file, named PROG.TSK. The object programs are OBJ1.OBJ, OBJ2.OBJ, and OBJ3.OBJ. The /LB switch references the BP2OTS library. LB: is the system library device, and the Task Builder assumes a default file type of .OLB for libraries.

You end the command line and indicate that you want to enter options by typing a single slash (/) on a separate line. The Task Builder responds with ENTER OPTIONS: and another TKB> prompt. You then enter the LIBR option, designating BP2SML as the resident library to be mapped read-only. RMSRES is the RMS resident library; it also is to be mapped read-only. (Symbols not resolved by the resident library, BP2SML, will be resolved by BP2OTS.OLB.)

The UNITS option declares the maximum number of I/O channels (units) that your program will use. The ASG option relates these channels to devices. For instance, the following example shows a maximum of twelve channels are used by the program. Defaults are accepted for channels 1 through 4. Channels 5 through 12 are the public structure (SY:). EXTTSK allocates an additional 512 words of memory to your program. You then end Task Builder input by typing two slash characters (//) on a separate line.

This BASIC-PLUS-2 example shows the use of cluster libraries:

```
RUN $TKB
TKB> MYPROG=PROG1,SUB1,SUB2,LB:BP2OTS/LB
TKB> /
ENTER OPTIONS:
TKB> CLSTR=BP2RES,RMSRES:RO
TKB> UNITS=12
TKB> ASG=SY:5:6:7:8:9:10:11:12
TKB> EXTTSK=512
TKB> //
```

In this example, you request the executable file MYPROG.TSK, consisting of the object modules PROG1.OBJ, SUB1.OBJ, and SUB2.OBJ. The resident libraries BP2RES and RMSRES are to be built to form a cluster using the upper 8K words of address space (APRs 6 and 7). The libraries are to be mapped read-only. The language library BP2OTS is the default library.

2.4.2 PDP-11 COBOL Example Including Two Disk Libraries

To build a PDP-11 COBOL program, you can type:

```
RUN $TKB
TKB> OUT=PROG,SUB,SUB2,LB:COBLIB/LB,LB:RMSLIB/LB
TKB> //
```

This command tells the Task Builder to create one file, the executable file, named OUT.TSK. The compiled object programs are PROG.OBJ, SUB.OBJ, and SUB2.OBJ. Two libraries are referenced; COBLIB.OLB and RMSLIB.OLB. The /LB switch indicates that the libraries are located in the library account (LB:).

2.4.3 COBOL-81 Examples Including Disk Library and Cluster Libraries

The following example illustrates building a COBOL-81 program:

```
RUN $TKB
TKB> FINAL=PROG1,PROG2,LB:C81CIS/LB
TKB> //
```

With this command, the Task Builder creates the executable file FINAL.TSK from the compiled object programs PROG1.OBJ and PROG2.OBJ, and from necessary routines from the library for the Commercial Instruction Set (CIS), C81CIS.OLB.

The second example for COBOL-81 shows the use of cluster libraries:

```
RUN $TKB
TKB> FINAL=PROG1,PROG2,LB:C81CIS/LB
TKB> /
ENTER OPTIONS:
TKB> CLSTR=C81CIS,FDVRES,RMSRES:RO
TKB> //
```

In the example above, you request the executable file FINAL.TSK, consisting of the object modules PROG1.OBJ and PROG2.OBJ. The /LB switch references the disk library C81CIS.OLB. The resident libraries C81CIS, FDVRES, and RMSRES are to be built to form a cluster using the upper 8K words of address space (APRs 6 and 7). The libraries are to be mapped read-only. The language library C81CIS is the default library. Note that while C81CIS in the command line refers to the disk library, C81CIS in the CLSTR option refers to the resident library.

2.4.4 DIBOL Example Including Disk and Resident Libraries

The following example illustrates building a typical RMS DIBOL program:

```
RUN $TKB
TKB> PAY=HOURS,EMPLK,CHECK,MYLIB/LB,LB:DBRLIB/LB
TKB> /
ENTER OPTIONS:
TKB> LIBR=DBRRES:RO:4
TKB> LIBR=RMSRES:RO:6
TKB> //
```

This example requests the executable file PAY.TSK. The object modules used are HOURS.OBJ, EMPLK.OBJ, and CHECK.OBJ. Modules are included from the library MYLIB.OLB (on the system disk in your account) and the library DBRLIB.OLB (in the system library account LB:). DBRRES is the DIBOL resident library for RMS; it is to be mapped read-only, beginning in APR 4. RMSRES is the RMS resident library; it also is to be mapped read-only, beginning in APR 6.

2.4.5 FORTRAN-77 Examples Including One Disk Library

To build a FORTRAN-77 program, you can type:

```
RUN $TKB
TKB> BURNS=KNIGHT,DAY,LB:F4POTS/LB
TKB> //
```

This example requests an executable file named BURNS.TSK. The files KNIGHT.OBJ and DAY.OBJ are the compiled files to be used, along with referenced routines from the library F4POTS.OLB.

2.4.6 MACRO Examples Including Resident Libraries

The following examples show the use of the LIBR and RESLIB options.

The first uses LIBR:

```
RUN $TKB
TKB> FINAL=FINAL,SUB1,SUB2
TKB> /
ENTER OPTIONS:
TKB> LIBR=RMSRES
TKB> //
```

This example requests the executable file FINAL.TSK, constructed using the compiled files FINAL.OBJ, SUB1.OBJ, and SUB2.OBJ. The resident library RMSRES is linked in also. Note that in this case, no APR is given; the Task Builder will use APRs 6 and 7 for RMSRES, so the system manager must have installed the system with RSX directive emulation code as a part of the monitor.

The LIBR option is used because the files RMSRES.TSK and RMSRES.STB are located on the system library device (LB:).

The next example uses the RESLIB option:

```
RUN $TKB
TKB> FINAL=FINAL,SUB1,SUB2
TKB> /
ENTER OPTIONS:
TKB> RESLIB=DR0:[1,150]RMSRES
TKB> //
```

The same requests are made as in the previous example. In this case, the files RMSRES.TSK and RMSRES.STB are located on the device DR0: in account [1,150]. The RESLIB option is used instead of LIBR because the library is not in LB:.

Part II
Overlays
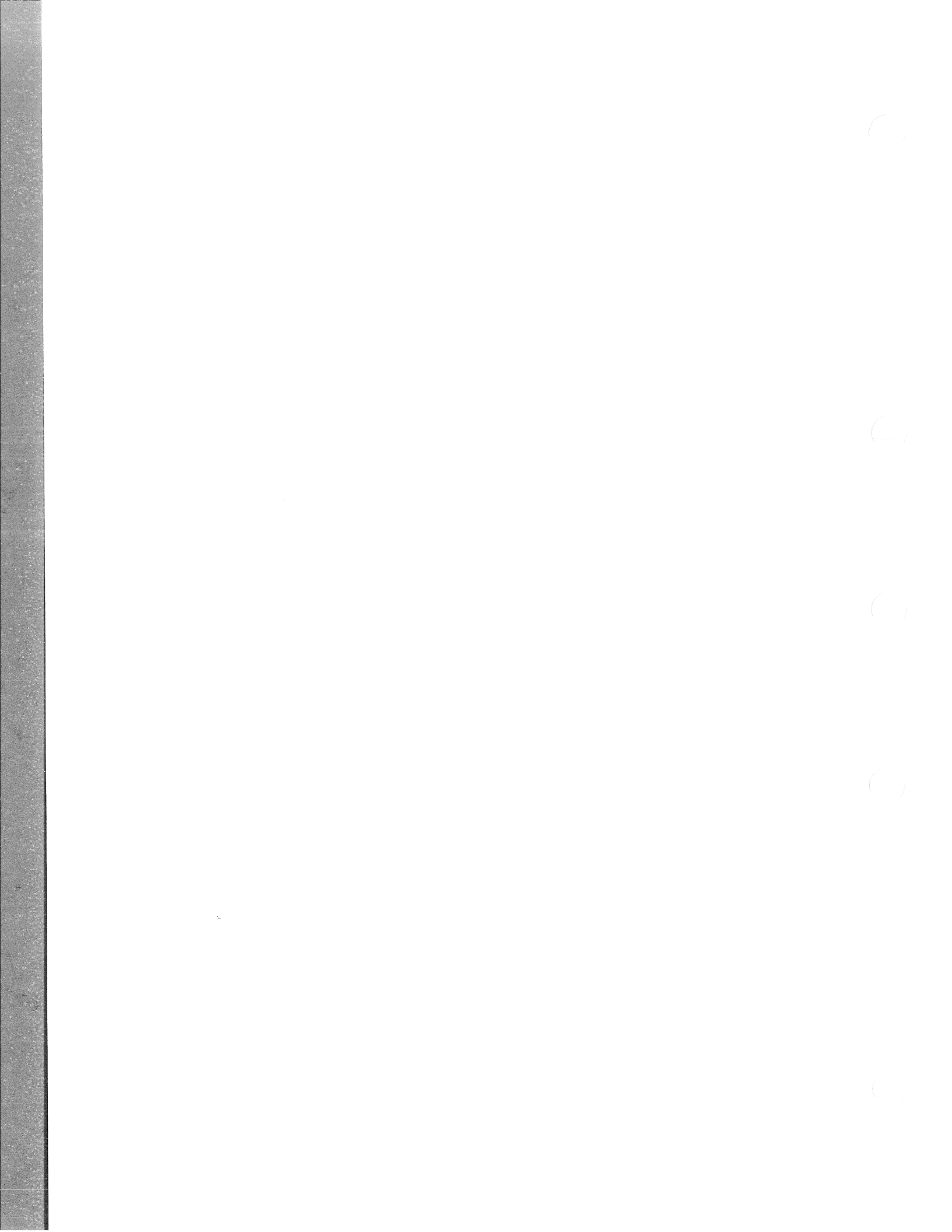
The Basic Concepts

If your program is too large to fit in the space available, you must specify an overlay structure for it. The easiest way to find out if your program is too large is to try to build it, using the steps outlined in Chapter 2. If you get the following error message, your program is too large:

```
?Task has illegal memory limits
```

Languages that can dynamically allocate memory (such as BASIC-PLUS-2) may not give this error at task build. Rather, they may produce another message at run time, such as:

```
?Maximum memory exceeded
```

This chapter tells how to specify an overlay structure to eliminate this problem. You design an overlay structure, such as the diagram in Figure 3-1, and describe the structure to the Task Builder using an "ODL file"; a file written in the Overlay Description Language.

Figure 3-1: The ODL File Is Your "Blueprint" for Overlays


```
        .ROOT MAIN-*(AWL, BWL, CWL)
AWL:    .FCTR A-LIB- (A1-LIB, A2-LIB)
BWL:    .FCTR B-LIB
CWL:    .FCTR C-LIB
LIB:    .FCTR LB:BP2OTS/LB
        .END
```
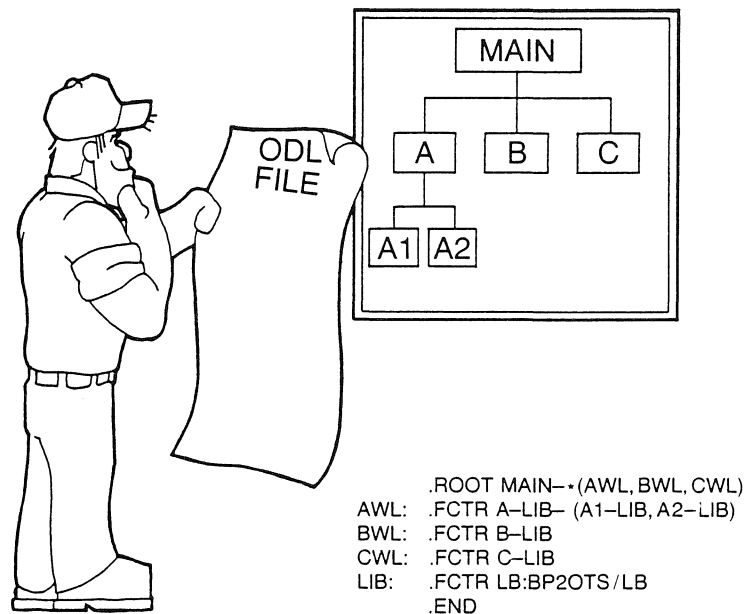
MK-00573-00

COBOL PROGRAMMERS NOTE

You cannot use the specific techniques described in this chapter to construct overlays. In COBOL, you begin working with overlays within the language itself by using the segmentation facility of the COBOL compiler. Techniques are described in the *PDP-11 COBOL User's Guide* and the *RSTS/E COBOL-81 User's Guide*.

COBOL programmers may want to read this chapter to get an idea of what the PDP-11 COBOL or COBOL-81 compiler and MRG utility (for PDP-11 COBOL) or BLDODL utility (for COBOL-81) are doing for you. Chapter 6 describes overlays in terms of program sections and may also be of interest to you.

3.1 What are Overlays?

The best way to explain overlays is by example. Suppose that the program you have written consists of a main program (called MAIN) and two separately compiled subroutines (called SUB1 and SUB2). Suppose further that MAIN calls both SUB1 and SUB2, and that neither SUB1 nor SUB2 contain any calls to separately compiled subroutines or to MAIN (see Figure 3–2).

Figure 3–2: Outlining the Call Structure
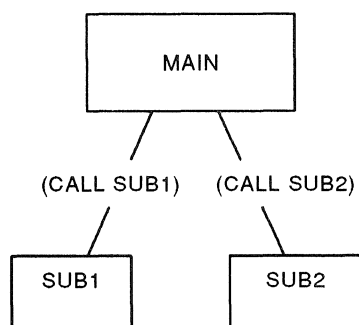
MK-00574-00

You can specify an overlay structure such that the run-time system (described in Section 2.1) loads MAIN when the program is first run. When MAIN calls SUB1, code built into MAIN by the Task Builder loads SUB1 for execution. Then, when control passes back to MAIN and it calls SUB2, the loading code that was built into MAIN brings SUB2 into memory overlaying SUB1 (see Figure 3–3).

Note that SUB1 and SUB2 do not call or use data from each other. This "logical independence" is necessary for program pieces that overlay each other. In this example, calls to routines or references to data that are not currently in memory must be made from the "root": the MAIN program.

Figure 3–3: A Simple Overlay in Memory

```
┌────────────────────┐        ┌────────────────────┐
│                    │        │                    │
│        MAIN        │        │        MAIN        │
│                    │        │                    │
├────────────────────┤        ├────────────────────┤
│                    │        │                    │
│        SUB1        │        │                    │
│                    │        │        SUB2        │
├────────────────────┤        │                    │
│                    │        │                    │
│  (UNUSED MEMORY)   │        ├────────────────────┤
│                    │        │  (UNUSED MEMORY)   │
│                    │        │                    │
└────────────────────┘        └────────────────────┘


         TIME 1  ──────────────────▶  TIME 2              MK–00575–00
```
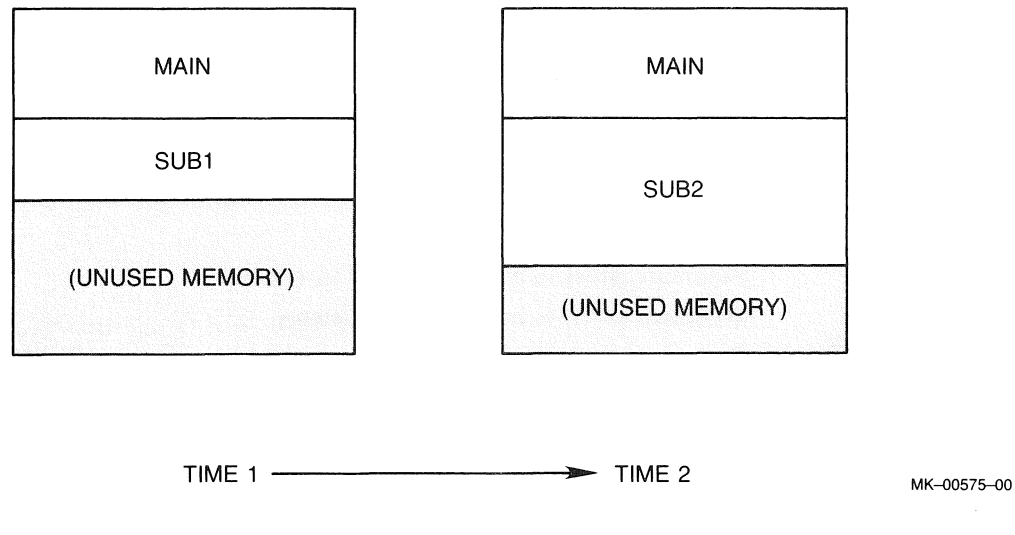
3.2 Constructing an ODL File: .ROOT, .FCTR, and .END Commands

To define an overlay structure to the Task Builder, you construct an "overlay map": a file consisting of instructions written in a language called the "Overlay Description Language." This file is often referred to as an ODL file.

Three commands form the heart of the Overlay Description Language: .ROOT, .FCTR, and .END. To give you an idea of its simplicity, here is an ODL file for the example shown in Figure 3–2:

```
          .ROOT   MAINWL-*(SUB1WL,SUB2WL)
MAINWL:   .FCTR   MAIN-LIBR
SUB1WL:   .FCTR   SUB1-LIBR
SUB2WL:   .FCTR   SUB2-LIBR
LIBR:     .FCTR   LB:BP2OTS/LB
          .END
```

The .ROOT, .FCTR, and .END commands for this example are described in the following sections.

3.2.1 The .ROOT Command

Every ODL file has one and only one .ROOT command; this command describes the entire overlay structure. In the example at the start of Section 3.2, the .ROOT command defines the entire structure in terms of "factors" defined in following .FCTR commands. This is simply for the convenience of saving space in the command line. You could have referred to the actual object files MAIN, SUB1, SUB2, and the library LB:BP2OTS in the .ROOT command and eliminated the .FCTR commands entirely (see Section 3.2.4). However, the .ROOT command would have been long and somewhat hard to read and interpret.

The syntax of the .ROOT and .FCTR commands defines the overlay structure. The first item following the .ROOT command indicates the root item, to be assigned the lowest virtual addresses:

```
.ROOT  MAINWL-*(SUB1WL,SUB2WL)
```

The root item in this example is MAINWL. This item—named to denote "MAIN With Library"—is defined in the following .FCTR command:

```
MAINWL:   .FCTR  MAIN-LIBR
```

For the moment, however, consider the .ROOT command. The following symbols define the structure of the overlay:

-    Separates pieces to be concatenated in memory
,   Separates pieces to be overlaid in memory
( )   Groups pieces to be overlaid

Thus, the hyphen in the .ROOT command indicates that MAINWL is to be concatenated with the structure (SUB1WL,SUB2WL). The parentheses indicate grouping; they enclose items that are to overlay each other. The structure inside—SUB1WL,SUB2WL—indicates SUB1WL and SUB2WL are to occupy the same space, or overlay each other as necessary.

In other words, a comma separating two or more items within parentheses indicates that they are to overlay each other. A dash between two items indicates they are to be concatenated, with the item on the left assigned the lowest addresses.

The asterisk (*) symbol shown in the example is an autoload indicator. It does not affect the overlay structure, although it is very important. It tells the Task Builder to generate what are called autoload vectors to ensure that overlay pieces can be loaded properly when the program is executed.

The use of asterisks is discussed in detail in Chapter 5; you can save a little space in your program if you use them carefully. However, the simplest rule, and one that always ensures proper loading for overlay structures described in this chapter, is to put an asterisk before the outermost left parenthesis in your ODL file.

3.2.2 The .FCTR Command

Consider the example under discussion again:

```
          .ROOT  MAINWL-*(SUB1WL,SUB2WL)
MAINWL:   .FCTR  MAIN-LIBR
SUB1WL:   .FCTR  SUB1-LIBR
SUB2WL:   .FCTR  SUB2-LIBR
LIBR:     .FCTR  LB:BP2OTS/LB
          .END
```

MAINWL, SUB1WL, and SUB2WL are all defined as factors in the lines following the first line. The term "factor" is used in the sense of "ingredient." That is, .FCTR commands are used to further define elements used in a .ROOT command or a preceding .FCTR command.

Note that the names used in the .ROOT command are defined in each .FCTR command by the first field: the name terminated by a colon. Likewise, the name LIBR, used in several of the .FCTR commands, is defined in the last .FCTR command. In general, factor names can consist of 1-6 characters from the set A-Z, 0-9, and the dollar sign ($).

The .FCTR command also specifies an overlay structure; the same items and operators used in a .ROOT command can also be used in a .FCTR command. In the example, the first three .FCTR commands consist of two items separated by a hyphen. Again, the hyphen separating two items means that the first item is assigned the lowest addresses, and the second item is to be concatenated following the first.

In the example shown at the start of Section 3.2, however, the concatenated item is LIBR, defined by a later .FCTR command as the BP2OTS library. When an item in a hyphenated series is a file with the /LB switch, it means that the first item's unresolved references are to be resolved from routines within that disk library. In other words, the entire library is not concatenated. Only those routines referenced are actually concatenated and added to the executable file. The items MAIN, SUB1, and SUB2 are the compiled or assembled object files. As with a simple build, the default file type for such files is .OBJ. The default file type for a file with the /LB switch is .OLB.

Note that a .FCTR command can contain an item defined in another .FCTR command. In general, .FCTR commands can be "nested" in this fashion up to 16 levels.

3.2.3 The .END Command

The .END command ends the ODL file; every ODL file must have one .ROOT command and end with .END.

3.2.4 Flexibility of the Overlay Description Language

From the preceding discussion, you probably have observed that there are many ways to construct ODL files using the three basic commands and their operators. For example, the following ODL file has the same effect as the example in Section 3.2:

```
.ROOT MAIN-LB:BP2OTS/LB-*(SUB1-LB:BP2OTS/LB,SUB2-LB:BP2OTS/LB)
.END
```

The ODL file above has no .FCTR statements. The following file also produces the same structure as the example at the beginning of Section 3.2:

```
        .ROOT MAIN-LIBR-*(SUB1-LIBR,SUB2-LIBR)

LIBR:   .FCTR LB:BP2OTS/LB
        .END
```

3.3 Using an ODL File When You Run TKB

To tell the Task Builder to build a program according to the structure specified in an ODL file, you simply give the ODL file name with the switch /MP instead of the object files in an ordinary command line. For example, to build the program described in Sections 3.1 and 3.2, you can type:

```
RUN $TKB
TKB>MYPROG,MPFILE=OVERLY/MP
ENTER OPTIONS:
TKB>LIBR=BP2RES:RO
TKB>UNITS=12
TKB>ASG=SY:5:6:7:8:9:10:11:12
TKB>EXTTSK=512
TKB>//
```

When you specify /MP on the input file for your task, it must be the only input file that you specify. Note that when you specify an ODL file, TKB automatically prompts for option input. Therefore, do not use the single slash (/) to direct TKB to prompt for options when you specify /MP on your input file.

The /MP switch indicates the file is an "overlay map," or ODL file. The default file type for files with the /MP switch is .ODL. Thus, the file used here as an overlay map is named OVERLY.ODL, on the system disk in the user's account.

The LIBR option declares that your program will access the resident library BP2RES. UNITS, ASG, and EXTTSK are other options often useful with BASIC-PLUS-2 programs. For another language, use the appropriate command as described in Chapter 2.

Note that you request a map file in this example. The map file is very useful when working with overlays.

3.4 The Memory Map File

This section discusses how to determine the size of the programs and subprograms you want to overlay.

Suppose that the build in Section 3.3 produces the error "SEGMENT seg-name HAS ADDR OVERFLOW: ALLOCATION DELETED". The program is too large; you must reexamine it. Now, though, you have an important tool: the memory map file, called in this case MPFILE.MAP. The first page of this map is shown in Example 3–1. Note the highlighted section, titled "MYPROG.TSK OVERLAY DESCRIPTION".

This section of the memory allocation map appears only if you request overlays by using the /MP switch appended to an ODL file specification. To get this information, then, you must specify the most reasonable overlay structure possible without actually knowing the length of the pieces.

In the first three columns, this section gives, in octal, the base address, top address, and length, in bytes, of each overlay piece. The most relevant information is given in the second of the two LENGTH columns: the decimal length of the piece, in bytes. The MAIN program, for example, is 49,152 bytes long. SUB1 is 34,164 bytes, and SUB2 is 16,384 bytes.

You are building the program to run under the RSX run-time system, which allows 32K words, or 64K bytes for your program. Thus, you must restructure the program to divide MAIN into further pieces to be overlaid. At 49152 bytes, MAIN with SUB1 and SUB2 is too large to fit in the space available. To do this intelligently, however, you need to know more about the Task Builder.

Note that total task size and task image size show the space allocated for the program minus a calculated overflow.

Example 3–1: Overlay Description of Memory Allocation Map

```
MAIN.TSK    Memory allocation map   TKB 08.006      Page 1
                        19-MAR-90    14:56


Partition name  : GEN
Identification  : 600319
Task  UIC       : [1,234]
Stack     limits: 001000 001777 001000 00512.
PRG xfr address: 012000
Total address windows: 2.
Task    extension   : 512. words
Task    image  size : 25280. words
Total   task   size : 25792. words
Task address limits: 000000 035047
R-W disk blk limits: 000002 000060 000057 00047.



MYPROG.TSK Overlay description:
Base    Top         Length
----    ---         ------
000000  042563 140000   49152.    MAIN
            .           34164.              SUB1
            .           16384.              SUB2

            .


(Other pages of memory map)
```

3.5 Designing Overlays Intelligently: Considering Space and Time

The same considerations are necessary in designing an overlay structure as in other aspects of computing: space and time. Some aspects of the problem of space (how to get the pieces to fit) have been discussed. To do the job well, you must also consider the problem of time: how to get the pieces to fit so that they execute in the least possible time.

3.5.1 Considering Space: Two Possibilities for Example

Suppose that examining the program in our example reveals two possibilities for dividing the program so that the pieces will fit.

In the first case, you divide MAIN into five parts by inserting calls in MAIN in the source code. Now you have a "root" segment, MAIN, with five branches, MAIN1, MAIN2, MAIN3, SUB1, and SUB2. The call structure is outlined in Figure 3–4.

Figure 3–4: Outline of First Call Structure for Example
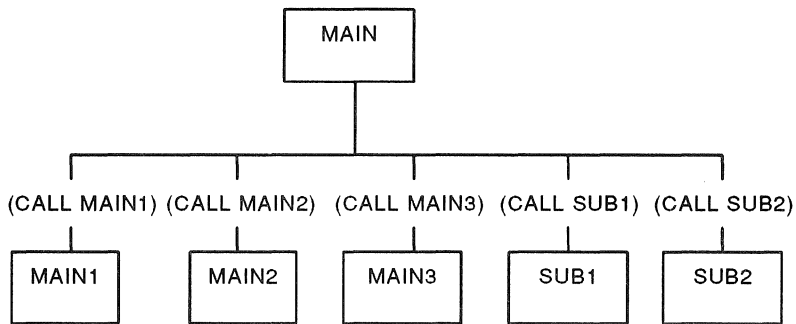
MK–00576–00

Note again the logical independence of the call structure for the items to be overlaid. Defining the overlay structure based on the call structure is one way to ensure the logical independence of the items in the overlay structure. In this case, MAIN1, MAIN2, MAIN3, SUB1, and SUB2 could not call each other or refer to data in each other. These items overlay each other and will not reside in memory at the same time. The ODL file for such a structure could look as follows:

```
        .ROOT MAINWL-*(MAIN1L,MAIN2L,MAIN3L,SUB1L,SUB2L)
MAINWL: .FCTR MAIN-LIBR
MAIN1L: .FCTR MAIN1-LIBR
MAIN2L: .FCTR MAIN2-LIBR
MAIN3L: .FCTR MAIN3-LIBR
SUB1L:  .FCTR SUB1-LIBR
SUB2L:  .FCTR SUB2-LIBR
LIBR:   .FCTR LB:BP2OTS/LB
        .END
```

In the second case, you divide MAIN into two pieces and divide SUB2 into two pieces called SUB2A and SUB2B. The outline for the call structure is shown in Figure 3–5.

Figure 3-5: Outline of Second Call Structure for Example
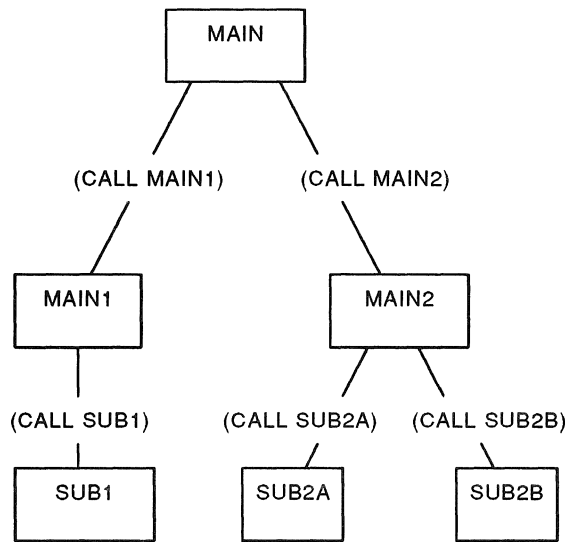
MK-00577-00

The ODL file for such a structure could look as shown below. Note the nested parentheses used to group the pieces that overlay each other. In general, parentheses can be nested to 16 levels.

```
          .ROOT   MAINWL-*(MAIN1L-SUB1L,MAIN2L-(SUB2AL,SUB2BL))
MAINWL:   .FCTR   MAIN-LIBR
MAIN1L:   .FCTR   MAIN1-LIBR
SUB1L:    .FCTR   SUB1-LIBR
MAIN2L:   .FCTR   MAIN2-LIBR
SUB2AL:   .FCTR   SUB2A-LIBR
SUB2BL:   .FCTR   SUB2B-LIBR
LIBR:     .FCTR   LB:BP2OTS/LB
          .END
```

Now suppose you build the program successfully in both of the above cases. The problem with space is resolved with either the structure shown in Figure 3-4 or in Figure 3-5. You would choose the structure that requires the least time to execute, as described in the following section.

3.5.2 Considering Time: Reducing Disk Access

When you ask for overlays, the Task Builder inserts code into your program to load the overlays properly. For the example in Figure 3-4, the Task Builder inserts code into MAIN to load MAIN1, MAIN2, MAIN3, SUB1, and SUB2 from disk into memory when they are called. (MAIN itself is loaded by the run-time system when the program is first run.)

Thus, when MAIN calls MAIN1, the code inserted by the Task Builder is executed to load MAIN1 from disk into memory for execution. When MAIN calls MAIN2, this code is again executed to load MAIN2 from disk into memory, and so forth. These disk accesses take time. You want to design your overlays to reduce the number of disk accesses.

In general, the Task Builder analyzes your ODL file to determine the best way to store pieces on disk so that they can be loaded quickly. It constructs the executable program file in "segments" that are loaded with one disk access. Pieces connected by a dash (-) are stored in one segment. Pieces separated by a comma are stored in separate segments.

Thus, the executable file for the ODL file in Figure 3–4 consists of the main program (loaded by the run-time system) and five segments each requiring a separate disk access for loading. The executable file for the ODL file in Figure 3–5 consists of the main program and four segments. MAIN1 and SUB1, connected by a dash in the ODL file, are stored as one segment. When MAIN calls MAIN1, MAIN1 and SUB1 are loaded together. Then, when MAIN1 calls SUB1, no separate disk access is necessary: SUB1 is already in memory.

MAIN2, SUB2A, and SUB2B are stored as separate segments. Each requires a separate disk access.

Thus, assuming each call is made only once, the structure in Figure 3–4 calls for five disk accesses. The structure in Figure 3–5 calls for four disk accesses. Since both fit, you would choose the structure shown in Figure 3–5.

3.6 Logical Independence of Items in Overlay Structure

This section discusses the need for the logical independence of items in an overlay structure and suggests that basing the overlay structure on the call structure is a reasonable way to approach overlay structure design. If your program has a complex call structure, however, this approach may not be feasible.

You can still visualize the tree-like structure we have shown previously, and you can still specify an overlay structure in terms of separately assembled or compiled program and subroutine files. However, you must consider the sequence of calls these pieces make to each other. In general, you must structure the overlay tree so that calls (or references to data) take place between pieces that are along the same path. Calls or references to data cannot take place between pieces that are along different paths. A path is simply any route from the root of the structure that follows a series of branches to an outermost piece of the tree.

Figure 3–6 shows a structure that is specified by the following ODL commands:

```
        .ROOT    AL-BL-*(CL,FCTR1)
FCTR1:  .FCTR    DL-(EL,FL,GL)
AL:     .FCTR    A-LIBR
BL:     .FCTR    B-LIBR
CL:     .FCTR    C-LIBR
DL:     .FCTR    D-LIBR
EL:     .FCTR    E-LIBR
FL:     .FCTR    F-LIBR
GL:     .FCTR    G-LIBR
LIBR:   .FCTR    LB:F4POTS/LB
        .END
```

Figure 3–6 shows pieces that overlay each other as separate "branches" of the tree. C and D would start at the same virtual address, as would E, F, and G. The paths in this structure are A-B-C, A-B-D-E, A-B-D-E, and A-B-D-G. Calls may be made between pieces on any of these paths. However, F could not call G, E, or C; C could not call D, E, F, or G; and so forth.

Figure 3-6: Separate Paths in an Overlay Structure
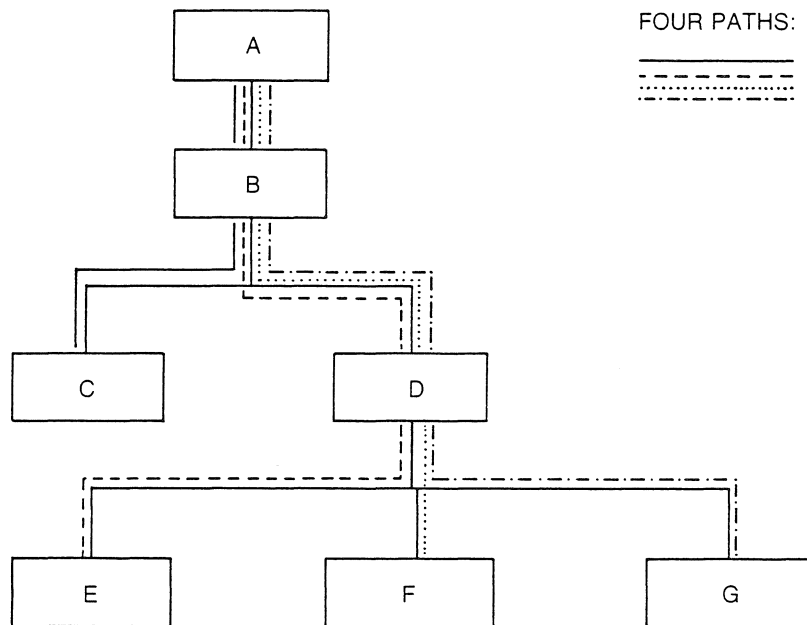
MK-00578-00

3.7 Resolution of Global Symbols

In the last section, it was noted that overlay pieces that are on separate paths cannot call each other or refer to data in each other. If you specify such a structure, the Task Builder gives you error messages about multiply defined or ambiguously defined global symbols. Since these errors can be one of the most frustrating aspects of task building, further clarification of the underlying concepts is necessary.

3.7.1 What Is a Global Symbol?

All languages provide the facility for defining and referring to symbols. In general, a symbol is a name that is eventually translated to an address for a location in computer memory. The location may contain data or a computer instruction.

Symbols can be classified as either local or global. A local symbol is one that is both defined and referenced within one program or subprogram. That is, its definition and usage are in the same (local) area.

A global symbol is one that can be defined in one program or subprogram and referred to by another separately compiled or assembled program or subprogram.

While you may not be aware of it, for example, the FORTRAN compiler defines a name you give to a COMMON area as a global symbol. Similar translations take place in all languages for common areas and entry points to programs and subprograms, including subprograms contained in libraries.

3.7.2 Undefined, Multiply Defined, and Ambiguously Defined Global Symbols

The Task Builder resolves references to global symbols at build time. In general, you can define two global symbols with the same name if they are on separate paths and are not referenced from a piece that is common to both paths.

If you define a global symbol on one path but refer to it on another path, the symbol is diagnosed as undefined where it is referenced.

If you define two global symbols with the same name on the same path, the symbol is multiply defined.

If you define two global symbols with the same name on different paths, but the symbol is referenced from a piece that is common to both, the symbol is ambiguously defined.

Examine the overlay structure in Figure 3–7. The global symbol Q is defined in A0 and B0. The references to Q in A22 and 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; the definitions and references occupy separate paths.

The global symbol R is defined in 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.

The global symbol S is defined in both A0 and B0. References to S from A1, A21, or A22 are resolved by the definition in A0. 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 different definitions of S on separate paths through CNTRL. The global symbol S is ambiguously defined.

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

Figure 3-7: Resolving Global Symbols
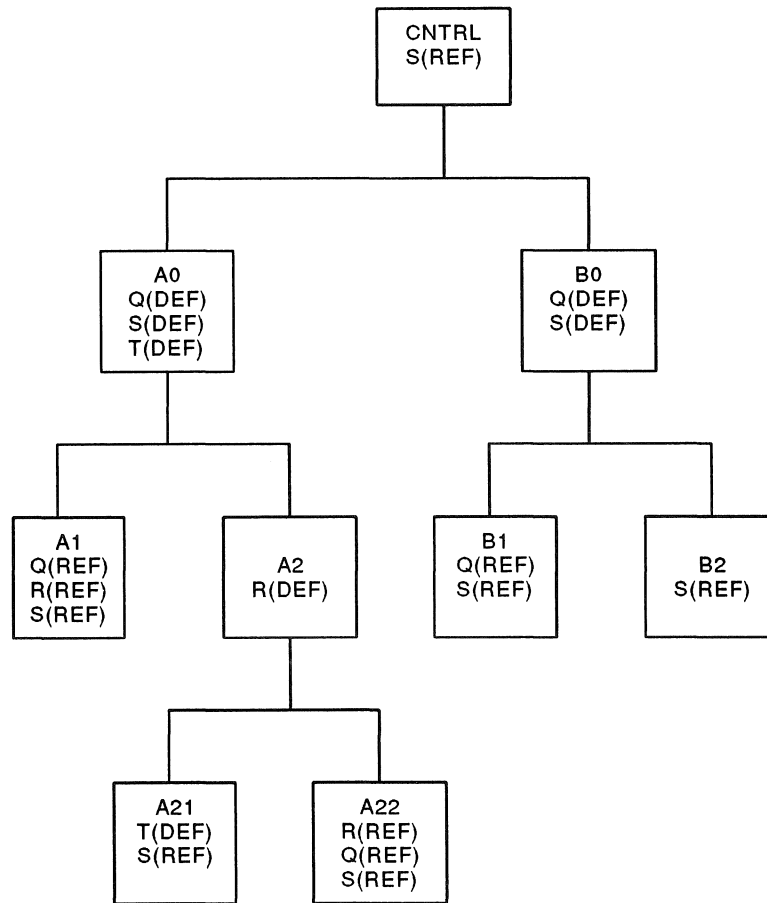
MK-00579-00

3.7.3 How Routines Are Inserted from Libraries

In all the examples so far, we have shown a library concatenated (using the dash in the ODL file) at the end of the root and every segment in the overlay structure. Unless you know which library routines are used by each piece of your program, this is the best way to ensure that library routines are properly inserted from the desired disk libraries. The Task Builder then ensures that routines referred to by more than one piece are accessible to all pieces. For example, consider the following ODL file for the overlay structure shown in Figure 3-8:

```
        .ROOT   ROOTL-*(AL,BL-(B1L,B2L))
ROOTL:  .FCTR   ROOT-LIBR
AL:     .FCTR   A-A1-LIBR
BL:     .FCTR   B-LIBR
B1L:    .FCTR   B1-LIBR
B2L:    .FCTR   B2-LIBR
LIBR:   .FCTR   LB:F4POTS/LB
        .END
```

Figure 3–8: Resolving Global Symbols from Disk Libraries


```
                    ┌─────────────┐
                    │    ROOT     │
                    │  REF $READ  │
                    │  REF $WRITE │
                    ├─────────────┤
                    │    $READ    │
                    ├─────────────┤
                    │   $WRITE    │
                    └─────────────┘
```

┌──────────────┐   ┌──────────────┐
│   A   │   │   B   │
│ REF $READ   │   │ REF $READ   │
│ REF $ASIN   │   │ REF $ASIN   │
├──────────────┤   │ REF $COS   │
│   A1   │   ├──────────────┤
│ REF $WRITE │   │ $ASIN   │
├──────────────┤   ├──────────────┤
│ $ASIN   │   │ $COS   │
└──────────────┘   └──────────────┘

            ┌──────────────┐   ┌──────────────┐
            │   B1   │   │   B2   │
            │ REF $COS   │   │ REF $READ   │
            │ REF $ABS   │   └──────────────┘
            ├──────────────┤
            │ $ABS   │
            └──────────────┘
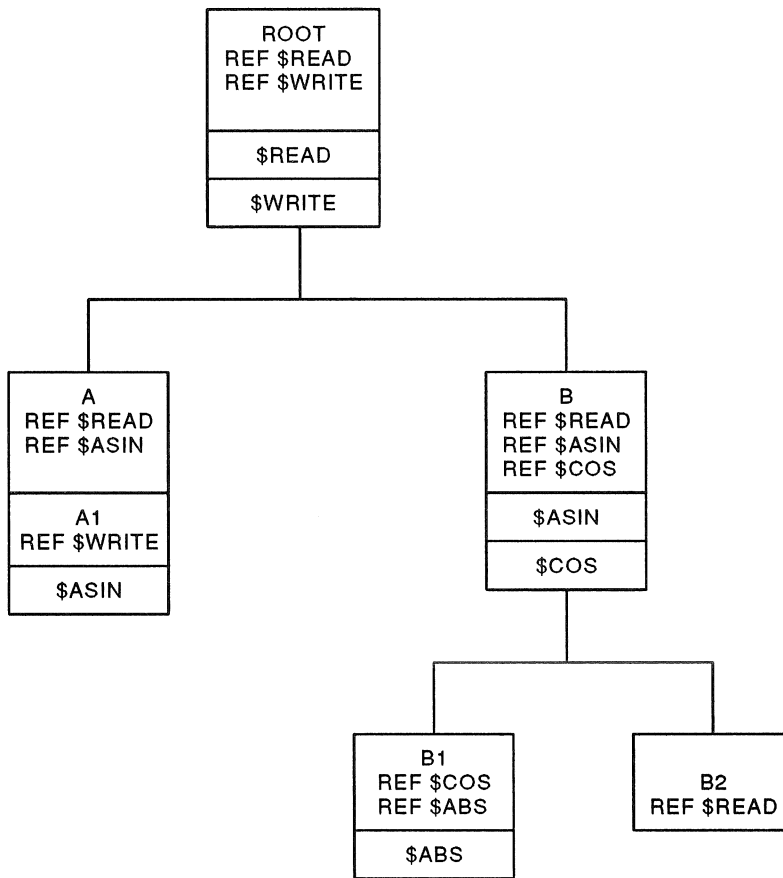
MK-00580-00

As shown in Figure 3–8, the ROOT section calls the routines $READ and $WRITE. The Task Builder resolves these references by building the routines into ROOT. The references to $READ in A and B are then resolved from ROOT. The reference to $WRITE in A1 is likewise resolved from ROOT.

Both A and B refer to the $ASIN routine. Since A and B are on different paths, the Task Builder puts the $ASIN routine in both A and B.

Both B and B1 refer to the $COS routine; it is built into B because B is closer to the root than B1. The reference to $COS in B1 is resolved by referring to the routine in B.

The $ABS routine is referred to in B1 only. It is built into B1.

If you know which routines are called from the various pieces of your program, you can shorten the time necessary to build your program by specifying the routines directly. You make a direct specification by appending a colon and the routine name to the /LB switch. For example, to build the structure shown in Figure 3–8, you could use:

```
          .ROOT   ROOTL-*(AL,BL-(B1L,B2))
ROOTL:    .FCTR   ROOT-LB:F4POTS/LB:$READ:$WRITE
AL:       .FCTR   A-A1-LB:F4POTS/LB:$ASIN
BL:       .FCTR   B-LB:F4POTS/LB:$ASIN:$COS
B1L:      .FCTR   B1-LB:F4POTS/LB:$ABS
          .END
```

You could also request a different structure. To build all the required routines into the root, for example, you could specify:

```
          .ROOT ROOTL-*(A-A1,B-(B1,B2))
ROOTL:    .FCTR ROOT-LB:F4POTS/LB:$READ:$WRITE:$ASIN:$COS:$ABS
          .END
```

In general, up to eight routines can be specified on one /LB switch.

3.7.4 The Default Library

The Task Builder searches through the overlay structure to resolve global symbols. If any symbols are undefined after it examines all the pieces you specify, it will search the default library (normally LB:SYSLIB.OLB). If it can resolve a global symbol by inserting a piece from this library, it will do so.

Note that because SYSLIB.OLB used the MACRO assembly language judiciously, the code is not inserted as described in Section 3.7.3. Units called program sections have been carefully defined using MACRO, such that the code in SYSLIB takes as little space as possible. Program sections, and how the Task Builder builds programs with them, are described in Chapter 6.

For libraries built from compiler-language routines, however, the information in Section 3.7.3 would apply.

Co-Trees: Another Way to Save Space

Chapter 3 describes the basics of specifying an overlay structure in an ODL file. This chapter discusses another overlay structure: co-trees. Co-trees are slightly more complex structures than any previously discussed. If applicable to your call structure, however, they can be extremely useful in cutting down the virtual address space your program takes (see Figure 4–1).

Figure 4–1: Co-Trees Save More Space Than Simple Overlays
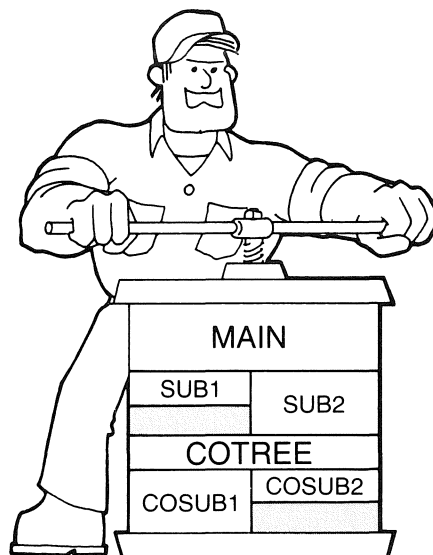

```
ODL FILE:

          .ROOT MANTRE, COTRE
MANTRE:   .FCTR MAIN–LIB–*(SUB1–LIB, SUB2–LIB)
 COTRE:   .FCTR *COTREE–LIB–*(COSUB1–LIB, COSUB2–LIB)
   LIB:   .FCTR LB:BP20TS/LB
          .END
```
MK–00581–00

4.1 The Co-Tree Structure

As the name implies, co-trees allow you to define more than one tree structure in an overlay description. For example, suppose that A and B are routines that are called from several branches of a tree. You could define A and B as part

of the root, so that they are always accessible from any branch of the tree (see Figure 4–2).

Figure 4–2: Putting A and B in the Root
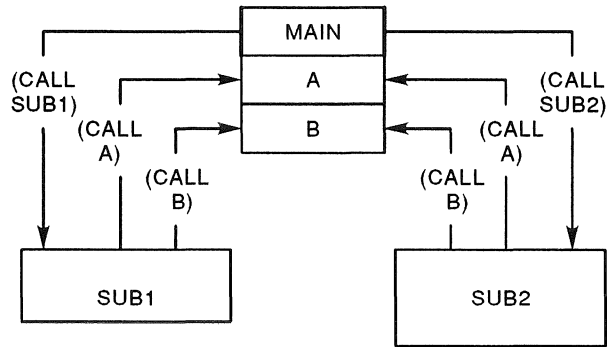
MK–00582–00

The ODL file for such a structure could appear as follows:

```
        .ROOT MAIN-A-B-LIBR-*(SUB1-LIBR,SUB2-LIBR)
LIBR:   .FCTR LB:BP2OTS/LB
        .END
```

Since A and B never call each other, however, they do not need to reside in memory at the same time. To save space, you could define A and B as part of a co-tree, such that they overlay each other. Like the main tree, co-trees must have a root. In this case, the root is called "COTRE" (see Figure 4–3).

Figure 4–3: A Co-Tree Structure
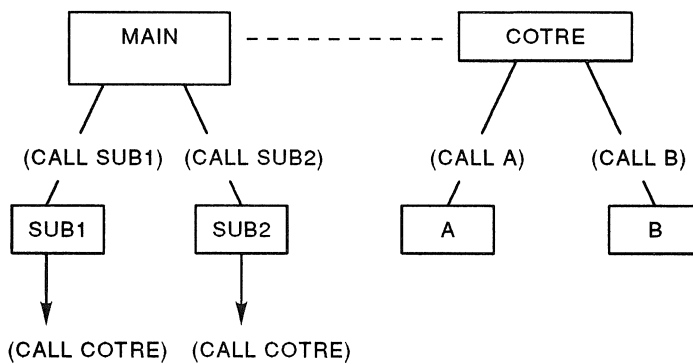
MK–00583–00

The ODL file for such a structure could be:

```
        .NAME   COTRE
        .ROOT   MANTRE,COTREE
MANTRE: .FCTR   MAIN-LIBR-*(SUB1-LIBR,SUB2-LIBR)
COTREE: .FCTR   *COTRE-LIBR-*(A-LIBR,B-LIBR)
LIBR:   .FCTR   LB:BP2OTS/LB
        .END
```

To separate co-trees, use the comma—not enclosed in parentheses – as between MANTRE and COTREE in the .ROOT statement above. (When the comma is used within parentheses, it separates pieces to be overlaid.) Note also that you put an autoload indicator (*) before the co-tree root and before the outermost left parenthesis in the co-tree overlay description.

To get an idea of how co-trees are loaded during execution, see Figure 4–4. This figure assumes that the call sequence is: MAIN calls SUB1, which calls COTRE twice: once to execute A and once to execute B. MAIN then calls SUB2, which calls COTRE to call A and B again.

The run-time loader loads MAIN. The remaining pieces are loaded by code inserted in MAIN by the Task Builder. Once called (at time 3 in Figure 4–4), COTRE is resident in memory for the rest of the run. Note that it begins at the place where the longest part of the main tree ends (after SUB2).

As shown in Figure 4–4, storage is not shared between trees. Any piece in a tree can call or refer to data in another tree without displacing pieces from the calling tree. However, calls back and forth between trees (called cross-tree calls) can cause problems. For example, suppose that at time 4 in Figure 4–4 the subprogram A had called SUB2. SUB2 would be loaded, displacing SUB1 from the main tree. In the normal course of events, SUB2 would return control to A, which would return control to an address generated for SUB1 at build time. SUB1 is no longer in memory, however. Control would be passed to some location in SUB2, which has displaced SUB1 in memory.

To keep this from happening inadvertently, the Task Builder restricts its search through the structure for references to the default library if you specify co-trees. The Task Builder makes one pass through the entire structure trying to resolve global symbols from the pieces you have specified in the ODL file. If there are unresolved symbols after this first pass, the Task Builder makes another pass, attempting to resolve undefined symbols from the default system library. If you have specified co-trees, the Task Builder restricts its search during this second pass.

For example, if the Task Builder resolves a symbol in one tree by inserting a module from the default system library, it does not search through co-trees to see if there are other unresolved references to this module. It restricts its search to the current tree and the root of the main tree. This procedure eliminates cross-tree calls like that described above; necessary code is not inadvertently displaced.

Figure 4-4: How a Co-Tree Is Loaded During Program Execution


```
      TIME 1          TIME 2           TIME 3          TIME 4          TIME 5
      MAIN          MAIN CALLS        SUB1 CALLS      COTRE CALLS     COTRE CALLS
      LOADED        SUB1; SUB1        COTRE; COTRE    A; A LOADED     B; B LOADED
                    LOADED            LOADED

      TIME 6          TIME 7           TIME 8
   CONTROL FALLS    SUB2 CALLS        COTRE CALLS
   BACK TO MAIN,    COTRE (ALREADY    B; B LOADED
   WHICH CALLS SUB2:  THERE) COTRE
   SUB2 LOADED      CALLS A; A LOADED
```
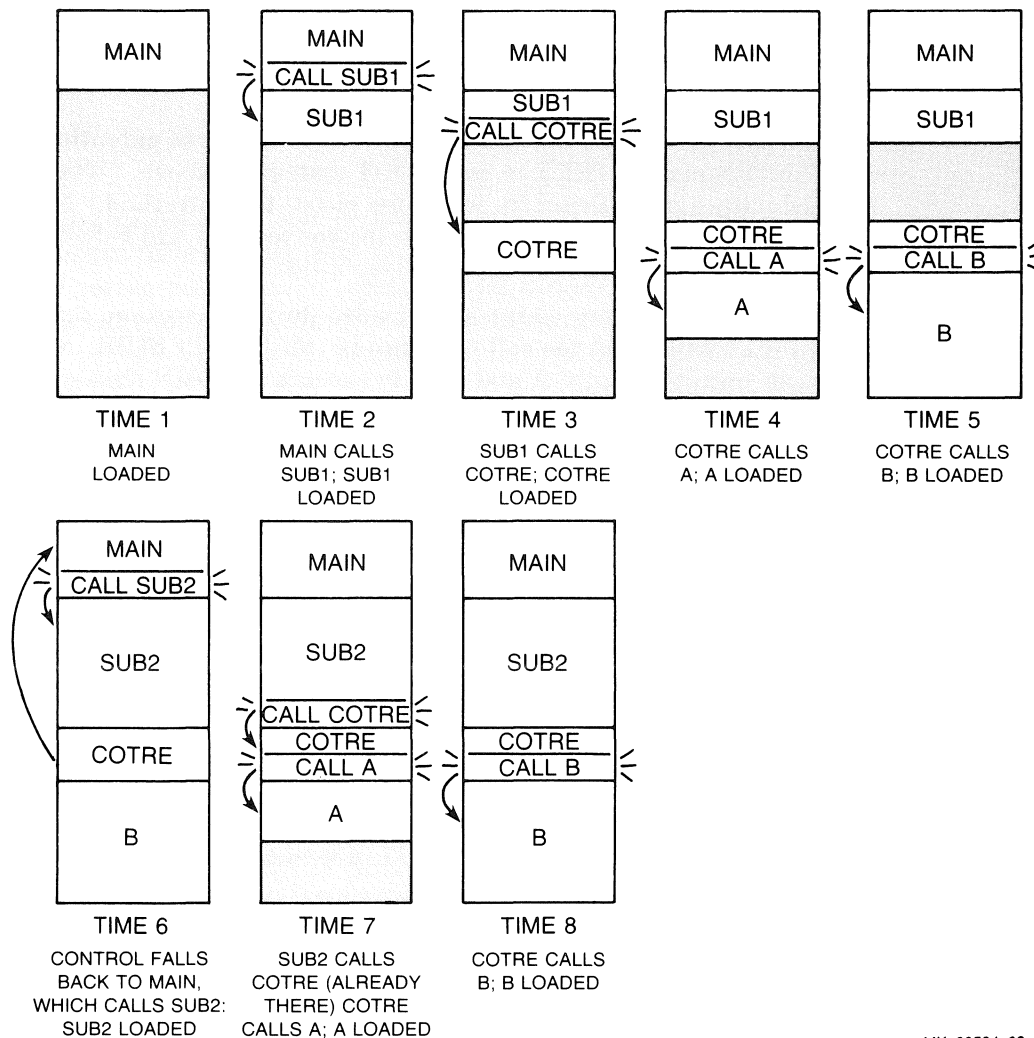
MK-00584-00

4.2 Using the .NAME Command for a Co-Tree Root

The example in Section 4.1 defined a separate co-tree root routine called COTRE. You can eliminate the need for a real routine (which takes space) by using the .NAME command to define the name of a dummy root for a co-tree. The calls out of SUB1 and SUB2 can then refer to A and B directly, and they will be loaded properly. For example, the ODL file could be:

```
             .NAME   NULL
             .ROOT   MANTRE,COTREE
MANTRE:      .FCTR   MAIN-LIBR-*(SUB1-LIBR,SUB2-LIBR)
COTREE:      .FCTR   NULL-*(A-LIBR,B-LIBR)
LIBR:        .FCTR   LB:BP2OTS/LB
             .END
```

The .NAME command lets you give a name to a piece of the overlay structure. (You can also use it to assign certain attributes to pieces of the overlay structure, described further in Section 6.5.) The .NAME command is described in detail in Section 13.4.

Note in the preceding example that you do not need to use an autoload indicator
(*) before a null root in a co-tree.

4.3 Designing the Most Space-Saving Co-Trees

Co-trees can save the most space when the pieces being overlaid in each tree are
about the same size. For example, look at the structure of the example from the
previous sections. Figure 4–5 shows three different structures. Figure 4–5 also
shows a new way of looking at overlay structures that takes the size of the pieces
into account. You may find this particularly useful when dealing with co-trees.

Figure 4–5: Co-trees Save More Space When Pieces Are the Same Size

NO CO-TREE   FIRST CO-TREE   REDESIGNED CO-TREE
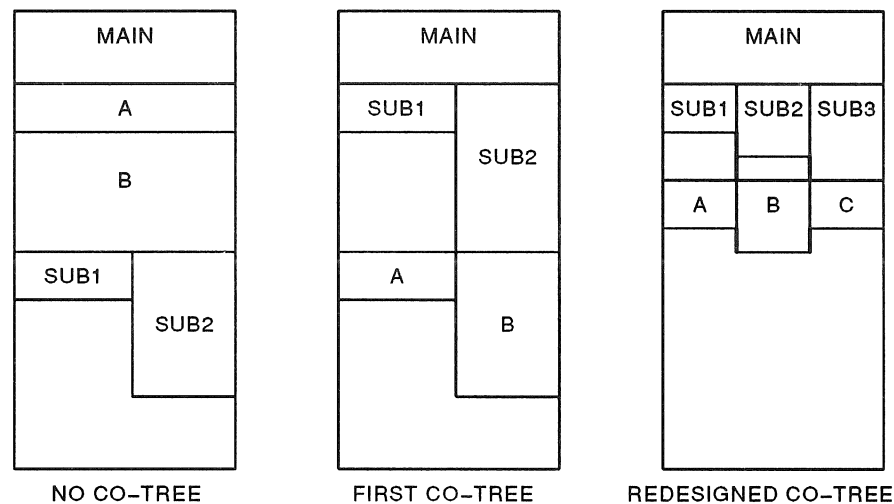
MK–00585–00

The first structure in Figure 4–5 shows the original overlay design, with A
and B built after MAIN in the root of the overlay structure. SUB1 and SUB2
overlay each other, after the root. The second structure in Figure 4–5 shows the
structure arrived at in the previous section: A and B are overlaid in a co-tree. By
comparing the first and second structures in Figure 4–5, you can see that co-trees
can save space. The total size of the program in virtual address space is smaller
using the co-tree.

Note in the second structure, however, that SUB2 and B are much larger than
their counterparts SUB1 and A. The space shown between SUB1 and the A-B
co-tree is essentially wasted. If you cut down SUB2, you can "squeeze" the co-tree
down further in virtual address space. Furthermore, if you reduce the size of B,
you can also reduce the total size of the program in virtual address space.

The third structure in Figure 4–5 shows a better co-tree. SUB2 has been divided
into two routines, SUB2 and SUB3. These routines are overlaid and are about
the same size as each other and SUB1. B has also been divided into two separate
routines, B and C. Again, the routines are overlaid and are about the same size as
each other and A. Note that the total size of the program in virtual address space
has been reduced even more than the second structure shown in Figure 4–5.

Thus, the general rule for constructing tight programs using co-trees is: use small subprograms of uniform size. If you are using co-trees at all, you are probably more concerned with space than with your program's execution time. You may not lose that much time, though, depending on how the calls are structured. Remember that calls can be made between co-trees without necessarily causing a new overlay segment to be loaded from disk.

For example, suppose that in the third structure of Figure 4–5, SUB1 calls A. A will remain in memory until B or C is called. Suppose control passes back to SUB1 to call A again, back to MAIN, and on to SUB2, which is loaded and calls A. A is still in memory. It does not need to be loaded again.

4.4 Co-Trees and High-Level Languages

As you can see from the previous sections, using co-trees can save space. The first time you try to build one using a high-level language, however, you will likely get a number of diagnostic error messages flagging multiply defined, ambiguously defined, and undefined symbols. This can be a bit disconcerting, especially since many of the symbols will not be any that you have referred to or defined in your program. Furthermore, the total virtual address space taken by the program may be even larger than that taken without co-trees.

The symbols are from library routines that have been inserted by the Task Builder. Note that the Task Builder is very careful about where it puts routines that are called from outside the main tree root on different trees in a co-tree structure. When you put general library references in your ODL file the Task Builder builds any routine that is called from outside the main tree root on two or more paths in two or more trees, into all the paths and trees where it is referenced. The Task Builder then resolves references to a routine in a particular path in a tree by referring to the routine built into that path on that tree.

So, the program will run as it has been built (unless you have made errors in your program, of course). Still, you may not want these routines built into each tree, where they can take more space than might actually be necessary. So, the Task Builder flags the symbols it finds as multiply defined or ambiguously defined, so that you can correct the situation if you want to.

4.4.1 Sample Source Program and Subprograms

Consider the following BASIC-PLUS-2 program and subprograms.

The main program, USER, simply calls three subprograms: INTRO, CRUNCH, and CHATR. INTRO displays a question at the user's terminal, and accepts the user's response: two integers. CRUNCH performs addition, multiplication, and subtraction on the two input numbers and calls CALC2 and CALC1, passing on the two input numbers. CHATR takes the sum, product, and difference calculated by CRUNCH and displays the values on the user's terminal. It then calls CALC1 and CALC2. CALC1 subtracts the two values and displays the difference. CALC2 adds the two values and displays the sum.

Source for Program USER

```
10         CALL INTRO(A1%,B1%)
20         CALL CRUNCH(A1%,B1%,SUMM%,PRODUCT%,DIFFER%)
30         CALL CHATR(A1%,B1%,SUMM%,PRODUCT%,DIFFER%)
40         END
```

Source for Subprogram INTRO

```
10         SUB INTRO(AA%,BA%)
20         INPUT "INPUT TWO NUMBERS";AA%,BA%
30         SUBEND
```

Source for Subprogram CRUNCH

```
10         SUB CRUNCH(AB%,BB%,CA%,CB%,CC%)
20         CA% = AB% + BB%
30         CB% = AB% * BB%
40         CC% = AB% - BB%
50         CALL CALC2(AB%,BB%)
60         CALL CALC1(AB%,BB%)
70         SUBEND
```

Source for Subprogram CHATR

```
10         SUB CHATR(AC%,BC%,CA%,CB%,CC%)
20         PRINT "THE SUM OF ";AC%;" AND ";BC%;" IS ";CA%
30         PRINT "THE PRODUCT OF ";AC%;" AND ";BC%;" IS ";CB%
40         PRINT "THE DIFFERENCE OF ";AC%;" AND ";BC%;" IS ";CC%
50         CALL CALC1(AC%,BC%)
60         CALL CALC2(AC%,BC%)
70         SUBEND
```

Source for Subprogram CALC1

```
10         SUB CALC1(AD%,BD%)
20         DA%=AD%-BD%
30         PRINT "THE COTREE DIFFERENCE IS ";DA%
40         SUBEND
```

Source for Subprogram CALC2

```
10         SUB CALC2(AE%,BE%)
20         EA%=AE%+BE%
30         PRINT "THE COTREE SUM IS ";EA%
40         SUBEND
```

4.4.2 Outlining the Sample Program's Call Structure

As Figure 4–6 shows, the sample program and its subprograms fit the general situation where co-trees can help: one or more subprograms called by one or more other subprograms.

Figure 4-6: Call Structure for Sample Program

```
                              USER
                                |
        ┌───────────────────────┼───────────────────────┐
   (CALL INTRO)           (CALL CRUNCH)             (CALL CHATR)
        |                       |                         |
      INTRO                  CRUNCH                     CHATR
                      ┌────────┴────────┐       ┌────────┴────────┐
                 (CALL CALC2)     (CALL CALC1)  (CALL CALC1)  (CALL CALC2)
                      |                |             |             |
                    CALC2            CALC1         CALC1         CALC2
```
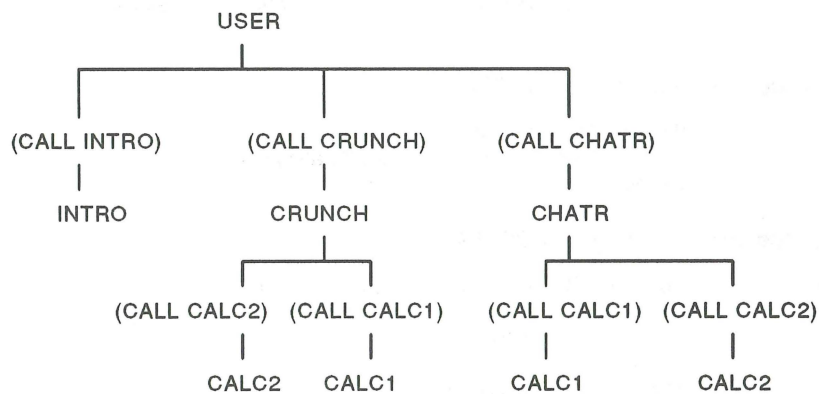
MK-00586-00

4.4.3 Compiling the Sample Program and Subprograms

The general steps for compiling a BASIC-PLUS-2 program are:

RUN $BASIC2

BASIC2 ◄───────────────── (the prompt from BASIC-PLUS-2)

OLD source-file

BASIC2 ◄─────────────────┘

COMPILE /OBJ

For example, to compile a source file named USER.B2S, type:

RUN $BASIC2

BASIC2

OLD USER

BASIC2

COMPILE /OBJ

These commands compile the file USER.B2S, creating the file USER.OBJ. (.B2S and .OBJ are the default file types assumed by BASIC-PLUS-2.)

4.4.4   First Build for Sample Program: Putting Subprograms in the Root

After creating the object files, the next step is task building. For the first build, without co-trees, we put CALC1 and CALC2 in the root (Figure 4–7).

Figure 4–7:   First Build Structure for Sample Program

```
                    USER
                     |
                    CALC1
                     |
                    CALC2
          _____|_____
         |           |           |
       INTRO      CRUNCH       CHATR


                MK-00586-01
```
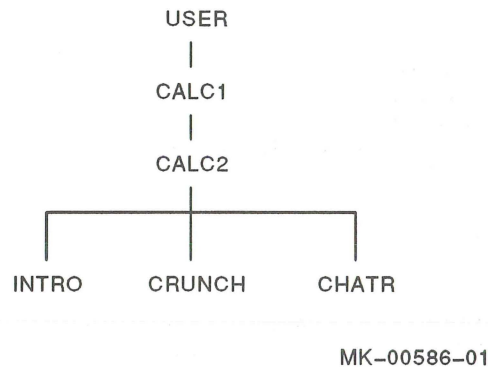
The ODL file for such a structure could be:

```
            .ROOT  USER-CALC1-CALC2-LIBR-*(INTWL,CHATWL,CRUNWL)
INTWL:      .FCTR  INTRO-LIBR
CRUNWL:     .FCTR  CRUNCH-LIBR
CHATWL:     .FCTR  CHATR-LIBR
LIBR:       .FCTR  LB:BP2OTS/LB
            .END
```

Calling the above file OVER1.ODL, the build process is:

```
RUN $TKB
TKB>TRY1,TRY1=OVER1/MP
ENTER OPTIONS:
TKB>UNITS=12
TKB>ASG=SY:5:6:7:8:9:10:11:12
TKB>EXTTSK=512
TKB>//
```

The build proceeds without error. Examining the map file, TRY1.MAP, you see the first page shown in Example 4–1.

The significant information is highlighted in Example 4–1. The TASK IMAGE SIZE is 6528 words, or 13056 bytes. This means that the total amount of virtual address space that the program will take is 13056 bytes. Further down, the size is itemized by segment. Segment USER (constructed from the files USER.OBJ, CALC1.OBJ, and CALC2.OBJ, plus library routines from BP2OTS.OLB) requires 11348 bytes; INTRO, 1696 bytes; CHATR, 380 bytes; and CRUNCH, 196 bytes. In subsequent builds, you will see the structure (shown by the way the segment names are indented) and the sizes change.

Example 4-1: First Page of Map File for Sample Program

```
TRY1.TSK    Memory allocation map   TKB 08.006        Page 1
                        15-MAY-90   14:46


Partition name : GEN
Identification : 000708
Task  UIC      : [1,196]
Stack      limits: 001000 001777 001000 00512.
PRG xfr address: 022462
Total address windows: 1.
Task extension : 512 words
Task  image  size  : 6528. words
Total  task  size  : 7040. words
Task address limits: 000000 031363
R-W disk blk limits: 000002 000036 000035 00029.



TRY1.TSK Overlay description:

Base    Top        Length

000000  026123  026124  11348.       USER
026124  031363  003240  01696.           INTRO
026124  026717  000574  00380.           CHATR
026124  026427  000304  00196.           CRUNCH
```

MK-01049-00

4.4.5 Second Build for Sample Program: Using a Co-Tree

For the second build of the sample program, use the co-tree structure shown in
Figure 4-8.

Figure 4-8: Structure for Second Build of Sample Program
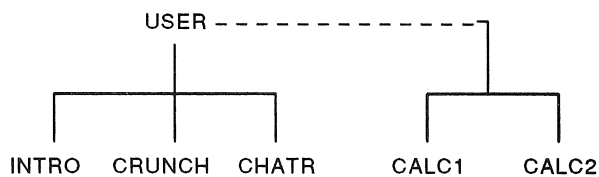
MK-00586-02

The ODL file for such a structure could be:

```
           .NAME  NULL
           .ROOT  USERWL,COTRWL
COTRWL:    .FCTR  NULL-*(CALC1-LIBR,CALC2-LIBR)
USERWL:    .FCTR  USER-LIBR-*(INTWL,CHATWL,CRUNWL)
INTWL:     .FCTR  INTRO-LIBR
CHATWL:    .FCTR  CHATR-LIBR
CRUNWL:    .FCTR  CRUNCH-LIBR
LIBR:      .FCTR  LB:BP2OTS/LB
           .END
```

Calling the file above OVER2.ODL, the build process is:

```
RUN $TKB
TKB>TRY2,TRY2=OVER2/MP
ENTER OPTIONS:
TKB>UNITS=12
TKB>ASG=SY:5:6:7:8:9:10:11:12
TKB>EXTTSK=512
TKB>//
```

This run, unlike the first, produces a blizzard of diagnostic error messages:

```
TKB -- *DIAG* --   MODULE CALC1 AMBIGUOUSLY DEFINES SYMBOL xxxx
         .
         .
         .
TKB -- *DIAG* --   MODULE $ICINI MULTIPLY DEFINES SYMBOL xxxx
         .
         .
         .
TKB -- *DIAG* --   MODULE $JPMOV MULTIPLY DEFINES SYMBOL xxxx
         .
         .
         .
```

It is useful to take note of the modules mentioned, for reasons that become clear later when you look at the memory map. The modules are, in order: CALC1, $ICINI, $JPMOV, $ICWRT, $ECONV, $ICFNS, $STFN1, CALC2, (and again) $ICINI, $JPMOV, $ICWRT, $ECONV, $ICFNS, and $STFN1.

If you try running the program, (by typing RUN TRY2.TSK), it works. So the Task Builder's error messages are only diagnostic messages, as indicated.

Example 4–2 shows two pages of the relevant information from the map file TRY2.MAP. Page 2 of the map shows that the total size of the program has grown from 6528 words in the first build to 7552 words for the second build. This would seem an inauspicious start for a memory-saving co-tree structure, but you can be prepared for this and look for the reasons.

Page 5 shows the start of the information you are most interested in at this point. The highlighted portion under the TITLE column shows the names of library routines that have been built into the INTRO piece of the overlay structure. You know that they are library routines because the FILE column shows them as from the library file BP2OTS.OLB.

Following pages of the map (not shown in Example 4–2) would show similar entries for library routines in the overlay pieces CRUNCH, CHATR, CALC1, and CALC2.

Example 4-2: Excerpts from Map File for Second Build of Sample Program

```
TRY2.TSK    Memory allocation map   TKB 08.006       Page 2
CALC2                    15-MAY-90    13:17


Partition name : GEN
Identification : 000708
Task  UIC      : [1,196]
Stack     limits: 001000 001777 001000 00512.
PRG xfr address: 016030
Total address windows: 1.
Task extension      : 512. words
Task  image  size   : 7552. words
Total  task  size   : 8064. words
Task address limits: 000000 035313
R-W disk blk limits: 000002 000054 000053 00043.



TRY2.TSK Overlay description:

Base    Top       Length
----    ---       ------
000000  021043  021044  08740.      USER
021044  030523  007460  03888.          INTRO
021044  026173  005130  02648.          CHATR
021044  021577  000534  00348.          CRUNCH
030524  030523  000000  00000.      NULL
030524  035313  004570  02424.          CALC1
030524  035303  004560  02416.          CALC2
                  .
                  .
                  .


TRY2.TSK    Memory allocation map   TKB 08.006       Page 5
USER                     15-MAY-90    13:17


*** Segment: INTRO

R/W mem  limits: 021044 030523 007460 03888.
Disk blk limits: 000024 000033 000010 00008.

Memory allocation synopsis:

Section                                 Title   Ident  File
-------                                 -----   -----  ----
. BLK.:(RW,I,LCL,REL,CON) 021044 000000 00000.
BP2OTS:(RW,I,LCL,REL,CON) 021044 007272 03770.
                          021044 000422 00274. $ICINI 23CM   BP2OTS.OLB
                          021466 002004 01028. $ICRED 53CM   BP2OTS.OLB
                          023472 000656 00430. $ICWRT 40CM   BP2OTS.OLB
                          024350 000642 00418. $STMOS 16CM   BP2OTS.OLB
                          025212 002404 01248. $ECONV 24CM   BP2OTS.OLB
                          027616 000226 00150. $ICFNS 11RE   BP2OTS.OLB
                          030044 000202 00130. $STLSS 08CM   BP2OTS.OLB
                          030246 000070 00056. $STFN1 06CM   BP2OTS.OLB
$ARRAY:(RW,D,LCL,REL,CON) 030336 000000 00000.
                  .
                  .
                  .
```

At this point, it is useful to sketch the information shown in the map file, listing the routines included in each overlaid piece from the library BP2OTS.OLB. Figure 4–9 is such a sketch, showing the relative sizes of the pieces and naming the library routines built into each of the overlaid pieces.

NOTE

Library routines have also been built into the root segment, USER, but they are of no concern. It is theoretically possible that some of these routines could be overlaid in their own tree. However, it is difficult to know the sequence in which such routines are called from other trees. You would have to look at an assembly-language listing of the compiled code to determine the sequence of calls.

That is, overlaying such routines in their own tree could, and probably would, result in the cross-tree call problem mentioned in Section 4.1. An overlay piece could inadvertently displace another overlay piece in memory, causing errors at execution time.

Figure 4–9 is the basis of the analysis for "fine-tuning" a build using co-trees with a high-level language. Now you can see more clearly just what the Task Builder has done and why.

First, it has built the routines $ICINI, $ICWRT, $ECONV, $ICFNS, and $STFN1 into two paths in each of the two trees: that is, into INTRO and CHATR in the main tree, and CALC1 and CALC2 in the co-tree. This was indeed the most reasonable thing for the Task Builder to do; it had no way of knowing whether to resolve the references in CALC1 and CALC2 with the definitions in INTRO or the definitions in CHATR. So, it built the routines into CALC1 and CALC2 again, and flagged the routines (modules) and symbols for your examination.

Figure 4–9: Sketch of the Structure for Second Build of Sample Program

```
┌─────────────────────────────────────────────────────┐
│                                                       │
│                                                       │
│              USER    —   8740 BYTES                   │
│                                                       │
│                                                       │
├─────────────┬─────────────┬─────────────────────────┤
│   INTRO     │  CRUNCH     │    CHATR                  │
│ 3888 BYTES  │ 348 BYTES   │  2648 BYTES               │
│  $ICINI     │  $JPADD     │   $ICINI                  │
│  $ICRED     │  $JPMOV     │   $JPMOV                  │
│  $ICWRT     │  $JMUL      │   $ICWRT                  │
│  $STMOS     │  $JPSUB     │   $ECONV                  │
│  $ECONV     │             │   $ICFNS                  │
│  $ICFNS     ├─────────────┤   $STFN1                  │
│  $STLSS     │             │                           │
│  $STFN1     │             │                           │
│             │             │                           │
│             │             │                           │
│             │             ├───────────────────────────┤
│             │             │                           │
├─────────────┴───────┬─────┴───────────────────────────┤
│      CALC1          │        CALC2                     │
│    2424 BYTES       │      2416 BYTES                  │
│     $ICINI          │       $ICINI                     │
│     $JPMOV          │       $JPMOV                     │
│     $ICWRT          │       $ICWRT                     │
│     $ECONV          │       $ECONV                     │
│     $ICFNS          │       $ICFNS                     │
│     $STFN1          │       $STFN1                     │
│                     │                                  │
│                     └──────────────────────────────────┤
│                     │                                   │
└─────────────────────┘
```
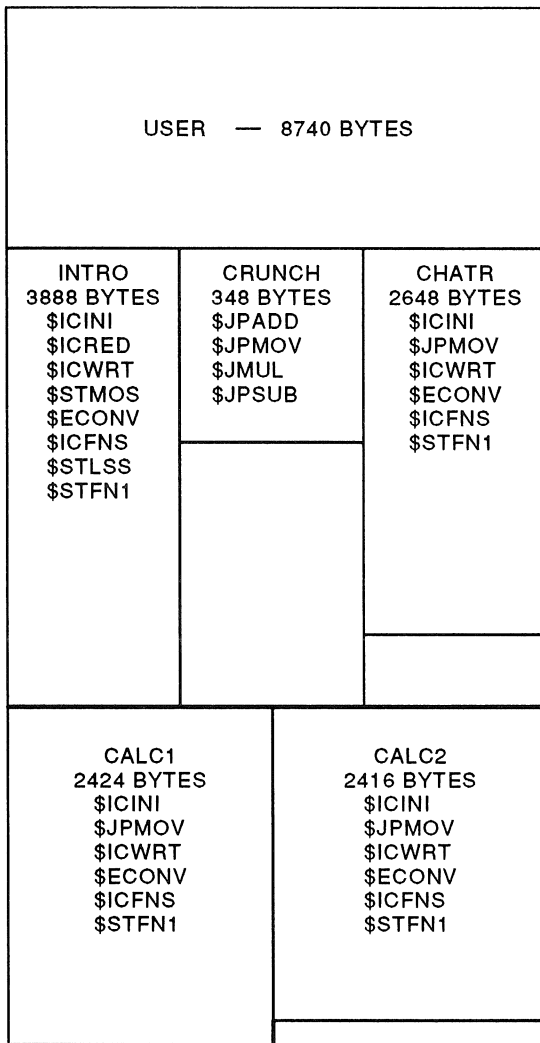
MK–01050–00

To save space, then, you can build one copy of each of these routines into the root, where they would be accessible from all branches of all trees (as shown in Section 4.4.6. First, however, continue with the analysis.

The only other routine that appears in more than one path on more than one tree is $JPMOV, appearing in CRUNCH, CHATR, CALC1, and CALC2. Now – look at the structure from the viewpoint of "overlaid pieces should be about the same size." CRUNCH is quite small at 348 bytes. CHATR and CRUNCH together about equal the size of INTRO. You can link CRUNCH and CHATR together using the following ODL commands:

```
          .
          .
          .
USERWL:   .FCTR  USER-LIB-*(INTWL,CRUNWL)
          .
          .
          .
CRUNWL:   .FCTR  CRUNCH-CHATR-LIBR
          .
          .
          .
```

This combination has the advantage in that it also consolidates the references to $JPMOV from two paths in the main tree to only one path. The Task Builder can then resolve the references to $JPMOV in CALC1 and CALC2 with the routine built into the branch in the main tree. Since in this branch, CRUNCH and CHATR both call CALC1 and CALC2, and since CALC1 and CALC2 will not make calls to any other paths in the main tree, you will not have any problem with inadvertently displaced pieces at runtime.

4.4.6 Third Build for Sample Program: Restructured Tree and Library Routines in Root

You are ready to build the program again using the structure shown in Figure 4–10.

Figure 4–10: Structure for Third Build of Sample Program
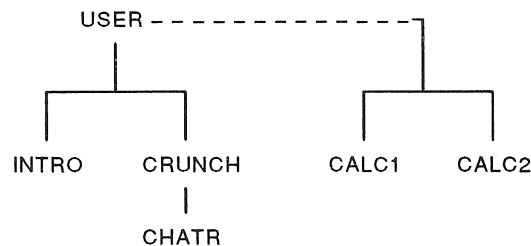
MK–00586–03

In addition to specifying the main program and subprograms, you also want to build the library routines $ICINI, $ICWRT, $ECONV, $ICFNS, and $STFN1 into the root. Do this by using the /LB switch, followed by specific routine names separated by colons. For example:

```
        .NAME  NULL
        .ROOT  USERWL,COTRWL
COTRWL: .FCTR  NULL-*(CALC1-LIBR,CALC2-LIBR)
USERWL: .FCTR  USER-LIB-*(INTWL,CRUNWL)
INTWL:  .FCTR  INTRO-LIBR
CRUNWL: .FCTR  CRUNCH-CHATR-LIBR
LIB:    .FCTR  LIB1-LIBR
LIB1:   .FCTR  LB:BP2OTS/LB:$ICINI:$ICWRT:$ECONV:$ICFNS:$STFN1
LIBR:   .FCTR  LB:BP2OTS/LB
        .END
```

In general, you can specify up to eight routines as modifiers to one LB switch.

Calling this file OVER3.ODL, the build process is:

```
RUN $TKB
TKB>TRY3,TRY3=OVER3/MP
ENTER OPTIONS:
TKB>UNITS=12
TKB>ASG=SY:5:6:7:8:9:10:11:12
TKB>EXTTSK=512
TKB>//
```

The build proceeds without error. The first page of the map file TRY3.MAP is shown in Example 4–3. As shown, the total amount of virtual address space taken by the program is 6400 words, smaller than the first build without co-trees, which took 6528 words. And, as can be determined by typing RUN TRY3.TSK, the program runs.

Example 4–3: First Page of Map File for Third Build of Sample Program

```
TRY3.TSK    Memory allocation map   TKB 08.006      Page 1
                      15-MAY-90    15:24

Partition name : GEN
Identification : 000708
Task  UIC      : [1,196]
Stack     limits: 001000 001777 001000 00512.
PRG xfr address: 022252
Total address windows: 1.
Task extension    : 512. words
Task  image  size : 6400. words
Total  task  size : 6912. words
Task address limits: 000000 030773
R-W disk blk limits: 000002 000037 000036 00030.

TRY3.TSK Overlay description:

Base    Top       Length
----    ---       ------
000000  025253    025254   10924.      USER
025254  030513    003240   01696.              INTRO
025254  026603    001330   00728.              CRUNCH
030514  030513    000000   00000.      NULL
030514  030773    000260   00176.              CALC1
030514  030763    000250   00168.              CALC2
```

4.4.7 Further Tips

The example program discussed in the previous sections is a simple one. For a complex program having many overlays in many co-trees, some of the steps described above are harder to follow; it may be very tedious to write down all the routine names in the diagnostic error messages resulting from a "first-try" co-tree build.

One way to get around this problem is to eliminate all library references in your preliminary build. The routines and symbols will then show up on the map file listing as "undefined symbols," and you can work from there.

4.4.8 Using Co-Tree Techniques with the Default Library

You can use the techniques discussed in this chapter for default library routines. However, you must be even more careful than you were with the language libraries. Routines in the default library were coded in the MACRO assembly language using expert manipulation of program sections (see Chapter 6). For example, a routine may use a data section that can be overlaid in low virtual address space while instruction sections are built into separate paths of the tree. Unless you are a MACRO programmer, aware of these program sections and capable of dealing with them, you should not try overlaying routines from the default library.

If you do want to try it, you will find the /MA and /FU switches useful. These switches are described in detail in Chapter 11.

Briefly, the /MA switch appended to the map file specification calls for more detail in the listing on routines built into the program from the default library.

Appending the /FU switch to the executable program file (default extension .TSK) tells the Task Builder to make a "full search" during its second pass to resolve undefined symbols from the default library. Suppose, for example, that the Task Builder builds a definition from the default library into one path on the main tree to resolve an undefined symbol. If this symbol is referred to from a path on a co-tree and the /FU switch has been used, the Task Builder resolves the reference in the co-tree with the definition in the main tree.

Note that if the symbol were referred to in more than one path on more than one tree, the Task Builder proceeds as it does when it searches through library references specified with a general-purpose /LB switch. That is, it builds the piece into all paths on all trees and flags the symbols as multiply or ambiguously defined. You can specify where you want individual program sections by using the Task Builder's .PSECT command (Chapter 6).

Again, you should not try to overlay pieces from the default library unless you are experienced in MACRO and can determine that cross-tree calls will not inadvertently displace portions of the overlay structure.

The Autoload Indicator

Now that you understand the rules for specifying an overlay structure, it is necessary to better understand the autoload indicator (*). The asterisk tells the Task Builder to generate what are called "autoload vectors" for pieces of your overlaid program. Autoload vectors are necessary for the pieces to be loaded properly. As mentioned in Section 3.2.1, the easiest way to use the autoload indicator is to put an asterisk (*) before the outermost parenthesis of the main tree and call co-trees, and before any non-null co-tree roots (Figure 5–1).

This chapter tells what is happening when you use the autoload indicator, and why. It also explains how you can save a small amount of space in your program by using autoload indicators judiciously.

Figure 5–1: The Easiest Way to Use Autoload Indicators

FOR A SINGLE TREE STRUCTURE:

```
.ROOT A-*(A1,A2,A3-(A31,A32))
                    (BEFORE THE OUTERMOST PARENTHESIS)
  .
  .
.END
```

FOR A CO-TREE STRUCTURE:

```
              .ROOT MAIN, COTRE1, COTRE2
MAIN:         .FCTR A-*(A1,A2,A3,)
              .NAME NULL              (BEFORE OUTERMOST LEFT
COTRE1:       .FCTR NULL-*(B1,B2)     PARENTHESIS  IN MAIN TREE
COTRE2:       .FCTR *C-*(C1,C2-(C21,C22))   AND   EACH CO-TREE)
                 .
                 .
                 .         (BEFORE A NON-NULL CO-TREE ROOT)
              .END
```

MK–00588–00

5.1   What are Autoload Vectors?

When overlays are not requested, the Task Builder resolves references to symbols in transfer-of-control statements by figuring out the location (address) where the symbol will reside when the program is executing. The Task Builder then puts

this address in the transfer-of-control instruction. For example, consider the simple build:

```
RUN $TKB
TKB>OBJ=MAIN,SUB1,LB:F4POTS/LB
TKB>//
```

As described in Chapter 3, the Task Builder concatenates MAIN and SUB1 and resolves undefined references by concatenating modules from the library. When MAIN calls SUB1, the Task Builder resolves the reference by substituting the address it has calculated for the entry point for SUB1.

With overlays, the Task Builder does not assume that such direct substitutions will work. When a call is made to a piece further away from the root, there is no guarantee that the piece referenced will be in memory when the call is executed. So, you must tell the Task Builder to generate autoload vectors for global symbols outside the root that are referenced in transfer-of-control statements by a piece closer to the root. And, where such a reference is made to a global symbol, the Task Builder will then substitute the address of the autoload vector instead of the direct address reference.

The generated autoload vectors are stored in every piece of your program that calls another piece further away from the root. The general form of an autoload vector is the four-word structure shown in Figure 5–2.

Figure 5–2:   The Four-Word Structure of a Vector Autoload

Autoload Vector Entry

JSR @PC+2
Offset to pointer to autoload code
Segment descriptor address
Entry point address

MK–01055–00
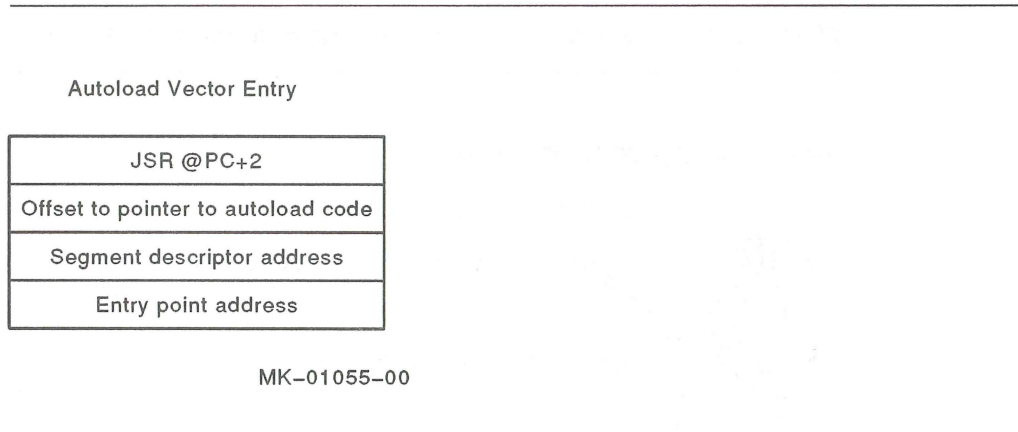
The JSR instruction passes control to the autoload processor, $AUTO. These two words are followed by the address of the descriptor for the segment to be loaded as well as the direct address calculated for the entry point of the piece to be loaded, if necessary.

Thus, when your program executes a call to a piece of your program further away from the root, control transfers to the autoload vector address and on to the autoload routine (inserted as a part of your program by the Task Builder). The autoload routine checks to see if the piece referred to is already in memory. If so, control is transferred to the location where the called routine resides, that is, to the entry point specified in the last word of the autoload vector.

If the desired piece is not currently in memory, the autoload routine loads the piece from disk (using information from the segment descriptor pointed to by the third word of the autoload vector). Once the appropriate piece is loaded, control is transferred to the entry point specified in the last word of the autoload vector.

5.2 Where are Autoload Vectors Really Needed?

You can request autoload vectors for all the pieces of your program, if you want to. If you use the easiest rule (described on the first page of this chapter) each global symbol referred to in a transfer-of-control statement closer to the root will have an autoload vector. But autoload vectors are only necessary when a transfer of control is made to a piece that is not currently in memory, so that it can be properly loaded. You can save four words for each unnecessary autoload vector you eliminate by using the autoload indicator only where it is really needed.

To understand how to request specific autoload vectors, you must understand how the Task Builder has stored the pieces of your program on disk, and how and when the appropriate pieces are loaded. This was discussed in Section 3.5.2; that information is reviewed here with a more complex example.

When you request overlays, the Task Builder constructs your executable program file such that pieces will be loaded in the most efficient manner. It does this by analyzing your ODL file. Pieces connected by a hyphen (-) are stored such that they will be loaded in one disk access. Pieces separated by a comma are stored such that they require a separate disk access for each piece.

For example, consider the overlay structure in Figure 5–3.

Figure 5–3: An Overlay Structure Without Autoload Vectors

MK–00589–00
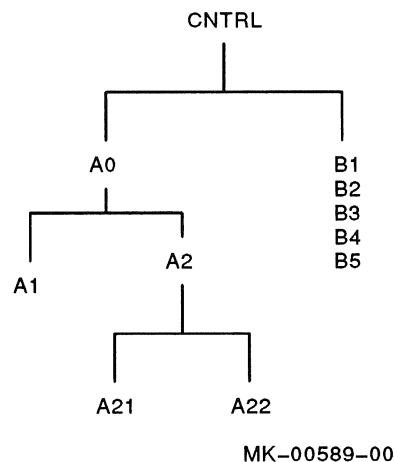
This structure can be represented (including the FORTRAN library F4POTS.OLB) by the following ODL file. Note that only the structure is shown—no autoload vectors for the moment, although they will be needed.

```
            .ROOT  CNTRL-LIBR-(AFCTR,BFCTR)
AFCTR:      .FCTR  AOWLIB-(A1WLIB,A2WLIB-(A21WLB,A22WLIB)
BFCTR:      .FCTR  B1-B2-B3-B4-B5-LIBR
AOWLIB:     .FCTR  A0-LIBR
A1WLIB:     .FCTR  A1-LIBR
A2WLIB:     .FCTR  A2-LIBR
A21WLB:     .FCTR  A1-LIBR
A22WLB:     .FCTR  A22-LIBR
LIBR:       .FCTR  LB:F4POTS/LB
            .END
```

Ignoring the factors needed to include the FORTRAN library, note that B1 through B5 are connected by hyphens. Hence, they are stored on disk as one segment. Suppose that the root, CNTRL, contains a call to B3. As long as an autoload vector has been generated for B3, it and B1, B2, B4, and B5 are loaded when the call is made, since these pieces have been stored as one segment on disk.

Once they are all loaded, B3 can call B1, B2, B4 and/or B5 using direct references (without autoload vectors). Likewise, they can all call each other using direct address references. The only piece you must request an autoload vector for is B3; you could have eliminated autoload vectors for B1, B2, B4, and B5. (Or, if CNTRL had called B5 first, you need have requested autoload vectors only for B5—the call sequence is as important a consideration as the fact that the items are connected by hyphens.)

For items separated by commas, the loading sequence is different. The Task Builder stores items separated by commas in separate segments on disk. At execute time, a reference to a piece "further out" (away from the root of the tree) causes all pieces between the calling piece and the called piece to be loaded.

Suppose that in the above example, CNTRL calls A21. Assuming that an autoload vector exists for the reference to A21, the pieces A0 and A2 are loaded along with A21. At this point, any routine in that path can call any other routine using a direct reference. That is, CNTRL could call A0, A2, or A21; A21 could call A2 or A0. A2 could call A21 or A0, and A0 could call A2 or A21 using direct references. So, as long as A21 is called first, it is the only piece along that path that needs an indirect reference, an autoload vector, to ensure that it is loaded when a piece closer to the root calls it.

Suppose, on the other hand, that CNTRL calls A0 first. Only A0 is loaded when that particular call is made. A0 needs an autoload vector. If A0 then called A22, A22 would also need an autoload vector. However, at that point, A2 and A22 will be loaded into memory. CNTRL could call A0, A2, and A22; A22 could call A2, A2 could call A0, and A0 could call A2 and A22 using direct references. No autoload vector is necessary for A2 in this case.

5.3 How to Request Specific Autoload Vectors

You request autoload vectors for a specific piece of the overlay by using an asterisk (*) in your ODL commands. For example, the following command causes autoload vectors to be generated for global symbols in A and C that are referred to in transfer-of-control statements from segments closer to the root. No autoload vectors are generated for such global symbols in B, however. (No autoload vectors are necessary for CNTRL since it is loaded by the run-time system when the program is first executed.)

```
.ROOT CNTRL-(*A,B-*C)
```

You can put the asterisk before any item in an ODL .ROOT or .FCTR command.

5.3.1 Asterisk Before File Names and Program Sections

If you put an asterisk before a file name or a program section name with the I (instruction) attribute † , an autoload vector is generated for each global symbol in the file or section (such as an entry point) referred to in a transfer-of-control statement from another piece closer to the root.

---

† Program sections are units processed by the Task Builder; they are described in Chapter 6.

5.3.2 Asterisk Before Items in Parentheses

If you put an asterisk before items enclosed in parentheses, autoload vectors are generated for each item within the parentheses.

For example:

```
BRNCH1:  .FCTR A-*(B,C,D)
```

Autoload vectors are generated for B, C, and D.

5.3.3 Asterisk Before Names Defined in .FCTR Commands

If you put an asterisk before a name later defined in a .FCTR command, an autoload vector is generated for the first item in the .FCTR command. If the first item in the .FCTR command is preceded by a left parenthesis, all items within the parentheses in the .FCTR command will have autoload vectors, as long as they are referred to in transfer-of-control statements from pieces closer to the root.

For example:

```
        .ROOT MAIN-(*AFCTR,*BFCTR)
AFCTR:  .FCTR A0-ASUB1-ASUB2
BFCTR:  .FCTR (B0-(B1,B2))
        .END
```

Autoload vectors are generated for A0, B0, B1, and B2, as long as they are referred to in transfer-of-control statements from pieces closer to the root. Autoload vectors are not generated for ASUB1 and ASUB2.

5.3.4 Asterisk Before Names Defined in .NAME Command

As mentioned in Section 4.2, the .NAME command assigns a name to a piece of the overlay structure. The piece, as it turns out, is the "segment" defined immediately following the name when the name is used in a .ROOT or .FCTR command. (Remember that a segment was defined as loadable with one disk access. See Section 3.5.2.)

For example, consider the following (unlikely) ODL file:

```
        .NAME   WECAN
        .NAME   ONLY
        .NAME   WONDER
        .ROOT   CNTRL-(WECAN-(A,B),BFCTR,CFCTR)
BFCTR:  .FCTR   *ONLY-B0-B1-B2-B3
CFCTR:  .FCTR   C1-(*WONDER-C2-(C3,C4))
        .END
```

The name WECAN applies to a null segment; this is how the .NAME command would be used to define a null root for a co-tree, as described in Section 4.2. There is no reason to generate an autoload vector for a null segment.

The name ONLY applies to the segment formed by the pieces B0, B1, B2, and B3. Hence, the asterisk before ONLY means that autoload vectors are generated for each of these pieces.

The name WONDER applies to the segment formed by C2. Hence, the asterisk before WONDER means that autoload vectors are generated for C2.

5.4 Example of Specific Autoload Vector Requests

Now that you understand how autoload indicators apply to the various elements possible in an ODL file, return to the specific example in Figure 5–3 and the ODL file following it.

Suppose that CNTRL calls B3, which makes various calls to B1, B2, B4, and B5 before it returns control. CNTRL then calls A21, which calls A2 and A0. Control returns to A21, which returns control to CNTRL. CNTRL then calls A1 and A22.

Thus, you need apply the autoload indicator only to B3 in the "B" branch of the structure. In the "A" branches, you must supply an autoload indicator for A21, A1, and A22. You can accomplish this with the following ODL file:

```
            .ROOT  CNTRL-LIBR-(AFCTR,BFCTR)
AFCTR:      .FCTR  AOWLIB-(A1WLIB,A2WLIB-(A21WLB,A22WLIB)
BFCTR:      .FCTR  B1-B2-*B3-B4-B5-LIBR
AOWLIB:     .FCTR  A0-LIBR
A1WLIB:     .FCTR  *A1-LIBR
A2WLIB:     .FCTR  A2-LIBR
A21WLB:     .FCTR  *A21-LIBR
A22WLB:     .FCTR  *A22-LIBR
LIBR:       .FCTR  LB:F4POTS/LB
            .END
```

5.5 If You Make a Mistake

If you make an error in placing asterisks, you will have problems within your program. Suppose you leave out an asterisk so that an autoload vector is not generated for a piece of your program. That piece will then be called with a direct reference, and if it is not actually in memory, control passes to whatever happens to be there. The Task Builder cannot diagnose the error at build time, because it does not attempt to analyze the sequence of calls in your program.

So, be very careful in requesting that the Task Builder generate autoload vectors for only some of the pieces making up your program.

Working with Program Sections

So far, overlay techniques have been discussed in terms of units you are familiar with: files consisting of separately compiled or assembled programs or subprograms, or of library files. This chapter discusses the units these files consist of—the units the Task Builder actually works with—program sections.

6.1 What is a Program Section?

All the language translators produce program sections. With MACRO, you work directly with program sections. The .PSECT directive of the MACRO language lets you name and define exactly what goes into these units. With the higher-level languages, the compiler handles most aspects of generating and assigning attributes to program sections for you (Figure 6–1).

Figure 6–1: The Task Builder Works with Program Sections
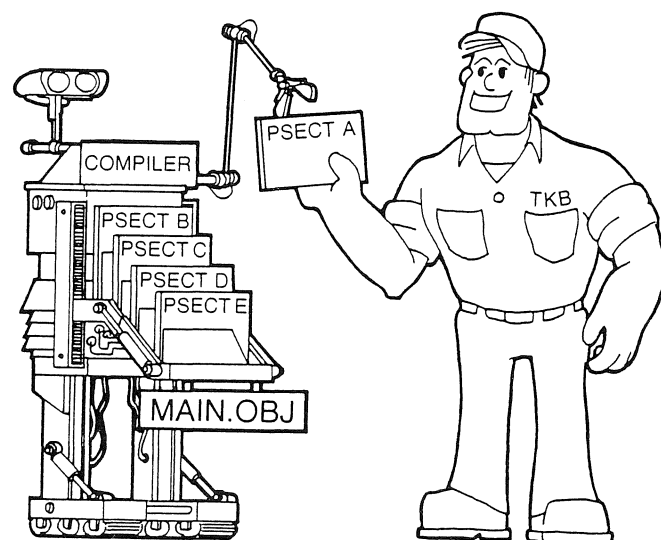
MK–00590–00

The Task Builder allocates space differently for different parts of a program or subprogram, depending on certain attributes. With program sections, you can take full advantage of these attributes. A case where you would need program sections involves common areas, a programming feature of all languages.

(COBOL programmers will recognize common areas as the LINKAGE SECTION of the DATA DIVISION in their program or subprogram.)

Common areas are areas in memory that are shared between programs and subprograms. A program can place data in a common area and pass control to another program or subprogram that uses the data in the common area for processing. Thus, one of the most obvious attributes of a program section is whether or not it consists of instructions or data.

Another attribute inherent in common areas is that the space allocated in both the program and subprogram should be overlaid, rather than concatenated. For example, suppose two FORTRAN programs define a common area named A. In each compilation, the compiler defines a program section named A with the "overlay" attribute. The Task Builder, when requested to build an executable program file from the two object files, can allocate one area of memory to A. It can resolve references to A such that both programs refer to the same area. (Note the word "can." Whether the Task Builder actually uses the same area or not depends upon whether the two definitions reside along the same path of the overlay structure, as described in Section 6.2.)

This illustrates yet another attribute of a program section: whether it is local or global. The compiler defines common areas as global; that is, they can be referred to by other separately compiled programs or subprograms. (MACRO programmers, again, state the attributes of program sections directly.) That is, global program section names can be referred to by other, separately compiled, programs or subprograms.

A program section has two other attributes: (1) whether it can be accessed read/write or read-only, and (2) whether its address is absolute or relocatable. See the *PDP-11 MACRO-11 Language Reference Manual* if you are interested in further detail about program section attributes.

6.2 Allocating Space for Global Program Sections

As mentioned in Section 6.1, one of the attributes of a program section is whether it is local or global. The Task Builder must determine where to allocate space for global program sections. If a FORTRAN program and subprogram both define a common area A, for example, where is the space for A to be allocated?

Figure 6-2 shows two examples. The common block COMA is defined in the pieces A2 and A21. The Task Builder allocates the space for COMA in A2 because that segment is closer to the root. The common block COMB, however, is defined in A0 and B0. These pieces are not on the same path, so the Task Builder allocates space for COMB in both A0 and B0. Note that A0 and B0 cannot communicate through COMB. When the overlay containing B0 is loaded, for example, any data stored in COMB by A0 is lost.

Figure 6–2: Allocating Space for Global Program Sections

ROOT

A0
DEF COMB

B0
DEF COMB

A1

A2
DEF COMA

SPACE FOR COMB ALLOCATED IN
BOTH A0 AND B0 BECAUSE THEY
ARE ON DIFFERENT PATHS

SPACE FOR COMA ALLOCATED
IN A2 BECAUSE IT IS CLOSER
TO THE ROOT

A21
DEF COMA

A22
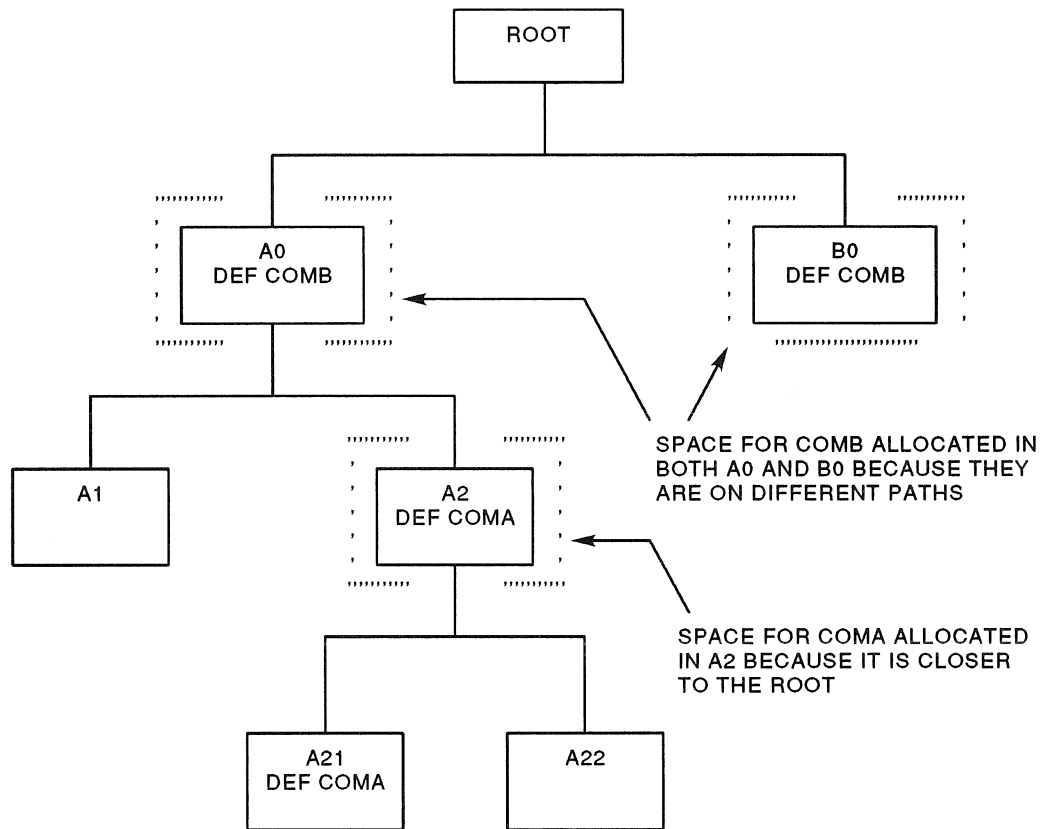
MK-00591-00

6.3 How the Task Builder Orders Program Sections

As discussed in Chapter 3, the Task Builder creates the executable file in
segments such that each segment is loaded with one disk access. It also orders
program sections within each segment.

The Task Builder groups the program sections according to access code.
Read/write program sections are assigned the lower addresses; read-only
program sections follow in higher-address space. Within these groups, program
sections are ordered alphabetically. The higher-level language translators are
designed to create names for program sections to take advantage of this feature
of the Task Builder. (If you are programming in MACRO and for some reason
do not want alphabetic ordering of program sections, you can use the /SQ switch
(Section 11.25) to request sequential ordering of program sections.)

As an example, suppose that the Task Builder is working with a segment
consisting of three pieces named IN1, IN2, and IN3, specified in the ODL file as
follows:

```
FCTR1:  .FCTR IN1-IN2-IN3
```

Table 6–1 shows the program sections each file contains and their access codes and allocation codes.

Table 6–1: Program Sections for IN1, IN2, AND IN3

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

The Task Builder first determines the amount of space to allocate for each program section. Program section A appears in two files and has the overlay (OVR) attribute. The OVR attribute causes the Task Builder to allocate the largest of the two sizes, or 300 bytes, for A. Program section B appears twice and has the concatenate (CON) attribute. Thus, the total allocation for B is the sum of the lengths of each occurrence, or 220 bytes. The portion allocated for IN1 (100 bytes) is assigned the lowest addresses, since it appears first in the input file list. Program section C appears twice and, since it has the concatenate attribute, is allocated 200 bytes.

The Task Builder then groups the program sections according to their access codes, with the read/write sections being allocated the lowest addresses, followed by the read-only sections (see Figure 6–3).

Figure 6–3: Allocation of Program Sections for IN1, IN2, and IN3

A (300)	Read/write
B (220)	Read/write
C (200)	Read-only
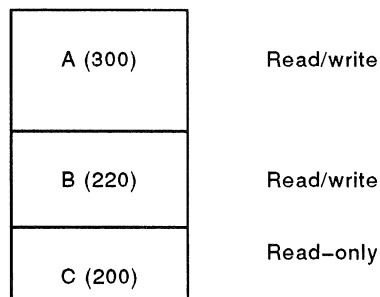
MK–00592–00

The only other factor contributing to how the Task Builder allocates and orders program sections is whether the section consists of instructions or data. The Task Builder always allocates address space for a program section beginning on a word boundary. If the program section has the instruction (I) and concatenate (CON) attributes, the Task Builder appends the space so that each piece begins on a word boundary. (This eliminates the possibility of an odd-address transfer.) If the program section has the data (D) and concatenate (CON) attributes, however, and the space contributed by a piece ends on a byte rather than a word boundary, the space for the next piece is appended starting with the next byte.

6.4 The Task Builder's .PSECT Command

You can direct the placement of program sections at build time with the .PSECT command. Suppose, for example, that you wanted to place the common block COMB from the example in Figure 6–2 in the root section of the program. This would make the area accessible from both A0 and B0; they would be able to pass data in the common area, as desired. This could be accomplished with the following ODL commands:

```
.PSECT  COMB,RW,GBL,REL,OVR,D
.ROOT   ROOT-COMB-LIBR-*(AOWL-*(A1WL,A2WL-*(A1WL,A22WL)))
    .
    .
    .
.END
```

In the .PSECT command, you specify the program section name first, followed by its attributes in any order. The attributes shown here are typical for a common block: read/write (RW), global (GBL), relocatable (REL), overlaid (OVR), and data (D).

6.5 Using .NAME to Make a Data PSECT Autoloadable

You can construct an object file consisting only of data and make that file autoloadable. For example, suppose that, using MACRO, you constructed a file consisting of a program section containing error messages. Suppose further that you wanted to overlay this file, because it was needed only when a certain subprogram was running. Such overlaying can be accomplished, and is perhaps best explained by example.

Consider the subprogram ERDAT.OBJ, which processes an error value. If the value is 0, the subprogram calls a routine named ALRIT.OBJ. If the value is not 0, however, it displays one of the error messages contained in the file MSG.OBJ, consisting of a program section with the D attribute, using, say, the MACRO language .ASCII command to define each particular error message.

There is no reason why ALRIT and MSG must be in memory at the same time. You can overlay ALRIT and MSG, by using the .NAME directive to define a name and attributes for the file MSG. For example:

```
        .ROOT   MAIN-*(OTHER,ERMSG-(ALRIT,MSG))
        .NAME   MESAGE,GBL
MSGF:   .FCTR   MESAGE-MSG
        .END
```

That is, you must include a .NAME command defining a global name (MESAGE in this case) for the data file (MSG in this case). The Task Builder generates the global symbol MESAGE and enters it into the symbol table for segment MESAGE. Since this segment is included for the generation of autoload vectors, an autoload vector is created for the segment referred to by the global symbol MESAGE. Because it consists only of a program section with the data (D) attribute, however, the Task Builder constructs a special autoload vector. The last word, normally an "entry point address" for an autoloaded segment, instead refers to the symbol $$RTS (generated by the Task Builder, and containing a simple return instruction).

So, program ERMSG includes a CALL MESAGE statement or JSR PC,MESAGE instruction to load the error message file MSG. At execution time, the CALL or JSR would transfer control to the autoload vector, which would call the $AUTO routine to load the necessary segment, if necessary, and then transfer control inline (back to the statement or instruction in ERMSG following the CALL or JSR).

In short, to make a data segment autoloadable:

1. Use a .NAME command with the GBL attribute to define a global name for the segment.

2. Make sure that an autoload vector is generated for the segment by using the autoload indicator as appropriate.

3. Load the segment by using a control instruction, such as CALL or JSR, referencing the name defined in the .NAME command.

Section 13.4 describes the .NAME command in detail.

6.6 More About Program Sections: Deciphering the Map

The first time you look at a memory map file, particularly if you use one of the higher-level languages, you may wonder "What has happened to my program?" The listing for even a small program is lengthy, and contains symbols that you never saw before.

The Task Builder presents a rather dazzling array of information in the memory map. You have seen parts of a map file in previous chapters dealing with overlays. Now that you understand what program sections are, and how the Task Builder orders them, more of the pieces can be fit together.

The following pages show a BASIC-PLUS-2 program consisting of a main program and three subroutines. (This program is a slight simplification of the one used to describe co-trees in Chapter 4.) USER simply calls three subroutines; INTRO accepts user input from the terminal; CRUNCH performs numeric computation; and CHATR prints the results.

Following the source listing, the ODL file for building the compiled object files USER.OBJ, INTRO.OBJ, CRUNCH.OBJ, and CHATR.OBJ is shown as a root segment with three overlaid segments.

Next, you see the map produced from the build. Note that this is a "short" map; you can request more information and more detail by using the /MA and /-SH switches on the map file name (see Sections 11.14 and 11.23). Section 11.23 gives a complete description of all the information given in a map file, organized for easy reference.

We used the /-WI switch to produce the 80-column listing for this manual. If you use a typical line printer for your map file, you would probably want to produce a 132-column listing (the default).

In this chapter, focus is on the information contained in a map for each segment. This information is interesting and may be helpful if you are trying to debug a program at the machine-language level. The Task Builder provides complete information on the structure of each segment it constructs.

Page 2 of the map, for example, shows the start of the information for the segment USER (the root segment of the program). Note the memory allocation synopsis; each program section in the segment USER is listed (under the heading SECTION), continuing to page 3 of the map.

The three columns of numbers to the right give the starting address (in octal) and length (in octal and decimal) of each program section or piece of a program section in the root. Note that the starting address of the first program section in the root segment is 2000[8]. The starting address is not 0 (even though you are dealing with relocatable addresses) because the Task Builder uses the first 2000[8] bytes in your program for some special-purpose information.

The run-time systems that your program can run under use the first 1000[8] bytes of this space to pass information to the RSTS/E monitor. MACRO programmers may be interested in this area (described in the *RSTS/E System Directives Manual*). Note that programmers in other languages cannot access this area as easily as MACRO programmers.

Likewise, the second 1000[8] bytes are allocated for what is called the stack. It is an area that can be used by assembly language programmers to pass values between programs or subprograms. A value can be "pushed on the stack" by one program and "popped from the stack" by another. Assembly language and C language programmers may be interested to know that you can allocate more (or less) than 1000[8] bytes for the stack by using the stack option, as described in Section 12.26. Higher-level programmers should not try to save space by reducing the amount of space taken by the stack; the compiler may well have generated code in your program to push data on the stack. Trying to decrease the size can cause unpredictable results when your program is executed. C programmers, on the other hand, may need to increase the amount of STACK space.

In any case, the first program section listed in segment USER is named ".BLK"—this is the name given by the assembler and compilers to what is called the blank common area. MACRO, FORTRAN, and BASIC programmers may recognize this; it is simply the area assigned to common areas that you do not define a specific name for in your program. In this case, no common areas are used, and so the program section shows a length of 00000. bytes.

Next, the BP2OTS section shows the library routines that the Task Builder has inserted into the root segment from the BP2OTS library. The routine names are listed in the "TITLE" column.

The program sections whose names begin with $ have been generated by the compiler. They contain the code and data generated by the compilation of the source files. It is interesting to note, for example, that the program section named $CODE is the only program section with the instruction (I) attribute. Logically enough, this is the name that the compiler gives to the machine instructions it generates from the source code, in this case for the USER program. If you look on following pages, you see this same $CODE section appearing in all the segments, and can begin to understand how the compilers have been designed to generate names to take advantage of the Task Builder's alphabetic ordering of program sections.

The program sections whose names begin with $$ have either been generated by the Task Builder itself or are routines supplied from the default library SYSLIB.OLB. For example, $$ALVC is the name of the program section containing the autoload vectors for each segment. (Appendix D lists this and other symbols reserved for use by the Task Builder.) Likewise, $$AUTO is the autoloading code needed to load overlay segments outside the root. It is supplied from the default library. The PSECT $$RTS is another example. This is the return instruction, mentioned in Section 6.5, which can be used to load a data segment.

Following this listing of program sections and their memory allocations is a list of global symbols defined or used in each segment. Note that the only global symbols recognizable from the source program are USER, INTRO, CRUNCH, and CHATR, assigned to the entry point of each routine by the compiler. The rest have been assigned by the compiler, or belong to the library routines that have been inserted into the segment.

The last page of the listing contains Task Builder statistics on the build. These statistics are mainly useful in analyzing Task Builder performance on your system, as described in Appendix E.

6.6.1 Source for Program USER

```
10          CALL  INTRO(A1%,B1%)
20          CALL  CRUNCH(A1%,B1%,SUMM%,PRODUCT%,DIFFER%)
30          CALL  CHATR(A1%,B1%,SUMM%,PRODUCT%,DIFFER%)
40          END
```

6.6.2 Source for Subprogram INTRO

```
10          SUB  INTRO(AA%,BA%)
20          INPUT  "INPUT TWO NUMBERS";AA%,BA%
30          SUBEND
```

6.6.3 Source for Subprogram CRUNCH

```
10          SUB  CRUNCH(AB%,BB%,CA%,CB%,CC%)
20          CA%  =  AB%  +  BB%
30          CB%  =  AB%  *  BB%
40          CC%  =  AB%  -  BB%
50          SUBEND
```

6.6.4 Source for Subprogram CHATR

```
10          SUB  CHATR(AC%,BC%,CA%,CB%,CC%)
20          PRINT  "THE SUM OF ";AC%;" AND ";BC%;" IS ";CA%
30          PRINT  "THE PRODUCT OF ";AC%;" AND ";BC%;" IS ";CB%
40          PRINT  "THE DIFFERENCE OF ";AC%;" AND ";BC%;" IS ";CC%
50          SUBEND
```

6.6.5 Overlay Description File FRED.ODL

```
            .ROOT  USWL-*(INTRWL,CRUNWL,CHATWL)
USWL:       .FCTR  USER-LIBR
INTRWL:     .FCTR  INTRO-LIBR
CRUNWL:     .FCTR  CRUNCH-LIBR
CHATWL:     .FCTR  CHATR-LIBR
LIBR:       .FCTR  LB:BP2OTS/LB
            .END
```

6.6.6   Task Builder Command File

```
USER,USER=FRED/MP
UNITS=12
ASG=SY:5:6:7:8:9:10:11:12
EXTTSK=512
//
```

6.6.7   Task Builder Listing

```
USER.TSK   Memory allocation map   TKB 08.006      Page 1
                    15-MAY-90    14:00


Partition name : GEN
Identification : 000708
Task  UIC      : [30,21]
Stack      limits: 001000 001777 001000 00512.
PRG xfr address: 016030
Total address windows: 1.
Task   extension    : 512. words
Task   image   size : 6304. words
Total  task    size : 6816. Words
Task address limits: 000000 030453
R-W disk blk limits: 000002 000041 000040 00032.




USER.TSK Overlay description:

Base    Top        Length
----    ---        ------
000000  020773  020774  08700.     USER
020774  030453  007460  03888.       INTRO
020774  021427  000434  00284.       CRUNCH
020774  026023  005030  02584.       CHATR


USER.TSK   Memory allocation map   TKB 08.006      Page 2
USER                  15-MAY-90    14:00




*** Root segment: USER


R/W mem  limits: 000000 020773 020774 08700.
Disk blk limits: 000002 000022 000021 00017.
```

Memory allocation synopsis:

Section					Title	Ident	File
-------					-----	-----	----
. BLK.:(RW,I,LCL,REL,CON)	002000	000000	00000.				
BP2OTS:(RO,I,LCL,REL,CON)	002000	014030	06168.				
	002000	000352	00234.		$CALLS	21CM	BP2OTS.OLB
	002352	000026	00022.		$ICEND	07CM	BP2OTS.OLB
	002400	000460	00304.		$ERTHR	70CM	BP2OTS.OLB
	003060	000132	00090.		$JMOVS	05CM	BP2OTS.OLB
	003212	000746	00486.		$IEULT	31RE	BP2OTS.OLB
	004160	001114	00588.		$IVOPN	60RE	BP2OTS.OLB
	005274	000000	00000.		$ICIO0	04CM	BP2OTS.OLB
	005274	001554	00876.		$BINIT	71RE	BP2OTS.OLB
	007050	000450	00296.		$CNTRL	14CM	BP2OTS.OLB
	007520	001102	00578.		$STMSC	25CM	BP2OTS.OLB
	010622	000022	00018.		$CALLR	06CM	BP2OTS.OLB
	010644	002156	01134.		$ERROR	76RE	BP2OTS.OLB
	013022	000162	00114.		$ICRCL	16CM	BP2OTS.OLB
	013204	000374	00252.		$IMALQ	12CM	BP2OTS.OLB
	013600	000260	00176.		$ICFSS	27RE	BP2OTS.OLB
	014060	000062	00050.		$ICCRL	00CM	BP2OTS.OLB
	014142	000222	00146.		$STGTA	04CM	BP2OTS.OLB
	014364	000230	00152.		$ICEOL	21CM	BP2OTS.OLB
	014614	000000	00000.		$BFPEI	06CM	BP2OTS.OLB
	014614	000114	00076.		$ERROT	71RE	BP2OTS.OLB
	014730	000242	00162.		RQLCB	69CM	BP2OTS.OLB
	015172	000004	00004.		$BFPER	06CM	BP2OTS.OLB
	015176	000070	00056.		$PROCT	00CM	BP2OTS.OLB
	015266	000266	00182.		$JCONV	03CM	BP2OTS.OLB
	015554	000062	00050.		$BXTRA	05RE	BP2OTS.OLB
	015636	000034	00028.		$BBTKS	01RE	BP2OTS.OLB
	015672	000112	00074.		$ICULT	04CM	BP2OTS.OLB
	016004	000024	00020.		SAVRG	69CM	BP2OTS.OLB
$ARRAY:(RW,D,LCL,REL,CON)	016030	000000	00000.				
	016030	000000	00000.		USER	000708	USER.OBJ
$CODE :(RO,I,LCL,REL,CON)	016030	000204	00132.				
	016030	000204	00132.		USER	000708	USER.OBJ
$FLAGR:(RW,D,GBL,REL,CON)	016234	000000	00000.				
	016234	000000	00000.		USER	000708	USER.OBJ
$FLAGS:(RW,D,GBL,REL,CON)	016234	000010	00008.				
	016234	000002	00002.		USER	000708	USER.OBJ
$FLAGS:(RW,D,GBL,REL,CON)	016244	000000	00000.				
	016244	000000	00000.		USER	000708	USER.OBJ
$ICIO0:(RW,D,GBL,REL,OVR)	016244	000030	00024.				
	016244	000000	00000.		USER	000708	USER.OBJ
	016244	000030	00024.		$ICIO0	04CM	BP2OTS.OLB

```
$ICIO1:(RW,D,GBL,REL,OVR)       016274 000200 00128.
                                016274 000200 00128. USER   000708 USER.OBJ
$IDATA:(RW,D,LCL,REL,CON)       016474 000012 00010.
                                016474 000012 00010. USER   000708 USER.OBJ
$PDATA:(RO,D,LCL,REL,CON)       016506 000000 00000.
                                016506 000000 00000. USER   000708 USER.OBJ
$STRNG:(RW,D,LCL,REL,CON)       016506 000000 00000.
                                016506 000000 00000. USER   000708 USER.OBJ
$TDATA:(RW,D,LCL,REL,CON)       016506 000000 00000.
                                016506 000000 00000. USER   000708 USER.OBJ
$$ALER:(RO,I,LCL,REL,CON)       016506 000024 00020.
$$ALVC:(RO,I,LCL,REL,CON)       016532 000030 00024.
$$AUTO:(RO,I,LCL,REL,CON)       016562 000142 00098.
$$BP2 :(RW,D,GBL,REL,OVR)       016724 001440 00800.
                                016724 001440 00800. USER   000708 USER.OBJ
$$MRKS:(RO,I,LCL,REL,OVR)       020364 000076 00062.
$$OVDT:(RW,D,LCL,REL,OVR)       020462 000024 00020.
$$OVRS:(RW,I,LCL,ABS,CON)       000000 000000 00000.
$$PDLS:(RO,I,LCL,REL,OVR)       020506 000002 00002.
$$RDSG:(RO,I,LCL,REL,OVR)       020510 000144 00100.
$$RESL:(RO,I,LCL,REL,CON)       020654 000034 00028.
$$RGDS:(RW,D,LCL,REL,CON)       020710 000000 00000.
$$RTQ :(RO,I,GBL,REL,OVR)       020710 000000 00000.
$$RTR :(RO,I,GBL,REL,OVR)       020710 000000 00000.
$$RTS :(RO,I,GBL,REL,OVR)       020710 000002 00002.
$$SGD0:(RW,D,LCL,REL,OVR)       020712 000000 00000.
$$SGD1:(RW,D,LCL,REL,CON)       020712 000060 00048.
$$SGD2:(RW,D,LCL,REL,OVR)       020772 000002 00002.
$$WNDS:(RW,D,LCL,REL,CON)       020774 000000 00000.
```

Global symbols:

```
BEQ$   007202-R   ERT$   002612-R   MSI$IM 003100-R   ULK$    003212-R
BGE$   007212-R   ERT$X  002626-R   NOBRA  007204-R   $ABNEX  011416-R
BGT$   007210-R   FLN$   002400-R   NOI$A  003204-R   $AFTS1  010634-R
BLE$   007200-R   FPUERR 016272-R   NOI$M  003176-R   $ATLIN  012260-R
BLT$   007222-R   FSS$   014002-R   NOI$S  003170-R   $ATOI   015266-R
BNE$   007220-R   GSC$   007066-R   OEA$   002646-R   $BALBF  013474-R
BRA$   007214-R   GSU$   007050-R   OEG$   002634-R   $BALMP  013456-R
CAL$   002000-R   INTRO  016532-R   OGB$   002706-R   $BCL    005052-R
CBR$   010622-R   JMC$   007130-R   OGS$   002736-R   $BINPT  003760-R
CHATR  016552-R   LIN$   002400-R   ONI$A  003162-R   $BOFS   004220-R
CLB$S  003132-R   LYN$   002562-R   ONI$M  003154-R   $BOP    004160-R
CLI$A  003142-R   MAD$   010602-R   ONI$S  003146-R   $BOPX   004176-R
CLI$M  003136-R   MOI$IA 003104-R   RCL$   013022-R   $BOUTP  003342-R
CLI$S  003132-R   MOI$IM 003100-R   REG$   007164-R   $BREAD  003476-R
CRUNCH 016542-R   MOI$IS 003074-R   RLI$M  003074-R   $BRTBF  013562-R
DPI$   003060-R   MOI$MA 003126-R   RLI$P  003116-R   $BRTMP  013542-R
END$   002352-R   MOI$MM 003122-R   RSI$M  003116-R   $CALFP  004114-R
EOL$   014364-R   MOI$MS 003116-R   RSI$P  003110-R   $CALIN  012246-R
ERL$   002562-R   MOI$SA 003070-R   RSM$   002414-R   $CCHDL  014644-R
ERN$   002542-R   MOI$SM 003064-R   RSU$   002424-R   $CCXIT  014726-R
ERR$   002600-R   MOI$SS 003060-R   SBE$   002102-R   $CHKRL  014060-R
```

Working with Program Sections 6–11

```
$CLFQB 003720-R   $FPHSK 015172-R   $IOERV 006710-R   $SETSC 006236-R
$CLOSR 013034-R   $FPUER 014614-R   $IOTST 004124-R   $SPEC  003224-R
$CLSAL 013164-R   $FRCER 014634-R   $MEMPR 014630-R   $STCRE 010116-R
$CLSFQ 004106-R   $FSS   004030-R   $MEMP1 011674-R   $STCRX 010122-R
$CLSHD 013152-R   $FSSCN 013600-R   $MNIUS 010456-R   $STMOV 010150-R
$CLSTK 010220-R   $FSSCZ 013602-R   $MNSUB 010522-R   $STMVX 010162-R
$CLXRB 003742-R   $GSACM 013204-R   $MREST 011004-R   $SYSHD 011274-R
$CNVIA 015456-R   $GTPTN 007734-R   $NOREX 011520-R   $VALDC 015710-R
$DATRC 006626-R   $GTPTR 010022-R   $ODDAD 014624-R   $VALID 015672-R
$DATRS 006604-R   $GTROM 007520-R   $ODDA1 011654-R   $VREAD 003252-R
$DOIT  012476-R   $GTR01 014142-R   $ONERG 011126-R   $XWRT  004002-R
$DOIT1 012520-R   $GTR23 014246-R   $OTSVA 016724-R   $$MAXC 000017
$ERROR 014614-R   $GTSTN 007724-R   $POMSK 007440-R   ..B2TK 015640-R
$ERRT1 010660-R   $GTSTR 010010-R   $PROCT 015176-R   ..CRLF 015576-R
$ERTXT 011532-R   $ICIO0 016244-R   $POMSK 007242-R   ..PMD  015574-R
$EXTSP 007666-R   $ICJMP 015754-R   $PSMS2 007234-R   ..PTXT 015602-R
$FLSAL 014426-R   $ICJM1 015754-R   $PSMS3 007226-R   ..RSTT 015554-R
$FLSFR 014436-R   $INITM 005274-R   $RELCB 014730-R   ..SVFQ 004136-R
$FLSNL 014454-R   $INITS 002270-R   $REQCB 015032-R
$FLUSH 014464-R   $INTCM 006432-R   $RSU2  010644-R
$FPASX 015174-R   $IOERS 006724-R   $SAVRE 016004-R
```

USER.TSK   Memory allocation map   TKB 08.006   Page 5
INTRO   15-MAY-90   14:00

*** Segment: INTRO

R/W mem limits: 020774 030453 007460 03888.
Disk blk limits: 000023 000032 000010 00008.

Memory allocation synopsis:

Section			Title	Ident	File
-------			-----	-----	----
. BLK.:(RW,I,LCL,REL,CON)	020774 000000 00000.				
BP2OTS:(RO,I,LCL,REL,CON)	020774 007272 03770.				
	020774 000422 00274.		$ICINI	23CM	BP2OTS.OLB
	021416 002004 01028.		$ICRED	53CM	BP2OTS.OLB
	023422 000656 00430.		$ICWRT	04CM	BP2OTS.OLB
	024300 000642 00418.		$STMOS	16CM	BP2OTS.OLB
	025142 002404 01284.		$ECONV	24CM	BP2OTS.OLB
	027546 000226 00150.		$ICFNS	11RE	BP2OTS.OLB
	027774 000202 00130.		$STLSS	08CM	BP2OTS.OLB
	030176 000070 00056.		$STFN1	06CM	BP2OTS.OLB
$ARRAY:(RW,D,LCL,REL,CON)	030266 000000 00000.				
	030266 000000 00000.		INTRO	000708	INTRO.OBJ
$CODE :(RO,I,LCL,REL,CON)	030266 000134 00092.				
	030266 000134 00092.		INTRO	000708	INTRO.OBJ
$FLAGR:(RW,D,GBL,REL,CON)	016234 000000 00000.				
	016234 000000 00000.		INTRO	000708	INTRO.OBJ
$FLAGS:(RW,D,GBL,REL,CON)	016234 000010 00008.				
	016236 000002 00002.		INTRO	000708	INTRO.OBJ
$FLAGT:(RW,D,GBL,REL,CON)	016244 000000 00000.				
	016244 000000 00000.		INTRO	000708	INTRO.OBJ
$IDATA:(RW,D,LCL,REL,CON)	030422 000004 00004.				
	030422 000004 00004.		INTRO	000708	INTRO.OBJ
$PDATA:(RO,D,LCL,REL,CON)	030426 000026 00022.				
	030426 000026 00022.		INTRO	000708	INTRO.OBJ
$STRNG:(RW,D,LCL,REL,CON)	030454 000000 00000.				
	030454 000000 00000.		INTRO	000708	INTRO.OBJ
$TDATA:(RW,D,LCL,REL,CON)	030454 000000 00000.				
	030454 000000 00000.		INTRO	000708	INTRO.OBJ
$$ALVC:(RO,I,LCL,REL,CON)	030454 000000 00000.				
$$RTS :(RO,I,GBL,REL,OVR)	020710 000002 00002.				

Global symbols:

```
ASC$    030234-R   IPT$   021246-R   LIT$    021166-R   MOS$AP 024516-R
BUF$    027752-R   IRD$   021002-R   LMA$1   030070-R   MOS$AS 024300-R
CCP$    027630-R   IVF$A  021416-R   LSS$AA  030000-R   MOS$MA 024622-R
CHR$    030256-R   IVI$A  021524-R   LSS$AM  030020-R   MOS$MM 024756-R
III$    021226-R   IVS$A  021564-R   LSS$AP  027774-R   MOS$MP 024716-R
IIN$    021120-R   LAM$1  030022-R   LSS$MA  030050-R   MOS$MS 024340-R
ILI$    021206-R   LAM$2  030026-R   LSS$PA  030044-R   MOS$PA 024626-R
ILS$    021102-R   LEN$   030224-R   MOS$AA  024556-R   MOS$PM 024510-R
INTRO   030266-R   LIS$   021064-R   MOS$AM  024502-R   MOS$PP 024712-R
```

```
MOS$PS 024330-R   NSS$PA 030156-R   STS$    027616-R   $FTOAX 026256-R
MOS$SA 024554-R   PVD$SI 023422-R   TAB$    027546-R   $INPTT 022170-R
MOS$SM 024444-R   PVF$SI 023432-R   WAT$    027732-R   $ISETP 022034-R
MOS$SP 024374-R   PVI$SI 023572-R   $ATOD   025142-R   $I4ER  021504-R
MOS$SS 024360-R   PVS$AI 023442-R   $CRLF   023724-R   $POS   027646-R
MOS$01 025056-R   RCT$   027604-R   $DMAXD  027544-R   $PRNSP 024060-R
MS1$01 024334-R   RST$   020774-R   $DTOA   026320-R   $PRNTL 024076-R
NMA$1  030172-R   SPC$   030176-R   $DTOAX  026324-R   $SETUP 023772-R
NSS$AA 030136-R   SPC$01 030176-R   $FMAXD  027542-R
NSS$MA 030162-R   STR$   030204-R   $FTOA   026252-R
```

*** Segment: CRUNCH

R/W mem limits: 020774 021427 000434 00284.
Disk blk limits: 000033 000033 000001 00001.

Memory allocation synopsis:

Section				Title	Ident	File
. BLK.:(RW,I,LCL,REL,CON)	020774	000000	00000.			
BP2OTS:(RO,I,LCL,REL,CON)	020774	000234	00156.			
	020774	000046	00038.	$JPADD	01CM	BP2OTS.OLB
	021042	000074	00060.	$JPMOV	02CM	BP2OTS.OLB
	021136	000024	00020	$JMUL	01CM	BP2OTS.OLB
	021162	000046	00038.	$JPSUB	01CM	BP2OTS.OLB
$ARRAY:(RW,D,LCL,REL,CON)	021230	000000	00000.			
	021230	000000	00000.	CRUNCH	000802	CRUNCH.OBJ
$CODE :(RO,I,LCL,REL,CON)	021230	000154	00108.			
	021230	000154	00108.	CRUNCH	000802	CRUNCH.OBJ
$FLAGR:(RW,D,GBL,REL,CON)	016234	000000	00000.			
	016234	000000	00000.	CRUNCH	000802	CRUNCH.OBJ
$FLAGS:(RW,D,GBL,REL,CON)	016234	000010	00008.			
	016240	000002	00002.	CRUNCH	000802	CRUNCH.OBJ
$FLAGT:(RW,D,GBL,REL,CON)	016244	000000	00000.			
	016244	000000	00000.	CRUNCH	000802	CRUNCH.OBJ
$IDATA:(RW,D,LCL,REL,CON)	021404	000012	00010.			
	021404	000012	00010.	CRUNCH	000802	CRUNCH.OBJ
$PDATA:(RO,D,LCL,REL,CON)	021416	000012	00010.			
	021416	000012	00010.	CRUNCH	000802	CRUNCH.OBJ
$STRNG:(RW,D,LCL,REL,CON)	021430	000000	00000.			
	021430	000000	00000.	CRUNCH	000802	CRUNCH.OBJ
$TDATA:(RW,D,LCL,REL,CON)	021430	000000	00000.			
	021430	000000	00000.	CRUNCH	000802	CRUNCH.OBJ
$$ALVC:(RO,I,LCL,REL,CON)	021430	000000	00000.			
$$RTS :(RO,I,GBL,REL,OVR)	020710	000002	00002.			

Global symbols:

```
ADI$IP 020774-R   CRUNCH 021230-R   MUI$IS 021150-R   SUI$PA 021222-R
ADI$MP 021010-R   MOI$IP 021042-R   MUI$MS 021144-R   SUI$PM 021214-R
ADI$PA 021034-R   MOI$MP 021056-R   MUI$PS 021136-R   SUI$PP 021172-R
ADI$PM 021026-R   MOI$PA 021074-R   MUI$SS 021152-R   SUI$PS 021206-R
ADI$PP 021004-R   MOI$PM 021102-R   NOI$P  021120-R   SUI$SP 021164-R
ADI$PS 021020-R   MOI$PP 021052-R   ONI$P  021110-R
ADI$SP 020776-R   MOI$PS 021066-R   SUI$IP 021162-R
CLI$P  021130-R   MOI$SP 021044-R   SUI$MP 021176-R
```

*** Segment: CHATR

R/W mem limits: 020774 026023 005030 02584.
Disk blk limits: 000034 000041 000006 00006.

Memory allocation synopsis:

Section			Title	Ident	File
. BLK.:(RW,I,LCL,REL,CON)	020774 000000 00000.				
BP2OTS:(RO,I,LCL,REL,CON)	020774 004316 02254.				
	020774 000422 00274.	$ICINI	23CM	BP2OTS.OLB
	021416 000074 00060.	$JPMOV	02CM	BP2OTS.OLB
	021512 000656 00430.	$ICWRT	04CM	BP2OTS.OLB
	022370 002404 01284.	$ECONV	24CM	BP2OTS.OLB
	024774 000226 00150.	$ICFNS	11RE	BP2OTS.OLB
	025222 000070 00056.	$STFN1	06CM	BP2OTS.OLB
$ARRAY:(RW,D,LCL,REL,CON)	025312 000000 00000.				
	025312 000000 00000.	CHATR	000802	CHATR.OBJ
$CODE :(RO,I,LCL,REL,CON)	025312 000352 00234.				
	025312 000352 00234.	CHATR	000802	CHATR.OBJ
$FLAGR:(RW,D,GBL,REL,CON)	016234 000000 00000.				
	016234 000000 00000.	CHATR	000802	CHATR.OBJ
$FLAGS:(RW,D,GBL,REL,CON)	016234 000010 00008.				
	016242 000002 00002.	CHATR	000802	CHATR.OBJ
$FLAGT:(RW,D,GBL,REL,CON)	016244 000000 00000.				
	016244 000000 00000.	CHATR	000802	CHATR.OBJ
$IDATA:(RW,D,LCL,REL,CON)	025664 000012 00010.				
	025664 000012 00010.	CHATR	000802	CHATR.OBJ
$PDATA:(RO,D,LCL,REL,CON)	025676 000126 00086.				
	025676 000126 00086.	CHATR	000802	CHATR.OBJ
$STRNG:(RW,D,LCL,REL,CON)	026024 000000 00000.				
	026024 000000 00000.	CHATR	000802	CHATR.OBJ
$TDATA:(RW,D,LCL,REL,CON)	026024 000000 00000.				
	026024 000000 00000.	CHATR	000802	CHATR.OBJ
$$ALVC:(RO,I,LCL,REL,CON)	026024 000000 00000.				
$$RTS :(RO,I,GBL,REL,OVR)	020710 000002 00002.				

Global symbols:

```
ASC$   025260-R   IRD$   021002-R   NOI$P  021474-R   STS$   025044-R
BUF$   025200-R   LEN$   025250-R   ONI$P  021464-R   TAB$   024774-R
CCP$   025056-R   LIS$   021064-R   PVD$SI 021512-R   WAT$   025160-R
CHATR  025312-R   LIT$   021166-R   PVF$SI 021522-R   $ATOD  022370-R
CHR$   025302-R   MOI$IP 021416-R   PVI$SI 021662-R   $CRLF  022014-R
CLI$P  021504-R   MOI$MP 021432-R   PVS$AI 021532-R   $DMAXD 024772-R
III$   021226-R   MOI$PA 021450-R   RCT$   025032-R   $DTOA  023546-R
IIN$   021120-R   MOI$PM 021456-R   RST$   020774-R   $DTOAX 023552-R
ILI$   021206-R   MOI$PP 021426-R   SPC$   025222-R   $FMAXD 024770-R
ILS$   021102-R   MOI$PS 021442-R   SPC$01 025222-R   $FTOA  023500-R
IPT$   021246-R   MOI$SP 021420-R   STR$   025230-R   $FTOAX 023504-R
```

$POS   025074-R $PRNSP 022150-R $PRNTL 022166-R $SETUP 022062-R

*** Task builder statistics:

    Total work file references: 25363.
    Work file reads: 0.
    Work file writes: 0.
    Size of core pool: 9630. words (37. pages)
    Size of work file: 8960. words (35. pages)

    Elapsed time:00:00:26

Part III
System Aspects

Building Your Own Memory-Resident Areas

Chapter 2 describes how to link a program to a resident library, one type of resident area. This chapter describes how to build your own resident area of routines or data.

7.1 What is a Resident Area?

A resident library (see Section 2.2.2) is one type of resident area. A resident library, when in memory, can be shared by many programs. It can consist of data or reentrant subroutines and is generally mapped read-only. (The term "reentrant" means that the program does not change any values within itself during execution. Thus the same code can be executed by one job while another job is also executing it; it can be "reentered" before the first job finishes.)

A resident common is another type of resident area. A resident common provides a way for two or more programs to communicate. One program can store data in the resident common for another program to retrieve at a later time. The resident common area is accessible to both. Resident common areas are generally mapped read/write.

7.2 The Steps in Creating a Resident Area

There are three steps in creating a resident area: the first involves the Task Builder. You must build the resident area and create a symbol table file that later allows other executable programs to link to the resident area. The symbol table file contains the global symbols defined in the library or common and either relative or absolute addresses for the symbols. This symbol table file is later used by the Task Builder when other builds reference the resident library or common.

The steps in building a resident area are described in the previous sections. The steps in creating a resident library are:

1. Using the MAKSIL utility to format Task Builder output to produce suitable input to the RSTS/E monitor. This step is described in the *RSTS/E Programmer's Utilities Manual*.

2. Establishing the area as memory-resident. You can perform this operation with the INSTALL DCL command as described in the *RSTS/E System Manager's Guide*.

7.3 How to Build Memory-Resident Areas

Building a memory-resident area is similar to building an executable program. The differences are:

1. You must declare that the task file † is not to contain a "header." The header on a task file contains information that is used by the loader in the run-time system when it loads an executable program. The information is used to set certain areas in the low 1000[8] bytes of address space in the user job area. Since resident areas do not occupy this low-address range, you do not need (and should not have) a header for the task file. You omit the header by appending the switch /-HD or /NOHD to the task file specification in the command line.

2. You must declare that no space is to be allocated for the "stack." The stack is an area of memory that can be used for temporary storage. The stack is accessed by the user program in low virtual address space (see Section 12.26). It should not be built into a resident area, which will occupy high virtual address space. You omit the stack by using the option STACK=0 in the build.

3. You must request a symbol table file as well as a task file. The symbol table file is the third output field on the TKB line. As described in Section 7.2, this file is used by the Task Builder when it links the resident area to a program that references the resident area.

4. You must declare whether the area is to be position independent (have relative addresses, so that it can be loaded anywhere in the job area) or absolute (have absolute addresses, so that it must be loaded into the same place in the job area each time it is used). The next two sections explain how to do this.

7.3.1 Building Position-Independent Resident Areas

A resident area can be either position independent or absolute. Position-independent areas can be placed anywhere in the user job area.

Declaring an area to be position independent causes the Task Builder to:

1. Include definitions for each root segment program section in the symbol table (.STB) file. A program can later reference this shared storage by program section name.

2. Generate relative addresses for the resident area, such that the resident area can be located anywhere in the user job area when it is linked to a program that references it. (This allows you to choose the automatic selection of the highest APR, or to select an APR in the LIBR, RESLIB, COMMON, RESCOM, RESSUP, OR SUPLIB option.)

You should declare an area to be position independent if:

1. The area contains code that executes correctly regardless of its position in the address space of the program that references it.

2. The area contains data that is not address dependent.

---

† The term "task file" is used instead of "executable file," since a memory-resident area is not executed with a RUN command. Rather, it is eventually linked to an executable program in a later build. The task file is the first file that you specify in a Task Builder command line, with default file type .TSK.

3. The area contains data that is referenced by a program (such data must reside in a named common block).

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

1. No code or data in the area should be included in the blank (unnamed) program section.

2. No code or data in a program that refers to the area should have a program section with the same name as a program section in the resident area.

3. The order in which address space is allocated to program sections (alphabetic or sequential) must be the same for the resident area and the program that refers to it.

To make an area position independent, use the /PI switch on the task or symbol table file name and specify the PAR option just to name a "partition" that the area is to occupy. (Do not use PAR to indicate a starting address and length in this case.) The partition name must be the same as the file name portion of the task and symbol table files.

Example

The following command line builds a position-independent area from the input files DAT1.DAT, DAT2.DAT, and DAT3.DAT:

```
RUN $TKB
TKB> DATLIB/-HD/PI,,DATLIB=DAT1.DAT,DAT2.DAT,DAT3.DAT
TKB> /
ENTER OPTIONS:
TKB> PAR=DATLIB
TKB> STACK=0
TKB> //
```

The /-HD and /PI switches are described in Chapter 13.

7.3.2 Building Absolute Resident Areas

Absolute resident areas must always occupy the same place in the user job area when linked to a program. If you build this type of area, only one program section, named .ABS., is included in the symbol table file. All references to code or data in such an area must be by global symbol name. Further, when you link a program to an absolute resident area, you must use the APR parameter to specify the correct location where the area is to be linked.

Use the PAR option to build an absolute resident area. The PAR option is described in Chapter 12. Briefly, the format is:

PAR=pname:base:length

where:

pname	is the partition name; this must be the same as the file name portion of the task file and symbol table file in the command line.
base	is the base address, in octal, that the resident area is always to occupy.
length	is the octal number of bytes of the area. If you omit the length argument, the length of the task file is used.

For example:

```
RUN $TKB
TKB>MYLIB/-HD,,MYLIB=CODE1,CODE2,CODE3
TKB>/
ENTER OPTIONS:
TKB>PAR=MYLIB:140000
TKB>STACK=0
TKB>//
```

The area above would always have to be linked as follows:

```
RUN $TKB
TKB>PROG=PROG
TKB>/
ENTER OPTIONS:
TKB>LIBR=MYLIB:RO:6
TKB>//
```

That is, since it was built to begin at location 140000, it must always be linked using APR 6. Note also that MYLIB.TSK and MYLIB.STB must be on the device in the account denoted by the system logical LB:, because LIBR was used rather than the RESLIB option.

7.4 Resident Areas with Memory-Resident Overlays

The Task Builder lets you construct what are called "memory-resident overlays" for resident areas. Memory-resident overlay segments are loaded from disk when your program is loaded; thereafter, they reside in memory as long as any other program in memory is using them. Memory-resident overlays share virtual address space, just as the disk-resident overlays that we have discussed. Unlike disk-resident overlays, memory-resident overlays do not share actual memory. Instead, they reside in separate areas of actual memory. The virtual address space is shared by the mapping technique described in Chapter 2.

For example, consider Figure 7–1. At time 1, the job area in virtual address space contains OVLY1, one segment of a resident area with memory-resident overlays. At time 2, the job area in virtual address space contains OVLY2, the other segment of the resident area with memory-resident overlays. Both segments OVLY1 and OVLY2 reside in physical memory; they are mapped into the virtual address space at different times.

Figure 7-1: Memory-Resident Overlays
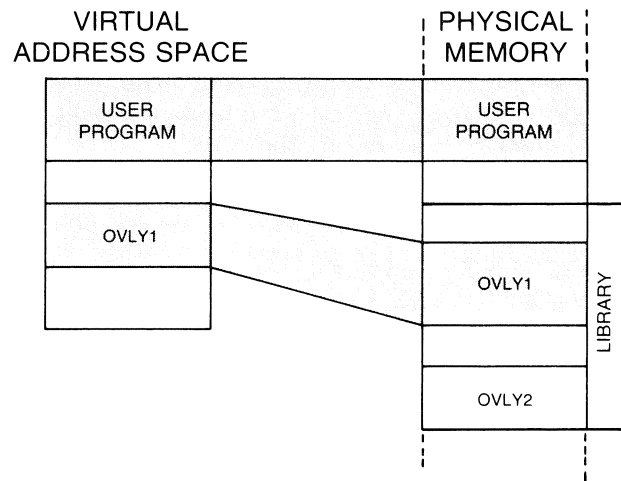
A) TIME1: USER PROGRAM REFERS TO OVLY1.
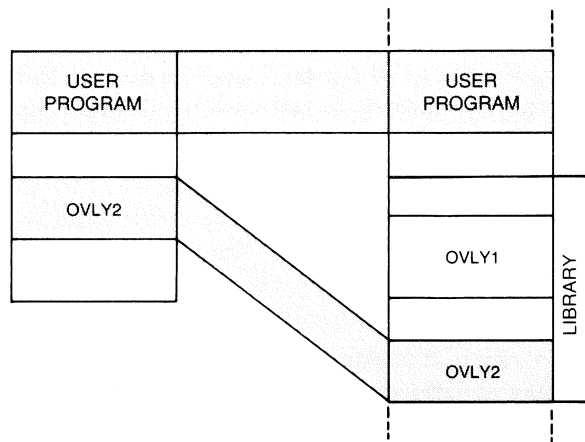
B) TIME2: USER PROGRAM REFERS TO OVLY2.

MK-00593-00

7.4.1 Specifying Memory-Resident Overlays

You can use many of the same techniques in doing memory-resident overlays for resident areas as you use for disk-resident overlays. As with disk-resident overlays, the branches of an overlay tree must be logically independent (see Section 3.6). In the example in Figure 7-1, OVLY1 cannot call or refer to data in OVLY2, or vice versa.

In the ODL file, use an exclamation point to specify memory-resident overlay segments. Memory-resident overlay segments are indicated by placing an exclamation point immediately before the left parenthesis enclosing the desired segments. For example:

```
.ROOT A-!(B,C)
```

In this example, segments B and C are declared resident in separate areas of memory. The Task Builder determines the addresses for the resident area (relative or absolute, depending on whether the resident area is built as position independent or absolute) as follows. The starting address of segment A is 0 (position independent) or as specified in the PAR option. The length of segment A is rounded up to the next 4K-word boundary; this determines the starting address for B and C. The length of B and C are rounded up to the next 32-word boundary to determine the total memory required by the area.

The exclamation point applies only to segments at the first level inside a pair of parentheses; segments nested within the first level are not affected.

Note the significance of rounding up to the next 4K-word boundary; the least amount of space that a memory-resident overlay can occupy is 4K words. Likewise, each segment occupies some multiple of 4K words. An overlay segment of 4097 words (one word over the 4096-word limit) will take 8K words of virtual address space.

There is another consideration for memory-resident overlays. The user program that is eventually linked to the resident area in the example in Figure 7–1 can make calls to A, B, or C and they will be mapped properly. However, A cannot call B or C. The reason is that the Task Builder does not build the necessary autoloading code into the resident area; it (eventually) builds it into the root segment of the user program to which the resident area is linked. A cannot call B or C, because there is no way to tell whether B or C has been mapped at any given time. B or C can call A, however, because it is known that A will be resident in the next-lower 4K of virtual address space, regardless of whether B or C is currently mapped.

7.4.2 Building Memory-Resident Overlays

As described above, a resident library containing memory-resident overlays must define the overlay structure in an ODL file. The build for such an overlay structure proceeds somewhat differently than for disk-resident overlays.

Specifically, the Task Builder does not include the overlay data base (segment descriptions, autoload vectors, and so forth) or the code for loading overlays as part of the resident area task file. Rather, the data base is made part of the symbol table file. This data base is later built into the program that refers to the resident area. Note that this increases the size of the program that refers to a resident area.

The symbol table file contains global definitions for only those symbols that are defined or referenced in the root segment of the area. Such symbols consist of the following:

1. Entry points to routines and data elements that are in the root.

2. Autoload vector addresses that point to definitions within a memory-resident overlay.

3. Definitions of symbols defined in a memory-resident overlay and referenced in the root.

No global symbol appears in the symbol table file unless it is either defined in the root segment, or referenced in the root segment and defined elsewhere in the overlay structure.

You can force the inclusion of a global reference in the root segment of the resident area by using the GBLREF option (Section 12.15). Thus, the necessary autoload vectors and definitions can be generated without explicitly including such references in a segment. The syntax of the GBLREF option is:

GBLREF=name

where:

name is the one- to six-character global symbol name. If the definition for the symbol resides within an autoloadable segment, the Task Builder creates an autoload vector, and includes it in the symbol table file. If the definition is not in an autoloadable segment, the real value is obtained and defined in the root segment.

You need to include in the GBLREF option all global symbols that will be used in transfer-of-control statements but are not defined or referenced in the root segment of the resident overlay area.

For example, suppose you are building a resident library out of the programs ADD, SUB, MULT, and DIV and that you want these four routines to be memory-resident overlays. The ODL file would be specified as follows:

```
.NAME   NULL
.ROOT   NULL-*!(ADD,SUB,MULT,DIV)
.END
```

ADD, SUB, MULT, and DIV are entry points that will probably be called by any program that references the resident library, and none of these are defined or referred to in the root segment of the overlay structure. Include these four global symbols in the GBLREF option when the library is built so that data for these symbols will be included in the symbol table file. For example:

```
RUN $TKB
TKB>MATHLB/-HD/PI,,MATHLB/PI=OVERLY/MP
TKB>/
ENTER OPTIONS:
TKB>GBLREF=ADD,SUB,MULT,DIV
TKB>PAR=MATHLB
TKB>STACK=0
TKB>//
```

Any program can then later refer to ADD, SUB, MULT, and DIV, and the Task Builder will resolve the references properly from the information in the library's symbol table file. For example:

```
RUN $TKB
TKB>MYPROG=MYPROG
TKB>/
ENTER OPTIONS:
TKB>RESLIB=DR0:[1,210]MATHLB
TKB>//
```

Note that we are using the RESLIB option, so the option includes the device and account where the files MATHLB.TSK and MATHLB.STB reside.

7.5 Building Your Own Cluster Libraries

This section assumes that you already know how to build a resident library. It is more difficult to build cluster libraries than noncluster libraries because of the additional rules imposed, as described in the rest of this section.

As discussed in Section 2.3.5, resident libraries that have been built to take advantage of the "clustering" feature of the Task Builder can be mapped to occupy the same virtual address space, taking less space in the user job area than they would otherwise.

You can build your own resident libraries to take advantage of the clustering feature. It requires that you follow the rules summarized below. Following subsections discuss these rules in detail.

1. All libraries in a cluster must be position independent or built for the same address.

2. All libraries except the default library in a user's CLSTR option must use memory-resident overlays.

3. A called library routine must not require parameters on the stack by the caller.

4. No library may be entered using synchronous or asynchronous system traps.

5. A library should not call routines from other libraries in the same cluster directly.

7.5.1 Rule 1: Position Independent or Built for Exact Address

The Task Builder must be able to place each library in a cluster at the same virtual address. To do this, the libraries must be built as position independent or be built to the exact address specified in a user's CLSTR option.

7.5.2 Rule 2: Use Memory-Resident Overlays

If you want your library to be referenced as other than the default library in a user's CLSTR option, it must use memory-resident overlays. Furthermore, the root of the memory-resident overlay structure must be null (of zero length).

If your library does not require overlays, you can still build it so it seems that resident overlays are being used. This will build the code necessary for cross-library linkage into your resident library.

For example, suppose you have a disk library file LB:US1LIB.OLB that requires 7K words of memory that you wish to make into a cluster library. You do not wish to use memory-resident overlays; the library will simply use two APRs when it is linked with a user's program. To build the necessary linkage into the library, you can specify an ODL file with a "null root" and a "null branch" (Figure 7–2).

The ODL file for such a structure could be:

```
        .NAME  US1CLS
        .ROOT  US1CLS-*!(NULLA,US1FAC)
NULLA:  .FCTR  LB:SYSLIB/LB:NULL
US1FAC: .FCTR  LB:US1LIB/LB
        .END
```

Figure 7–2: Using a Null Memory-Resident Overlay
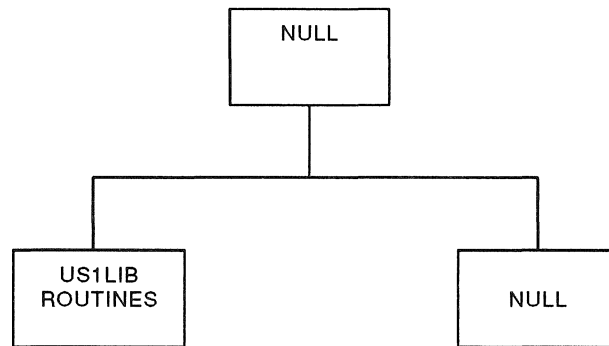
MK–00831–00

7.5.3   Rule 3: No Required Parameters on the Stack

This rule applies to routines contained in libraries other than the default library.
A routine in a cluster library should not expect to receive information from the
caller that was pushed on the stack. This is because the Task Builder autoload
routines ($AUTO) may have used the stack for its own purposes in remapping
to the called library. There is no way for your library routine to determine at
run-time whether the autoload code has or has not placed mapping information
on the stack. So, the best way to handle this type of information exchange is to
pass the address of call parameters in general-purpose registers, for example, R0.
If parameters must be passed on the stack, then the calling program can push
the information on the stack, and save the contents of the stack pointer register
(SP) to another register, for example, R0. The called library routine can then use
R0 to find the information it needs from the stack.

NOTE

Assembly language programmers must use a JSR PC instruction to
transfer control to the desired library routine, and the library routine
must use an RTS PC instruction to return control to the caller.

7.5.4   Rule 4: No Trap or Asynchronous Entry

A routine built as part of a library that is to be used in a cluster cannot be
specified as the service routine for a synchronous trap or for asynchronous entry
as a result of a Ctrl/C, for example. This is because the library may not be the
one that is mapped at the time of the trap. For example, if the default library
contains the service routine to display an error message upon odd address trap,
and the odd address fault occurs within one of the other libraries of the cluster,
the routine will not be available to service the trap.

7.5.5 Rule 5: No Calls to Routines in Another Cluster Library

A resident library routine cannot directly call a routine in another resident library in the same cluster. The called resident library may not be in memory when the call is made. There are rather elaborate techniques for routing the call through autoload vectors that must be built into the user program in the low segment. These techniques are described in the next section; however, Digital recommends that you do not make calls between resident libraries that may be in the same cluster.

7.5.6 Revectoring Cluster Libraries

A cluster library cannot directly call a routine in another library in the cluster. The general technique involves indirect references, routing calls through the user program in the "low segment," so that control eventually passes through the correct autoload vectors to the desired routine in the called library. Thus, the called library can be loaded from disk and, if necessary, mapped. The called routine is then executed, and eventually control is returned to the calling library.

The approach involves including a "vector table" in the calling library, and a corresponding "jump table" in the user program in the low segment. Ideally, the code necessary for the vector table and jump table would be included in the system library, so this is what our example shows.

The vector table defines as entry points the desired entry points in the called library. Each definition in the calling library defines an offset for the entry point. The offset defines the location in the library, and a corresponding "jump table" in the user program in the low segment. Ideally, the code necessary for the vector table and jump table would be included in the system library, so this is what our example shows.

The vector table defines as entry points the desired entry points in the called library. Each definition in the calling library defines an offset for the entry point. The offset defines the location in the jump table for the address of the desired autoload vector into the called library. The vector table also includes common dispatch code to transfer control. This code transfers control through the jump table, through the appropriate autoload vector in the user program, to the entry point in the called library (see Figure 7–3).

Figure 7-3: Overview of How Inter-Cluster-Library Calls Work
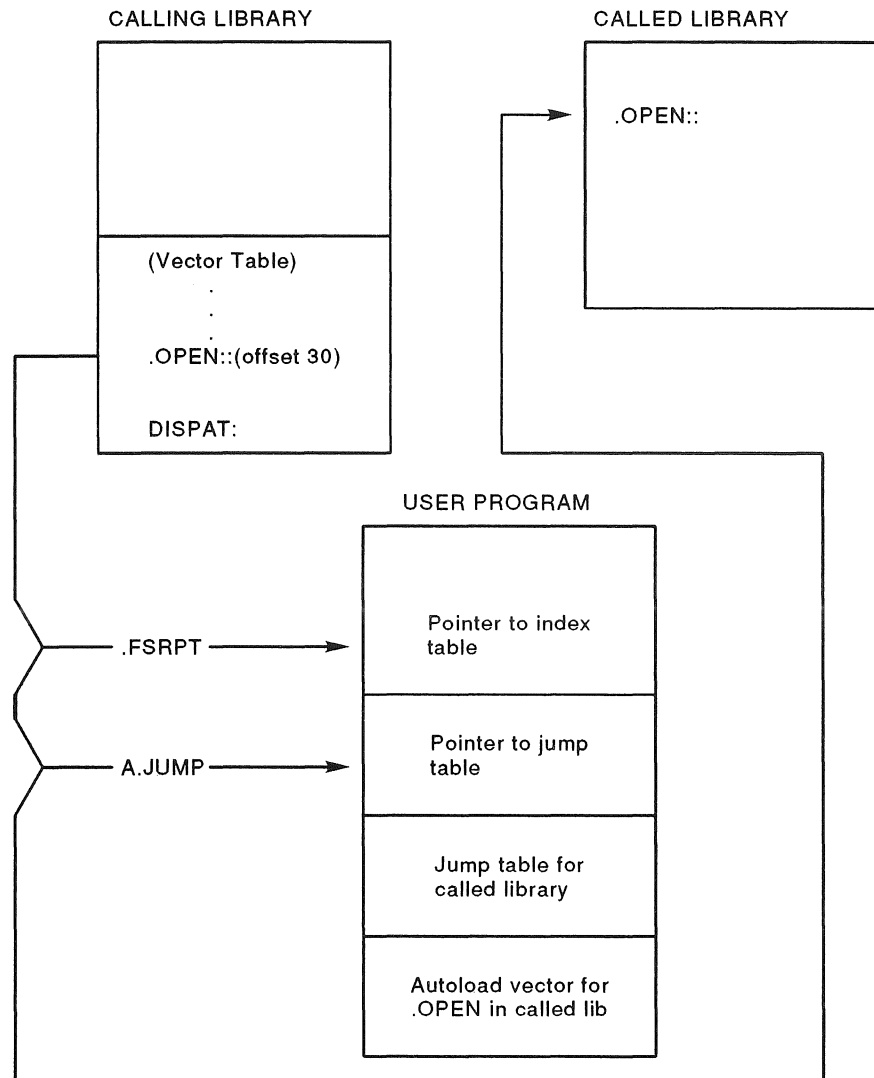
MK-01054-00

7.5.7 Sample Vector Table Code

The code below illustrates part of a sample vector table.

```
.OPEN:: MOV    #30,-(SP)           ;PUT OFFSET INTO USER PROGRAM
                                   ;JUMP TABLE ON THE STACK
        BR     DISPAT              ;JOIN COMMON DISPATCH
               .
               .
               .
DISPAT: MOV    R0,-(SP)            ;SAVE REGISTER
        MOV    @#.FSRPT,R0         ;GET POINTER TO DATA AREA
        ADD    A.JUMP(R0),2(SP)    ;ADD VECTOR BASE TO OFFSET
        MOV    (SP)+,R0            ;RESTORE REGISTER
        MOV    @(SP)+,-(SP)        ;PICK UP ADDRESS OF TARGET
        JMP    @(SP)+              ;AND TRANSFER TO TARGET
```

In the example above, the code at .OPEN pushes the known offset into the jump table (30) onto the stack and transfers control to dispatch code, common for all the revectored entry points. The code at DISPAT:

1. Pushes the contents of R0 onto the stack, to save it.

2. Moves the address of the base of the data area into R0.

3. Adds the base address of the jump table to the index onto the stack.

4. Restores R0.

5. Puts the address of the desired routine's autoload vector onto the stack.

6. Jumps to the autoload vector for the desired routine (.OPEN).

7.5.8 GBLXCL and GBLINC Options

In the preceding example, notice that both the calling library and the called library contain an entry point named .OPEN. You must exclude global symbols for such revectored entry points from the calling library's symbol table (.STB) file, or the Task Builder has a hard time figuring out which one to use when the libraries are referenced during a user program's build. To do this, use the GBLXCL option when you are building the calling library.

Another aspect of building the calling library is ensuring that the needed jump tables are built into the user program when the calling library is referenced there. This involves placing the code for the jump tables in the system library, or a library always referenced by the user program, and ensuring that such code is always included in the user program when the calling library is referred to in the user program. You use the GBLINC option in the calling library to do this.

The example below shows the build for the "calling library," called US1CLS.

```
RUN $TKB
TKB>US1CLS/-HD,US1CLS/CR/-SP/MA,US1CLS=US1LIB.OBJ
TKB>LB:SYSLIB/LB:FCSVEC
TKB>/
ENTER OPTIONS:
TKB>STACK=0
TKB>PAR=US1CLS:140000:4000
TKB>GBLINC=.FCSJT
TKB>GBLXCL=.OPEN
TKB>//
```

The example above shows the vector table (FCSVEC) as a module in the system library. Building such a vector table as part of a commonly used library, such as SYSLIB, makes it easier for more than one library to access the called library.

The GBLINC option shown forces the Task Builder to add a global reference entry for .FCSJT in the library's .STB file. This ensures that the Task Builder links the jump table modules required by the library into the user program. These modules should be in the system library, or in a library always referenced by the user program. Thus, this forced loading mechanism is invisible to the user.

7.6 FORTRAN Virtual Arrays

The FORTRAN-77 programming system automatically provides a virtual array mechanism for up to 255K words of data. The FORTRAN user can generate a unique dynamic region for data by making the proper declarations within the control of the FORTRAN language. Refer to the FORTRAN-77 *User's Guide* and other language manuals for more information.

NOTE

The size of the virtual arrays created may be limited by the dynamic region limit. See the *RSTS/E System Manager's Guide*.

If a FORTRAN program requires more memory-based data than the virtual arrays can provide, use virtual program selectors.

Use the following FORTRAN-77 statement to invoke the automatic virtual array mechanism:

```
VIRTUAL var(index), var2(index2),...
```

The FORTRAN compiler informs the Task Builder of the required size of the virtual array area. The TKB instructs the RSTS monitor to create the proper-sized dynamic region automatically. In RSX-11M-PLUS, this offset size is additional memory allocated to the task partition. In RSTS/E, it is the size of the dynamic region that will hold the virtual array data. This unnamed dynamic region will be created when the task is loaded at run-time and will always have the region ID value of zero. If not enough memory to allocate the dynamic region exists (see the *RSTS/E System Manager's Guide*), the task will not start. The FORTRAN language will automatically map the region as needed to access the data.

You can use the automatic region creation feature in MACRO-11 programs by including one of the following:

* The program of a .PSECT of the form:

```
        .psect NAME,D
varnam::
        UNORG
```

where:

name   is a unique section name

- The TKB VSECT option related to the PSECT above as follows:

```
VSECT=name:base:max_window_size:region_size
```

where:

name	is the related PSECT name
base	is the byte address of the window to map the dynamic region (for example, 160000 for APR 7)
max_window_size	is the number of bytes the largest mapped window in the region will have.
region_size	is the size the dynamic region will be in memory when created in 32-word blocks (the maximum value on RSTS/E is 17740, which equates to 255K words).

See Chapter 12 for more details on the VSECT option.

When the MACRO-11 task has been started, there will be a dynamic region with a region ID of zero already created and attached to the job. Before the region can be used, the program must create the window (CRAW$) and map it to the APR (MAP$). It does not need to be the same APR indicated by the "base" in the VSECT option, but if it is not, then the "varnam" global has no utility in addressing the region.

7.7 Virtual Program Selectors

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

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

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

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

By specifying the physical size, you can allocate, before run time, the physical memory that the program section will be mapped into at run time. TKB allocates this physical memory within an area that is treated by RSTS/E as a dynamic range. This area is called the mapped array area and can be up to 255K words in size.

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

Note also that TKB attaches the virtual attribute to a relocatable program section you have specified in the VSECT option only if the section is defined in the root segment of your task through either a FORTRAN COMMON or a MACRO-11 .PSECT statement. The relocatable program section with the virtual attribute in the root does not use address space in your task; using this procedure merely assigns an address, window size, and physical length to a region yet to be mapped at run time by your task. For example:

```
VSECT=MARRAY:160000:20000:2000
```

In this example, virtual program section MARRAY is allocated with a window size of 4K words (20000 $_8$ bytes) and a base virtual address of 160000. In physical memory, 32K words are reserved for mapping the section at run time.

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

```
COMMON/MARRAY/ARRAY(4096) ...
```

This statement generates a program section to which TKB attaches the virtual attribute. However, this program section is not a FORTRAN virtual array; the user program must manually map it. A reference to the first element of the section, ARRAY(1), is translated by TKB to the virtual address 160000. Figure 7–4 shows the effect of this use on the VSECT option.

Figure 7–4: VSECT Option Usage


```
                      .
                      .
                      .
TKB>/
Enter Options:
TKB>VSECT – MARRAY: 160000: 20000: 2000
                      .
                      .
                      .
```

* Size of dynamic region is the total of all VSECT physical lengths.
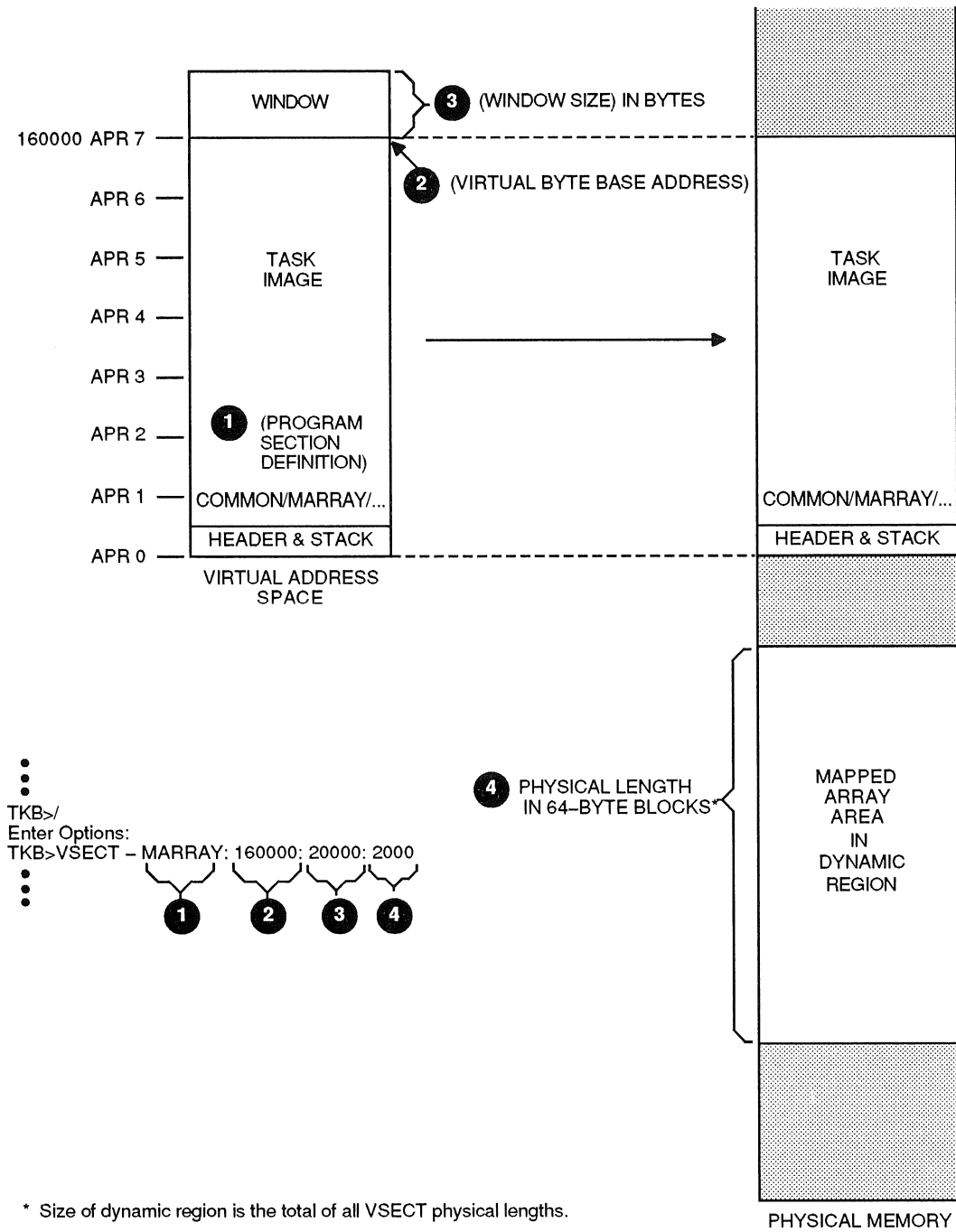
As mentioned previously, TKB restricts the amount of virtual address space allocated to the section to a value that is less than or equal to the window size, wrapping around to the bases if the window size is exceeded.

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

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

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

7.7.1 FORTRAN Run-Time Support for Virtual Program Sections

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

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

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

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

You can ask the ALSCT routine to either use the automatically created region or create a region of its own. By creating new regions, it is possible for a FORTRAN program to have over 1.5 megabytes of data resident in memory (if it physically exists) without affecting program size. This is because the two D-APRs that are not used by the RMSRES can each be used for a 255K-word region in addition to the 255K-word region automatically created.

NOTE

FORTRAN does not provide any array bounds checking for these arrays.

7.7.1.1 The ALSCT Subroutine

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

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

The ireg parameter is a one-dimensional integer array that is nine words long. Elements 1 through 8 of the array contain a region descriptor for the physical memory to be mapped. The format of the descriptor is shown in Table 7-1.

Table 7–1: Format of Region Descriptor

ireg(1)	Region ID (0 is the automatic region)
ireg(2)	Size of region in units of 64-byte blocks
ireg(3)	Name of region in Radix-50 format (first three characters)[1]
ireg(4)	(Second three characters) [1]
ireg(5)	Name of main partition region in Radix-50 format (first three characters)[1]
ireg(6)	(Second 3 characters)[1]
ireg(7)	Region status word[1]
ireg(8)	Region protection code[1]
ireg(9)	Thread word—links window descriptors used to map portions of the region; maintained by the subroutine

[1]Not supported on RSTS/E.

The elements of the array that you set up consist of ireg(1) and ireg(3) through ireg(8). The thread word, ireg(9), must be 0 on the initial call each region used; thereafter, the subroutine maintains it.

On the initial call to ALSCT for each region ID used, (ireg (1)) has special meaning in that it tailors the region descriptor to what has been specified. If ireg(1)=0 and ireg(2)=0, the system will use the automatic region and fill in ireg(2) with the size of the dynamic region that was created at run-time. If ireg (1)=-1, the system will create a new dynamic region of the size given in ireg(2) (if memory is available) and return the region ID of the new region in ireg(1). After the initial call to ALSCT, the program should not change either ireg(1) or ireg(2).

The subroutine does not refer to elements 3 through 8; the caller must set them up as required by the applicable system directives (for example, CRRG$). For a detailed description of these parameters, refer to the *RSTS/E System Directives Manual*.

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

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

iwnd(8)	Address of send/receive buffer
iwnd(9)	Base offset of physical block allocated to section in units of 64-byte blocks
iwnd(10)	Length of block in units of 64-byte blocks (supplied by caller); set to maximum block offset by subroutine
iwnd(11)	Thread word—links window descriptors used to map other portions of the region; maintained by the subroutine

You must set up iwnd(10) before calling ALSCT.

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

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

The remaining elements must be set up as required by the Executive directives. Consult the *RSTS/E System Directives Manual* for a detailed description of these parameters.

The ists is a variable that receives the results of the call and returns one of the following values:

+1	Block successfully allocated; in this case, the region and window descriptors arrays are set up as described above.
-200.	Insufficient physical memory available for allocating the block. will contain the region size available.

Ireg(1) will contain the new region ID or 0 for auto region. Ireg (2) will contain the region size used. If the program uses both the automatic virtual array and the manual virtual arrays in the automatically created region, you must first allocate a place-holder window equal to the size of that used by the automatic virtual array. Example 7-2 demonstrates how to do this.

7.7.1.2 The RLSCT Subroutine

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

`CALL RLSCT (ireg, iwnd)`

As for ALSCT, ireg is a one-dimensional array that is nine words long. The contents of the array are the same as those described for the ALSCT subroutine. Likewise, iwnd is the same as for ALSCT, that is, a one-dimensional integer array that is 11 words long. The contents of the array are the same as those described for subroutine ALSCT.

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

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

1. Allocate storage in the program for one window descriptor for each virtual program section and for a single region descriptor.

2. Your task calls the subroutine ALSCT to reserve the physical memory to which the virtual program section will be mapped.

3. Your task issues the mapping directives to map the virtual address space into a portion of the physical memory. It is the task's responsibility to ensure that the physical memory to be mapped is always within the limits defined by iwnd(9) and iwnd(10).

4. When the space is no longer required, the task unmaps and releases it with a call to RLSCT.

7.7.2 Building a Program That Uses a Virtual Program Section

Example 7-2 shows the FORTRAN source file for a task named VSECT.FTN. The example illustrates the use of the ALSCT FORTRAN subroutine, and uses the automatic region for both VARRAY and IARRAY. When you build, install, and run VSECT2, it allocates the mapped array area in a dynamic region, creates a 4K-word window, and maps to the area through the window. ALSCT then initializes the area and prompts for an array subscript at your terminal, as follows:

```
SUBSCRIPT?
```

When you input a subscript, it responds with ELEMENT= and the contents of the array element for the subscript you typed. VSECT2 continues to prompt until you press Ctrl/Z at which time it exits.

Once you have compiled VSECT2, you can build it with the following TKB command sequence:

```
TKB @VSECT2
```

Example 7–1: VSECT2.CMD

```
>
;               VSECT2.CMD
SY:VSECT2/ID/FP,vsect2=vsect2.odl/MP
UNITS =12
ASG    =SY:7:8:9:10:11:12
EXTTSK=512
MAXBUF=512
WNDWS=2
LIBR=RMSRES:RO:6:0
VSECT=MARRAY:160000:20000:2000
//
```

Note the following in Example 7-1:

- The /ID allows the use of the D-APRs where the RMSRES is placed.

- The RMSRES is at APR 6 and the APRs are unprotected in the D-space.

- The VSECT option has a physical length of 4K words which will be added to the length needed for the VARRAY( ).

- The WNDWS option is set allow for both types of arrays used.

This command file sequence directs TKB to create a task image file named VSECT.TSK and a map file named VSECT2.MAP. (See Example 7-3.)

The WNDS option directs TKB to add a window block to the header in the task image. The Executive initializes this window block when it processes the mapping directives within the task. This allows two programmed address windows (CRAW$) to be created in this region.

The VSECT option directs TKB to establish for the program section named MARRAY a base address of 160000 (APR 7) and a length of (20000) $_8$ bytes (4K words). The program section MARRAY is defined within the task through the FORTRAN COMMON statement. The VSECT option also specifies that TKB is to

allocate 200 64-byte blocks of physical memory in the task's mapped array area in the automatic dynamic region. For more information on the switches and options used in this example, refer to Chapters 11 and 12, respectively.)

This command sequence results in the map shown in Example 7-3.

The DCL link for FORTRAN-77 also creates the following ODL file:

```
;           VSECT2.ODL
; TKB .ODL file created by DCL LINK  V10.0-F    09-May-90    09:49 AM
@LB:RMS11M.ODL
@LB:RMSRLX.ODL
ROOT$$: .FCTR   OTSROT-RMSROT
        .NAME   LBR$$
        .ROOT   AO$,OTSALL,RMSALL
AO$:    .FCTR   SY:VSECT2-ROOT$$-LBR$$
.END
```

Example 7-2: Source Listing for VSECT2.FTN

```
c
c       vsect2.ftn
c
        VIRTUAL VARRAY(1500)
        integer *2 sub,irdb(9),iwdb(11),iwdb2(11),dsw
        integer *2 IARRAY(4096)
        common /MARRAY/IARRAY
c       Set up the window for the manual virtual array.
        iwdb(1)="3400                  !use apr 7 for window
        iwdb(3)=128                    !window size=128*32 words=4096 (1 apr)
        iwdb(5)=0                      !offset =0
        iwdb(7)="422                   !status=WS.64B!WS.WRT!WS.UDS
        iwdb(10)= 128                  !size to allocate =4Kw
c       Set up the window for the automatic virtual array.
c       This window will not actually be used--it is just a place-holder.
        iwdb2(1)="3400                 !use apr 7 for window
        iwdb2(3)=96                    !window size=96*32 words=3072
        iwdb2(5)=0                     !offset =0
        iwdb2(7)="422                  !status=WS.64B!WS.WRT!WS.UDS
        iwdb2(10)= 96                  !size to allocate =3Kw
        irdb(2)=340                    !region size expected (total)
        irdb(9)=0                      !assure link is zero for first call
c
c       Allocate 3K mapped area for the VARRAY (not to be used)
c
        call ALSCT(irdb,iwdb2,dsw)
        if (dsw .ne. 1) goto 100
c
c       Allocate 4K mapped array
c
        call ALSCT(irdb,iwdb,dsw)
        if (dsw .ne. 1) goto 100
c
c       Create a 4K window
c
        call CRAW(iwdb,dsw)
        if (dsw .ne. 1) goto 200
c
c       Map the array into virtual space
c
        call MAP(iwdb,dsw)
        if (dsw .ne. 1) goto 300
```

Example 7–2 (Cont.): Source Listing for VSECT2.FTN

```
          do 10 i=1,4096
10        IARRAY(i)=i
c
c         Mapped array initialized
c
30        write (5,35)
35        format ('subscript?')
          read (5,40, END=1000) sub
40        format (i7)
          write(5,60) IARRAY(sub)
60        format (' element = ',I7)
          goto 30
c
c         error entries
c
100       write (5,101)dsw
101       format (' error from ALSCT, error= ',I7)
          goto 1000
200       write (5,201)dsw
201       format (' error from CRAW, error= ',I7)
          goto 1000
200       write (5,201)dsw
201       format (' error from CRAW, error= ',I7)
          goto 1000
300       write(5,301)dsw
301       format (' error from mapping, error= ',I7)
          goto 1000
1000      call exit
          end
```

Example 7-3: Task Builder Map (Edited) for VSECT2.TSK

```
VSECT2.TSK   Memory allocation map   TKB M43.00      Page 1
                   18-MAY-90    09:19
Partition name : GEN
Identification : 18MAY
Task  PPN      : [216,1]
Stack     limits: 001000 001777 001000 00512.
PRG xfr address: 002424
Task attributes: ID
Total address windows: 7.
Mapped  array  area: 35840. words
Task   extension   : 512. words
Task  image  size  : 6720. words, I-Space
                     5376. words, D-Space
Total  task  size  : 6720. words, I-Space
                     41728. words, D-Space
Task Address limits: 000000 032133 I-Space
                     000000 024703 D-Space
R-W disk blk limits: 000002 000142 000141 00097.

Memory allocation synopsis:

Section                                        Title  Ident  File
-------                                        -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)   001000 000000 00000.
MARRAY:(RW,D,GBL,REL,OVR)   160000 020000 08192.
                            160000 020000 08192.  .MAIN. 18MAY  VSECT.OBJ

                .
                .
$VARS  :(RW,D,LCL,REL,CON)  002262 000104 00068.
                            002262 000104 00068.  .MAIN. 18MAY
VSECT.OBJ
$VIRT  :(RW,D,GBL,REL,CON)  002000 000140 00096.
                            002000 000136 00094.  .MAIN. 18MAY
VSECT.OBJ
$$ALER:(RO,I,LCL,REL,CON)   003220 000040 00032.
                .
                .
```

In the following examples, the IARRAY is put into a newly created dynamic region separate from the automatic region containing VARRAY( ). In this case, there is no need to set aside space for VARRAY( ) in the VSECT.

The following is the VSECT.CMD for Example 7-5.

```
TKB @VSECT
```

Example 7-4: VSECT.CMD

```
;            VSECT.CMD
SY:VSECT/ID/FP,vsect=vsect.odl/MP
UNITS =12
ASG    =SY:7:8:9:10:11:12
EXTTSK=512
MAXBUF=512
WNDWS=2
LIBR=RMSRES:RO:6:0
VSECT=MARRAY:160000:20000:0
//
```

In this command sequence, note that:

- The /ID allows the use of the D-aprs where the RMSRES is placed.

- The RMSRES is placed at APR 6 and the APR's are unprotected in the D-space.

- No physical length is given on the VSECT option because a new region will be created.

- The WNDWS option is set allow for both types of arrays used.

The DCL link command creates the same ODL file for Example 7-5 as it did for Example 7-2:

```
;        VSECT.ODL
; TKB .ODL file created by DCL LINK  V10.0-F     09-May-90     09:49 AM
@LB:RMS11M.ODL
@LB:RMSRLX.ODL
ROOT$$: .FCTR    OTSROT-RMSROT
        .NAME    LBR$$
        .ROOT    AO$,OTSALL,RMSALL
AO$:    .FCTR    SY:VSECT-ROOT$$-LBR$$
.END
```

Example 7–5: Source Listing for VSECT.FTN

```
c       vsect.ftn
c
        VIRTUAL varray(1500)
        integer *2 sub,irdb(9),iwdb(11),dsw
        integer *2 IARRAY(4096)
        common /MARRAY/IARRAY
        iwdb(1)="3400              !use apr 7 for window
        iwdb(3)=128               !window size=128*32 words=4096 (1 apr)
        iwdb(5)=0                 !offset =0
        iwdb(7)="422              !status=WS.64B!WS.WRT!WS.UDS
        iwdb(10)= 128             !size to allocate =4Kw
        irdb(2)=200               !region size expected
        irdb(3)=0                 !set name to no-name
        irdb(4)=0
        irdb(1)=-1                !set to create new region
        irdb(9)=0                 !assure link is zero for first call
c
c       Allocate 4K mapped array in a newly created dynamic region
c
        call ALSCT(irdb,iwdb,dsw)
        if (dsw .ne. 1) goto 100
c
c       create a 4K window
c
        call CRAW(iwdb,dsw)
        if (dsw .ne. 1) goto 200
c
c       Map the array into virtual space
c
        call MAP(iwdb,dsw)
        if (dsw .ne. 1) goto 300
        do 10 i=1,4096
10      IARRAY(i)=i
c
c       Mapped array initialized
c
30      write (5,35)
35      format ('subscript?')
        read (5,40, END=1000) sub
40      format (i7)
        write(5,60) IARRAY(sub)
60      format (' element = ',I7)
        goto 30
c
c       error entries
c
100     write (5,101)dsw
101     format (' error from ALSCT, error= ',I7)
        goto 1000
200     write (5,201)dsw
201     format (' error from CRAW, error= ',I7)
        goto 1000
200     write (5,201)dsw
201     format (' error from CRAW, error= ',I7)
        goto 1000
300     write(5,301)dsw
301     format (' error from mapping, error= ',I7)
        goto 1000
1000    call exit
        end
```

7.8 Advanced Programmed Region Control

The Task Builder can now provide information about the allocation of D-space APRs for a task to the program at execution time. Together with the new EXTM$ memory extend function (see *RSTS/E System Directives Manual*), the TKB can maximize the amount of D-space memory that a task can request.

Programmers can build tasks with libraries that have both I- only and I- and D-space code. The Task Builder will calculate the minimum number of APRs that each task requires to satisfy the D-space needs of all the libraries used by the task.

To effectively utilize this feature, each library should be organized such that all DATA space required in the library is located at the highest address in the library space. You must know at the time the library is built how many APRs contain D-space because this information is required as input to the Task Builder in the /LI switch.

As part of the information the Task Builder associates with each individual resident library, a bit mask representing the APRs used by the library is kept. The default value represents the I-space APRs. RSTS will load the library with the I- and D-space equivalent, thereby automatically protecting any D-space existing in the library.

However, the saved bit mask value may be modified by the programmer in the library switch given to the Task Builder. The /LI switch, which is required to build a library, may now include a value in the range 0 to 377 in the following format:

/LI[:nnn]

where:

nnn   is the desired mask value.

The meaning of each bit in the mask value is the same as described in the EXTM$ function and is given in Table 7-2.

Table 7–2: Bit Meanings in the Mask Value

Bit 7	Represents APR 7	<200>
Bit 6	Represents APR 6	<100>
Bit 5	Represents APR 5	<40>
Bit 4	Represents APR 4	<20>
Bit 3	Represents APR 3	<10>
Bit 2	Represents APR 2	<4>
Bit 1	Represents APR 1	<2>
Bit 0	APR 0 not usable by library	

Through this switch value, the programmer can inform the Task Builder which APRs contain D-space requirements. This switch value, rather than the default, will be associated with this library when it is used by the Task Builder in future applications.

It is important to note that for libraries built by Task Builders prior to V9.6, the mask value is zero by definition. This means that the library is defined as having I-only code. If, however, the library does have D-space requirements, they may be

protected at the time the application task is built, as described in the following section.

7.8.1 The EXTM$ Feature

At the coding level, the application programmer must do the following two things to use the EXTM$ feature:

- Use the EXTM$ memory extension function instead of the EXTK$ function

- Provide a properly addressed memory location for the Task Builder to fill in the APR usage mask data

The following lines of code provide the properly addressed location for the Task Builder to fill in:

```
          .PSECT   $$TSKP,D
$TSKP::
          .PSECT   ......
```

This PSECT can go anywhere in the code. However, if it is not at the end of the code, reset the PSECT back to your required PSECT.

For the EXTM$ function to work beyond what the EXTK$ does, the application task must be built with I- and D-space on (/ID switch).

The value loaded in to $TSKP by the Task Builder and readable on execution of the program will be one of two values:

- The value computed by the Task Builder based on the appplication task and all reference libraries

- An override value provided by an option line

If the Task Builder generates the value in $TSKP, it will be the logical OR of the application tasks allocated D-space APR mask and the associated APR bit masks for all libraries referenced by the task build.

For example, for a task that has two libraries where LIB1 uses APRs 6 and 7 and LIB2 uses APRs 5, 6, and 7, and 5KW of data space, the computed value would be 343 octal as determinable from the APR representation in Table 7-2.

It is possible also to explicitly set the value in $TSKP at the time of the application task build. This is how a programmer who knows a library has additional available D-space can inform the task of such knowledge. The following three option lines have been amended to include this new level of information:

```
LIBR  =name:accesscode[:baseAPR[:bitmask]]
```

```
RESLIB =file/accesscode[:baseAPR[:bitmask]]
```

```
CLSTR =lib1,lib2,lib3...libn:accesscode[:baseAPR[:bitmask]]
```

where:

bit mask   is the value logically ORed with the task's allocated D-space mask and loaded by the Task Builder in location $TSKP.

See Chapter 12 for further details on the use of the listed options.

When using cluster libraries, the bit mask value applies to all libraries; therefore, the most limiting required value must be used.

When relocation of a position-independent coded library (PIC) is being done by the base APR value, the bit mask value must be specified because the value associated with the library was based on its original base and is not valid at the new base. To spcify the bit mask value of the library, shift it up or down one bit for each APR of relocation shift, and then combine it with the other mask values required.

If using libraries that were built by versions of the Task Builder before V9.6 and the library contains any D-space, the bit mask value must be specified to protect that D-space at the time the application is task built. As of RSTS/E V9.6, the only Digital-supplied library that can take advantage of this feature is RMSRES. RMSRES is built as I- only code; therefore, all the D-space APRs are available to the MACRO-11 programmer through this feature.

If a mask value of 0 is given, the EXTM$ command behaves just as EXTK$ does under RSX-11M-PLUS; that is, any request for more data space used by a library will automatically turn that D-space over to the task D-space all the way to the top of the virtual address space.

Refer to the EXTM$ in Chapter 5 of the *RSTS/E System Directives Manual* for more details.

7.9 Fast-Mapping Facility

A new mapping facility has been added as an alternative to the MAPFQ subfunction of the .PLAS directive, providing greatly enhanced performance and compatibility with the fast-mapping facility of RSX-11M-PLUS. Applications that are dependent on remapping of libraries in the normal course of execution could see significant speed improvements.

However, the fast-mapping facility has the following restrictions:

* Only the offset to the map field (FQSIZE) and, optionally, the length to the map field (FQBUFL), may be modified by the fast-mapping facility.

* The interface to the fast-mapping facility is designed for speed, not for ease of programming. Debugging a task using fast-mapping may be more difficult than using the .PLAS directive. Specifically, protecting the operating system and its data structures is the only validation of parameters that is done.

* The interface uses the PDP-11 IOT instruction. Tasks may use IOT for internal communications and other functions, but tasks that use fast-mapping cannot use the IOT instruction for any purpose other than fast-mapping.

* The interface uses registers for passing parameters rather than using a FIRQB, a savings of significant instructions over the .PLAS directive. This means that the MACRO-11 programmer must be careful about register usage when using fast-mapping.

These restrictions (particularly the second) should not deter the use of fast-mapping in high-performance applications. However, Digital recommends that you first get the application running with the .PLAS directive, varying only the FQSIZE and FQBUFL fields, and then replace the directive with fast-mapping.

7.9.1 Fast-Mapping Code Provided by TKB

If a program does not use the IOT instruction or ODT, fast-mapping is still available when using resident libraries. Much of the mapping of resident libraries is automatically provided by the Task Builder in the form of autovectoring. Fast-mapping is possible by using the /FO and /FM switches (see Section 11). When the /FO switch is used, the autovectoring code loads fast-mapping routines rather than .PLAS calls. No code changes are required by the user.

7.9.2 Programming Considerations

Before issuing a fast-mapping call, the task must create *and* map the window by using the Create Address Window (CRAFQ) and MAPFQ subfunctions of the .PLAS directive. The FIRQB+12 (octal) in the CRAFQ directive must be at least as large as the largest FQBUFL to be requested in fast-mapping.

Three parameters are required for the fast-mapping call. The first parameter is the base APR times 10 (octal) plus the I/D space flag (0=I-space and 100 (octal)= D-space).

The second parameter is the offset field to map from the start of the library to the base of the window. The third parameter is an optional length to map. If the length to map is given, the high bit of the first parameter (the APR times 10 (octal)) must be set.

Table 7-3 lists the first parameter values.

Table 7–3: Values for the First Fast-Mapping Call Parameter

Space	Base APR	If No Length Given	With Length Given
User-I	0	APR I-0 cannot be changed	
User-I	1	000010	100010
User-I	2	000020	100020
User-I	3	000030	100030
User-I	4	000040	100040
User-I	5	000050	100050
User-I	6	000060	100060
User-I	7	000070	100070
User-D	0	APR D-0 cannot be changed	
User-D	1	000110	100110
User-D	2	000120	100120
User-D	3	000130	100130
User-D	4	000140	100140
User-D	5	000150	100150
User-D	6	000160	100160
User-D	7	000170	100170

The offset field (second parameter) is specified in 32-word slivers, in the same way it would be for the FQSIZE value in the MAPFQ subfunction of the .PLAS directive, and has the same meaning.

If the length to map is not specified (high bit of R0 is 0), the length is assumed to be the length given in the last PLAS mapping (either the value in XRB at W.NLEN is CRAW$ or MAP$, or FQBUFL if CRAFQ or MAPFQ).

If the length is specified (high bit of R0 is 1), then that length is mapped. If a length of 0 is given in R2, then the smaller of either the last .PLAS mapping length or the remaining length in the library between the offset and the library top is mapped. This handling is identical to that for the .PLAS mapping for the same case.

Note the difference between the RSTS/E and RSX-11M-PLUS versions is that User D-space must remain mapped to user's low core (APR0) at all times in RSTS/E. RSX-11M-PLUS does not have such a restriction.

Note also that the speed of fast-mapping is affected by the parameter values. Not specifying the length to map is the fastest and specifying a length equal to zero is the slowest. The improvement over the .PLAS directive is in the range of six to twelve times faster. The amount of improvement depends on the amount of remapping done by the task.

7.9.2.1 Calling Sequence

R0	Parameter 1	See Table 7-2
R1	Parameter 2	Offset field
R2	Parameter 3	Optional length specification

7.9.2.2 Returned Data

R0	Status	IS.SUC if successful
		IE.ITS or IE.ALG if failure
R2	Length mapped	In slivers if successful
R1,R3		Indeterminate

7.9.2.3 Programming Examples

Changing only window offset in I-space:

```
MOV   #40,R0      ;window at APR 4 in I-space
MOV   #200,R1     ;offset from base of library is 4K words
                  ;200 slivers * 32 words/sliver
                  ;R2 is not needed

IOT               ;issue fast map call
TST   R0          ;successful?
BMI   ERROR       ;no
```

Changing window offset and actual length specified:

```
MOV   #100150,R0  ;window at APR 5 in D-space
MOV   #100,R1     ;offset 2K words from library base
MOV   #500,R2     ;specified length = 10K words
                  ;this will use APRs 5, 6, & 7
IOT               ;issue fast map call
TST   R0          ;successful?
BMI   ERROR       ;no
```

Changing window offset and defaulted length specified:

```
MOV     #100140,R0   ;window base at APR 4 in D-space
CLR     R1           ;offset=0, window begins at library base
CLR     R2           ;force calculation to FIRQB+12(octal) or
                     ;remaining size of library
IOT                  ;issue fast map call
TST     R0           ;successful?
BMI     ERROR        ;no
MOV     R2,MAPLEN    ;copy of size actually mapped, in slivers
```

In the above text and examples, the term "library" means a resident library or dynamic data region.

The following are additional notes on the use of the fast map facility:

- Fast-mapping cannot be used if ODT is included in the task build, because the IOT instruction is used differenctly for both facilities.

- Fast map cannot be used for mapping supervisor modes or called from supervisor mode.

- Fast-mapping can be turned on and off in the program by doing a .SET or .CLEAR system call to RSTS/E (see the *RSTS/E System Directives Manual* for details of usage). Note that RSX-11M-PLUS does not support this programmable control.

User-Mode I- and D-Space

This chapter describes how the Task Builder divides a user task into instruction and data space (I- and D-space). A series of figures and text explain task mapping and the use of task windows in a RSTS/E system with an I- and D-space task. In the text, comparisons are made between conventional tasks and I- and D-space tasks. A conventional task is one that does not separately map instruction space and data space.

I- and D-space is available only on specific PDP-11 processors. You can run conventional tasks on an I- and D-space system; however, you cannot run I- and D-space tasks on a system that does not have the hardware available. PDP-11 processors that can run I- and D-space tasks are: 11/44, 11/45, 11/50, 11/53, 11/55, 11/70, 11/73, 11/83, 11/84, 11/93 and 11/94.

8.1 User-Task Data Space

User-task data space contains data. The user task accesses data space through D-space APRs. The I- and D-space feature allows a total of 16 APRs to map your task: eight APRs for data space and eight APRs for instruction space. If your task uses both I- and D-space to its maximum capacity, it can contain 64K words of virtual address space.

Your task must use .PSECTs to contain data or instructions or both.

Conventional tasks and tasks that separate instruction space and data space differ in only a few areas. The following sections discuss these areas.

8.2 I- and D-Space Task Identification

The monitor determines whether a task uses I- and D-space at run time. Therefore, you can run tasks built without I- and D-space on a system that supports I- and D-space without rebuilding the task.

The I- and D-space task is one in which the Task Builder separates the data areas and instructions. In this task, data areas should be defined by the MACRO-11 .PSECT directive that has the data attribute. Similarly, the .PSECT directive with the instruction attribute defines instruction areas.

8.3 Comparison of Conventional Tasks and I- and D-Space Tasks

A conventional task operating in user mode can contain 32K words of virtual address space and access up to 32K words of physical memory. However, a task using both I- and D-space APRs can contain 64K words of virtual address space and access up to 64K words of memory.

The conventional task in an I- and D-space system uses both sets of APRs. However, the relocation addresses in both I- and D-space APRs are identical.

An I- and D-space task can use both I- and D-space APRs independently; that is, APRs used in this way are not overmapped. Because of this, the task can use eight D-space APRs to access and use data, and eight I-space APRs to access and execute instructions. Using 16 APRs allows the I- and D-space task to access a total of 64K words of physical memory at one time.

Table 8–1 summarizes APR mapping for various combinations of I- and D-space tasks and I- and D-space systems.

Table 8–1:   Mapping Comparison Summary

I/D Task	I/D System	Mapping Summary
No	Yes	I-space APRs and D-space APRs contain the same relocation addresses.
Yes	Yes	I-space APRs map instruction space. D-space APRs map data space.
Yes	No	Missing special feature error at run time.

8.4 Conventional Task Mapping

Conventional tasks map their virtual addresses to their logical addresses through both I-space and D-space APRs. That is, the Task Builder does not separate instruction space or data space, and the system does not differentiate the spaces except by the logic inherent in the task. Therefore, the task must map to its logical address space by both sets of APRs, which are overmapped.

Figure 8–1 shows an 8K-word task that does not use I- and D-space, and that is built against an 8K-word resident library.

Figure 8–1:   Conventional Task Linked to a Region in I- and D-Space System
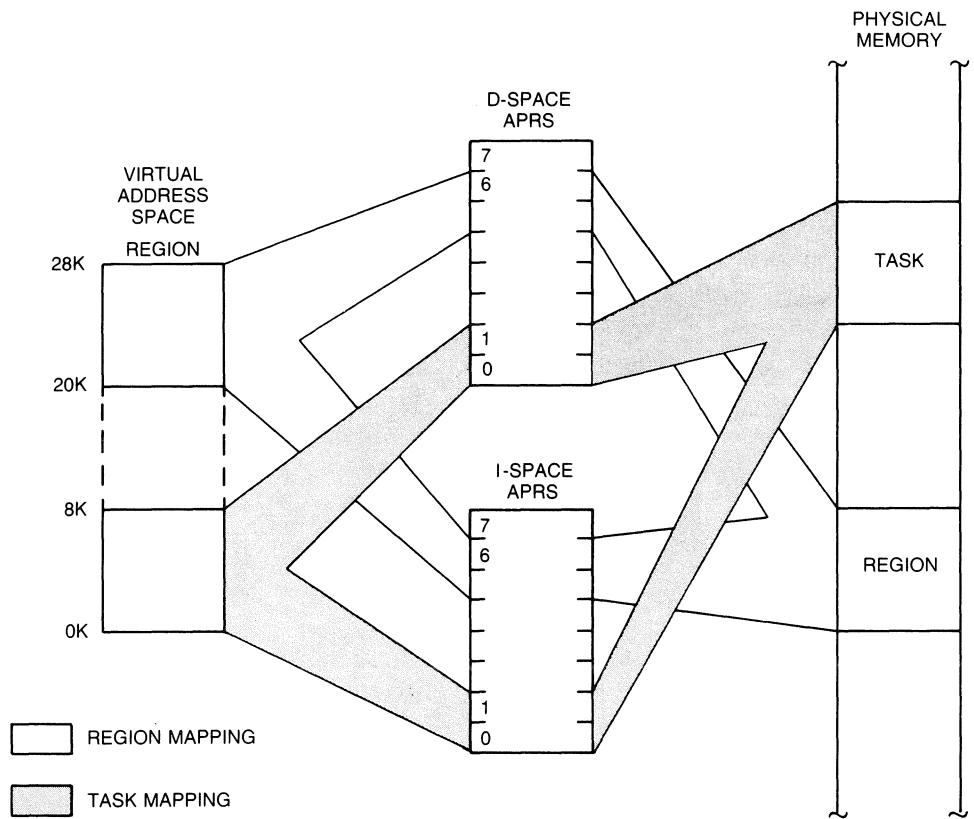
MK-01678-00

8.5   I- and D-Space Task Mapping

Figure 8–2 shows an 8K-word I- and D-space task. The Task Builder separated the data and instructions in this task. Because of the way the Task Builder processes task space, the task header must physically reside at the beginning of the task in I-space. The Task Builder puts the header that the monitor uses for task control in D-space. The task's stack is also in D-space.

The task in Figure 8–2 uses two APRs because of its size (8K-word). D-space APR 0 maps the task's header and stack and part of D-space.

Figure 8–2:   I- and D-space Task Mapping in an I- and D-space System

VIRTUAL
ADDRESS
SPACE

D-SPACE

8K

TASK
D-SPACE

STACK

HEADE

0K

I-SPACE

8K

TASK
I-SPACE

HEADER

0K

D-SPACE
APRS

7
6

1
0

I-SPACE
APRS

7
6

1
0

PHYSICAL
MEMORY

TASK
D-SPACE

STACK

HEADER   HEADER USED BY MONITOR

TASK
I-SPACE

HEADER   UNUSED HEADER COPY
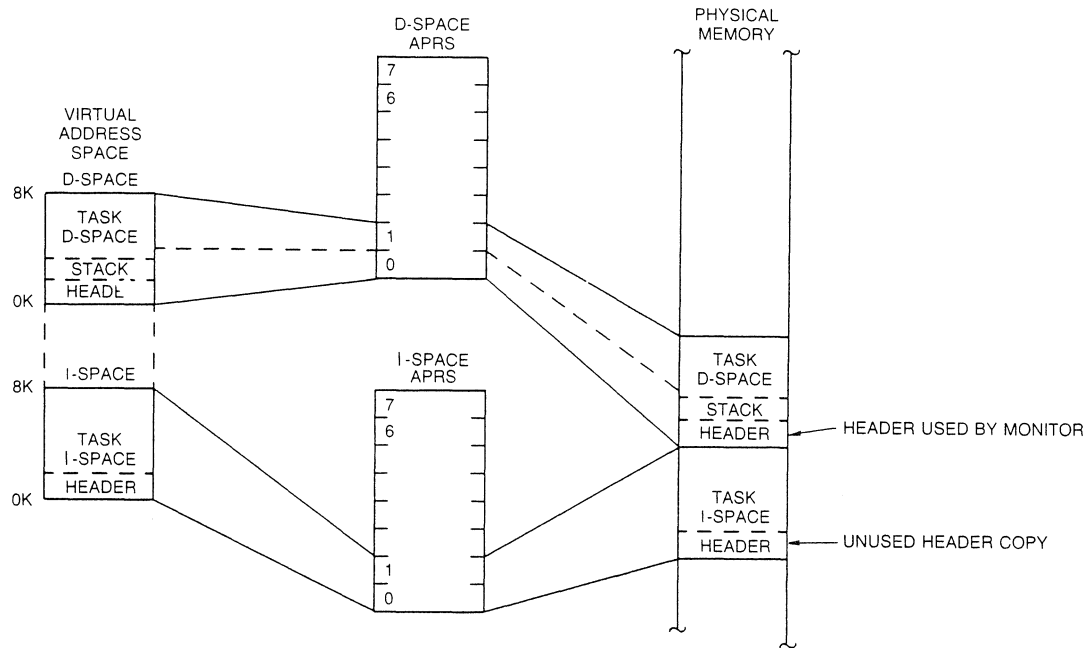
MK-02072-00

8.6   Designing an I-and D-Space Task

You design an I- and D-space task by specifying data space separately from instruction space. Good programming practice suggests that all data areas and buffers should be located in adjacent locations. Similarly, all instructions should be located in adjacent locations. However, the Task Builder will separate instruction and data space when it builds the task. For the Task Builder to do this, you must tell it which statements are data and which are instructions.

If you program in MACRO-11, use the MACRO-11 .PSECT directive to separate instructions and data. Use this directive with the instruction (I) attribute for all the instruction locations in your task's code. Use .PSECT with the data (D) attribute for all the data locations. You must define a data .PSECT in an I- and D-space task even if no actual data is contained in the task. In this case, the .PSECT can be of 0 length.

Note that a configuration consisting of libraries that map separate I- and D-space is not possible.

8.7   Concurrent Libraries

The Task Builder allows the optimization of a task's virtual address space on machines that support separated I- and D-spaces through the RSTS/E concurrent libraries feature. This feature allows a data region to share the same APR as a library that only contains I-space code (such as RMSRES).

When the RESCOM or COMMON options are used, the resulting task maps the specified region in D-space only. Depending upon how the task is designed, the equivalent I-space is mapped according to one of the following arrangements:

- As part of the task's instruction low segment area

- To a library containing I-space only code specified in a RESLIB or LIBR option

- Equal to the D-space for task builds done with no separate I- and D-spaces (/-ID)

- Not mapped at all if none of the above conditions is true

When RESLIB or LIBR options are used, the Task Builder initially gives both the I-space and the D-space APRs to the library. However, the D-space can be stripped from the library and given to another region under the following conditions:

- When a COMMON or RESCOM option is assignned to the same APR (see above)

- When an EXTTSK option expands the task's D-space into the same APR

- If the program executes a EXTM$ call that expands the tasks D-space into the same APR

- If the program executes a CRAW$ (or CRAFQ) call with the User Data Space (UDS) bit set that uses the APR

- If the program creates a dynamic region that uses the APR

- When a VSECT option is given that uses the APR

The D-space of one or more APRs of a library can be forced to stay with the I-space (protected) by the use of the proper Bit Mask values. The protection can occur at the following levels:

- When the library is built (/LI:mask)

- When the task is built (for example, RESLIB=MYLIB:RO:6:300 (300 is a mask)

- When the EXTM$ is executed (the mask is given as a parameter)

For more details on the protection mask and EXTM$, refer to Secton 7.8.1.

If the task is built without the /ID switch, the D-space will remain with the I-space for all APRs. Most of the items above will create some form of error if used as concurrent libraries without the /ID switch.

NOTE

The order in which options are presented to the Task Builder has no bearing on the resulting mapping.

Chapter 9

Supervisor-Mode Library

A supervisor-mode library is a resident library that doubles a user task's virtual address space by mapping the instruction space of the processor's supervisor mode.

A call from within a user task to a subroutine within a supervisor-mode library causes the processor to switch from user mode to supervisor mode. The user task transfers control to a mode-switching vector that the Task Builder includes within the task. After performing the mode switch, the vector transfers control to the called subroutine within the supervisor-mode library. When the library routine finishes executing, it transfers control to a completion routine within the library, which switches the processor back to user mode. The user task continues executing with the processor in user mode at the return address on the stack.

9.1 Mode-Switching Vectors

In a task that links to a supervisor-mode library, TKB includes a 4-word, mode-switching vector in the user task's address space for each entry point referred to in a subroutine in the library.

The following example shows the contents of a mode-switching vector:

```
MOV #COMPLETION-ROUTINE,-(SP)
CSM #SUPERVISOR-MODE-ROUTINE ADDRESS
```

NOTE

When switching from user mode to supervisor mode, all registers of the referencing task are preserved. All condition codes in the PSW are saved on the stack before they are cleared and must be restored by the completion routine.

9.2 Completion Routines

After the subroutine finishes executing, its RETURN statement transfers control to a completion routine that switches from supervisor mode to user mode. The completion routine returns program control back to the referencing task at the instruction after the call to the subroutine. The system library (SYSLIB) contains the following two completion routines:

$CMPCS   restores only the carry bit in the user mode PSW.

$CMPAL   restores all condition code bits in the user mode PSW.

9.3 Programming Considerations for the Contents of Supervisor-Mode Libraries
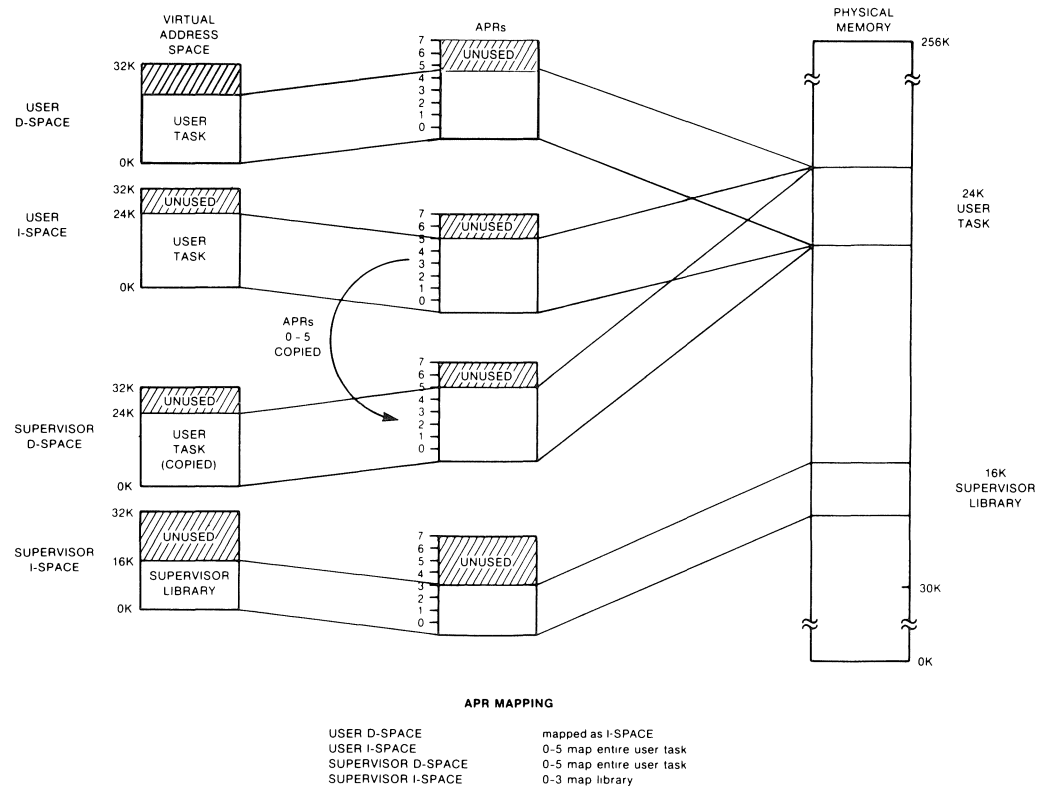
The following requirements must be followed for code in a supervisor-mode library:

- Only subroutines entered in the form JSR PC, should be used to invoke the library. Once within the library, other forms of calls are allowed.

- The library must not contain subroutines that use the stack pointer (R6) to pass parameters.

- Data may be placed on the stack prior to calling a supervisor routine as long as the data is pointed to by a register other than R6 and that the calling parameters are removed from the stack only after return to user mode.

- No asynchronous I/O calls (.WRITA/.READA) can be made from supervisor mode.

- If both the library and the referencing task link to a subroutine from the system library, then the entry point name of the subroutine must be excluded from the STB file for the library.

- The library normally does not contain data of any kind (even read-only) because the user supervisor D-space APRs map the user D-space of the task by default. This includes user data, buffers, I/O status blocks, Directive Parameter Blocks (only the $S directive form can be used, because the DPB for this form is pushed onto the stack at run time).

9.4 Supervisor-Mode Library Mapping

Supervisor-mode libraries are mapped with the supervisor I-space APRs. Supervisor D space APRs map, by default, either the user I space in non- I- and D-tasks or the user D space in tasks with separate I-and D spaces. An example of non-I- and D-task mapping is shown in figure 9-1. The MSDS$ directive can be used to cause the D-space APRs to map supervisor I-space.

Figure 9–1: Mapping of a 24K Word Conventional User Task Linking to a 16K
Word Supervisor-Mode Library

VIRTUAL
ADDRESS
SPACE

APRs

PHYSICAL
MEMORY

256K

USER
D-SPACE

32K

USER
TASK

0K

7
6
5
4
3
2
1
0

UNUSED

24K
USER
TASK

USER
I-SPACE

32K
24K

UNUSED

USER
TASK

0K

7
6
5
4
3
2
1
0

UNUSED

APRs
0 - 5
COPIED

SUPERVISOR
D-SPACE

32K
24K

UNUSED

USER
TASK
(COPIED)

0K

7
6
5
4
3
2
1
0

UNUSED

16K
SUPERVISOR
LIBRARY

SUPERVISOR
I-SPACE

32K

16K

UNUSED

SUPERVISOR
LIBRARY

0K

7
6
5
4
3
2
1
0

UNUSED

30K

0K

APR MAPPING

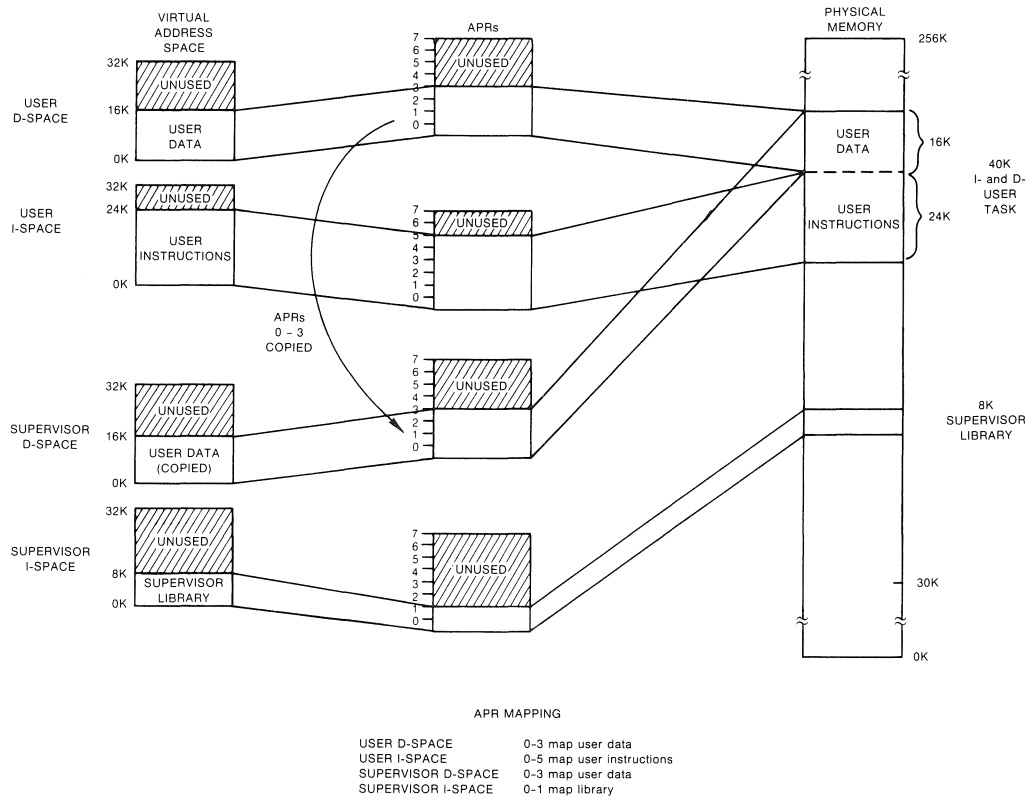|	
USER D-SPACE	mapped as I-SPACE
USER I-SPACE	0-5 map entire user task
SUPERVISOR D-SPACE	0-5 map entire user task
SUPERVISOR I-SPACE	0-3 map library

ZK-439-81

9.4.1 Supervisor-Mode Library Data

The supervisor D-space APRs always over-map the user D-space APRs (for
programs built using /ID). The supervisor D-space APRs over-map the user
I-space APRs (for programs that are built with /-ID).

9.4.2 Supervisor-Mode Libraries with I- and D-Space Tasks

I- and D-space tasks may link to supervisor-mode libraries. Instead of mapping to
the entire user task, the supervisor-mode library's D-space APRs map the task's
data space. Because the I- and D-space task maps its data with the D-space
APRs, the task's D-space APRs are copied into the supervisor-mode library's
D-space APRs. Therefore, the supervisor-mode library maps its own instructions
with supervisor-mode I-space APRs and the task's data with supervisor-mode
D-space APRs (Figure 9-2).

Figure 9–2: Mapping of a 40K Word I- and D-Space Task Linking to an 8K
Word Supervisor-Mode Library

APR MAPPING

USER D-SPACE   0–3 map user data
USER I-SPACE   0–5 map user instructions
SUPERVISOR D-SPACE   0–3 map user data
SUPERVISOR I-SPACE   0–1 map library

ZK-1105-82

9.5 Building and Linking to Supervisor-Mode Libraries

Building and linking to a supervisor-mode library is essentially the same as
building and linking to a conventional resident library (discussed in Chapter 7).
When you build a supervisor-mode library using the TKB command line, you
suppress the header by attaching /-HD to the task image file. During option
input, you suppress the stack area by specifying STACK=0. You specify the
partition in which the library is to reside and, optionally, the base address and
length of the library with the PAR option.

9.5.1 Relevant TKB Options

Use the following options to build and reference supervisor-mode libraries:

CMPRT	Indicates that you are building a supervisor-mode library and specifies the name of the completion routine.
RESSUP (SUPLIB)	indicates that your task references a supervisor-mode library.
GBLXCL	excludes a global symbol from the STB file of the supervisor-mode library.

These options are discussed briefly in the next sections and are fully documented in Chapter 12.

9.5.2 Mode-Switching Instruction

Mode switching occurs with a hardware instruction available with all processors that support the CSM instruction. Processors that do not support the hardware CSM instruction but do support supervisor mode (that is, the PDP-11/45, 11/50, 11/55 and 11/70) have the CSM instruction emulated by RSTS/E. Throughout the remainder of the chapter, supervisor-mode libraries are referred to as Change Supervisor-Mode (CSM) libraries.

9.5.2.1 Required Memory Layouts for Supported CSM Instructions

Only two forms of the CSM instruction are supported by Digital software:

CSM	# address	(7027)
CSM	(sp)+	(7026)

Any other form of the instruction does not put the target address where the vector software expects. Under normal uses, the TKB will supply the CSM as part of the vectoring.

9.5.2.2 The CSM Library Dispatching Process

When you build the referencing task and specify the SV or SW argument to the RESSUP or SUPLIB option, TKB includes a 4-word context-switching vector for each call to a subroutine in the library. This has been very generally discussed in Section 9.2. This section discusses the CSM library vector in more detail.

CSM mode switching occurs as follows:

* The vector is entered with the return address on top of the stack (TOS).

* The vector pushes the completion-routine address on the stack.

* A CSM instruction is executed with the supervisor-mode entry point as the immediate addressing mode parameter.

The CSM instruction executes the following steps either through hardware action or software emulation:

* Evaluating the source parameter and storing the entry point address in a temporary register

* Copying the user stack pointer to the supervisor stack pointer

- Placing the current PSW and PC on the supervisor stack, clearing the condition codes in the PSW

- Pushing the entry point address on the supervisor stack

- Placing the contents of location 10 in supervisor I-space into the PC and transferring execution to that address in supervisor mode

The stack looks like this when the processor begins to execute at the contents of virtual 10 in supervisor mode:

```
higher memory address
     user sp ---->   return address
                     completion routine address in super mode
                     PSW
                     PC
     super sp --->   entry point address of routine in super mode
lower memory address
```

Because the CSM library mode-switching vector processor begins executing at the contents of virtual 10 in supervisor mode, the completion routine must be located at virtual 0. In this way, virtual location 10 is within properly mapped memory.

9.6 CSM Libraries

This section discusses how you build and link to CSM libraries. It also shows an extended example of building and linking to a CSM library and explains the context-switching vectors and completion routines for CSM libraries.

9.6.1 Building a CSM Library

You can indicate to the Task Builder that you are building a CSM library by specifying the name of the completion routine as the argument for the CMPRT option. This option places the name of the completion routine into the library's STB file. Link the completion routine, either $CMPAL or $CMPCS, located in LB:SYSLIB.OLB, as the first input file. Although the completion routines are located in the system library (which is ordinarily referenced by default), you must explicitly indicate it and link it as the first input file. You must also specify in the PAR option a base of 0 for the partition in which the library will reside. These two steps locate the completion routine at virtual 0 of the library's virtual address space. This placement is a requirement.

Specify the name of any global symbols that you would like to exclude from the library's STB file as the argument to the GBLXCL option. You must exclude from the STB file of a supervisor-mode library any symbol defined in the library that represents the following:

- An entry point to a subroutine that uses the stack pointer to address parameters

- An entry point to a subroutine mapped in user mode that the referencing user task calls and also exists in the CSM library

The following sample TKB command sequence builds a CSM library called SUPER in directory [30,55] on device SY:

```
TKB>SUPER/-HD/LI/PI,SUPER/MA,SUPER=
TKB>LB:SYSLIB/LB:CMPAL,SY:[30,55]SUPER
TKB>/
Enter Options:
TKB>STACK=0
TKB>PAR=GEN:0:2000
TKB>CMPRT=$CMPCS
TKB>GBLXC=$SAVAL
TKB>//
>
```

The second line in this example is the key line. The first LB:SYSLIB indicates the .OLB file from which the /LB:CMPAL is to be loaded. Note the different meanings for the LB:s, and that by locating the CMPAL before the actual library code, the contents of location IO in supervisor mode is assured.

RSTS/E requires the output of the Task Builder to be converted to a loadable library by the MAKSIL program (refer to the *RSTS/E Programmer's Utilities Manual* for additional information on MAKSIL). An example of MAKSIL is shown here:

```
$ run $maksil
MAKSIL  V10.0-L
Resident Library name? SUPER
Task-built Resident Library input file <SUPER.TSK>?
Include symbol table (Yes/No) <Yes>?
Symbol table input file <SUPER.STB>?
Resident Library output file <SUPER.LIB>?
SUPER built in 1 K-words, 21 symbols in the directory
SUPER.TSK renamed to SUPER.TSK <40>
```

The library is built without a header or stack, like all shared regions. It is position-independent and has only one program section named .ABS. The /LI switch in TKB accomplishes this, eliminating program section name conflicts between the library and the referencing task.

The completion routine module CMPAL is specified first in the input line. The library will run in partition GEN at 0 and is not more than 1K words. These are two aspects of building supervisor-mode libraries specific to these libraries: the completion routine must be linked first and must reside at virtual 0. (Why the CSM library must reside at virtual 0 is discussed in the next section.)

The CMPRT option specifies the global symbol $CMPCS, which is the entry point of the completion routine. Note that the name for the system library object module is CMPCS and its corresponding global entry symbol is $CMPCS.

The GBLXCL option excludes $SAVAL from the library's STB file because the user task must reference a copy of $SAVAL that is mapped with user mode APRs.

9.6.2 Linking to a CSM Library

If your task links to a user-owned CSM library, use the RESSUP option. If your task links to a system-owned CSM library, use the SUPLIB option. These options tell TKB that the task will link to a supervisor-mode library. The option takes up to three arguments, as follows:

- The file specification (RESSUP option) or name (SUPLIB option) of the library

- A switch that tells TKB whether to use system-supplied, mode-switching vectors

- A switch that determines whether the library is to be attached read/write or read-only

- For position-independent libraries, an APR that must be APR 0 so that the library's completion routine is mapped at virtual 0

This information enables TKB to find the STB file for the CSM library, include a 4-word mode-switching vector within the user task for each call to a subroutine within the library, and correctly map the library at virtual 0 in the library image.

The following examples of TKB command sequences build a task named REF, which references the library SUPER that you built in Section 9.6.1:

```
TKB> REF,REF=REF
TKB> /
Enter Options:
TKB> RESSUP=SUPER/SV:0
TKB> //
>
```

This sequence tells TKB to include in the logical address space of REF a user-owned, supervisor-mode library named SUPER. TKB includes a 4-word mode-switching vector within the user task for each call to a subroutine within the library. The CSM library is position-independent and is mapped with APR 0.

```
TKB> REF/ID/DA,REF=REF
TKB> /
Enter Options:
TKB> RESSUP=SUPER/SW:0
TKB> //
>
```

This sequence does the same function except that it also includes ODT for use in debugging the task together with the CSM library. Note the /ID is required to select the proper ODT version that contains supervisor-mode commands. The /SW switch attaches the CSM library read/write so that breakpoints may be placed in the library. The CSM library must also be INSTALLed with read/write access for ODT to be able to make changes in it.

9.6.3 Example of a CSM Library and Building a Task

This example shows you the code and maps and the TKB command sequence for building with a CSM library. Example 9-1 shows the code for the library SUPER and Example 9-2 shows its accompanying map. Example 9-3 shows the code for the completion routine $CMPCS that is linked into SUPER from the system library. Example 9-4 shows the code for referencing task TSUP, and Example 9-5 shows its accompanying map.

Example 9-1: Code for SUPER.MAC

```
        .TITLE  SUPER
        .IDENT  /01/

SORT::
        CALL    $SAVAL          ; SAVE ALL REGISTERS
        TST     (R5)+           ; SKIP OVER NUMBER OF ARGUMENTS
        MOV     (R5)+,R0        ; GET ADDRESS OF LIST
        MOV     (R5)+,R4        ; GET ADDRESS OF LENGTH OF LIST
        MOV     (R4),R4         ; GET LENGTH OF LIST
        BEQ     40$             ; IF NO ARGUMENTS
        MOV     R0,R5           ;
        DEC     R4              ;
10$:
        MOV     R5,R0           ; COPY
        MOV     R4,R3           ; COPY LENGTH OF LIST
20$:
        TST     (R0)+           ; MOVE POINTER TO NEXT ITEM
        CMP     (R5),(R0)       ; COMPARE ITEMS
        BLE     30$             ; IF LE IN CORRECT ORDER
        MOV     (R5),R2         ; SWAP ITEMS
        MOV     (R0),(R5)       ;
        MOV     R2,(R0)         ;
30$:
        DEC     R3              ; DECREMENT LOOP COUNT
        BGE     20$             ; IF NE LOOP
        DEC     R4              ; DECREMENT
        BLE     40$             ; IF EQ SORT COMPLETED
        TST     (R5)+           ; GET POINTER TO NEXT ITEM
        BR      10$             ; TO BE COMPARED
40$:
        RETURN

SEARCH::
        CALL    $SAVAL          ; SAVE ALL THE REGISTERS
        CMP     #4,(R5)+        ; FOUR ARGUMENTS?
        BNE     20$             ; IF NE NO
        MOV     (R5)+,R0        ; GET ADDRESS OF NUMBER TO LOCATE
        MOV     (R5)+,R1        ; ADDRESS OF LIST SEARCHING
        MOV     (R5)+,R2        ; GET ADDRESS OF LENGTH OF LIST
        MOV     (R2),R2         ; GET LENGTH OF LIST
        BEQ     20$             ; IF NO ARGUMENTS
        MOV     (R5),R5         ; ADDRESS OF RETURNED VALUE
        MOV     R2,R3           ; COPY LENGTH

10$:
        CMP     (R0),(R1)+      ; IS THIS THE NUMBER?
        BEQ     30$             ; IF EQ YES
        BMI     20$             ; IF MI NUMBER NOT THERE
        DEC     R2              ; DECREMENT LOOP COUNT
        BNE     10$             ; IF NE NOT AT END OF LIST
20$:
        MOV     #-1,(R5)        ; END OF LIST PASS BACK ERROR
        RETURN
30$:
        SUB     R2,R3           ; NUMBER FOUND - GET INDEX INTO LIST
        INC     R3              ;
        MOV     R3,(R5)         ; RETURN INDEX
        RETURN
        .END
```

Note the use of the routine $SAVAL in both the Library SUPER.MAC and the main program TSUP.MAC in Example 9-4. This double usage is the reason the global is excluded from the library's STB with the GBLXCL option.

Example 9-2: Memory Allocation Map for Super

```
SUPER.TSK    Memory Allocation Map   TKB M43.00       Page 1
                     11-AUG-90    15:41

Partition name : GEN
Identification : 03.01
Task   UIC      : [30,55]
Task attributes: -HD,PI
Total address windows: 1.
Task   image   size  : 128. words
Task address limits: 000000 000343
R-W disk blk limits: 000002 000002 000001 00001.

Root segment: CMPAL

R/W mem  limits: 000000 000341 000342 00226.
Disk blk limits: 000002 000002 000001 00001.

Memory allocation synopsis:

Section                                     Title  Ident  File
-------                                     -----  -----  ----
.BLK.:(RW,I,LCL,REL,CON) 000000 000342 00226.
                         000000 000140 00096. CMPAL  03.01  SYSLIB.OLB
                         000140 000140 00096. SUPER  01     SUPER.OBJ
                         000300 000042 00034. SAVAL  00     SYSLIB.OLB

Global symbols:

SEARCH 000220-R  SORT   000140-R  $CMPAL  000022-R  $CMPCS 000110-R
$SAVAL 000300-R  $SRTI  000002-R

Task builder statistics:

    Total work file references: 300.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 6466. words (25. pages)
    Size of work file: 1024. words (4. pages)

    Elapsed time:00:00:08
```

Example 9–3: Completion Routine $CMPCS from SYSLIB.OLB

```
        .TITLE   CMPAL
        .IDENT   /0204/
;
;
;              COPYRIGHT (c) 1987,1989 BY
;        DIGITAL EQUIPMENT CORPORATION, MAYNARD
;         MASSACHUSETTS.  ALL RIGHTS RESERVED.
;
; THIS  SOFTWARE  IS  FURNISHED  UNDER  A LICENSE AND MAY BE USED
; AND  COPIED  ONLY IN  ACCORDANCE WITH THE TERMS OF SUCH LICENSE
; AND WITH  THE INCLUSION  OF THE ABOVE  COPYRIGHT  NOTICE.   THIS
; SOFTWARE, OR ANY OTHER  COPIES  THEREOF, MAY NOT BE PROVIDED OR
; OTHERWISE MADE  AVAILABLE TO ANY OTHER PERSON.  NO TITLE TO AND
; OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.
;
; THE INFORMATION  IN THIS DOCUMENT IS SUBJECT  TO CHANGE WITHOUT
; NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT  BY  DIGITAL
; EQUIPMENT CORPORATION.
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF
; ITS SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
;
        .ENABL   LC
;
;
; This module supports the "new" transfer vector format generated
; by the Task Builder for entering super mode libraries.  This
; format is optimized for speed and size and supports user data
; space tasks.
;
; The CSM dispatcher routine and the standard completion routines
; $CMPAL and $CMPCS are included in this module due to the close
; interaction between them.
;
;
; **-CSM Dispatcher-Dispatch CSM entry
;
; This module must be linked at virtual zero in the supervisor-mode
; library.  It is entered via a four word transfer vector of the
; form:
;
;       MOV     #completion-routine,-(SP)
;       CSM     #routine
;
; Note: Immediate mode emulation of the CSM instruction is required
;       in the Executive for 11/70s.
;
; The CSM instruction transfers control to the address contained in
; supervisor mode virtual 10.  At this point the stack is the
; following:
;
;          (SP)   Routine address
;         2(SP)   PC (past end of transfer vector)
;         4(SP)   PSW with condition codes cleared
;         6(SP)   Completion-routine address
;        10(SP)   Return address
;
```

Example 9–3 (Cont.):   Completion Routine $CMPCS from SYSLIB.OLB

```
; A routine address of 0 is special-cased to support return to
; supervisor mode from a user mode debugging aid (ODT).  In this
; case stack is the following:
;
;           (SP)    zero
;          2(SP)    PC from CSM to be discarded
;          4(SP)    PSW from CSM to be discarded
;          6(SP)    Super mode PC supplied by debugger
;         10(SP)    Super mode PSW supplied by debugger
;
; To allow positioning at virtual zero, this code must be in the
; blank PSECT which is first in the TKB's PSECT ordering.

        .PSECT
        .ENABL  LSB

; Debugger return to super mode entry. Must start at virtual zero

        CMP     (SP)+,(SP)+      ; Clean off PSW and PC from CSM

;
: **-$SRTI-SUPER mode RTI
;
; This entry point performs the necessary stack management to
; allow an RTI from super mode to either super mode or user mode.
; In this case, the stack is the following:
;
;           (SP)    Super mode PC
;          2(SP)    Super mode PSW

$SRTI:: TST     2(SP)            ; Returning to user mode?
        BR      70$              ; Join common code
; CSM transfer address, this word must be at virtual 10 in super
; mode

        .WORD   CSMSVR           ; CSM dispatcher entry

; Dispatch CSM entry

CSMSVR: MOV     6(SP),2(SP) ; Set completion routine address for RETURN
        JMP     @(SP)+      ; Transfer to super mode library routine

;
; **-$CMPAL-Completion routine which sets up NZVC in the PSW
;
; Copy all condition codes to stacked PSW.  Current stack:
;
;           (SP)    PSW with condition codes cleared
;          2(SP)    Completion routine address (to be discarded)
;          4(SP)    Return address
;
```

Example 9–3 (Cont.):   Completion Routine $CMPCS from SYSLIB.OLB

```
$CMPAL::BPL       40$               ;
        BNE       20$               ;
        BVC       10$               ;
        BIS       #16,(SP)          ; Set NZV
        BR        $CMPCS            ;
10$:    BIS       #14,(SP)          ; Set NZ
        BR        $CMPCS            ;
20$:    BVC       30$               ;
        BIS       #12,(SP)          ; Set NV
        BR        $CMPCS            ;
30$:    BIS       #10,(SP)          ; Set N
        BR        $CMPCS            ;
40$:    BNE       60$               ;
        BVC       50$               ;
        BIS       #6,(SP)           ; Set ZV
        BR        $CMPCS            ;
50$:    BIS       #4,(SP)           ; Set Z
        BR        $CMPCS            ;
60\:    BVC       $CMPCS            ;
        BIS       #2,(SP)           ; Set V

;
; **-\CMPCS-Completion routine which sets up only C in the PSW
;
; Copy only carry to stacked PSW.  Current stack:
;
;       (SP)      PSW with condition codes cleared
;       2(SP)     Completion routine address (to be discarded)
;       4(SP)     Return address
;
$CMPCS::ADC       (SP)              ; Set up carry
        MOV       4(SP),2(SP)       ; Set up return address for RTT
        MOV       (SP)+,2(SP)       ; And PSW. Returning to super mode?
70$:    BPL       80$               ; If PL yes
        MOV       #6,-(SP)          ; Number of bytes for (SP), PSW, and PC
        ADD       SP,(SP)           ; Compute clean stack value
        MTPI      SP                ; Set up previous stack pointer
80\:    RTT                         ; Return to previous mode and caller
        .DSABL    LSB
        .END
```

Example 9–4: Code for TSUP.MAC

```
        .TITLE  TSUP
        .IDENT  /01/
        .MCALL  QIOW$,DIR$,QIOW$S
WRITE:  QIOW$   IO.WVB,5,1,,,,<OUT,,40>
READIN: QIOW$   IO.RVB,5,1,,,,<OUT,5>
IARRAY: .BLKW   12.
LEN:    .BLKW   1
IART:   .BLKW   1
INDEX:  .WORD   0
OUT:    .BLKW   100.
ARGBLK:
EDBUF:  .BLKW   10.
FMT1:   .ASCIZ  /%2SARRAY(%D)=/
FMT2:   .ASCIZ  /%N%2SNUMBER TO SEARCH FOR?/
FMT3:   .ASCIZ  /%N%2S%D WAS FOUND IN ARRAY(%D)/
FMT4:   .ASCIZ  /%N%2S%D WAS NOT IN ARRAY/
FMT5:   .ASCIZ  /%2SARRAY(%D)=%D/
        .EVEN
START:
        MOV     #IARRAY,R0      ; GET ADDRESS OF ARRAY
        MOV     #10,R1          ; SET LENGTH OF ARRAY
5$:
        CLR     (R0)+           ; INITIALIZE ARRAY
        DEC     R1              ; LOOP
        BNE     5$
        MOV     #IARRAY,R0      ;
        MOV     #INDEX,R2
10$:
        MOV     #FMT1,R1        ; FORMAT SPECIFICATION (ADDRESS
                                ; OF INPUT STRING)
        MOV     (R2),EDBUF      ; GET INDEX
        INC     EDBUF           ;
        CALL    PRINT           ; PRINT MESSAGE
        CALL    READ            ; READ INPUT
        MOV     IART,(R0)+      ; PUT BINARY KEYBOARD INPUT INTO ARRAY
        BEQ     20$             ; ZERO MARKS END OF INPUT
        INC     (R2)            ;
        CMP     (R2),#10.
        BNE     10$             ; IF NE YES
20$:
        MOV     (R2),LEN        ; CALCULATE LENGTH OF ARRAY
        MOV     #ARGBLK,R5      ; GET ADDRESS OF ARGUMENT BLOCK
        MOV     #2,(R5)+        ; NUMBER OF ARGUMENTS
        MOV     #IARRAY,(R5)+   ; PUT ADDRESS OF ARRAY
        MOV     #LEN,(R5)       ;
        MOV     #ARGBLK,R5      ;
        CALL    SORT            ; SORT ARRAY
;+
;Task Builder replaced call to SORT subroutine in SUPLIB with
;4-word context switching vector. Flow of control switches to SUPLIB
;via the vector and back via the completion routine $CMPCS. TSUP
;continues executing at the next instruction.
;-
        CLR     R2              ;
        MOV     #IARRAY,R0      ; GET ARRAY ADDRESS
```

Example 9-4 (Cont.):   Code for TSUP.MAC

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

PRINT:
        CALL    $SAVAL          ; SAVE ALL REGISTERS
        MOV     #OUT,R0         ; ADDRESS OF OUTPUT BLOCK
        MOV     #EDBUF,R2       ; START ADDRESS OF ARGUMENT BLOCK
        CALL    $EDMSG          ; FORMAT MESSAGE
        MOV     R1,WRITE+Q.IOPL+2 ; PUT LENGTH OF OUTPUT
                                ; BLOCK INTO PARAMETER BLOCK
        DIR$    #WRITE          ; WRITE OUTPUT BLOCK
        RETURN
READ:
        CALL    $SAVAL          ; SAVE ALL REGISTERS
        DIR$    #READIN         ; READ REQUEST
        MOV     #OUT,R0         ; GET KEYBOARD INPUT
        CALL    $CDTB           ; CONVERT KEYBOARD INPUT TO BINARY
        MOV     R1,IART         ; PUT INPUT INTO BUFFER
        RETURN

        .END    START
```

TSUP prompts you to enter numbers at your terminal. It calls a subroutine
in SUPER to sort the numbers. Then it displays the numbers you entered as
array entries and prompts you to request a number to search for. TSUP calls a
subroutine in SUPER.LIB to search for the number. Finally, TSUP indicates that
either the number was not found or the array location in which the number is
stored.

Example 9–5: Memory Allocation Map for TSUP

```
TSUP.TSK    Memory allocation map  TKB M43.00     Page 1
                    11-AUG-90   15:41

Partition name : GEN
Identification : 01
Task  UIC     : [30,55]
Stack     limits: 000274 001273 001000 00512.
PRG xfr address: 002130
Total address windows: 2.
Task  image  size  : 1344. words
Task address limits: 000000 005133
R-W disk blk limits: 000002 000007 000006 00006.

*** Root segment: TSUP

R/W mem  limits: 000000 005133 005134 02652.
Disk blk limits: 000002 000007 000006 00006.

Memory allocation synopsis:

Section                                Title  Ident  File
-------                                -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)  001274 002334 01244.
                           001274 001234 00668. TSUP   01     TSUP.OBJ
CMPAL :(RW,I,LCL,REL,CON)  000000 000474 00316.
PUR$D :(RO,I,LCL,REL,CON)  003630 000076 00062.
PUR$I :(RO,I,LCL,REL,CON)  003726 000752 00490.
$$RESL:(RO,I,LCL,REL,CON)  004700 000212 00138.
$$SLVC:(RO,I,LCL,REL,CON)  005112 000020 00016.

TSUP.TSK;1 Memory Allocation Map  TKB M43.00     Page 2
                    11-AUG-90   15:41

*** Task builder statistics:

    Total work file references: 2477.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 6988. words (27. pages)
    Size of work file: 1024. words (4. pages)

    Elapsed time:00:00:05
```

9.6.3.1 Building the Library SUPER

To build SUPER in directory [30,55] on device SY:, use the following TKB command sequence:

```
TKB> SUPER/-HD/LI/PI,SUPER/MA,SUPER=
TKB> LB:SYSLIB/LB:CMPAL,SY:[30,55]SUPER
TKB> /
Enter Options:
TKB> STACK=0
TKB> PAR=GEN:0:2000
TKB> CMPRT=$CMPCS
TKB> GBLXCL=$SAVAL
TKB> //
>
```

```
$ run $maksil
MAKSIL  V10.0-L
Resident Library name? SUPER
Task-built Resident Library input file <SUPER.TSK>?
Include symbol table (Yes/No) <Yes>?
Symbol table input file <SUPER.STB>?
Resident Library output file <SUPER.LIB>?
SUPER built in 1 K-words, 21 symbols in the directory
SUPER.TSK renamed to SUPER.TSK<40>
```

SUPER is built without a header or stack. It is position-independent and has
only one program section, named .BLK. The /LI switch eliminates program
section name conflicts between the library and the referencing task.

The completion routine module CMPAL is specified first in the input line. The
library will run in partition GEN at 0 and is not more than 1K words.

The GBLXCL option excludes $SAVAL from the library's STB file. Exclude
$SAVAL from the STB file because the referencing task, TSUP, also calls $SAVAL.
If TSUP finds $SAVAL in the STB file of SUPER, it will not link a separate
copy of $SAVAL into its task image from the system library. If TSUP could not
link to a copy of $SAVAL that is mapped through user APRs, TSUP would call
$SAVAL as a subroutine residing within the supervisor-mode library but without
the necessary mode-switching vector and completion routine support. This option
forces TKB to link $SAVAL from the system library into the task image for TSUP.

The memory allocation map in Example 9-2 shows the following information:

- SUPER begins at virtual 0.

- The completion routine, $CMPAL, is linked into the library from the system
  library at virtual 0.

- The entry point $CMPAL is located at virtual 22, SEARCH is located at 220,
  and SORT is located at 140. All of these entry points are relocatable.

9.6.3.2 Building TSUP

Use the following TKB command sequence to build a task, TSUP, that links to
SUPER:

```
TKB> TSUP,TSUP=TSUP
TKB> /
Enter Options:
TKB> RESSUP=SUPER/SV:0
TKB> //
>
```

This command sequence tells TKB to include in the logical address space of TSUP
a user-owned supervisor-mode library named SUPER. TKB includes a 4-word
mode-switching vector within the task image for each call to a subroutine within
the library. The library is position-independent and is mapped with supervisor
I-space APR 0. This is a requirement for CSM libraries because the CSM library
expects to find the entry point of the completion routine at location 10.

The memory allocation map for TSUP in Example 9-5 shows the following infor-
mation:

- $CMPAL is linked from the STB file of the library and begins at location 0.

- The mode-switching vectors begin at 5112 and are 16 bytes in length. This
  means that TSUP calls subroutines within the library two times (four words
  for each vector).

- The initiation routine $SUPL is located at 4700.

- The SEARCH and SORT subroutines that were located at virtual 220 and 140, respectively, in the virtual address space of SUPER have been relocated to the mode-switching vectors residing at 5112 and 5122, respectively, in TSUP.

- The system library module SAVAL, containing $SAVAL, has been linked into the task image instead of including $SAVAL from the library's STB file.

9.6.3.3 Running TSUP

After building SUPER and TSUP as indicated in the task-build command sequence discussed previously, install the library SUPER and run TSUP. TSUP prompts you for the position in which to store the number in the array:

```
ARRAY (x)
x
```

Enter a number. TSUP stores the number in the array and prompts you again for a number. This continues until you have entered a 0, an invalid number, or 10 numbers. Then TSUP calls the SORT routine in SUPER.

When you enter a number, TSUP calls the SEARCH routine in SUPER. TSUP then outputs a message indicating whether the number was in the array.

9.6.4 Passing Parameters Using Stack Space

Note also the existence of the two independent stack pointers: (NOT stacks) one for user mode and one for supervisor mode. The fact that a variable amount of additional data is placed on one stack and not the other by the system is why the SP cannot be used as a pointer to parameters. The following example shows a method of properly using the stack to pass parameters to and from a CSM library routine:

In the user mode program:

```
        .
        .
        MOV    R0,-(SP)     ;make a register available
        MOV    DATA1,-(SP)  ;parameters needed by CSM library routine
        MOV    DATA2,-(SP)  ;placed on stack as a packet
        MOV    SP,R0        ;put address of packet in another register
        JSR    PC,SUPER_ROUTINE ;transfer to CSM library routine
                            ;on return to user mode
        ADD    #4,SP        ;remove data packet from user stack
        MOV    (SP)+,R0     ;restore register
        .
        .
```

In the CSM library code:

```
        SUPER_ROUTINE:
                MOV    (R0),R2      ;put DATA2 in R2
                MOV    2(R0),R3     ;put DATA1 in R3
                .
                .
```

Data may be returned from a CSM library in the same manner as long as the data packet was reserved on the stack and addressed in the user mode prior to the call to the CSM library routine.

9.7 Using Supervisor-Mode Libraries as User-Mode Resident Libraries

Supervisor-mode libraries can double as conventional resident libraries. For position-independent supervisor-mode libraries, rebuild the referencing task using the RESLIB option instead of the RESSUP option. Indicate the first available user mode APR that you want to map the library. For CSM libraries, this will always change because you cannot map a shared region with APR 0. You do not have to rebuild the library.

For absolute supervisor-mode libraries, rebuild the referencing task using the RESLIB option instead of the RESSUP option. Rebuild the library only if the beginning partition address in the PAR option is incompatible with the address limits of your referencing task.

Once installed, the library can be used in either mode at any time. All APR relocation is done when the task that references the library is built. There is no change to the library itself.

9.8 Multiple Supervisor-Mode Libraries

A user task can reference multiple supervisor-mode CSM libraries. However, all the CSM libraries must use the completion routine that begins at virtual 0 in supervisor-mode instruction space.

9.9 Linking Supervisor-Mode Libraries

You cannot link supervisor-mode libraries together, and you cannot link a supervisor-mode library to a user-mode library. Calling a user-mode library is not possible because its code is not mapped through the I-space APRs while in the supervisor-mode library. However, you can link user mode libraries to a supervisor-mode library.

9.10 Writing Your Own Vectors and Completion Routines

You can write your own mode-switching vectors and completion routines. This may be necessary for threaded code. If you use your own vectors, build them into the task and use the /-SV or the /-SW switch on the RESSUP or RESLIB option when you build the referencing task. If you create your own completion routines, write your completion routine to resemble the system-supplied completion routines (see Example 9-3) as much as possible. If you do not retain the last three lines of code as indicated in Example 9-3, the task may not complete if the Executive processes an interrupt before the switch back to user mode has completed.

The debugger ODT expects the presence of the system-supplied completion routines to handle breakpoints and all other traps while in supervisor mode. Lack of this code may cause the task to abort should a trap occur.

9.11 Overlaid Supervisor-Mode Libraries

It is possible to use overlaid supervisor-mode libraries. However, the following restrictions must be noted when building these libraries:

- The completion routine for the library must be in the root.
- Only one level of overlaying is allowed (see Figure 9-3).

Figure 9–3: Overlay Configuration Allowed for Supervisor-Mode Libraries

```
    Allowed            Allowed              Not Allowed

  A       B       A   B   C   D       A   B     C   D
 _|_____|_     _|___|___|___|_     _|___|_   _|___|_
        |               |               |_E       |_F
                                           |_____|
 ROOT |           ROOT |               ROOT |
```
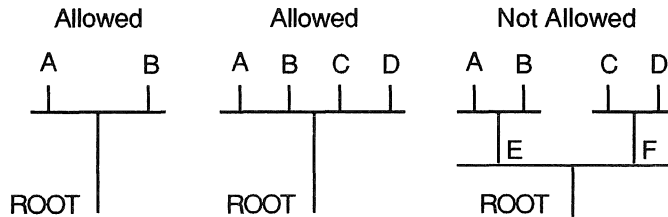
9.12 Using ODT to Debug CMS Library

The use of ODT with a CSM library is the same as with other types of tasks except for following:

- The CSM library must have been installed as RW and 1 user.
  INSTAL/LIB/NOREAD_ONLY/NOSHAREABLE file[ppn]
- When the task that uses the CSM library is Task built, the parameters change:
  + Indicate the inclusion of the proper ODT.OBJ file by using the /DA and /ID switches.
  + Indicate that the library should be loaded Read/Write by replacing the supervisor-mode vector switch /SV:0 with the /SW:0 switch.
- Once in ODT, the Z command switches ODT to the supervisor mode and the U command switches it back to user mode.
- Only the UI, UD and ZI spaces are defined in ODT. Any attempt to use the ZD (supervisor data) space will result in the nonexistent memory indicator from ODT. However, the data in supervisor D-space is mapped the same as the user data space and therefore is available using the UD command.

- Breakpoints and single step work the same as in user space. There is, however, no indicator as to which space, Z or U, the breakpoint was in. It is the task of the user to determine or keep track of which space the program is in when in ODT.

- ODT operating with supervisor mode requires the use of the Digital system-supplied supervisor-mode completion routines that handle the special mode switching needs of ODT.

9.13 Trap Handling with Supervisor Libraries

Although asynchronous I/O calls (READA \ & WRITA) are illegal from a task that is also using supervisor mode, other forms of traps may require handling by a task that is using supervisor mode.

The two system asynchronous traps, FPP exception and Control-C interception, as well as all forms of Synchronous Service Traps (SSTs) are permitted. The trap service routines for these may be located in either user or supervisor mode.

If the service routines reside in supervisor mode, they must adhere to additional requirements that their user mode counter-parts do not need.

- The routine must exit using either the SSTX$ or ASTX$ calls. Whereas in user mode, only service routines are allowed to clean the PC \ & PSW off the stack and continue without returning, tasks using supervisor mode must use an exit call. Note that the hardware prohibits an RTI instruction from returning a task to supervisor mode from a service routine in user mode.

- Routines located in supervisor space usually must obey all the rules of supervisor library routines such as not calling user mode routines and having no data within the code.

9.13.1 Locating Service Routines

The task informs RSTS/E in which space the service routine is at the time the respective service vectors are set up. The following calls set up the trap service vectors for the task:

```
FPPA$     Floating point exception
SCCA$     Control-C interception
SVTK$     SST trap vector table
SVDB$     SST debugging trap vector table
```

9.13.1.1 FPPA$ AND SCCA$

In the cases of FPPA$ and SCCA$, the task issues a single-address vector. Which space the service routine resides in is indicated by bit 0 of the vector. If it is zero, the service routine is in user space. If the bit is 1 (an odd address vector), the routine is in supervisor space.

9.13.1.2 SVTK$ and SVDB$

In the SVTK$ and SVDB$ calls, the task issues a list of vectors that will be associated with the different events (BPT, for example). The bit 0 in each of the individual service routine vector addresses indicates which mode that routine is in, but in a different way from FPPA$ and SCCA$.

An even vector entry causes the SST routine to be executed in the same mode (either user or supervisor) that the processor was in when the SVTK$ or SVDB$ call was issued. An odd vector entry causes the SST routine to be executed in the other mode. For example, if the processor was in supervisor mode when an SVTK$ was issued and the vector was odd (bit 0 set), the SST routine is executed in the user mode. This method of designation of service routine location is the same as RSX-11/M-Plus. This method allows the individual SSTs to be different modes at the same time (BPT in user and address trap in supervisor). Bit 0, which is the flag(s), is a member of the vector list, *not* the address pointer to the vector list.

9.14 Building to a Supervisor-Mode RMS Library

On the RSTS/E systems that support supervisor mode, you may choose to use RMSRES as a supervisor-mode library instead of user mode. Because this configuration uses two otherwise idle supervisor-mode APRs to map most of the RMS-11 code, the impact of the RMS-11 code on your user mode virtual address space is reduced to the absolute minimum. There also may be slight performance advantages over the clustered RMS-11 configuration.

To use RMSRES as a supervisor-mode library, use the following sequence of commands:

```
TKB> command string
TKB> /
ENTER OPTIONS:
TKB> RESSUP=RMS$:RMSRES/SV:0
TKB> //
```

The following modules must be included in the root of the task:

```
LB:RMSLIB/LB:R0EXSY:R0AUTS:R0IMPA
```

This can be done by either adding it to the task builder command string or by adding @LB:RMSSLX in the users's ODL file.

If the task requires global definitions of the user-visible RMS-11 symbols, the following should also be included: LB:RMSLIB/LB:RMSSYM

To include remote access (DAP) support while also using RMSRES as a supervisor-mode library, several options are available. Use the module R0AUTS for task resident DAP support. For resident library DAP support, use the module R0AULS and specify DAPRES as a LIBR or CLSTR option in the task builder command sequence. For overlaid DAP support, use the module R0AUOS. The following example includes DAP using the resident library support: LB:RMSDAP/LB:R0AULS

This can be done either by adding it to the task builder command string or by adding @LB:DAPSLX in the users's ODL file.

If inconsistencies are found in the modules at execution time, a BPT trap will be generated and the value 175744 (the error code ER$LIB) will be in R0. This can happen if not all segments of the library are installed or if the version numbers of one or more segments do not match the root segment, the RMSDAP code, or the task itself.

9.15 Map Supervisor D-Space

The Map Supervisor D-Space directive allows the issuing task to change the mapping of its supervisor-mode D-space APRs. This directive also provides information about the current mapping of the tasks supervisor-mode D-space APRs and about the library's current mode.

When supervisor-mode library code is executing, the supervisor-mode I-space APRs map supervisor-mode instruction space. However, the supervisor mode D-space APRs normally map the user mode D-space. Code that resides in a supervisor-mode library may need to use its instruction space as data space as well for such items as error messages. The Map Supervisor Data Space directive allows such code to over map the supervisor D-space and the supervisor I-space on an APR by APR basis. This over mapping is the case even for tasks built as separate I- and D-space in the user space.

The Map Supervisor D-space directive allows the task to specify a 7-bit mask that determines which space each supervisor D-space APR will use. The mask value contains one bit for each APR beginning at APR1. Supervisor-mode D-APR0 must over map user mode D-APR0 at all times and, therefore, is not included in the mask value.

On the return from the MSDS$ call, the current supervisor-mode D-space mapping mask is returned along with the high byte (mode bits) of the PSW. If bit 15 of the mask value is set to 1 on the call no change occurs to supervisor mapping but existing mapping is returned.

The MSDS$ call is designed to be made from the library whose mapping it is intended to alter. This can allow a library to determine for itself which mode it is in from the PSW mode bits returned. Further, if the MSDS$ call is issued from user mode rather than supervisor mode, no mapping change occurs; however, the mapping data and PSW are returned. Note that the MSDS$ call returns ONLY the supervisor-mode mapping status; if the library is in user mode, no useful mapping data is returned. Normally, a library in user mode will have its D-space already mapped to user mode D-space unless the task builder or the program has taken some overt action to remap the D-space.

Digital recommends that all required data be placed at the highest addresses in the library. Not only does this remove it from the APR0 restriction in supervisor mode, but also allows the library to make proper use of the EXTM$ directive. (See *RSTS/E System Directives Manual* if used in the user mode.)

Fortran Call

Not supported.

Macro Call

MSDS$ mask

Macro Expansion

```
.MACRO   MSDS$     mask
.BYTE    201.,2              ;MSDS$ DIC code, DPB size= 2 words
.WORD    mask
.ENDM
```

Parameters

A 7-bit APR mask plus the following action flag information:

bit 15	Flag: if =0, then set mapping; if =1, then report current supervisor-mode D-space mapping (no changes); if issued from user mode, value forced to =1
bit 14	APR 7 mapping switch
bit 13	APR 6 mapping switch
bit 12	APR 5 mapping switch
bit 11	APR 4 mapping switch
bit 10	APR 3 mapping switch
bit 9	APR 2 mapping switch
bit 8	APR 1 mapping switch
bits 0-7	Zero on input, set to PSW high byte on return

Note that if bits 8 through 14 equal 0, super D-space = user D-space. If they equal 1, super D-space = super I-space.

Error Codes

IE.SDP	DIC or DPB size invalid

Example

The following code could exist in a supervisor library to print error messages:

```
TST     (R0)            ;test some piece of user data
BPL     DATAOK          ;skip next if data ok

MSDS$S  #100000         ;get current mapping state
MOV     $DSW,R0         ;
MOV     R0,-(SP)        ;save for future restoration

BIS     #400,R0         ;update mask to map APR1 to super
MSDS$S  R0              ;change the mapping

MOV     #MESSAG,R1      ;pointer to error message which is data
CALL    OUTMSG          ;call a routine in super mode to output
                        ;text data message.
MOV     (SP)+,R0        ;get back the original mapping state
MSDS$S  R0              ;now back to the original state

..etc...
```

This is created somewhere in APR 1 of the supervisor library:

```
MESSAG:         .ASCIZ  /Error in data/
```

Part IV
Reference Section

Chapter 10

Task Builder Command Line Format

10.1 Running the Task Builder

To run the Task Builder, type:

RUN $TKB

Or, if the system manager has installed TKB as a concise command language
(CCL) command, you can type:

TKB

The Task Builder responds with the prompt TKB> and you type a command. If
TKB has been installed as a CCL command, you can type TKB and the command
on the same line:

TKB command

10.1.1 Command Line

The Task Builder produces up to three files as output from its analysis of the
object files you specify as input. The general form of the command is:

task-file,map-file,symbol-file=input-file,...,input-file

task-file	The file specification you give to the executable file produced by the Task Builder. If you do not want this file produced, type the comma delimiter. If you leave off the file type from the file specification, the Task Builder supplies a default type of .TSK.
map-file	The file specification you give to the memory map file produced by the Task Builder. If you do not want this file produced, type the comma delimiter. If you leave off the file type from the file specification, the Task Builder supplies a default type of .MAP.
symbol-file	The file specification you give to the symbol-table file produced by the Task Builder. If you do not want this file produced, simply leave out the file specification. If you leave off the file type from the file specification, the Task Builder supplies a default type of .STB.

input-files   The input to the Task Builder. For a simple (nonoverlaid) build, these are the object files produced from the assembly or compilation of your program and subroutines, plus disk library files containing subroutines needed to complete the program.

You signify disk library files by appending the switch /LB to the file specification. This notifies the Task Builder that the file named is a library to be searched. The Task Builder searches the library for any unresolved references in the object files appearing to the left of the library file in the command line.

If you do not specify file types, the Task Builder assumes a default type of .OBJ for object files and a default type of .OLB for object libraries.

For an overlaid build, the input file is an ODL file, signified with a /MP switch. The Task Builder assumes a default file type of .ODL for files with the /MP switch.

If you give a device designator or a project-programmer number in a file specification in the input list (to the right of the equal sign), they apply to all file specifications to the right in the list that do not have a device designator or a project-programmer number.

For a build using MACRO object programs, for example, a suitable command line is:

```
TKB EXE1,EXE1,EXE1=OBJ1,OBJ2,LB:RMSLIB/LB
```

The Task Builder constructs the executable file EXE1.TSK, the map file EXE1.MAP and the symbol table file EXE1.STB from the files OBJ1.OBJ, OBJ2.OBJ, and relevant modules from the library RMSLIB.OLB. (The relevant modules are those referenced in your program. You may have referred to them in source statements, or the MAC assembler may have translated source statements into calls referring to this library.)

To omit the map file, type:

```
TKB EXE1,,EXE1=OBJ1,OBJ2,LB:RMSLIB/LB
```

To produce only the executable file, type:

```
TKB EXE1=OBJ1,OBJ2,LB:RMSLIB/LB
```

To produce no output files, type:

```
TKB=OBJ1,OBJ2,LB:RMSLIB/LB
```

The example above is useful if you are running the Task Builder only to see error messages; that is, for a diagnostic run. Note how project-programmer numbers and device designators work when given for a file specification:

```
TKB=OBJ1,[2,243]OBJ2,OBJ3,LB:RMSLIB/LB,MYLIB/LB
```

For this command, the Task Builder would search for the file OBJ1.OBJ in the user's account. It would attempt to find the files OBJ2.OBJ and OBJ3.OBJ on the public disk structure in the account [2,243]. The project-programmer number also applies to the libraries. That is, the Task Builder would look on the system library disk for a file RMSLIB.OLB under the account [2,243]. Likewise, since the device name LB: also applies to MYLIB, the Task Builder would look on the system library disk in account [2,243] for the library file MYLIB.OLB.

If you do not want this to happen, you must respecify the project-programmer number and device that you want to apply to remaining files. The simplest way to accomplish this is to assign a logical name to the account [2,243] and use the system-wide logical SY: to "go back to" your account on the public disk structure. For example:

```
ASSIGN SY:[2,243] JOHN
```

```
Ready
```

```
TKB=JOHN:FILE1,SY:FILE2,FILE3,LB:RMSLIB/LB,SY:MYLIB/LB
```

This can also be accomplished using multiple command lines, as shown in the following section.

10.1.2 Multiline Command

Because you can specify any number of input files to the Task Builder, it is sometimes necessary to enter a command on more than one line.

If you run the Task Builder such that it prompts with TKB>, it continues prompting for input until it receives a line consisting only of two slash characters (//). For example:

```
RUN $TKB
TKB>IMG1,IMG1,IMG1=SY:[2,243]FILE1
TKB>FILE2,FILE3,LB:RMSLIB/LB
TKB>MYLIB/LB
TKB>//
```

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

```
TKB IMG1,IMG1,IMG1=JOHN:FILE1,SY:FILE2,FILE3,LB:RMSLIB/LB,SY:MYLIB/LB
```

In addition, it produces all three output files.

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

10.2 Options

You may need to specify options to build a particular program. An option modifies the action taking place during the build. To include options, you must use the multiline format. If you specify a line consisting of a single slash (/), the Task Builder assumes that the last input file has been entered and prompts for options by displaying "ENTER OPTIONS:" and another TKB> prompt. You then enter the options you want and terminate the build with the double slash. For example:

```
RUN $TKB
TKB>command
TKB>continued-command
TKB>/
ENTER OPTIONS:
TKB>option
TKB>//
```

10.3 Multiple Builds in One Run

If you want to build more than one program, you can use the single slash after typing options for the preceding program. The Task Builder stops accepting input, builds the program, and then requests information for the next build. For example:

```
RUN $TKB
TKB>IMG1=IN1,IN2,IN3
TKB>/
ENTER OPTIONS:
TKB>UNITS=4
TKB>ASG=SY:0:1,MT0:3,KB:4
TKB>COMMON=JRNAL:RO
TKB>/
TKB>IMG2=SUB1
TKB>//
```

The Task Builder accepts the input for the first build; it then stops accepting input when you type the single slash after the COMMON option. The Task Builder builds IMG1.TSK and then prints TKB> to accept the input for building IMG2.TSK.

10.4 Indirect Command Files

The descriptions of Task Builder commands, up to this point, assume that you are entering them from the keyboard. You can also create indirect command files containing Task Builder commands that you want executed. Later, when you run the Task Builder, you type an at sign character (@) followed by the name of the indirect command file. This capability is very useful if you repeat the same build operation often.

For example, you can use a text editor to create a file called AFIL.CMD, which contains:

```
IMG1,IMG1=IN1,IN2,IN3
/
UNITS=4
ASG=SY:0:1,MT0:3,KB:4
COMMON=JRNAL:RO
//
```

Later, you can type:

```
RUN $TKB
TKB>@AFIL
TKB>
```

Or, if TKB is installed as a CCL command, you can type:

```
TKB @AFIL
```

When the Task Builder finds a line consisting of two slashes, it stops processing the indirect command file, builds the program, and exits.

When the Task Builder finds a single slash on a line, and the slash is the last character in the file, the Task Builder displays a prompt for input and lets you finish the command from the terminal. For example, suppose the file AFIL.CMD in the last example is changed to read:

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

You run the command file as usual. The Task Builder accepts the command file input, and displays the prompt for options:

```
RUN $TKB
TKB>@AFIL
ENTER OPTIONS:
TKB>
```

From this point, processing is as usual for keyboard input.

Using a single slash after options in indirect command files is a handy way to return control to your terminal between successive builds. For example, suppose you create two indirect command files. The first, AFIL.CMD, contains:

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

The second, AFIL2.CMD, contains:

```
IMG2,IMG2=IN4,IN5,IN6
/
LIBR=RMSRES
//
```

The terminal interaction to build these two programs is:

```
RUN $TKB
TKB>@AFIL
TKB>@AFIL2
```

Note that you cannot use the CCL form to run the Task Builder to enter two indirect command files. You must use the multiline format.

You can use an indirect command file reference within an indirect command file. The Task Builder allows two levels of indirection. For example, you could put standard options in an indirect command file, and refer to that file from another command file. Suppose the file AFIL.CMD contains:

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

You must put the indirect file reference on a separate line. Now, suppose the file BFIL.CMD contains:

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

To build using these files, you type:

```
RUN $TKB
TKB>@AFIL
```

Note that you can also use an indirect command file to enter options only. For example:

```
RUN $TKB
TKB> IMG1=IN1,IN2,IN3
TKB> /
TKB> @OPTIONS
```

10.5 Comments in Lines

You can put comments anywhere in the command sequence. You begin a comment with a semicolon (;) and terminate it with a carriage return. For example, you could add comments to the indirect command file in the previous section as follows:

```
;
;BUILD 32T
;
;THE OUTPUT FILES ARE
;
IMG1,IMG2=
;
;THE INPUT FILES ARE
;
IN1,IN2,IN3
;
;
;OPTIONS ARE
;
/
COMMON=JRNAL:RO    ;RATE TABLES
;
//
```

10.6 File Specifications

You use the standard RSTS/E conventions for file specifications. In general, the format is:

device:[ppn]filename.type/sw1/sw2.../swn

If you do not specify a device, the public disk structure is assumed. The default for project-programmer number is your account. The exception is when you specify a device or project-programmer number for a file in a list of files. Such a device designator or project-programmer number "sticks" to all file specifications to the right. For example, consider the following input list:

```
TKB =OBJ1,[2,243]OBJ2,OBJ3,LB:RMSLIB/LB
```

The Task Builder looks for the file OBJ1.OBJ in the user's account. It looks for the files OBJ2.OBJ and OBJ3.OBJ on the public disk structure in account [2,243]. It looks for the file RMSLIB.OLB on the system library disk in account [2,243].

The default for file type depends on the switch you apply to the file specification. If no switches are used, the defaults are:

```
file.TSK,file.MAP,file.STB=file.OBJ,...,file.OBJ
```

Defaults assumed when you use various switches are described in Chapter 11. For example, the default file type when you use the /LB switch is .OLB.

Task Builder Switches

The Task Builder lets you modify the action taken on a file by appending a switch to the file specification. A switch is a slash (/) followed by a two- to four-character code. In general, you can precede the two- to four-character code with a minus sign (-) or the letters "NO", and the Task Builder negates the function of the code. For example, the Task Builder recognizes the following settings for the switch /MP:

/MP The file is an ODL file.
/-MP The file is not an ODL file.
/NOMP The file is not an ODL file.

The Task Builder assumes a default setting for each switch. For example, if you do not specify any setting for the /MP switch, the Task Builder assumes /-MP (that the file is not an ODL file). In the switch descriptions in this chapter, note that the "Syntax" section shows where the switch is placed by using the opposite of the default setting. (There is no need to specify a switch if you want to use the default.)

Table 11–1 lists the Task Builder switches available on RSTS/E systems. They are described in following subsections in alphabetical order.

Table 11–1: Task Builder Switches

Switch	Meaning	Applies to File	Default
/CC	Input file consists of concatenated programs or subprograms.	.OBJ	/CC
/CO	Causes the Task Builder to build a shared common.	.TSK, .STB	/CO
/DA	Executable program contains a debugging aid.	.TSK, .OBJ	/-DA
/DL	Specified library file is a replacement for the default system library.	.OLB	/-DL
/EL	Extend library	.TSK	/-EL
/FM	Enables the Fast Map feature of the executive.	.MAP	/-FM
/FO	Indicates that the memory resident overlay should use the Fast Map version.	.MAP	/FO
/FP	Program uses Floating-Point processor.	.TSK	/FP

(continued on next page)

Table 11–1 (Cont.): Task Builder Switches

Switch	Meaning	Applies to File	Default
/FU	All co-tree overlay segments are searched for matching definition or reference when subroutines from the default system library are processed.	.TSK	/-FU
/HD	Task file (executable program) includes a header.	.TSK, .STB	/HD
/ID	Creates I-and D-space tasks.	.TSK	/-ID
/LB	Input file is a library file.	.OLB	/-LB
/LI	Informs the Task Builder to build a shared library.	.TSK, .STB	/-LI
/MA	Memory map file contains information about the file.	.MAP, .OBJ	†
/MP	Input file is an ODL (memory map) file.	.ODL	/-MP
/MU	Program is a multiuser program.	.TSK	/-MU
/NM	No diagnostic messages on screen.	.TSK	/-NM
/PI	Resident area is position-independent.	.TSK, .STB	/-PI
/PM	Post-mortem dump requested.	.TSK	/-PM
/RO	Memory-resident overlay operator (!) is enabled.	.TSK	/RO
/SB	Causes the task to be built with the slow mode.	.TSK, .OBJ	/-SB
/SG	Allocates task program sections alphabetically by access code (RW followed by RO).	.TSK	/SG
/SH	Short memory-map file is produced.	.MAP	/SH
/SP	Spool map file to line printer.	.MAP	/SP
/SQ	Program sections are allocated sequentially, rather than alphabetically.	.TSK	/-SQ
/SS	Selective search for global symbols.	.OBJ	/-SS
/TR	Executable program is to be traced.	.TSK	/-TR
/WI	Memory map file is printed at width of 132 characters (for /-WI, 80 characters).	.MAP	/WI
/XT:n	Task Builder exits after n diagnostics.	.TSK	/-XT

†The default is /MA for an input file, and /-MA for system and resident area symbol table (.STB) files.

11.1 /CC—Concatenated Programs and Subprograms

File

Input

Syntax

file.TSK=file.OBJ/-CC

Description

This switch controls the way the Task Builder extracts programs and subprograms from your input file. Your input file can contain more than one program or subprogram. One way to achieve this is by concatenating more than one object module.

By default, the Task Builder includes all the programs and subprograms in your input file when it builds the executable program file. If you negate this switch (as in the "Syntax" section above), the Task Builder includes only the first program or subprogram of your input file.

This switch will not affect library files. If you try to use /CC and /LB, or /-CC and /LB, in an attempt to limit or expand the Task Builder's normal processing of libraries, the /LB simply overrides the /CC or /-CC.

Default

/CC

Example

```
RUN $TKB
TKB> FIRST1=BUNCH/-CC,LB:F4POTS/LB
TKB> //
```

11.2 /CO—Build a Common Block Shared Region

File

Task image
.STB file

Syntax

file.TSK/CO=file.OBJ

or

,,file.STB/CO=file.OBJ

Description

The /CO switch informs the Task Builder that a shared common is being built. If you build a shared common, you should use the /CO switch and the /-HD switch.

If you use the /-PI switch for an absolute shared common, all the program sections in the common are marked absolute. Using the /-PI/-HD switches without the /CO switch causes the Task Builder to build a shared library.

If you use the /PI switch for a relocatable shared common, all program sections in the common are marked relocatable.

In either case, the .STB file contains all the program section names, attributes, lengths, and symbols. The Task Builder links common blocks by program sections. Therefore, the .STB file of a shared region built with the /CO switch contains all defined program sections.

Using the /PI/-HD switches without the /CO switch causes the Task Builder to build a shared common.

The /CO switch does not have a /-CO form.

Effect

This switch causes the Task Builder to include all program section declarations in the .STB file.

Defaults

/CO

Example

```
RUN $TKB
TKB>VAL/CO/-HD=VAL.OBJ
TKB>//
```

> **NOTE**
>
> Commons (read/write libraries) must still be processed using the MAKSIL utility. See the *RSTS/E Programmer's Utilities Manual* for more details about MAKSIL.

11.3 /DA—Debugging Aid

File

Executable program file or input file

Syntax

file.TSK/DA=file.OBJ

or

file.TSK=file.OBJ,file.OBJ/DA

Description

If you use the /DA switch on the executable program file, the Task Builder automatically includes the system debugging aid LB:ODT.OBJ in the executable program (LB:ODTID.OBJ if /ID is also included).

If you use this switch on one of your input files, the Task Builder assumes that the file is a debugging aid that you have written.

In either case, /DA has the following effects:

1. The transfer address of the debugging aid overrides the executable program transfer address.

2. The Task Builder initializes the header of the program so that, when your program is loaded, register R0 through R4 contain the following values:

   R0   Transfer address of program.

   R1   Task name in Radix-50 format (word 1). The Task Builder derives this name from the TASK=option. If no TASK= is supplied, this value will be 0.

   R2   Second word of task name.

   R3   The first three of six RAD50 characters representing the version number of your program. The Task Builder derives this number from the first .IDENT directive it encounters in your program. If no .IDENT directives appear, this value will be 0.

   R4   The second three RAD50 characters representing the version number of your program.

Refer to your specific language reference manual for more information about debugging aids.

Default

/-DA

Example

```
RUN $TKB
TKB> PROG/DA=OBJ,OBJ2,LB:F4POTS/LB
TKB> //
```

11.4 /DL—Default Library

File

Input

Syntax

file.TSK=file.OBJ,file.OLB/DL

Description

The library file you specify replaces the file LB:SYSLIB.OLB as the library file that the Task Builder searches to resolve undefined global references. This file is searched only when undefined symbols remain after all the files you specify have been processed. The /DL switch can be used with only one input file.

Default

/-DL

Example

```
RUN $TKB
TKB> PROG=PROG,LB:F4POTS/LB,NEWLIB/DL
TKB> //
```

11.5 /EL—Extend Library

File

Executable program file

Syntax

file.TSK/LI/-HD/EL=file.OBJ

Description

The /EL switch places the upper-address limit as determined by the PAR option in the library's label block, though the actual size of the library may be smaller. This switch is useful when building vectored libraries subject to size changes, such as RMS.

The switch specifies the maximum possible size for the library according to the size specified in the PAR option. The switch specifies a larger library virtual address range than is actually present in the library to allow RMS to map its vectored library segments.

Default

/-EL

11.6 /FM—Fast Map

File

Memory allocation (map) file

Syntax

file.TSK/FM=file.obj

Description

The /FM quailfier tells TKB to set the proper bits in the task header to enable the Fast Map feature of the executive.

Default

/-FM

Example

```
RUN $TKB
TKB>PROG/FM=OBJ1
TKB>//
```

11.7 /FO—Fast Map Overlay

File

Memory allocation (map) file

Syntax

file.TSK/FO/FM=file.obj

Description

The /FO switch indicates that the overlay handler for memory resident overlay should use the Fast Map version. This version is faster, but it does take more space. Use the /FO switch in conjunction with the /FM switch so that RSTS/E will use the IOT instruction for Fast Map rather than for application use for this task.

Default

/FO

Example

```
RUN $TKB
TKB> PROG/FO/FM=OVERLY/MP
ENTER OPTIONS:
TKB> //
```

11.8 /FP—Floating Point

File

Executable program file

Syntax

file.TSK/FP=file.OBJ

Description

Setting the /FP switch causes the RSTS/E monitor to save the state of the
floating-point processor when the program is run. You must set this switch
on systems that have the floating-point processor (if the program uses the
floating-point processor), so that the run-time system can trap floating-point
errors properly. Setting or negating this switch has no effect on systems without
a floating-point processor.

Default

/FP

Example

```
RUN $TKB
TKB>PROG/FP=OBJ1,OBJ2,LB:F4POTS/LB
TKB>//
```

11.9 /FU—Full Search

File

Executable program file

Syntax

file.TSK/FU=file.ODL/MP

Description

The /FU switch affects how the Task Builder inserts code from the default
library when your overlay structure has co-trees. Normally, when the same
code (program section) is called or referenced from different co-trees, it is built
into both co-trees unless it can be resolved from code already built into the main
root. This prevents the problem of run-time errors caused by unintentionally
displacing segments with cross-tree calls, as described in Chapter 4.

If you use this switch, the Task Builder can resolve undefined global references
with code from the default library that is already built into other co-trees. This
can be useful if you want to try to cut down on the space taken by code inserted
into co-trees from the default library, as described in Section 4.4.8.

Default

/-FU

Example

```
RUN $TKB
TKB>PROG/FU=OVERLY/MP
ENTER OPTIONS:
TKB>//
```

11.10 /HD—Header

File

Executable program file or symbol definition file

Syntax

file.TSK/-HD,,file.STB=file.OBJ

or

file.TSK,,file.STB/-HD=file.OBJ

Description

The /HD switch causes the Task Builder to generate a header for your executable program file. This header is used by the run-time system when it loads your program for execution. The run-time system takes certain values from the header and inserts them in the low 1000 bytes of your program. (This area is used by the RSTS/E monitor, the run-time system, and—with a few languages—your program itself. For example, this area contains the "core common" area accessible to BASIC-PLUS-2 programs and the FIRQB and XRB areas used by MACRO programs. The contents of this area may be of interest to you if you are programming in MACRO. The area is described in the *RSTS/E System Directives Manual*.

In any case, you must have a header for executable program files (this is the default). If you are building a resident library or common, or a run-time system itself, you must negate this switch.

Default

/HD

Example

```
RUN $TKB
TKB> DATLIB/-HD/PI,,DATLIB/PI=DAT1.DAT,DAT2.DAT,DAT3.DAT
TKB> /
ENTER OPTIONS:
TKB> PAR=DATLIB
TKB> //
```

11.11 /ID—I- and D-Space

File

Task image (.TSK).

Syntax

file.TSK/ID=file.OBJ,file.OLB

Description

Use this switch to create I- and D-space tasks. The switch directs the Task
Builder to mark your task as one that uses I-space APRs and D-space APRs in
user mode. The Task Builder separates I-PSECTs from D-PSECTs. See Chapter
8 for more information.

Default

/-ID

Example

```
RUN $TKB
TKB> PROG1.TSK/ID=PROG1.OBJ,PROG1.OLB/LB
TKB> //
```

11.12 /LB—Library File

File

Input file, or any file in an ODL command.

Syntax

file.TSK=file.OBJ,file.OLB/LB

or

file.TSK=file.OBJ,file.OLB/LB:mod-1:mod-2:...:mod-8

or

file.OBJ=file.OBJ,file.OLB/LB:mod-1:mod-2,file.OLB/LB

or

(Any of the above forms in an ODL file)

Description

If you use the /LB switch, it indicates that the file is a library file. The Task Builder's interpretation depends upon the form you use. If you use the switch without arguments, the Task Builder assumes that your input file is a library file of relocatable object routines. The Task Builder searches the file to resolve undefined references in any files you have specified preceding the library specification. It extracts necessary routines (which contain definitions for undefined references) and includes them in your executable program file.

If you use the switch with the mod-i arguments (mod-1:mod-2:...and so forth), the Task Builder extracts from the library the routines named as arguments regardless of whether or not they contain definitions for unresolved references.

If you want the Task Builder to search a library both to resolve global references and to select named routines, you must name the library twice: once, with the routines named (/LB switch with modifiers) and a second time with the general form (/LB switch without modifiers).

The position of the library file in the command line is important. The following rules apply:

1. The library file must appear to the right of the input file(s) that contain references to be resolved from the library. For example:

    ```
    TKB>file.TSK=infile1.OBJ,infile2.OBJ,lib.OLB/LB
    ```

    In this command, unresolved references from infile1.OBJ and infile2.OBJ are resolved from the library.

    In the following command, unresolved references from infile1.OBJ are resolved from the library, but references from infile2.OBJ are not:

    ```
    TKB>file.TSK=infile1.OBJ,lib.OLB/LB,infile2.OBJ
    ```

2. When you are building an overlay structure, you specify the library within the ODL file. You use the hyphen to indicate concatenation; unresolved references from the segment to the left of the hyphen are resolved from the library specified to the right. For example:

```
            .ROOT    AFCTR-(BFCTR,CFCTR)
AFCTR:      .FCTR    A-LIBR
BFCTR:      .FCTR    B-LIBR-(B1-LIBR,B2-LIBR)
CFCTR:      .FCTR    C-C1-LIBR-(C01-LIBR,C02-LIBR)
LIBR:       .FCTR    LB:F4POTS/LB
            .END
```

Notice that in this example, there is no -LIBR entry after C. Since C and C1 are constructed as one segment, putting a -LIBR entry after C would only cause an unecessary and time-consuming search. Only one search is needed for each segment; you can place the -LIBR entry at the end of the segment, after C1. Section 3.7.3 explains how routines are inserted into segments from libraries.

Default

/-LB

Example

See the Description section above.

11.13   /LI—Build a Library Shared Region

File

Task Image
.STB file

Syntax

file.TSK/LI[:apr bit mask]

or

,,file.STB/LI[:apr bit mask]

Description

The /LI switch makes the Task Builder build a shared library. However, you must use the /-HD switch with the /LI switch to build the shared library. The /LI switch does not have a /-LI form.

See the discussion on APR masks in Chapter 7.

Effect

The Task Builder includes only one program section declaration in the .STB file.

If you use the /-PI switch for an absolute library, the Task Builder names the program section .ABS, makes the library position dependent, and defines all symbols as absolute. Also, if you use the /-PI switch without the /LI switch, the Task Builder assumes /LI to be the default.

If you use the /PI switch for a relocatable library, the Task Builder names the program section the same as the root segment of the library. The Task Builder forces this name to be the first and only declared program section in the library. The Task Builder declares all global symbols in the .STB file relative to that program section. Also, if you use the /PI switch without the /LI switch, the Task Builder assumes that a shared common is to be built. (/CO is the default.)

Default

/LI [:nnn]

where:

n   is the bit mask representing the APR in I-space used by the library

Example

```
RUN $TKB
TKB>PARODI/LI/-HD=PARODI.OBJ
TKB>//
```

NOTE

Libraries must still be processed using the MAKSIL utility. See the *RSTS/E Programmer's Utilities Manual* for more details about MAKSIL.

11.14 /MA—Map Contents of File

File

Input or memory allocation (map) file

Syntax

file.TSK,file.MAP=file.OBJ,file.OBJ/-MA

or

file.TSK,file.MAP/MA=file.OBJ

Description

If you negate this switch and apply it to an input file, the Task Builder leaves the file off the "file contents" portion of the memory map. Furthermore, it will exclude from the map all global symbols defined or referred to in the file.

If you set this switch for the map file, the Task Builder includes in the map the names of routines it has added to your program from the default library (LB:SYSLIB.OLB). It also includes in the map file information contained in the symbol definition file of any shared region referred to by the program.

Default

/MA for input files

/-MA for system library and resident library .STB files.

/-MA for map file

Example

```
RUN $TKB
TKB>PROG,PROG/MA=OBJ1,OBJ2,LB:F4POTS/LB
TKB>//
```

11.15 /MP—Overlay Map

File

Input

Syntax

file.TSK=file.ODL/MP

Description

Your input file is an overlay map (ODL file). The file contains directions for an overlay structure in the Overlay Description Language. When you use the switch, it must be the only input file that you specify. The default file type for a file with the /MP switch is .ODL.

Default

/-MP

Example

```
RUN $TKB
TKB>PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB>//
```

NOTE

If you use the multiline command format when you specify an ODL file, TKB automatically prompts for option input. Therefore, you must not use the single slash (/) to direct TKB to switch to option input mode when you have specified /MP on your input file.

11.16 /MU—Multiuser Program

File

Executable program file

Syntax

file.TSK/MU=file.OBJ

Description

The /MU switch tells the Task Builder to separate the program's read-only and read/write program sections. On RSTS/E systems, you only use this switch if you want to build a program so that the read-only code from the *root* is accessible to multiple users. For this reason, it is recommended that programs built with the /MU switch be nonoverlaid. Several steps are involved in this procedure.

When you use /MU on the executable file, the Task Builder places the read-only sections in your program's upper virtual address space and the read/write program sections in your program's lower virtual address space. You then have to use the MAKSIL program (described in the *RSTS/E Programmer's Utilities Manual* to make the read-only code accessible to multiple users, and add the read-only code as a resident area using the DCL INSTALL/LIBRARY command (described in the *RSTS/E System Manager's Guide*).

Multiple users can then run the program built, causing multiple copies of the read/write code to be executed, but with only one copy of the read-only code taking space in memory.

Note that a program built with the /MU switch cannot be run correctly until it has been converted by MAKSIL into a separate executable file (consisting of the read/write code) and a resident area, which in turn must be added with the DCL INSTALL/LIBRARY command (just like any resident area). Note also that, when you build a program and use the /MU switch, you can also use the HISEG option. If you build with HISEG, the Task Builder will put the read-only code to occupy virtual address space below the run-time system. If you do not build with HISEG, the Task Builder will put the read-only code so that it occupies the highest possible address space (using APR 7).

Default

/-MU

Example

```
RUN $TKB
TKB> PROG/MU=OBJ1,OBJ2,OBJ3
TKB> //
```

11.17 /NM—No Diagnostic Messages

File

Executable program file

Syntax

file.TSK/NM=file.OBJ

Description

Using the /NM switch eliminates the display of diagnostic messages from a build.

Default

/-NM

Example

```
RUN $TKB
TKB>PROG/NM=OBJ1,OBJ2,LB:F4POTS/LB
TKB>//
```

11.18 /PI—Position Independent

File

Executable program file or symbol definition

Syntax

file.TSK/PI=file.OBJ

or

file.TSK,,file.STB/PI=file.OBJ

Description

Use the /PI switch when you are building a resident area that is position independent, that is, a region that can be placed anywhere in the program's address space. (The other option is an absolute resident area, which is fixed in the program's address space.) See Section 7.3.1 for a discussion of position-independent resident areas.

Default

/-PI

Example

```
RUN $TKB
TKB> DATLIB/-HD/PI,,DATLIB=DAT1.DAT,DAT2.DAT,DAT3.DAT
TKB> /
ENTER OPTIONS:
TKB> STACK=0
TKB> PAR=DATLIB
TKB> //
```

11.19 /PM—Post-Mortem Dump

File

Executable program file

Syntax

file.TSK/PM=file.OBJ

Description

Setting the /PM switch causes the Task Builder to set an indicator in your executable program file. If your program terminates abnormally when it is executed, the indicator causes the system to write automatically the contents of the program in memory on a disk file. The file name for the created file is:

PMDnnn.PMD

where: nnn is your job number.

The file must be formatted by the PMDUMP program (see the *RSTS/E Utilities Reference Manual*) before you can read it.

Default

/-PM

Example

```
RUN $TKB
TKB> PROG/PM=OBJ1,OBJ2,LB:F4POTS/LB
TKB> //
```

11.20 /RO—Resident Overlay

File

Executable program file

Syntax

file.TSK/-RO=file.ODL/MP

Description

When you use /RO, the Task Builder processes any memory-resident overlay operators (!) in your ODL file. That is, the Task Builder uses the exclamation point operator to construct an executable program file that contains one or more memory-resident overlay segments.

If you negate this switch, the Task Builder checks the syntax of the exclamation point where it appears in the ODL commands but does not construct memory-resident overlay segments.

Default

/RO

Example

```
RUN $TKB
TKB> PROG/-RO=OVERLY/MP
ENTER OPTIONS:
TKB> //
```

11.21 /SB—Slow Build

File

Executable program file

Syntax

file.TSK/SB=file.obj

Description

The /SB qualifier causes the task to be built with the slow mode of the Task Builder. This option increases the amount on TKB internal storage and therefore allows you to build larger or more complex tasks.

Default

/-SB

Example

```
RUN $TKB
TKB>PROG/SB=OVERLY/MP
ENTER OPTIONS:
TKB//
```

11.22 /SG—Segregate Program Sections

File

Task image

Syntax

file.TSK/SG=file.OBJ

Description

The /SG switch allocates virtual address space to all read/write (RW) program sections and then to all read-only (RO) program sections.

Effect

The /SG switch gives you control over the ordering of program sections. By using the /SG switch, you cause the Task Builder to order program sections alphabetically by name within access code (RW followed by RO). If you specify the /SQ switch with the /SG switch, the Task Builder orders program sections in their input order by access code. (See the description of the /SQ switch for more information.)

You use the negated switch, /-SG, to make the Task Builder interleave the RW and RO program sections. Thus, the combination /-SG/SQ results in a task with its program sections allocated in input order and its RW and RO sections interleaved. Also, you can use /-SQ/-SG to make the Task Builder order program sections alphabetically with RW and RO sections interleaved. However, /SG is the default.

When task building multiuser tasks, the /MU switch causes the Task Builder to default to /SG. Therefore, to correctly build read-only tasks, you can use the /MU switch only.

Default

/SG

Example

```
RUN $TKB
TKB> BARBEL/SG=BARBEL.OBJ
TKB> //
```

11.23 /SH—Short Map

File

Memory allocation (map) file

Syntax

file.TSK,file.MAP/-SH=file.OBJ

Description

Negating this switch (-SH) requests the long version of the memory allocation map. The Task Builder produces the "file contents" section of the map. An example of the long version of the map is shown in Example 11–1. The letters in brackets and the numbers in cirlces in the figure correspond to the notes following the figure.

Default

/SH

Example

```
RUN $TKB
TKB> PROG,PROG/-SH=OBJ1,OBJ2,LB:F4POTS/LB
TKB> //
```

Example 11-1: Memory Allocation (Map) File

```
ROOTM.TSK    Memory allocation map   TKB 08.006      Page 1 ┐  ❶
                      05-MAY-90    13:50                     ┘

Task    name    : ROOTM  [A]                                ┐

Partition name : GEN   [B]
Identification : 01    [C]
Task  UIC       : [2,234]   [D]
Task  priority : 50.   [E]
Stack      limits: 001000 001777 001000 00512.   [F]        ❷
ODT xfr address: 011054   [G]
PRG xfr address: 002000   [H]
Task attributes: DA,MU   [I]
Total address windows: 2.   [J]
Task   extension   : 128. words   [K]
Task  image  size  : 9760. words   [L]
Total  task  size  : 9888. Words   [M]
Task address limits: 000000 046033   [N]
R-W disk blk limits: 000002 000101 000100 00064.   [O]
R-O disk blk limits: 000102 000112 000011 00009.   [P]      ┘

ROOTM.TSK Overlay description:                              ┐

Base    Top       Length
----    ---       ------
000000  016027  016030  07192.       ROOTM
016030  031247  013220  05776.            MULOV            ❸
016030  032043  014014  06156.            ADDOV
032044  046033  013770  06136.                 SUBOV
032044  045667  013624  06036.                 DIVOV        ┘

ROOTM.TSK    Memory allocation map   TKB 08.006      Page 2
ROOTM                   05-MAY-90    13:50

*** Root segment: ROOTM   [A]

R/W mem  limits: 000000 016027 016030 07192.   [B]
R-O mem  limits: 160000 170577 010600 04480.   [C]
Disk blk limits: 000002 000020 000017 00015.   [D]

Memory allocation synopsis:                                       ❹

Section                                    Title Ident  File
-------                                    ----- -----  ----
. BLK.:(RW,I,LCL,REL,CON)     002000 000000 00000.   [E]
CODE   :(RW,I,LCL,REL,CON)    002000 000024 00020.
                              002000 000024 00020. .MAIN.01 ROOTM.OBJ   [F]
                                 .
                                 .
                                 .
Global symbols:

ADD 170076-R  DATEND  010010-R DAT1  002024-R MUL  170066-R .ODTL1 010434-R  [G]
BEG 002000-R  DAT0    160000-R  DIV  170116-R SUB  170106-R .ODTL2 010436-R


File: ROOTM.OBJ  Title: .MAIN.   Ident: 01    [H]
   <. ABS.>: 000000 000000 000000 00000.   [I]
 >>>>>>>>>>> Undefined reference: NOSYMB   [J]
```

Example 11–1 (Cont.):   Memory Allocation (Map) File

```
     <CODE  >: 002000 002023 000024 00020.
        BEG    002000-R   [K]
     <DATA  >: 160000 170041 010042 04130.    [L]
                          .
                          .
                          .

 ************

 Undefined references:   [M]

     NOSYMB

 ROOTM.TSK   Memory allocation map   TKB 08.006     Page 4
 MULOV                    05-MAY-90   13:50

 *** Segment: MULOV

 R/W mem  limits: 016030 031247 013220 05776.
 Disk blk limits: 000021 000034 000014 00012.

 Memory allocation synopsis:

 Section                                       Title  Ident  File
 -------                                       -----  -----  ----
 . BLK.:(RW,I,LCL,REL,CON)      016030 013220 05776.
                                016030 013220 05776. .MAIN.      MULOV.OBJ
 $$ALVC:(RO,I,LCL,REL,CON)      031250 000000 00000.
 $$RTS :(RO,I,GBL,REL,OVR)      170570 000002 00002.

 Global symbols:

 MUL     016030-R

 File: MULOV.OBJ  Title: .MAIN.  Ident:
     <. BLK.>: 016030 031247 013220 05776.
        MUL    016030-R
                          .
                          .
                          .

 *** Task builder statistics:

     Total work file references: 4693.    [A]
     Work   file  reads: 0.    [B]
     Work   file  writes: 0.   [B]
     Size of core pool: 4814. words  (18. pages)   [C]
     Size of work file: 3072. words  (12. pages)   [D]

     Elapsed time:00:00:17    [E]
```

❺

❻

❶ The page header shows the name of the executable program file and the overlay segment name (if applicable), along with the date, time, and version of the Task Builder that created the map.

❷ The task attribute section contains the following information:

   A. Task Name. The name specified in the TASK option. If you do not use the TASK option, the Task Builder suppresses this field.

   B. Partition Name. The partition specified in the PAR option. If you do not specify a partition, the default is a partition named GEN.

   C. Identification. The version as specified in the .IDENT assembler directive. If you do not specify, the default is the same as the version of the Task Builder.

D. User Identification Code. The project-programmer number used to create the executable program file.

E. Priority. (On RSTS/E systems, this field is ignored.) Priority is suppressed if you do not use the PRI= option.

F. Stack Limits. The low and high octal addresses of the stack, followed by its length in octal and decimal bytes.

G. ODT Transfer Address. The starting address of the ODT debugging aid. If you do not specify the ODT debugging aid, this field is suppressed.

H. Program Transfer Address. The starting address of your program. For MACRO programmers, this is the address of the symbol specified in the .END directive of the source code of your program. (The compilers generate a starting address automatically.) If you do not specify a transfer address for your program, the Task Builder automatically establishes a transfer address of 000001 for it. The Task Builder also suppresses this field in the map if no transfer address is specified.

I. Attributes. Using certain switches indicates that your program has certain attributes. Such switch settings are shown only if they differ from the defaults. For example, the following could be displayed:

    DA - the program contains a debugging aid.

    MU - the program is broken into RO and RW sections for processing by MAKSIL. See the description of the /MU switch in Section 11.16 of this manual, and the MAKSIL chapter in the *RSTS/E Programmer's Utilities Manual*, for more information.

J. Total Address Windows. The number of window blocks allocated to the program.

K. Task Extension. The increment of physical memory (in decimal words) allocated through the EXTTSK or PAR option.

L. Task Image Size. The amount of memory (in decimal words) required to contain your program's code. This number does not include physical memory allocated through the EXTTSK option.

M. Total Task Size. The amount of memory (in decimal words) allocated to your program, including the physical memory allocated through the EXTTSK option or PAR option.

N. Task Address Limits. The lowest and highest virtual addresses allocated to the program, exclusive of resident areas.

O. Read/Write Disk Block Limits. From left to right: the first octal relative disk block of the program's read/write region; the last octal relative disk block number of the read/write region; the total contiguous disk blocks required to accomodate the read/write region in octal and decimal.

P. Read-Only Disk Block Limits. From left to right: the first octal relative disk block of the multiuser program's read-only region; the last octal relative disk block number of the read-only region; the total contiguous disk blocks required to accomodate the read-only region in octal and decimal. This field appears only when you are building a multiuser program with the /MU switch.

❸ The Overlay Description shows, for each overlay segment in the tree structure of an overlaid program, the beginning virtual address (the base), the highest virtual address (the top), the length of the segment in octal and decimal bytes, and the segment name. Indenting is used to illustrate the ascending levels in

the overlay structure. The Task Builder prints the Overlay Description only when an overlaid program is created.

❹ The Root Segment Allocation. This section has the following elements:

A. Root Segment. The name of the root segment. If your program has only one segment, the entire program is considered to be the root segment.

B. Read/Write Memory Limits. From left to right, the beginning virtual address of the root segment (the base), the virtual address of the last byte in the segment (the top), the length of the segment in octal and decimal bytes.

C. Read-Only Memory Limits. From left to right, the beginning virtual address of the root segment (the base), the virtual address of the last byte in the segment (the top), the length of the segment in octal and decimal bytes. This field appears only when you are building a multiuser program with the /MU switch.

D. Disk Block Limits. From left to right, the first relative block number of the beginning of the root segment, the last relative block number of the root segment, total number of disk blocks in octal, and the total number of disk blocks in decimal.

E. Memory Allocation Synopsis. From left to right: the program section name, the program section attributes, starting virtual address of the program section, total length of the program section in octal and decimal bytes.

F. Contributor. This field lists the pieces that have contributed to each program section. In this example, the program section ANS was defined in the file ROOTM.OBJ. The identification in this case is 01 as a result of an .IDENT assembler directive. If the program section ANS had been defined in more than one piece (for example, in more than one routine in a library (.OLB) file), each contributing piece and the file from which it was extracted would have been listed here.

G. Global Symbols. This section lists the global symbols defined in the segment. Each symbol is listed along with its octal value. -R is appended to the value if the symbol is relocatable. The list is alphabetized in columns.

The File Contents Section is produced only if you specify the /-SH switch in the Task Builder command sequence. The Task Builder then creates this section for each segment in an overlay structure. It lists the following information:

H. Input File. The file name, the name established by a .TITLE assembler directive, and the version as established by an .IDENT assembler directive.

I. Program Section. Program section name, starting virtual address of the program section, ending virtual address of the program section, and length in octal and decimal bytes.

J. Undefined Reference. This section provides the names of undefined symbols in the preceding program section.

K. Global Symbol. Global symbol names within each program section and their octal values. If the segment is autoloadable (see Chapter 5), this value will be the address of an autoload vector. The autoload vector in turn will contain the address of the symbol. -R is appended to the value if the symbol is relocatable.

L. Program Section. This field is identical to the field described in note I.

The following sections in the map file appear regardless of whether you use the /-SH switch or not.

M. Undefined References. This field lists the undefined global symbols in the segment.

❺ The Tree Segment Description is printed for every overlay segment in an overlay structure. Its contents are the same for each overlay segment as the Root Segment Allocation is for the root segment.

❻ Task Builder Statistics list the following information, which can be used to evaluate Task Builder performance:

A. Work File References. The number of times that the Task Builder accessed data stored in its work file.

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

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

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

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

E. Elapsed Time. The amount of wall-clock time required to construct the executable program and memory allocation (map) file. Elapsed time is measured from when you entered the last option to the completion of map output. This value excludes the time required to process the overlay description and parse the list of input file names.

See Appendix E for a more detailed discussion of the work file.

11.24 /SP—Spool Map Output

File

Memory allocation (map) file

Syntax

file.TSK,file.MAP/SP=file.OBJ

Description

This switch determines whether your map file is automatically queued to the line printer for output. If you use this switch, the Task Builder creates a map file and queues it for printing. The default (if you specify a map file) is to create the map file but not to queue it for printing.

Default

/-SP

Example

```
RUN $TKB
TKB>PROG,PROG/SP=OBJ1,OBJ2,LB:F4POTS/LB
TKB>//
```

11.25 /SQ—Sequential

File

Executable program file

Syntax

file.TSK/SQ=file.OBJ

Description

If you set the /SQ switch, the Task Builder does not reorder program sections alphabetically. Instead, it collects all the references to a given program section from your input files, groups them according to their access code (read-only or read/write), and within these groups, allocates memory for them in the order that you input them.

You use this switch to satisfy requirements that certain program sections be adjacent. Using this feature is otherwise discouraged because standard library routines (such as FORTRAN I/O handling routines and File Control System (FCS) routines from SYSLIB) will not work properly.

You can also make program sections adjacent by selecting their names alphabetically to correspond to the desired order.

Default

/-SQ

Example

```
RUN $TKB
TKB>PROG/SQ=OBJ1,OBJ2,OBJ3
TKB>//
```

11.26 /SS—Selective Search

File

Input file

Syntax

file.TSK=file.OBJ/SS

or

file.TSK=file.OBJ,file.STB/SS

or

file.TSK=file.OBJ,file.OLB/LB/SS

Description

Setting the /SS switch tells the Task Builder to include in its internal symbol table only those global symbols for which it has already encountered an undefined reference.

When processing an input file, the Task Builder normally includes into its internal symbol table each global symbol it encounters within the file whether or not there are references to it. When you attach the /SS switch to an input file, the Task Builder checks each global symbol it encounters within that file against its list of undefined references. If the Task Builder finds a match, it includes the symbol into its symbol table.

Default

/-SS

Example

Suppose that you are building a program consisting of input files containing global entry points and references (calls) to them, as shown in Table 11–2.

Table 11–2: Input Files for /SS Example

Input File Name	Global Definition	Global Reference
IN1.OBJ		A
IN2.OBJ	A	
	B	
	C	
IN3.OBJ		C
IN4.OBJ	A	
	B	
	C	

Files IN2 and IN4 contain definitions for global symbols of the same name. Assume that the global symbols represent entry points to different routines within these files.

Suppose that you want the Task Builder to resolve the reference to A in IN1 with the definition of A in IN2. Further, assume that you want the reference to global symbol C in IN3 to be resolved with the definition of C in IN4. You can accomplish this by ordering the input files and using the /SS switch. For example:

```
TKB>SELECT=IN1,IN2/SS,IN3,IN4/SS
```

The Task Builder processes input files from left to right. Thus, the Task Builder processes file IN1 first and finds the reference to symbol A. Since there is no definition for A within IN1, the Task Builder marks A as undefined and moves on to process IN2. Because IN2 has the /SS switch, the Task Builder limits its search of IN2 to symbols it has already marked as undefined, namely A. The Task Builder finds a definition for A and puts A in its symbol table.

The Task Builder moves on to IN3, and encounters the reference to symbol C. Since the Task Builder did not include symbol C from IN2 in its symbol table, it marks C as undefined and moves on to IN4. When the Task Builder processes IN4, it finds the definition for C, and includes that symbol in the table. Again, since the /SS switch is attached, only symbol C is included in the Task Builder's internal symbol table.

Thus, the reference to A in IN1 is resolved with the definition in IN2, and the reference to C in IN3 is resolved with the definition in IN4. Note that the /SS switch affects only the Task Builder's internal symbol table. The routines for which symbols B and C are entry points will be included in the executable program file even though there are no references to them.

11.27 /TR—Traceable Program

File

Executable program file

Syntax

file.TSK/TR=file.OBJ

Description

When this switch is set, the Task Builder sets the T-bit in the initial program status word (PSW) for your program. When your program is executed, a trace trap occurs when each instruction is completed.

The system library (SYSLIB.OLB) contains a trace routine (TRACE.OBJ) that processes the trap. You must explicitly build this routine into your executable file if you want to use it. To do this, you must use the LBR utility (see the *RSTS/E Programmer's Utilities Manual*) to remove TRACE.OBJ from the system library. You then build TRACE.OBJ into your program using the /DA switch. The example below shows TRACE.OBJ in the user's account on the public structure.

Default

/-TR

Example

```
RUN $TKB
TKB>PROG/TR=OBJ1,OBJ2,OBJ3,TRACE/DA
TKB>//
```

11.28 /WI—Wide Listing Format

File

Memory allocation (map) file

Syntax

file.TSK,file.MAP/-WI=file.OBJ

Description

Negating this switch causes the Task Builder to format the map file 80 columns wide. Setting the switch or accepting the default causes a map 132 columns wide.

Default

/WI

Example

```
RUN $TKB
TKB> PROG,PROG/-WI=OBJ1,OBJ2,LB:F4POTS/LB
TKB> //
```

11.29 /XT[:n]—Exit on Error

File

Executable program file

Syntax

file.TSK/XT:n=file.OBJ

Description

Setting the /XT:n switch causes the Task Builder to exit after it finds n errors. The number of errors can be specified in decimal or octal:

n.   Decimal number (decimal point must be there).

#n or n   Octal number.

If you do not specify n, the Task Builder assumes a value of 1.

Default

/-XT

Example

```
RUN $TKB
TKB>PROG/XT:10.=OBJ1,OBJ2,LB:F4POTS/LB
TKB>//
```

Chapter 12

Task Builder Options

The options you specify to the Task Builder modify the action taken during the build. The options available to RSTS/E users are listed in Table 12–1. Complete descriptions of the options follow, in alphabetical order.

Table 12–1: Task Builder Options

Option	Meaning
ABORT	Terminates command input and allows you to restart the input of command lines.
ABSPAT	Declares absolute patch values.
ACTFIL	Declares number of files that program can have open simultaneously.
ASG	Declares device assignment to logical units, or RSTS/E channels.
CLSTR	Declares a series of resident libraries to be clustered in one space in the user job area.
CMPRT	Declares the completion routine for the supervisor-mode library.
COMMON	Declares a resident common area on LB: to be accessed by the program.
DSPPAT	Declares a series of object-level patches starting at the specified base address.
EXTSCT	Declares extension of a program section.
EXTTSK	Declares extension of the program itself.
FMTBUF	Declares extension of buffer used by FORTRAN for processing format strings at run time.
GBLDEF	Global symbol definition.
GBLINC	Includes a definition for a global symbol in the symbol table (.STB) file.
GBLPAT	Declares a series of object-level patch values relative to a global symbol.
GBLREF	Declares a global symbol reference.
GBLXCL	Excludes a definition for a global symbol from the symbol table (.STB) file.
HISEG	Associates an executable program with a high segment or run-time system.
LIBR	Declares a resident library on LB: to be accessed by the program.
MAXBUF	Declares an extension to the FORTRAN record buffer.
ODTV	Declares the address and size of the debugging aid SST vector.
PAR	Used to build resident area; defines the partition that the resident area is to occupy.
RESCOM	Declares a resident common area to be accessed by the program.

Table 12–1 (Cont.): Task Builder Options

Option	Meaning
RESLIB	Declares a user-owned resident library to be accessed by the program.
RESSUP	Declares that your task intends to access a user-owned supervisor-mode library.
RNDSEG	Rounds the size of a named segment up to the nearest APR boundary while building a resident library.
STACK	Defines the size of the stack.
SUPLIB	Declares that your task intends to access a systemwide supervisor-mode library.
TASK	Names the executable program for SYSTAT.
TSKV	Declares the address of the program's SST vector.
UNITS	Declares the maximum number of units (channels).
VARRAY	Specifies an overlaid virtual array so that each segment of an overlaid task that uses it defines the array in the same way that it is defined in the root segment.
VSECT	Specifies the virtual base address, virtual length, and physical memory allocated to the named program section.
WNDWS	Declares the number of additional address windows to be used by the program.

12.1 ABORT—Abort the Build

The ABORT option is useful when you have made an error on an earlier line of Task Builder input. When you type the ABORT=n in response to a TKB> option prompt, the Task Builder stops accepting input for the current build and prepares to accept input for a new build operation. You can then restart the same or another command sequence.

Syntax

ABORT=n

where:

n   is any integer. (You must specify =n to satisfy the general form of the syntax for options, but the value is ignored.)

Note that typing a Ctrl/Z (pressing the Ctrl and Z keys at the same time) causes the Task Builder to stop accepting input and start building the current program. ABORT is the only way to restart the Task Builder if you find an error and do not want a build to take place.

Default

None

Example

```
 RUN $TKB
TKB> PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB> RESLIB=RMSRES
TKB> ABORT=1
?TKB -- *FATAL* -- TASK BUILD ABORTED VIA REQUEST

ABORT = 1

TKB>
```

12.2 ABSPAT—Absolute Patch

The ABSPAT option declares a series of object-level patch values starting at a specific base address. You can specify up to eight patch values.

Note that all patches must be within the segment address limits or the Task Builder will generate a fatal error:

TKB–*DIAG*–LOAD ADDRESS OUT OF RANGE IN file-name

Syntax

ABSPAT=seg-name:address:val1:val2:...:val8

where:

seg-name	is the one- to six-character name of the segment.
address	is the octal address of the first patch. The address can be on a byte boundary; however, two bytes are always modified for each patch: the addressed byte and the following byte.
val1	is an octal number in the range of 0 through 177777 to be stored at the address.
val2	is an octal number in the range of 0 through 177777 to be stored at the address plus 2.
.	
.	
.	
val8	is an octal number in the range of 0 through 177777 to be stored at the address plus 14.

Default

None

Example

```
RUN $TKB
TKB> PROG,PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB> /
ENTER OPTIONS:
TKB> ABSPAT=MYRTN:012156:143672:027001
TKB> //
```

The ABSPAT option sets the word at location 012156 in segment MYRTN to 143672, and the word at location 012160 in segment MYRTN to 027001.

12.3 ACTFIL—Number of Active Files

The ACTFIL option declares the number of files that your program can have open simultaneously. For each active file that you specify, the Task Builder allocates approximately 512 bytes.

If you specify less than four active files (the default), the ACTFIL option saves space. If you want your program to have more than four active files, you must use the ACTFIL option to make the additional allocation.

You must include a language library (object time system or OTS), and record I/O service routines (such as RMS-11) in your program for the extension to take place. The program section that is extended has the reserved name $$FSR1.

Syntax

ACTFIL=n

where:

n   is a decimal integer indicating the maximum number of files that can be open at the same time.

Default

ACTFIL=4

Example

```
 RUN $TKB
TKB> PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB> /
ENTER OPTIONS:
TKB> ACTFIL=2
TKB> //
```

12.4 ASG—Assign Devices

The ASG option declares physical devices assigned to one or more logical units. (A logical unit corresponds to a channel number in RSTS/E terminology). Note that you cannot assign a unit number higher than the maximum number of units declared in the UNITS option (Section 12.30).

Syntax

ASG=dev-name:unit-1:unit-2:...,dev-name:unit-n:...

where:

dev-name   is a two-character alphabetic device name followed by an optional one- or two-digit decimal unit number.

unit-i   are decimal integers indicating the logical unit numbers (channels).

Default

ASG=SY:1:2:3:4,TI:5,TT:6

Example

```
RUN $TKB
TKB> PROG1=OBJ1,OBJ2,LB:F4POTS/LB
TKB> /
ENTER OPTIONS:
TKB> UNITS=8
TKB> ASG=SY:1:2:3:4:5:6:7,LP0:8
```

The above example declares a maximum of 8 logical units (the UNITS option should be given before the ASG option). The channels 1-7 are allocated to the public disk structure, and channel 8 is allocated to line printer unit 0. Note that in order to assign more than 8 logical units (channels) to a single device, you must respecify the device name followed by the additional units to be assigned. For example:

```
RUN $TKB
TKB> PROG1=OBJ1,OBJ2,LB:F4POTS/LB
TKB> /
ENTER OPTIONS:
TKB> UNITS=11
TKB> ASG=SY:1:2:3:4:5:6:7:8,SY:9:10:11
```

12.5 CLSTR—Cluster Libraries

The CLSTR option lets you declare that multiple resident libraries are to share the same virtual address space in your program. (See Section 2.3.5 for a general discussion of how cluster libraries work.)

Syntax

CLSTR=default-library,library-2,...,library-5:access-code[:apr]

where:

default-library

                .

                .

library-5	The first library listed in the CLSTR option is the default library. Because of the way clustering works, only certain libraries can be default libraries. If you want to build libraries to be clusterable, see Chapter 7 for a description of the techniques. If you simply want to use libraries in a resident library cluster, the Digital-supplied libraries are designed so that the language library can always serve as the default library. Note that not all resident libraries that are available with RSTS/E can take advantage of the clustering feature. Those that can are:

BP2RES Clusterable resident library for BASIC-PLUS-2 programs.

BP2SML Clusterable resident library (a subset of BP2RES) for BASIC-PLUS-2 programs.

C81CIS Clusterable resident library for COBOL-81 programs compiled with /CIS switch (the normal default if your system has the Commercial Instruction Set [CIS] option).

C81LIB Clusterable resident library for COBOL-81 programs compiled with /-CIS switch (the normal default if your system does not have the CIS option).

DIBOLR Clusterable resident library for RMS DIBOL programs.

SMRES Clusterable resident library for SORT/MERGE.

F4PCLS Clusterable resident library for RMS FORTRAN-77 programs.

FDVRDB Clusterable resident library for the form driver for FMS (Form Management System), with debug mode support.

FDVRES Clusterable resident library for the form driver for FMS, without debug mode support.

RMSRES Clusterable resident library for RMS-11 that supports sequential, relative, and indexed file operations.

DAPRES Clusterable resident library for network record access through RMS.

Thus, you can use C81CIS or C81LIB (for COBOL-81 programs) as the default library, and FDVRES and/or RMSRES as secondary libraries in the cluster. Likewise, you can use BP2RES or BP2SML as the default library, and FDVRES and/or RMSRES as secondary libraries in the cluster. (See the reference manual for your specific language for more information.)

Up to five resident libraries can form a cluster on RSTS/E systems. A cluster for Digital-supplied libraries must occupy the upper 8K of your address space. If your site builds its own clusterable libraries, however, these libraries can occupy their own separate cluster, as long as the limit of seven resident libraries for each task build is not exceeded.

For example, you can cluster either of two variations of the COBOL-81 library (C81CIS or C81LIB) with the FMS library (FDVRES) and/or the RMS-11 library (RMSRES), and two or three of your own clusterable libraries either in the same cluster or in a separate cluster in lower virtual memory.

access-code

is either RW (read/write) or RO (read-only). This code indicates how your program intends to access the library. (It will be RO for Digital-provided resident libraries such as BP2RES, FDVRES, C81CIS, etc.) For example:

```
TKB>CLSTR=C81CIS,FDVRES:RO
```

apr

is an integer in the range of 1 to 7 that specifies the first Active Page Register (APR) reserved for the clustered libraries. (See Section 2.3.4 for information on APRs.) If you leave this parameter off, the Task Builder assigns the highest APRs it can to the cluster (APRs 6 and 7 for the above command line).

Currently, Digital-supplied libraries are built to use APRs 6 and 7 so that they can occupy 8K words at the highest end of the user job area.

Default

None

Example

```
RUN $TKB
TKB>PROG,PROG=PROG,LB:C81CIS/LB
TKB>/
ENTER OPTIONS:
TKB>CLSTR=C81CIS,FDVRES:RO
TKB>//
```

12.6 CMPRT—Completion Routine

Use this option to identify a shared region as a supervisor mode library. The CMPRT option requires an argument that specifies the entry point of the completion routine in the library. The completion routine switches the processor back from supervisor mode to user mode, and returns program control to the user task after the supervisor mode library subroutine finishes executing. The completion routine is invoked by a RTS PC at the end of the supervisor mode subroutine.

The following completion routines are available in the system library:

- $CMPCS restores only the carry bit in the user mode PSW.

- $CMPAL restores all the condition code bits in the user mode PSW.

These routines perform the necessary overhead to switch the processor from supervisor mode to user mode and return program control to the user task at the instruction following the call to supervisor mode library subroutine. Although you can write your own completion routines, you should use either $CMPCS or $CMPAL whenever possible. Chapter 9 discusses these routines in detail.

Syntax

CMPRT=name

where:

name   is a one- to six-character name identifying the completion to routine global entry point.

Default

None

12.7 COMMON—Access System Common Block

The COMMON option indicates a resident library that should contain only data. The format of the COMMON option is the same as the LIBR option (Section 12.18).

Syntax

COMMON=name:access-code[:apr[:mask]]

See the description of the LIBR option (Section 12.18) for a discussion of the parameters.

Example

```
RUN $TKB
TKB>PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB>COMMON=MYCOM:RW:5
TKB>//
```

12.8 DSPPAT—Absolute Patch for D-Space

Use the DSPPAT option to declare a series of object-level patches starting at the specified base address, and to make patches to the D-space and I- and D-space task. A maximum of eight patches can be specified. You can also use this option to patch a conventional task at any location.

Syntax

DSPPAT=seg-name:address:val1:val2:...:val8

where:

seg-name	is the 1- to 6-character Radix-50 name of the segment.
address	is the octal address of the first patch. The address can only be on a byte boundary; however, two bytes are always modified for each patch: the addressed byte and the following byte.
val1	is an octal number in the range of 0 to 177777 to be stored at address.
val2	is an octal number in the range of 0 to 177777 to be stored at address+2.
.	
.	
.	
val8	is an octal number in the range of 177777 to be stored at address+16.

Default

None

Example

```
RUN $TKB
TKB> PROG,PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB>
ENTER OPTIONS:
TKB> DSPPAT:MYDATA:100:1:2
```

The DSPPAT option sets the word at location 100 in segment MYDATA to 1, and the word at location 102 in segment MYDATA to 2.

NOTE

All patches must be within the segment address limits or TKB generates the following error message:

```
TKB--*DIAG*--Load address out of range in module-name
```

12.9 EXTSCT—Extend Program Section

The EXTSCT option extends the size of a program section. If the program section has the concatenated (CON) attribute, its size is extended by the length specified. If the program section has the overlay (OVR) attribute, its size is set equal to the length specified, if the length specified is greater than the current size. (If the length is less than the current size, the current size is allocated.)

Syntax

EXTSCT=psect-name:length

where:

psect-name   is the one- to six-character name of the program section to be extended.

length   is the octal number of bytes to extend the program section.

Default

None

Example

```
RUN $TKB
TKB> PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB> /
ENTER OPTIONS:
TKB> EXTSCT=BUFF:250
TKB> //
```

Suppose that BUFF is initially 200[8] bytes long. After the above option is specified, it will be allocated 450[8] bytes if it is concatenated (CON), or 250[8] bytes if it is overlaid (OVR).

12.10 EXTTSK—Extend Task Memory

The EXTTSK option directs the system to allocate additional memory for your executable program, up to a maximum of (32K-32736) words.

The amount of memory available to the program is the sum of the program's size plus the increment you specify in the EXTTSK option, rounded up to the next 32-word boundary. This option extends only the D-space of an I- and D-space task.

This option extends only the D-space of and I- and D-space task.

Syntax

EXTTSK=length

where:

length   is a decimal number in the range 0<n<32,736 specifying the increase in task memory allocation (in words).

Default

The program is extended to the next multiple of 1K words.

Example

```
 RUN $TKB
TKB> PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB> /
ENTER OPTIONS:
TKB> EXTTSK=4096
TKB> //
```

NOTE

1. You should not use the EXTTSK option to extend a task containing memory-resident overlays because the system does not map the extended area.

2. Be careful when extending an I- and D-space task that is linked to a library containing both data and instructions. Normally, libraries are mapped in both I- and D-space, allowing data and instructions to be intermixed. The extension length must not extend into the area mapped for the library or the library will be mapped in I-space only.

12.11 FMTBUF—Format Buffer Size

The FMTBUF option declares the length of the internal working storage that you want the Task Builder to allocate within your program for the compilation of format specifications at run time. The length of this area must equal or exceed the number of bytes in the longest format string to be processed.

Run-time compilation occurs whenever an array is referred to as the source of formatting information within a FORTRAN I/O statement. The program section that the Task Builder extends has the reserved name $$OBF1.

Syntax

FMTBUF=n

where:

n   is a decimal integer, larger than the default (132), that specifies the number of characters in the longest format specification.

Default

FMTBUF=132

Example

```
RUN $TKB
TKB>PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB>/
ENTER OPTIONS:
TKB>FMTBUF=140
TKB>//
```

12.12 GBLDEF—Define a Global Symbol

The GBLDEF option defines a global symbol and its value. The Task Builder considers this symbol definition to be absolute. It overrides any definition in your object program files.

Syntax

GBLDEF=symbol-name:symbol-value

where:

symbol-name	is the one- to six-character name assigned to the global symbol.
symbol-value	is an octal number in the range of 0 through 177777 assigned to the defined symbol.

Default

None

Example

```
RUN $TKB
TKB>PROG=OBJ1,OBJ2,F4POTS/LB
TKB>/
ENTER OPTIONS:
TKB>GBLDEF=LITVAL=1357
TKB>//
```

12.13 GBLINC—Include Global in .STB File

The GBLINC option includes a global symbol in the .STB file that would not otherwise be there. This option is used in Digital-supplied resident libraries that may need to call routines in other resident libraries in a cluster. It is also useful if you are building your own clusterable resident libraries.

Syntax

GBLINC=symbol

where:

symbol   is the global symbol name to be included in the symbol table file being built for the resident library.

Default

None

Example

```
 RUN $TKB
TKB> PROG,PROG,PROG=OVERLY/MP
TKB> /
Enter Options:
TKB> GBLINC=.FCSJT
TKB> GBLINC=USER
TKB> //
```

The GBLINC option includes the symbols named (.FCSJT and USER) in the symbol table file (PROG).

12.14 GBLPAT—Global Relative Patch

The GBLPAT option declares a series of object-level patch values starting at an offset relative to a global symbol. You can specify up to eight patch values.

Note that all patches must be within the segment address limits or the Task Builder will generate a fatal error.

Syntax

GBLPAT=seg-name:sym-name[+/-offset]:val1:val2:...:val8

where:

seg-name	is the one- to six-character name of the segment.
sym-name	is the one- to six-character name specifying the global symbol.
offset	is an octal number specifying the offset from the global symbol.
val1	is an octal number in the range of 0 through 177777 to be stored at the address of the global symbol plus or minus the offset.
val2	is an octal number in the range of 0 through 177777 to be stored at the address of the global symbol, plus or minus the offset, plus 2.
.	
.	
.	
val8	is an octal number in the range of 0 through 177777 to be stored at the address of the global symbol, plus or minus the offset, plus 14.

Default

None

Example

```
RUN $TKB
TKB> PROG,PROG=OVERLY/MP
TKB> /
ENTER OPTIONS:
TKB> GBLPAT=IN1:MRTN+4:10001
TKB> //
```

The GBLPAT option sets the word at location MRTN+4 in segment IN1 to 010001.

12.15 GBLREF—Global Symbol Reference

The GBLREF option declares a global symbol reference. The reference originates in the root segment of the executable program.

Syntax

GBLREF=symbol-name

where:

symbol-name   is the one- to six-character name of a global symbol.

Default

None

Example

```
RUN $TKB
TKB>PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB>GBLREF=MRTN
TKB>//
```

12.16 GBLXCL—Exclude Global from .STB File

The GBLXCL option excludes a global symbol from the .STB file that would otherwise be there. This option is used in Digital-supplied resident libraries that may need to call routines in other resident libraries in a cluster. It is also useful if you are building your own clusterable resident libraries.

Syntax

GBLXCL=symbol

where:

symbol   is the global symbol name to be excluded from the symbol table file being built for the resident library.

Default

None

Example

```
RUN $TKB
TKB> PROG,PROG,PROG=OVERLY/MP
TKB> /
Enter Options:
TKB> GBLXCL=.FCSJT
TKB> GBLXCL=USER
TKB> //
```

The GBLXCL option excludes the symbols named (.FCSJT and USER) from the symbol table file (PROG).

12.17 HISEG—Define High Segment

The HISEG option associates an executable program with a user-written high segment, or run-time system. If there are global definitions within the high segment that resolve references in the input files you specify, the Task Builder links them correctly. The symbol-table file (.STB file) for the named run-time system must be in the account specified by the system logical name LB:. If the HISEG option is not specified:

1. The run-time system associated with the executable program is the same as that associated with the Task Builder itself.

2. No global references to symbols in that high segment are resolved.

Note that the HISEG option is sometimes used when you build a multiuser program with the /MU switch. (See Section 11.16.)

Syntax

HISEG=high-segment-name

where:

high-segment-name   is a one- to six-character name specifying the run-time system.

Default

If no high segment is specified, the run-time system associated with the Task Builder is assumed.

Example

```
RUN $TKB
TKB> PROG=OBJ1,OBJ2
TKB> /
ENTER OPTIONS:
TKB> HISEG=USRTS
TKB> //
```

12.18 LIBR—Access System-Owned Resident Library

The LIBR option declares that your program intends to access a system-owned resident library.

Syntax

LIBR=name:access-code[:apr[:mask]]

where:

name	is the one- to six-character name specifying the library. The Task Builder expects to find a symbol table file and task image file of the same name (name.STB and name.TSK) on the device and under the account specified by the system logical name LB:.

The easiest way to find out if the files exist on LB: is to do a directory:

```
DIR LB:RMSRES.STB
  Name Typ  Size    Prot         DR3:[1,11]
RMSRES.STB    4   < 40>
DIR LB:RMSRES.TSK
  Name Typ  Size    Prot         DR3:[1,11]
RMSRES.TSK   16   < 40>
```

(If the files do not exist on LB:, you must use the RESLIB option, Section 12.23.)

access-code	is the code RW (for read/write) or RO (for read-only), indicating the type of access required by your program.
mask	can be an explicit D-APR usage mask for the library. The value given here will override what was defined for the mask when you built the library, and will be acceptable to all other APR masks associated with this task. See Section 7.7 for more details.
apr	is an integer in the range of 1 to 7 that specifies the first Active Page Register (APR) reserved for the library.

It is not really necessary to understand Active Page Registers to use this modifier. Think of your 32K-word user job area as divided into 8 parts of 4K words each, numbered from 0 through 7. Your program occupies one or more of the lowest-numbered segments.

You can "map" a resident library into the area between the top of the task and the top of the virtual address space. The map must begin on a 4K-word boundary. For example, if the program takes 6K words, there are six APRs available (24K words) for resident library mapping. You can map up to 20K words of resident library into your job, beginning with APR 2, as illustrated in the following diagram:

APR

MK–01047–00

Default

None

Example

```
RUN $TKB
TKB> PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB> LIBR=RMSRES:RO:5
TKB> //
```

This example causes the RMSRES library in LB: to be mapped through APRs 5 and 6. The run-time system is to be mapped through APRs 4 through 7.

12.19 MAXBUF—Maximum Record Buffer Size

The MAXBUF option declares the maximum record buffer size required for any file used by the program. If your program requires a maximum record size that exceeds the default buffer length (133 bytes), you must use this option to extend the buffer.

You must also include a language library (object time system, or OTS), such as FORTRAN's F4POTS, in your executable program for the extension to take place. The program section that is extended has the reserved name $$IOB1.

Syntax

MAXBUF=n

where:

n   is a decimal integer, larger than 133, that specifies the maximum record size in bytes.

Default

MAXBUF=133

Example

```
 RUN $TKB
TKB> PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB> /
ENTER OPTIONS:
TKB> MAXBUF=166
TKB> //
```

12.20 ODTV—ODT SST Vector

The ODTV option declares that a global symbol is the address of the ODT Synchronous System Trap vector table. You must define the global symbol in the main root segment of your program.

The vector address list contained at the global symbol will automatically be installed as the task is loaded at execution time. Also, the vectors are active on the execution of the first instruction when the task is started. This means that there is no window where the traps could be missed, and the program will not waste code space making explicit or SVTK$ or SVDB$ calls.

Syntax

ODTV=symbol-name:vector-length

where:

symbol-name   is a one- to six-character name of a global symbol.

vector-length   is a decimal integer in the range of 1 through 32, specifying the length of the SST vector in words.

Default

None

Example

```
 RUN $TKB
TKB> PROG/DA=OBJ1,OBJ2
TKB> /
ENTER OPTIONS:
TKB> ODTV=TRPVEC:8
TKB> //
```

For related information, refer to the *RSTS/E System Directives Manual* for the SVDB$ (Set SST Vector Table for Debugging Aid) macro.

12.21 PAR—Partition for Resident Area

The PAR option is used when building a resident area. The option identifies a "partition" for the resident area: the amount of space the resident area will occupy when linked into user programs in the user job area (and its location, if the resident library is to occupy absolute addresses).

Syntax

PAR=pname[:base:length]

where:

pname   is the name of the partition. This name must be the same as the file name portion of the executable and symbol table files in the command line. For example:

```
RUN $TKB
TKB>LIBRES/-HD,,LIBRES/PI=LIBRES
TKB>/
ENTER OPTIONS:
TKB>PAR=LIBRES
TKB>//
```

base   is the octal byte address that defines the start of the partition. If the library is position-independent (see Section 7.3.1), the base address is zero. If the library is absolute, the base address must be on a 4K word boundary. For example, if the library is always to be positioned beginning at APR 6, you would specify an octal address of 140000.

length   is the octal number of bytes contained in the partition. If the length is omitted or is zero, it is the size of the executable file.

If length is nonzero, and greater than the size of the executable file that the build produced, the Task Builder automatically extends the size of the resident area to make up the difference.

If the executable file size is greater than the length specified, the Task Builder issues the following error message:

```
%TKB--*DIAG*-TASK HAS ILLEGAL MEMORY LIMITS
```

Default

PAR=GEN

Example

See parameter description, above.

12.22 RESCOM—Access Resident Common Block

The RESCOM option indicates a resident area that should contain only data. The format of the RESCOM option is the same as the RESLIB option (Section 12.23).

Syntax

RESCOM=file-spec/access-code[:apr[:mask]]

12.23 RESLIB—Access Resident Library

The RESLIB option declares that your program intends to access a resident library.

Syntax

RESLIB=filespec/access-code[:apr[:mask]]

where:

filespec	is the file specification identifying the library. The Task Builder expects to find a symbol table file and task image file with the same filename (filename.STB and filename.TSK) on the device and account specified. You must omit the file type from the file specification.
access-code	is the code RW (for read/write) or RO (for read-only), indicating the type of access required by your program.
mask	can be an explicit D-APR usage mask for the library. The value given here will override what was defined for the mask when you built the library, and will be acceptable to all other APR masks associated with this task. See Section 7.7 for more details.
apr	is an integer in the range of 1 to 7 that specifies the first Active Page Register (APR) reserved for the library.
	It is not really necessary to understand Active Page Registers to use this modifier. Think of your 32K-word user job area as divided into 8 parts of 4K words each, numbered from 0 through 7. Your program occupies one or more of the lowest-numbered segments.
	You can "map" a resident library into the area between the top of the task and the top of the virtual address space. The map must begin on a 4K-word boundary. For example, if the program takes 6K words, there are six APRs available (24K words) for resident library mapping. You can map up to 20K words of resident library into your job, beginning with APR 2, as illustrated in the following diagram:

APR

```
0 ┬─────┬─────────────────────┬───
  │     │                     │
1 ┤     │   YOUR PROGRAM      ├───
  │     │                     │
2 ┤     ├─────────────────────┤───
  │     │                     │
3 ┤     │  SPACE AVAILABLE    ├───
  │     │   FOR LIBRARY       │
4 ┤     │                     ├───
  │     │                     │
5 ┤     │                     ├───
  │     │                     │
6 ┤     │                     ├───
  │     │                     │
7 ┤     │                     ├───
  │     │                     │
  └─────┴─────────────────────┘
```

MK–01052–00

Default

None

Example

```
RUN $TKB
TKB>PROG=OBJ1,OBJ2
TKB>/
ENTER OPTIONS:
TKB>RESLIB=DR2:MYLIB/RO:5:200
TKB>//
```

This example causes the library MYLIB on DR2: in the user's account to be mapped read-only beginning at APR 5. The user informs the Task Builder that this library has D-space usage in APR 7.

12.24 RESSUP—Resident Supervisor-Mode Library

The RESSUP option declares that your task intends to access a user owned supervisor-mode library. The term "user owned" means that the library and the symbol definition file associated with it can reside in any directory that you choose. You can specify the directory along with the other portions of the file specification. Do not place comments on the line with RESSUP.

Syntax

RESSUP=filespec/[-]access-code[:apr]

where:

filespec	is the specification identifying the supervisor mode library. The Task Builder expects to find a symbol table file and a task image file with the same file name (filename.STB and filename.TSK) on the device and account specified. You must omit the file type from the specification.
[-]	indicates whether TKB includes mode switching vectors within the user task. If you specify /SV or/SW, TKB includes a 4-word mode switching vector within the address space of the user task for each call to a supervisor-mode library subroutine. If you specify /-SV or /-SW, you must provide your own mode switching vectors. This is useful if your library contains threaded code. Digital recommends, however, using system supplied vectors whenever possible.
access-code	is the code SV (for read-only) or SW (for read/write), indicating the type of access required by your program.
apr	is an integer in the range 0 to 7 that specifies the first supervisor Active Page Register (APR) that you want TKB to reserve for this supervisor-mode library. For position-independent libraries only, you can specify that the default is the lowest available supervisor APR. One supervisor-mode library is required to be at virtual 0 (that is, /SV:0) and must have the CSM (change supervisor mode) dispatcher present together with the completion routines as described in Chapter 9. Most uses would be /SV:0.

12.25 RNDSEG—Round Segment

The RNDSEG option rounds the size of a named segment up to the nearest APR boundary while building a resident library.

Syntax

RNDSEG=seg-name

where:

seg-name   is the 1- to 6-character Radix-50 name of the segment

Default

None

NOTE

The RNDSEG option operates only during a library build. Attempting to use the option while building any other form of task will result in the following diagnostic error message:

```
TKB -- *DIAG* - Library build not requested - ignoring option
```

If you attempt to specify a nonexistent name, the following diagnostic error will be generated and the build will continue:

```
TKB -- *DIAG* - Segment not found to address round
```

12.26 STACK—Declare Stack Size

The STACK option declares the maximum size of the "stack" required by the executable program.

The stack is an area of memory used for temporary storage, subroutine calls, and other system functions. The stack is referenced by hardware register 6 (the stack pointer). The default stack size is 256_{10} words, or 1000_8 bytes. The Task Builder allocates space for the stack immediately following the low 1000_8 bytes of memory used by the RSTS/E monitor, your program, and the run-time system. (That is why, if you look at the Task Builder memory map file, the first location of your program begins at address 2000, unless you specify a different stack size with this option.)

CAUTION

Decreasing the size of the stack to less than the default size can cause unpredictable results for programs written in certain higher-level languages.

Syntax

STACK=n

where:

n   is a decimal integer specifying the number of words required for the stack.

Default

STACK=256

Example

```
RUN $TKB
TKB>PROG=OBJ1,OBJ2,OBJ3
TKB>/
ENTER OPTIONS:
TKB>STACK=512
TKB//
```

12.27 SUPLIB—Resident Supervisor-Mode Library

The SUPLIB option declares that your task intends to access a systemwide supervisor-mode library. The term "systemwide" means that the Task Builder expects to find the supervisor-mode library and the symbol definition file associated with it in the system library account (LB:).

Syntax

SUPLIB=name:[-]access-code[:apr]

where:

name	is the 1- to 6-character name specifying the supervisor mode library. The Task Builder expects to find a symbol table file and a task image file of the same name (name.STB and name.TSK) on the system device and under the account specified by the system logical LB:. If the files do not exist on LB:, you must use the RESSUP option.
[-]	indicates whether TKB includes mode switching vectors within the user task. If you specify :SV or :SW, TKB includes a 4-word mode switching vector within the address space of the user task for each call to a supervisor-mode library subroutine. If you specify :-SV or :-SW, you must provide your own mode switching vectors. Providing your own mode switching vectors is useful if your library contains threaded code. Digital recommends, however, using system supplied vectors whenever possible.
access-code	is the code SV (for read-only) or SW (for read/write), indicating the type of access required by your program.
apr	is an integer in the range 0 to 7 that specifies the first supervisor Active Page Register (APR) that you want TKB to reserve for this supervisor-mode library. You can specify an APR only for position-independent supervisor mode libraries. The default is the lowest available APR. One supervisor-mode library is required to be at virtual 0 (for example, :SV:0) and must have the CSM (change supervisor mode) dispatcher present together with the completion routines as described in Chapter 9. Most uses would be :SV:0.

12.28 TASK—Program Name for SYSTAT

The TASK option lets you specify the name of the program being built. This name is displayed by the SYSTAT program. You can use this option if you want to give a name to a program other than the name of the executable program file.

Syntax

TASK=program-name

where:

program-name   is the one- to six-character name to identify the program in SYSTAT. The characters within the name must be letters (A to Z), numbers (0 to 9), periods (.), or dollar signs ($).

Default

TASK=executable file name

Example

```
RUN $TKB
TKB> PROG,PROG=OVERLY/MP
ENTER OPTIONS:
TKB> TASK=USER
TKB> //
```

12.29 TSKV—Task SST Vector

The TSKV option declares that a global symbol is the address of the program Synchronous System Trap (SST) vector table. You must define the global symbol in the main root segment of your program.

The vector address list contained at the global symbol will automatically be installed as the task is loaded at execution time. Also, the vectors are active on the execution of the first instruction when the task is started. This means that there is no window where the traps could be missed, and the program will not waste code space making explicit or SVTK$ or SVDB$ calls.

Syntax

TSKV=symbol-name:vector-length

where:

symbol-name   is a one- to six-character name of a global symbol.

vector-length   is a decimal integer in the range of 1 through 32 specifying the length of the SST vector in words.

Default

None

Example

```
RUN $TKB
TKB> PROG=OBJ1,OBJ2
TKB> /
ENTER OPTIONS:
TKB> TSKV=VECNAM:8
TKB> //
```

For related information, refer to the *RSTS/E System Directives Manual* for the SVTK$ (Set SST Vector Table for Task) macro.

12.30 UNITS—Maximum Number of Units or Channels

The UNITS option declares the maximum number of logical units (often called channels in RSTS/E documentation) that are used by the program. The default number is 6.

NOTE

If you want to use more than 6 channels, specify the UNITS option before the ASG option (Section 12.4) that defines the devices for the units.

Syntax

UNITS=max-units

where:

max-units   is a decimal number from 0 to 14 specifying the maximum number of logical units. (The Task Builder allows you to specify more than 15 channels, but RSTS/E will give an error when the program is RUN.)

Default

UNITS=6

Example

```
RUN $TKB
TKB>PROG=OBJ1,OBJ2,LB:F4POTS/LB
TKB>/
ENTER OPTIONS:
TKB>UNITS=4
TKB>ASG=SY:0:1,LP0:3,TI:4
TKB>//
```

12.31 VARRAY - Virtual Array Specification and Usage

A virtual array in FORTRAN is a defined area outside of the virtual address space of a task, but within the task's logical address space. The TKB assigns the name $VIRT to the virtual array and automatically provides code to create a dynamic region in memory. Refer to Section 7.7 for more dtails on the use of virtual arrays. The VARRAY=OVR option specifies an overlaid virtual array so that it may be used similarly to the way a FORTRAN COMMON is used. This means that each segment of an overlaid task that uses it defines the array in the same way as it is defined in the root segment. Thereafter, the segment may access the array directly without passing arguments, as is necessary when the array has the concatenated attribute (the default, VARRAY=CON).

To use the VARRAY option with the OVR attribute as VARRAY=OVR, you must first define the array (for example, VIRTUAL DATA(10)), in the root segment of the task. Then, you must define the array in the same way in each segment of an overlaid task using the virtual array. Example 12-1 illustrates how a virtual array may be directly accessed by segments in a task. The example also shows the TKB command line and overlay description file for building the task.

Using the VARRAY option with the CON attribute as VARRAY=CON (the default operation) results in a virtual array subject to the restrictions and uses that are described in the reference manual for the specific kind of FORTRAN that you are using.

Syntax

VARRAY=option

where:

option   is either OVR or CON.

Default

VARRAY=CON

Example 12–1: A Task Using a Virtual Array with the OVR Attribute

```
C
C    Program to test the Task Builder option VARRAY
C
      PROGRAM MAIN
      IMPLICIT INTEGER *2 (A-Z)
      VIRTUAL DATA(10)
      CALL INPUT
      CALL CALC
      CALL OUTPUT
      CALL EXIT
      END
```

```
        SUBROUTINE INPUT
        IMPLICIT INTEGER *2 (A-Z)
        VIRTUAL DATA(10)
        TYPE 10
10      FORMAT (1X,'INPUT I '$)
        ACCEPT 20,DATA(1)
20      FORMAT (I2)
        TYPE 30
30      FORMAT (1X, 'INPUT J '$)
        ACCEPT 20,DATA(2)
        RETURN
        END

        SUBROUTINE CALC
        IMPLICIT INTEGER *2 (A-Z)
        VIRTUAL DATA(10)
        DATA(3) = DATA(1) + DATA(2)        !I + J
        DATA(4) = DATA(1) - DATA(2)        !I - J
        DATA(5) = DATA(1) * DATA(2)        !I * J
        DATA(6) = DATA(1) / DATA(2)        !I / J
        RETURN
        END

        SUBROUTINE OUTPUT
        IMPLICIT INTEGER *2 (A-Z)
        VIRTUAL DATA(10)
        TYPE 10.DATA(3),DATA(4),DATA(5),DATA(6)
10      FORMAT (1X, 'I + J =',I6,/,1X,'I - J =',I6,/
        1,1X'I * J=',I6,/,1X.'I / J =',I6)
        RETURN
        END
;
; Command file MAINFT.CMD to build MAINFT.TSK
;
MAINFT/FP,MAINFT/MA/-WI=MAINFT/MP
VARRAY=OVR
//
;
;Overlay description file MAINFT.ODL for MAINFT.TSK
;       .ROOT   $MAIN-*($INPT,$CALC,$OUTP)
$MAIN:  .FCTR   MAIN-LB:[1,1]F77OTS/LB
$INPT:  .FCTR   INPUT-LB:[1,1]F77OTS/LB
$CALC   .FCTR   CALC-LB:[1,1]F77OTS/LB
$OUTP   .FCTR   OUTPUT-LB:[1,1]F77OTS/LB
        .END
```

12.32 VSECT—Virtual Program Section

Use the VSECT option to specify the virtual base address, virtual length, and, optionally, the physical memory allocated to the named program section. It is possible to have multiple VSECT lines in a task build; however, the sum total of the phsyical lengths cannot exceed 255K words. The VSECT option does the following two things during task build:

* It adds the physical length to the offset value in the header so that the RSTS/E monitor will know at run-time how large a dynamic region the task will need.

* It places the given PSECT name at the specified absolute address (usually an APR boundary like 140000 octal) so that the program can relate the base of the window that will be mapped to the code. To do this in MACRO-11, declare the PSECT directly; in FORTRAN-77, use a COMMON statement of the form:

    COMMON MARRAY/IARRAY

    where:

    MARRAY   is the PSECT name
    IARRAY   is declared as an array

If the task being built is /ID and one or more APRs are being used of I-only library code (such as RMSRES), it is useful to use the D-APRs of that code with the VSECT option as it is D-only mapping.

Refer to Section 7.6 for more information on virtual program sections.

Syntax

VSECT=p-sect-name:[base:window][physical-length]

where:

p-sect-name	is a 1- to 6-character program section name.
base	is an octal value representing the virtual base address of the program section in the range of 0 to 177777. If you use the mapping directives, the value you specify must be a multiple of 4K.
window	is an octal value specifying the amount of virtual address space in bytes allocated to the program section. Base plus window must not exceed 177777. This is the maximum window size that can be created with the CRAW$ function.
physical-length	is an octal value specifying the minimum amount of physical memory to be allocated to the section in units of 64-byte blocks. TKB rounds this value up to the next 256-word limit. This value, when added to the task image size of any previous allocation, must not exceed 255K words on RSTS/E (maximum 17740 octal). If you do not specify a length, TKB assumes a value of 0.

Default

Physical length defaults to 0.

12.33 WNDWS—Number of Address Windows

The WNDWS option declares the number of address windows required by the program in addition to those needed to map the program and any declared (with CLSTR, RESLIB, LIBR, SUPLIB, RESCOM, RESSUP, or COMMON) resident area. In other words, you use this option to tell the Task Builder what windows your program will access directly using the .PLAS mapping directives (see the *RSTS/E System Directives Manual*). The number specified is equal to the number of such simultaneously mapped regions the program will use.

Syntax

WNDWS=n

where:

n   is an integer in the range 0 to 23.

Default

WNDWS=0

Example

```
 RUN $TKB
TKB> PROG=OBJ1,OBJ2
TKB> /
ENTER OPTIONS:
TKB> WNDWS=2
TKB> //
```

Overlay Description Language (ODL)

The Task Builder provides a language, called the Overlay Description Language (ODL), that allows you to describe the overlay structure of a program. You construct a text file containing a series of ODL commands, one command per line. You then refer to this file in a Task Builder command line, with an /MP switch, as described in Chapter 3. For example:

```
RUN $TKB
TKB>OUT,MAP=OVERLY/MP
```

The ODL command file is named OVERLY.ODL (.ODL is the default file type).

13.1 ODL Command Line

An ODL line takes the form:

label: directive argument-list ;comment

A label is required only for the .FCTR command (see Section 13.3).

The ODL commands are listed below and are described in alphabetical order in Sections 13.2 through 13.6.

.ROOT   specifies the entire overlay structure in terms of (1) your separately compiled or assembled program and subprogram files, (2) library files, (3) program sections, and (4) names defined in .NAME or .FCTR commands. These elements are connected by operators, which show the way the elements are to be linked. Operators include the symbols:

         - , * ( ) !

.FCTR   defines a "substructure" within the entire overlay structure. As with .ROOT, the substructure is specified in terms of object files, library files, program sections, and names defined in .NAME or other .FCTR commands. These elements are connected by the same operators used in the .ROOT command.

.PSECT   allows you to directly specify the placement of a global program section in an overlay structure. Thus, you can indicate the segment to which the program section will be allocated.

.NAME   allows you to define a name and attributes for an overlay segment. An overlay segment is a piece of the overlay structure that is stored on disk such that it is loaded with one disk access.

.END   is used to end the overlay description.

13.2   The .END Command

Use the .END command as the last line in the ODL file. The .END command tells the Task Builder where the input ends. The format of the .END command is:

.END

13.3   The .FCTR Command

The .FCTR command lets you build large, complex overlay structures and represent them clearly. The format of the .FCTR command is:

label: .FCTR structure

where:

label   at the beginning of the line is used as a part of the structure of a .ROOT or another .FCTR command. The label must be unique with respect to file names and other labels. The structure portion of the .FCTR command can be made up of the same components as the structure of a .ROOT command.

The .FCTR command lets you extend the overlay tree description beyond the one line possible in a .ROOT command. For example:

```
        .ROOT   AFCTR,BFCTR
AFCTR:  .FCTR   A-LIB-(A1-LIB,A2-LIB)
BFCTR:  .FCTR   B-LIB-(B1-LIB,B2FCTR)
B2FCTR: .FCTR   B2-LIB(B21-LIB,B22-LIB,B23-LIB)
LIB:    .FCTR   LB:F4POTS/LB
        .END
```

In the example above, the AFCTR and BFCTR items in the .ROOT command are expanded in following .FCTR commands. Likewise, B2FCTR and LIB are defined in the third and fourth .FCTR commands. The B2FCTR item is defined in a "nested" .FCTR command; that is, the B2FCTR item is defined by a .FCTR command nested within the BFCTR item's defining .FCTR command. The .FCTR command can be nested in this manner to 16 levels.

13.4   The .NAME Command

The .NAME command lets you give a name to a segment and then assign attributes to a segment. As described in Chapter 3, a segment is a piece of your overlay structure that can be loaded in one disk access.

The name you assign must be unique; that is, it must be different than file names, program section names, .FCTR labels, and other segment names used in the overlay description.

The chief purpose of the command is to assign a name to a null co-tree root (Section 4.2), and to make a data segment autoloadable (Section 6.5).

The format of the .NAME command is:

.NAME segment-name[,attr][,attr]

where:

segment-name	is a one- to six-character name consisting of the characters A-Z, 0-9, and $. The name applies to the segment defined immediately following the name when it is used in a .ROOT or .FCTR command. A segment is formed by pieces connected by a dash (-) without intervening parentheses. (Pieces connected by a comma are overlaid and are stored as separate segments.)
attr	is one of the following:

	GBL	Defines the segment-name as a global symbol. As such, it can be referred to in transfer-of-control statements from other pieces of the overlay structure. When such a transfer of control is executed, the segment is loaded, and control is returned to the statement or instruction immediately following the call. Used chiefly in making data segments autoloadable (see Section 6.5).
	NOGBL	Does not define the segment-name as a global symbol. Hence, the name cannot be referred to in transfer-of control statements from other pieces of the overlay structure. If the GBL attribute is not specified, NOGBL is assumed. You would use this attribute when using .NAME to define a null segment as a co-tree root (see Section 4.2).
	NODSK	No disk space is allocated to the named segment in the executable file. If a data overlay segment has no initial values but will be generated by the running program, there is no need to reserve space for it. If you request this option, and the code in your program assigns initial data values to the segment, the Task Builder terminates the build with a fatal error:

    LOAD ADDR OUT OF RANGE IN MODULE
    file-name

	DSK	Disk space is allocated to the named segment in the executable file. If you do not specify NODSK, DSK is assumed.

If more than one name is applied to a segment, the attributes of the last name given take effect.

13.5 The .PSECT Command

You use the .PSECT command to define the name and attributes of a program section that you want to place in the structure of a .ROOT or .FCTR command. In other words, you can directly specify the placement of a program section named in a .PSECT command.

The general form of the .PSECT command is:

.PSECT p-name[,attr1][,attr2]...[,attr4]

where:

p-name	is the name of the program section (a one- to six-character name consisting of the character A-Z, 0-9, or $).

attr[i]	The attr parameters can be any of the following:	
	GBL	A global program section, or
	LCL	A local program section.
	RW	A read/write program section, or
	RO	A read-only program section.
	REL	A relocatable program section, or
	ABS	An absolute program section.
	OVR	An overlaid program section, or
	CON	A concatenated program section.
	SAV	A program section with the save attribute.
	D	A program section contains data.
	I	A program section contains instructions and/or data.

For example, suppose a program consists of the file CNTRL as a root, with overlays A, B, and C. Suppose that CNTRL calls A, B, C, and A again, and that A contains a common block named DATA3. The first execution of A stores data in DATA3, and the second execution of A needs this data. The common block DATA3 must be moved to the root segment, where it will not be overlaid with the old values when A is read in from disk for its second execution. This is accomplished by the following ODL file:

```
        .PSECT DATA3,RW,GBL,REL,OVR
        .ROOT CNTRL-DATA3-LIB-*(A-LIB,B-LIB,C-LIB)
LIB:    .FCTR LB:F4POTS/LB
        .END
```

See the *PDP-11 MACRO-11 Language Reference Manual* for more information about .PSECTs.

13.6 The .ROOT Command

Each overlay description must have one, and only one .ROOT command. The .ROOT command defines the overlay structure. The general format of the command is:

.ROOT structure

where:

structure	is a series of file specifications for your separately compiled object programs, library files, program section names, or names defined in .FCTR or .NAME commands.

These items are connected by the following operators:

1. The hyphen (-) operator indicates the concatenation of two items. For example, X-Y means that sufficient virtual address space will be allocated to contain the items X and Y simultaneously. The Task Builder allocates X and Y in sequence.

2. The comma (,) operator, appearing within parentheses, indicates the overlaying of virtual address space. For example, (Y,Z) means that the virtual address space can contain either Y or Z; they overlay each other. Parentheses can be nested to 16 levels.

    The comma operator outside of parentheses is used to define multiple tree structures.

3. The exclamation point (!) operator indicates memory-resident overlays in a resident area (see Chapter 7).

4. The asterisk operator (*) indicates that autoload vectors are to be generated for the following piece or pieces of the overlay structure. Unless you want to save space by carefully applying autoload indicators (Chapter 5), the simplest way to use the asterisk is immediately before the outermost left parenthesis in your ODL file. And, for co-trees, put additional asterisks before a non-null co-tree root segment and any co-tree's outermost left parenthesis.

For example:

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

The .ROOT command in this ODL file describes the following overlay tree.
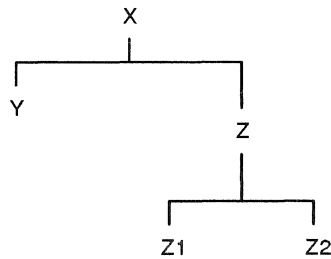
MK-01053-00

The units Y and Z overlay each other, as do Z1 and Z2.

13.7 Indirect Command Files

The ODL processor can accept ODL text specified in an indirect command file. If an at sign (@) appears as the first character in a line, the processor reads text from the file specified immediately after the at sign character.

For example, suppose you create a file, called BIND.ODL, that contains the text:

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

This text can be inserted by a line beginning with @BIND in another ODL file:

```
        .ROOT A-*(B,C)
C:      .FCTR C1-(C2,C3)
@BIND
        .END
```

This has the same effect as an ODL file with the following commands:

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

The Task Builder allows two levels of indirection. That is, you can place a reference to an indirect command file in an indirect command file. (However, note that excessive use of indirect command files will degrade Task Builder performance.)

Appendixes
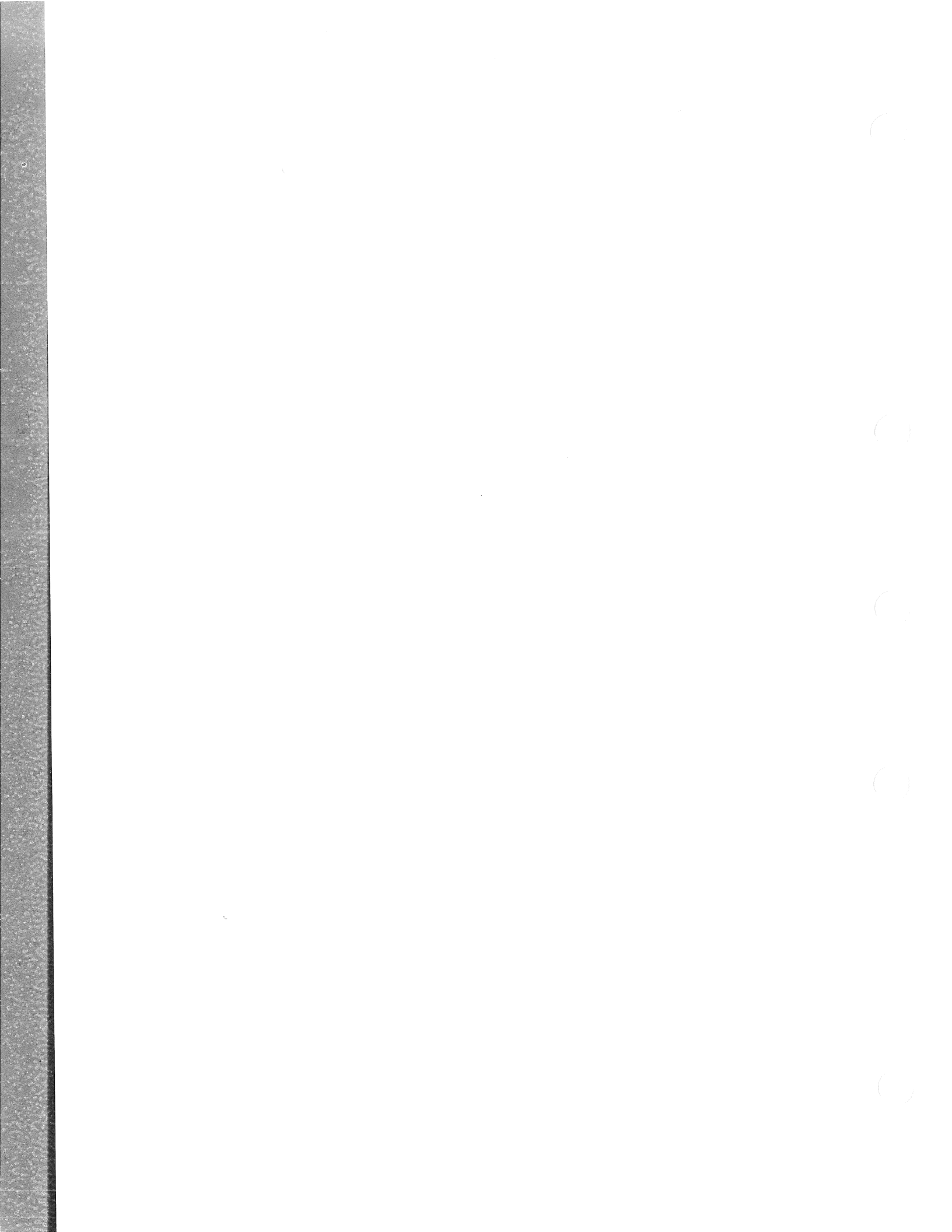
Appendix A

Error Messages

This appendix lists the error messages produced by the Task Builder. Error messages are printed in two forms:

- %TKB—*DIAG*-error-message

- ?TKB—*FATAL*-error-message

When you give commands to the Task Builder from a terminal (rather than from a command file), you can correct some errors as you go along. These errors are noted with the *DIAG* heading. Correct the error and the build will proceed. Indeed, some of the errors merely tell you about an unusual condition. If you can live with the condition, or consider it to be normal to your program, you can go ahead with the build and run the executable file produced.

The errors headed by *FATAL* abort the build; you have to start over.

Messages and their explanations are listed below. If the explanation refers to a system error, or says that the error should not occur on RSTS/E systems, please send a Software Performance Report (SPR) to Digital.

ALLOCATION FAILURE ON FILE filename

The Task Builder could not find enough disk space to store the executable program file or did not have write access to the account or disk that was to contain the file.

BLANK P-SECTION NAME IS ILLEGAL odl-line

The overlay description line printed contains a .PSECT command that does not have a program section name.

CLUSTER LIBRARY ELEMENT, element-name, IS NOT RESIDENT OVERLAID

The listed cluster element was built without memory-resident overlays. This kind of element cannot be used as a cluster library element. Cluster libraries 2-6 must be memory-resident and overlaid.

COMMAND I/O ERROR

An I/O error occurred for an input file in a Task Builder command. The device may not be on line, or there may be a hardware error.

COMMAND SYNTAX ERROR command-line

The command line given is specified incorrectly. See Part IV for the correct syntax for the command.

COMPLEX RELOCATION ERROR - DIVIDE BY ZERO: MODULE filename

A zero divisor was detected in a complex expression. The result of the division was set to zero. (A probable cause is division by a global symbol whose value is undefined. You can set a value for a global symbol with the GBLDEF option to correct this.)

FILE filename ATTEMPTED TO STORE DATA IN VIRTUAL SECTION

You should not get this error running the Task Builder on RSTS/E systems. It diagnoses an error for a capability not used on RSTS/E.

FILE filename HAS ILLEGAL FORMAT

The file named is in an invalid format. This can occur if you try to build a text file, such as a source file. Input files must be compiled or assembled object program files or library files containing compiled or assembled object routines.

ILLEGAL APR RESERVATION

An APR parameter specified in a COMMON, LIBR, SUPLIB, RESCOM, RESSUP, or RESLIB option is outside the range 0-7.

ILLEGAL DEFAULT PRIORITY SPECIFIED

Note that this error relates to the PRI option, which is ignored on RSTS/E systems. The error is returned if you specify an illegal value in the use of the PRI option.

ILLEGAL ERROR-SEVERITY CODE octal-list

System error (no recovery). Please send Digital a Software Performance Report (SPR) with a copy of the message containing the octal-list as printed.

ILLEGAL FILENAME invalid-line

The invalid line printed contains a wildcard (*) in a file specification. You cannot use wildcards in file specifications for the Task Builder.

ILLEGAL GET COMMAND LINE ERROR CODE

System error (no recovery). Please send an SPR to Digital.

ILLEGAL LOGICAL UNIT NUMBER invalid-line

You tried to assign a device (ASG option) to a logical unit number larger than the available number of logical units (UNITS option or the default of 6 if the UNITS option is not specified).

ILLEGAL MULTIPLE PARAMETER SETS invalid-line

You tried to specify more parameters for an option than the option format calls for. See Chapter 12 for the correct format for options.

ILLEGAL NUMBER OF LOGICAL UNITS invalid-line

You cannot specify a logical unit number greater than 14.

ILLEGAL ODT OR TASK VECTOR SIZE

You should not get this error on RSTS/E systems; it diagnoses an error for an option not processed by RSTS/E.

ILLEGAL OVERLAY DESCRIPTION OPERATOR invalid-line

The invalid line printed is an ODL line that contains an operator that the Task Builder does not recognize. This error occurs if the first character in a program section or segment name is a period (.).

ILLEGAL OVERLAY DIRECTIVE invalid-line

The invalid line printed contains an unrecognizable overlay command.

ILLEGAL PARTITION/COMMON BLOCK SPECIFIED

You tried to specify a partition option (PAR) or resident area access option (COMMON, LIBR, RESCOM, RESLIB, RESSUP, OR SUPLIB) defining a resident area as starting not on a 32-word boundary.

ILLEGAL P-SECTION/SEGMENT ATTRIBUTE

You tried to define an attribute for a program section or segment that is not recognizable to the Task Builder. See the description of the .PSECT command or .NAME command in Chapter 13.

ILLEGAL REFERENCE TO LIBRARY P-SECTION psect-name

Your program attempts to refer to a program section name that exists in a run-time system or resident area but has not named the run-time system or area in a COMMON, HISEG, LIBR, RESCOM, RESLIB, RESSUP, SUPLIB or option.

ILLEGAL SWITCH file-specification

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

INCOMPATIBLE OTS MODULE

TKB did not find the Overlay Run-Time System (OTS) module. The OTS modules are part of the system library. This error occurs if you are using an incompatible version of the system library (SYSLIB.OLB).

INCOMPATIBLE REFERENCE TO LIBRARY P-SECTION psect-name

Your program attempts to refer to more storage in a run-time system or resident library than exists in the run-time system or resident library definition.

INCORRECT LIBRARY MODULE SPECIFICATION invalid-line

The invalid line printed names a library routine name with an invalid character. Valid characters are A-Z, 0-9, space, dollar sign ($), or period (.).

INDIRECT COMMAND SYNTAX ERROR invalid-line

The invalid line printed is a command from an indirect file. You must correct the syntax of the command in the indirect file.

INDIRECT FILE OPEN FAILURE invalid-line

The invalid line printed refers to a command input file that could not be located.

INSUFFICIENT PARAMETERS invalid-line

The invalid line printed contains too few parameters. See Part IV for the correct format for command lines, switches, and options.

INVALID APR RESERVATION invalid-line

You specified an APR on an option dealing with an absolute resident area. An absolute resident area is built to occupy the same virtual address space each time it is used; you do not specify an APR in this case.

INVALID KEYWORD IDENTIFIER invalid-line

The invalid line printed contains an unrecognizable option.

INVALID PARTITION/COMMON BLOCK SPECIFIED

The base address of a partition defined in a PAR option is not on a 4K boundary or is not 0, or the memory bounds for the partition overlap a run-time system or other resident area.

INVALID REFERENCE TO MAPPED ARRAY BY MODULE filename

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

INVALID WINDOW BLOCK SPECIFICATION

The number of extra address windows requested with the WNDWS option cannot exceed 23.

I/O ERROR LIBRARY IMAGE FILE

An I/O error occurred during an attempt to open or read the symbol table file (.STB file type) for a run-time system or resident area.

I/O ERROR ON INPUT FILE filename

An I/O error occurred during an attempt to open or read an input file in the Task Builder command line. This error message can also occur if your command line is too long (greater than 80 characters).

I/O ERROR ON OUTPUT FILE filename

An I/O error occurred during an attempt to open or write to an output file in the Task Builder command line.

LABEL OR NAME IS MULTIPLY DEFINED invalid-line

The invalid line printed defines a name that has already appeared in 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. (The specified library has no routines.)

LIBRARY REFERENCES OVERLAID LIBRARY invalid-line

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

LOAD ADDR OUT OF RANGE IN MODULE filename

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

* An attempt to initialize a program section contained in a run-time system or resident area.

- An attempt to initialize an absolute location outside the limits of the segment or in the header.

- A patch outside the limits of the segment it applies to.

- An attempt to initialize a segment having the NODSK attribute.

LOOKUP FAILURE ON FILE filename invalid-line

The invalid line printed contains a file name that cannot be located.

LOOKUP FAILURE ON SYSTEM LIBRARY FILE

The Task Builder cannot find the system library file to resolve undefined symbols. The system library is LB:SYSLIB.OLB unless defined otherwise with a /DL switch.

LOOKUP FAILURE RESIDENT LIBRARY FILE invalid-line

No symbol table (.STB) file or executable file (.TSK) can be found for the run-time system or resident area.

MAXIMUM INDIRECT FILE DEPTH EXCEEDED invalid-line

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

MODULE filename AMBIGUOUSLY DEFINES P-SECTION psect-name

The program section named has been defined in two pieces of the overlay structure that are not on a common path and is referred to by a segment that is common to both paths.

MODULE filename AMBIGUOUSLY DEFINES SYMBOL sym-name

The file named refers to or defines a symbol. The symbol definition exists on two different paths but is referenced by a segment common to both paths.

MODULE filename ILLEGALLY DEFINES XFR ADDRESS psect-name addr

This error is caused by one of the following:

1. The start address printed is odd (it must be even).

2. The file containing the start address is in an overlay segment. The start address must be in the root segment of the main tree.

3. The address is in a program section that has not yet been defined. Please send an SPR to Digital if this is caused by Digital-supplied software.

MODULE filename MULTIPLY DEFINES P-SECTION psect-name

The program section named has been defined more than once in the same segment with different attributes.

Or, a global program section has been defined more than once with different attributes in more than one segment along a common path.

MODULE filename MULTIPLY DEFINES SYMBOL sym-name

Two definition for the relocatable symbol sym-name have occurred on a common path.

Or, two definitions for an absolute symbol with the same name but two different values have occurred.

MODULE filename MULTIPLY DEFINES XFR ADDR IN SEG segment-name

More than one file making up the root has a start address.

MODULE routine-name NOT IN LIBRARY

The Task Builder could not find the routine named on the /LB switch in the library specified.

NO DYNAMIC STORAGE AVAILABLE

The Task Builder needs additional storage for a symbol table and cannot find it. Refer to Appendix E for ways to optimize Task Builder performance.

NO MEMORY AVAILABLE FOR LIBRARY library-name

The Task Builder could not find enough free virtual memory to map the specified run-time system or resident area. Refer to Appendix E for ways to optimize Task Builder performance.

NO ROOT SEGMENT SPECIFIED

You must specify one .ROOT command in the overlay description file.

NO VIRTUAL MEMORY STORAGE AVAILABLE

The maximum allowable size of the Task Builder work file was exceeded. See Section 7.5.6 for suggestions on reducing the size of the work file.

ONLY ONE HISEG MAY BE SPECIFIED

You attempted to specify more than one high segment. The command that generated this error is ignored.

OPEN FAILURE ON FILE filename

An I/O error occurred while the Task Builder was attempting to open the specified file. Try the build again. If you get the same error, see your system manager and report the I/O error.

OPTION SYNTAX ERROR invalid-line

The invalid line printed contains an option that the Task Builder cannot process because it is specified incorrectly. See Chapter 12 for the correct syntax for options.

OVERLAY DIRECTIVE HAS NO OPERANDS

All overlay commands except .END require operands.

OVERLAY DIRECTIVE SYNTAX ERROR invalid-line

The invalid line printed contains a syntax error or refers to a line that contains an error.

PARTITION par-name HAS ILLEGAL MEMORY LIMITS

The partition named is longer than the available address space.

PASS CONTROL OVERFLOW AT SEGMENT segment-name

System error. Please send an SPR to Digital with a copy of the ODL file associated with the error.

PIC LIBRARIES MAY NOT REFERENCE OTHER LIBRARIES

You have tried to build a position-independent resident area that refers to another resident area.

P-SECTION psect-name HAS OVERFLOWED

You have tried to create a program section larger than (32K-32) words.

REQUIRED INPUT FILE MISSING

At least one input file is required for a build.

REQUIRED PARTITION NOT SPECIFIED

You should not get this error on RSTS/E systems. It diagnoses an error for a capability not used on RSTS/E.

RESIDENT LIBRARY HAS INCORRECT ADDRESS ALIGNMENT invalid-line

The invalid line printed specifies a resident area that has one of the following problems:

1. The library refers to another library with invalid address bounds (that is, not on 4K word boundary).

2. The library has invalid address bounds.

RESIDENT LIBRARY MAPPED ARRAY ALLOCATION TOO LARGE invalid-line

The invalid-line displayed contains a reference to a shared region that has allocated too much memory in the task's mapped array area. The total allocation exceeds the system limit; the maximum usable size on RSTS/E is 255K words.

RESIDENT LIBRARY MEMORY ALLOCATION CONFLICT option

One of the following problems has occurred. You tried to specify:

*   More than 7 resident areas.

*   The same resident area more than once.

*   Absolute resident areas whose memory allocations overlay.

ROOT SEGMENT IS MULTIPLY DEFINED invalid-line

The invalid line printed contains the second .ROOT command encountered in an ODL file. Only one .ROOT command is allowed.

SEGMENT seg-name HAS ADDR OVERFLOW: ALLOCATION DELETED

Within a segment, the program attempted to allocate more than (32K-32) words. A map file is produced if it was specified, but no executable file is produced.

TASK HAS ILLEGAL MEMORY LIMITS

You have tried to build a program whose size exceeds the allowable memory size. (This may be the size defined in a PAR option.) If an executable file was produced, delete it.

TASK HAS ILLEGAL PHYSICAL MEMORY LIMITS
mapped-array executable-program program extension

The sum of the values displayed—mapped array size, executable program size, and program extension size—exceeds 2.2 million bytes. The quantities are shown as octal numbers in units of 64-byte blocks. Delete any resulting executable program file.

TASK IMAGE FILE filename IS NON-CONTIGUOUS

This error will only occur if your disk is so full that RSTS/E cannot find contiguous space for your program. The file is therefore created noncontiguous; you can otherwise ignore the error message.

TASK REQUIRES TOO MANY WINDOW BLOCKS

The number of address windows required by the program and any resident areas is more than 16. Only 16 are available.

TASK-BUILD ABORTED VIA REQUEST option-line

The option-line printed contains your request to abort the build. You can now retype commands to rerun the Task Builder.

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

The invalid line printed contains a .FCTR command that exceeds the maximum of 16 nested .FCTR commands.

TOO MANY PARAMETERS invalid-line

The invalid line printed contains an option with more parameters than required.

TOO MANY PARENTHESES LEVELS invalid-line

The invalid line printed contains nested parentheses that exceed the maximum of 16 nested parentheses.

TRUNCATION ERROR IN MODULE filename

You tried to load a global value greater than +127 or less than -128 into a byte. Only the low-order eight bits are loaded.

UNABLE TO OPEN WORK FILE

This error can result from several conditions. For example, the device is full, or the work file is assigned to a private pack where you do not have an account, or the work file device is either not mounted or is mounted read-only.

UNBALANCED PARENTHESES invalid-line

The invalid line printed contains unbalanced parentheses. The number of left parentheses must equal the number of right parentheses.

n UNDEFINED SYMBOLS SEGMENT seg-name

The segment named contains n undefined symbols. If you did not request a memory map file, the symbols are also printed at your terminal.

VIRTUAL SECTION HAS ILLEGAL ADDRESS LIMITS option

This error means that the virtual section was declared larger than the limit on RSTS/E.

WORK FILE I/O ERROR

An I/O error occurred during an attempt to refer to data stored by the Task Builder in its work file.

Task Builder Input Data Formats

A compiled or assembled program (.OBJ file), called an object module, consists of variable-length record information. Six record (or block) types are included in the object language. These records guide the Task Builder in the translation of the object language into a task image.

The six record types are:

- Type 1 - Declare Global Symbol Directory (GSD)

- Type 2 - End of Global Symbol Directory

- Type 3 - Text Information (TXT)

- Type 4 - Relocation Directory (RLD)

- Type 5 - Internal Symbol Directory (ISD)

- Type 6 - End of Module

Each object module must consist of at least five of the record types. The only record type that is not mandatory is the internal symbol directory. The appearance of the various record types in an object module follows a defined format. See Figure B–1.

An object module must begin with a GSD record and end with an end-of-module record. Additional GSD records can occur anywhere in the file but must appear before an end-of-GSD record. An end-of-GSD record must appear before the end-of-module record, and at least one relocation directory record (RLD) must appear before the first text information record (TXT). Additional RLDs and TXTs can appear anywhere in the file. The internal symbol directory records (ISDs) can appear anywhere in the file between the initial GSD and end-of-module records.

Object module records are of variable length and are identified by a record type code in the first byte of the record. The format of additional information in the record depends on the record type.

Figure B–1: General Object Module Format

GSD	INITIAL GSD
RLD	INITIAL RELOCATION DIRECTORY
GSD	ADDITIONAL GSD
TXT	TEXT INFORMATION
TXT	TEXT INFORMATION
RLD	RELOCATION DIRECTORY

GSD	ADDITIONAL GSD
END GSD	END GSD
ISD	INTERNAL SYMBOL DIRECTORY
ISD	INTERNAL SYMBOL DIRECTORY
TXT	TEXT INFORMATION
TXT	TEXT INFORMATION
TXT	TEXT INFORMATION
END MODULE	END OF MODULE
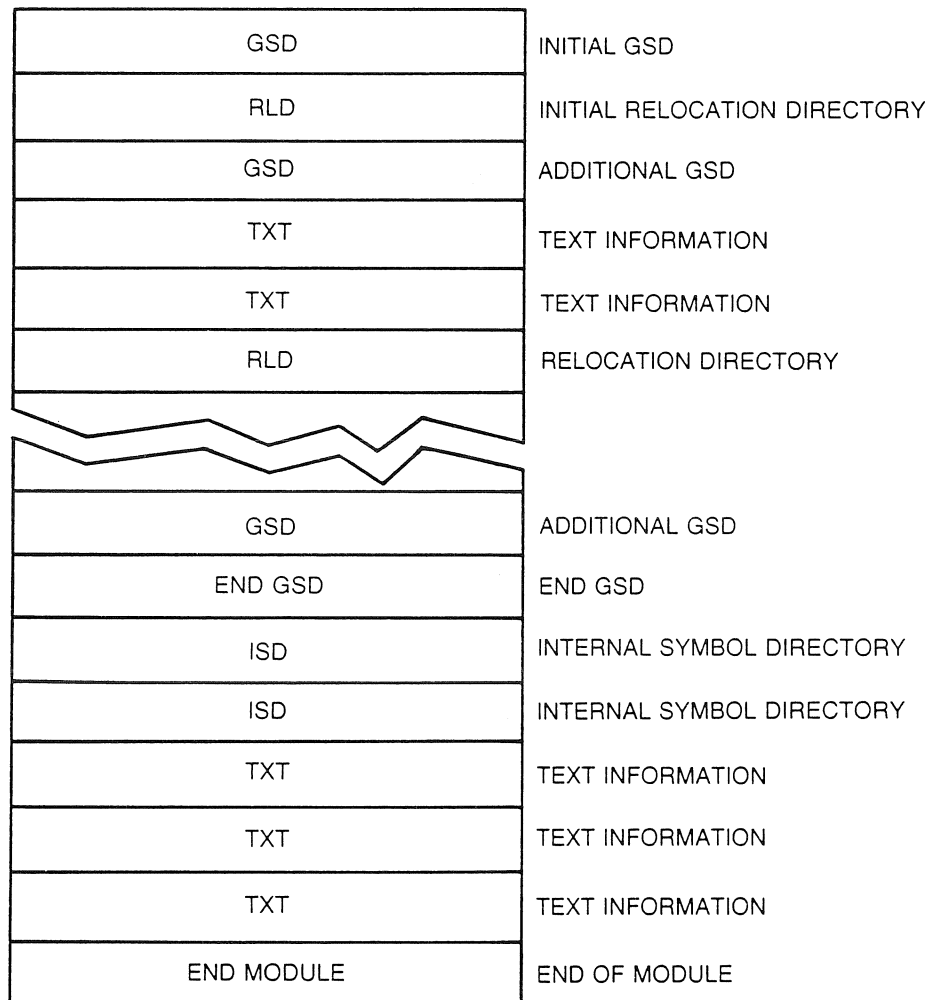
MK–01056–00

B.1 Global Symbol Directory

Global symbol directory (GSD) records contain all the information necessary to assign addresses to global symbols and to allocate the memory required by a task.

GSD records are the only records processed in the first pass. You can save a significant amount of time if you put all GSD records at the beginning of a module, because less of the file must be read on the first pass.

GSD records contain the nine types of entries listed in Table B-1.

Table B–1: GSD Entry Types

Type (Octal)	Entry
0	Module Name
1	Control Section Name
2	Internal Symbol Name
3	Transfer Address
4	Global Symbol Name
5	Program Section Name
6	Program Version Identification
7	Mapped Array Declaration
10	Completion Routine Name

There are four words in the GSD record for each entry type. The first two words contain six Radix-50 characters. The third word contains a flag byte and the entry type identification. The fourth word contains additional information about the entry. See Figure B-2.

Figure B-2: GSD Record and Entry Format
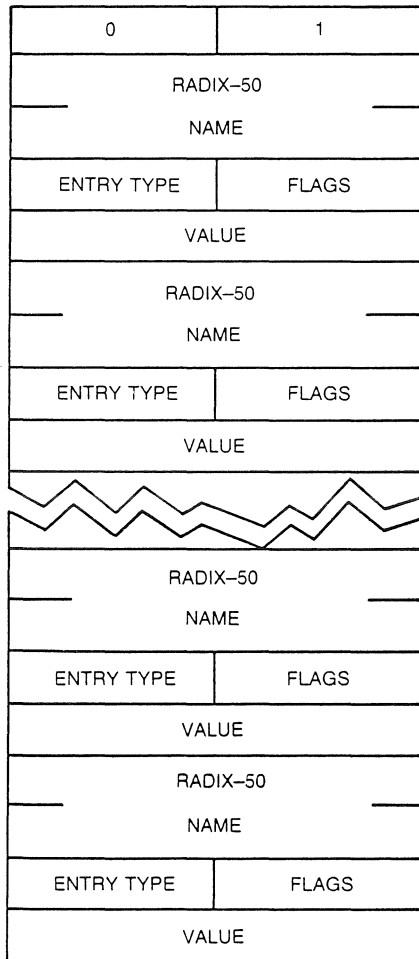
MK-01057-00

B.1.1 Module Name

The module name entry, as illustrated in Figure B–3, declares the name of the object module. The name need not be unique with respect to other object modules because modules are identified by file, not module name. Only one module name entry can occur in any given object module.

Figure B–3:   Module Name Entry Format

MK–01058–00

B.1.2   Control Section Name

Control sections, which include ASECTs, blank CSECTs, and named CSECTs, are supplanted by PSECTs. For compatibility with other systems, Task Builder processes ASECTs and both forms of CSECTs. Section B.1.6 details the entry generated for a PSECT statement. In terms of the PSECT directive, ASECT and CSECT statements can be defined as follows:

- For a blank CSECT, the PSECT definition is:

  `.PSECT ,LCL,REL,CON,RW,I,LOW`

- For a named CSECT, the PSECT definition is:

  `.PSECT name,GBL,REL,OVR,RW,I,LOW`

- For an ASECT, the PSECT definition is:

  `.PSECT . ABS.,GBL,ABS,I,OVR,RW,LOW`

ASECTs and CSECTs are processed by the Task Builder as PSECTs with the fixed attributes defined above. The entry generated for a control section is shown in Figure B–4.

Figure B–4: Control Section Name Entry Format

```
+-------------------------------+
|        CONTROL SECTION        |
|             NAME              |
+---------------+---------------+
|       1       |   (Ignored)   |
+---------------+---------------+
|        MAXIMUM LENGTH         |
+-------------------------------+
```

MK-01058-01

B.1.3 Internal Symbol Name

The internal symbol name entry declares the name of an internal symbol (with respect to the module). The Task Builder does not support internal symbol tables, so the detailed format of this entry is not defined (Figure B–5). Any internal symbol entry encountered while the Task Builder reads the GSD is ignored.

Figure B–5: Internal Symbol Name Entry Format

```
+-------------------------------+
|            SYMBOL             |
|             NAME              |
+---------------+---------------+
|       2       |       0       |
+---------------+---------------+
|           UNDEFINED           |
+-------------------------------+
```

MK-01059-00

B.1.4 Transfer Address

The transfer address entry, as illustrated in Figure B–6, declares the transfer address of a module relative to a PSECT. The first two words of the entry define the name of the PSECT, and the fourth word defines the relative offset from the beginning of that PSECT. If no transfer address is declared in a module, a transfer address entry either must not be included in the GSD or a transfer address 000001 relative to the default absolute PSECT (.ABS.) must be specified.

Figure B-6: Transfer Address Entry Format

```
┌──────────────────────────────────┐
│            PSECT                  │
│ ─────            ─────            │
│            NAME                   │
├───────────────┬──────────────────┤
│      3        │       0          │
│               │                  │
├───────────────┴──────────────────┤
│           OFFSET                  │
└──────────────────────────────────┘
```

MK-01059-01

NOTE

If the PSECT is absolute and OFFSET is not 000001, then OFFSET is the actual transfer address.

B.1.5 Global Symbol Name

The global symbol name entry, as illustrated in Figure B-7, declares either a global reference or a definition. All definition entries must appear after the declaration of the PSECT they are defined in and before the declaration of another PSECT. Global references can appear anywhere within the GSD.

Figure B-7: Global Symbol Name Entry Format

```
┌──────────────────────────────────┐
│           SYMBOL                  │
│ ─────            ─────            │
│            NAME                   │
├───────────────┬──────────────────┤
│      4        │     FLAGS         │
│               │                  │
├───────────────┴──────────────────┤
│           VALUE                   │
└──────────────────────────────────┘
```

MK-01059-02

The first two words of the entry define the name of the global symbol. The flag byte declares the attributes of the symbol, and the fourth word declares the value of the symbol relative to the PSECT it is defined in.

The flag byte of the symbol declaration entry has the following bit assignments:

Bit 0 - Weak Qualifier

> 0 = Symbol is a strong definition or reference and is resolved in the normal manner.
>
> 1 = Symbol is a weak definition or reference. A weak reference (Bit 3 = 0) is ignored. A weak definition (Bit 3 = 1) is ignored unless a previous reference has been made.

Bit 1 - Not used

Bit 2 - Definition Type

> 0 = Normal Definition of reference.
>
> 1 = Library definition. If the symbol is defined in a resident library .STB file, the base address of the library is added to the value, and the symbol is converted to absolute (bit 5 is reset); otherwise, the bit is ignored.

Bit 3 - Reference or Definition

> 0 = Global symbol reference.
>
> 1 = Global symbol definition.

Bit 4 - Not used

Bit 5 - Relocation

> 0 = Absolute symbol value.
>
> 1 = Relative symbol value.

Bit 6-7 - Not used

B.1.6   PSECT Name

The PSECT name entry, as illustrated in Figure B–8, declares the name of a PSECT and its maximum length in the module. It also declares the attributes of the PSECT in the flag byte.

Figure B–8: PSECT Name Entry Format
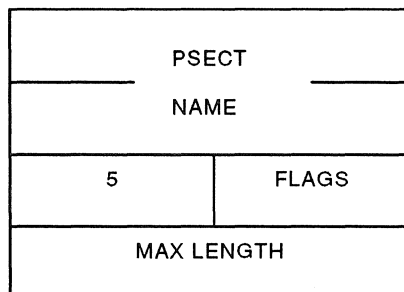
MK–01060–00

GSD records must be constructed such that, once a PSECT name has been declared, all global symbol definitions pertaining to it must appear before another PSECT name is declared. Global symbols are declared in symbol declaration entries. Thus, the normal format is a series of PSECT names each followed by optional symbol declarations.

The flag byte of the PSECT entry has the following bit assignments:

Bit 0 - SAV PSECT

    0   =   Normal PSECT.

    1   =   PSECT is forced into the root of the task.

Bit 1 - Library PSECT

    0   =   Normal PSECT.

    1   =   Relocatable PSECT that references a resident library or common block.

Bit 2 - Allocation

    0   =   PSECT references are to be concatenated with other references to the same PSECT to form the total memory allocated to the PSECT.

    1   =   PSECT references are to be overlaid. The total memory allocated to the PSECT is the largest request made by individual references to the same PSECT.

Bit 3 - Reserved for the Task Builder

Bit 4 - Access

    0   =   PSECT has read/write access.

    1   =   PSECT has read-only access.

Bit 5 - Relocation

    0   =   PSECT is absolute and requires no relocation.

    1   =   PSECT is relocatable, and references to the control PSECT must have a relocation bias added before they become absolute.

Bit 6 - Scope

    0   =   The scope of the PSECT is local. References to the PSECT are collected only within the segment in which the PSECT is defined.

    1   =   The scope of the PSECT is global. References to the PSECT are collected across segment boundaries. The segment in which a global PSECT is allocated storage is determined either by the first module that defines the PSECT on a path or by direct placement of a PSECT in a segment by the .PSECT directive.

Bit 7 - Type

    0   =   The PSECT contains instruction (I) references.

    1   =   The PSECT contains data (D) references.

NOTE

The length of all absolute PSECTs is zero.

B.1.7 Program Version Identification

The program version identification entry, as illustrated in Figure B–9, declares the version of the module. The Task Builder saves the version identification of the first module that defines a nonblank version. This identification is then included on the memory allocation map and is written in the label block of the task image file.

The first two words of the entry contain the version identification. The flag byte and fourth words are not used and contain no meaningful information.

Figure B–9: Program Version Identification Entry Format

```
+-------------------------------------+
|                                     |
|              VERSION                |
|      ----              ----         |
|            IDENTIFICATION           |
|                                     |
+------------------+------------------+
|                  |                  |
|        6         |        0         |
|                  |                  |
+------------------+------------------+
|                                     |
|                 0                   |
|                                     |
+-------------------------------------+
```
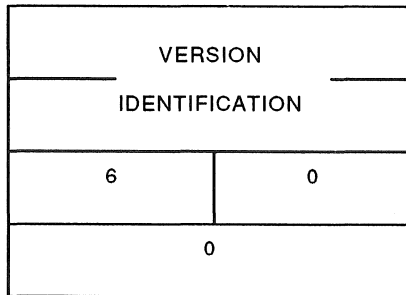
MK–01060–01

B.1.8 Mapped Array Declaration (Type 7)

The mapped array declaration entry allocates space within the mapped array area of task memory. The array name is added to the list of task program section names and may be referred to by subsequent RLD records. The length (in units of 64-byte blocks) is added to the task's mapped array location. The total memory allocated to each mapped array is rounded up to the nearest 512-byte boundary. The contents of the flag byte are reserved and assumed to be 0.

One additional window block is allocated whenever a mapped array is declared. Figure B-10 illustrates the mapped array declaration entry format.

Figure B-10:   Mapped Array Declaration Entry Format

```
┌─────────────────────────────────────────────────────────┐
│                      MAPPED ARRAY                         │
├─────────────────────────────────────────────────────────┤
│                         NAME                              │
├──────────────────────────┬──────────────────────────────┤
│      ENTRY               │                                │
│      TYPE   = 7          │           FLAGS                │
├──────────────────────────┴──────────────────────────────┤
│         LENGTH (NUMBER OF 64-BYTE BLOCKS)                 │
└─────────────────────────────────────────────────────────┘
```

B.1.9 Completion Routine Definition (Type 10)

The completion routine defintion declares the entry point for completion routine of a supervisor-mode library. The data structure is created by the Task Builder and appears only in symbol definition files of supervisor mode libraries.

As shown in figure B-11, the first two words of the entry define the name of the entry point. The third word contains the entry type byte and the flag byte. The flag byte contains no meaningful information. The fourth word contains the symbol value.

Figure B-11:   Completion Routine Entry Format

```
┌─────────────────────────────────────────────────────────┐
│                 COMPLETION ROUTINE                        │
│                       NAME                                │
├──────────────────────────┬──────────────────────────────┤
│      ENTRY               │                                │
│      TYPE   = 10         │              0                 │
├──────────────────────────┴──────────────────────────────┤
│                       VALUE                               │
└─────────────────────────────────────────────────────────┘
```

B.2 End of Global Symbol Directory

The end-of-global-symbol-directory record, as illustrated in Figure B–12, declares that no other GSD records are contained farther on in the module. Exactly one end-of-GSD record must appear in an object module. Its length is one word.

Figure B–12: End-of-GSD Record Format

0	2
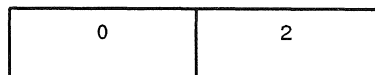
MK–01060–02
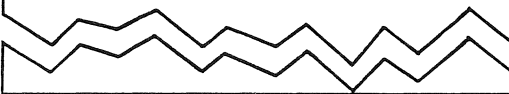
B.3 Text Information

The text information record, as illustrated in Figure B–13, contains a byte string of information that is to be written directly into the task image file. The record consists of a load address followed by the byte string.

Text records can contain words or bytes or a combination of both of information whose final contents have not been determined yet. This information will be bound by a record (see Section B.4). If the test record does not need modification, then no relocation directory record is needed. Thus, multiple text records can appear in sequence before a relocation directory record.

The load address of the text record is specified as an offset from the current PSECT base. At least one relocation directory record must precede the first text record. This directory must declare the current PSECT.

Figure B-13: Text Information Record Format

0	3
LOAD ADDRESS	
TEXT	TEXT
TEXT	TEXT
TEXT	TEXT
⌇⌇⌇⌇	⌇⌇⌇⌇
TEXT	TEXT
TEXT	TEXT
TEXT	TEXT
TEXT	TEXT
TEXT	TEXT

MK-01061-00

The Task Builder writes a text record directly into the task image file and computes the value of the load address minus four. This value is stored in anticipation of a subsequent relocation directory that modifies words and bytes that are contained in the test record. When added to a relocation directory displacement byte, this value yields the address of the word and byte to be modified in the task image.

B.4 Relocation Directory

Relocation directory records (see Figure B–14) contain the information necessary to relocate and link the preceding text information record. Every module must have at least one relocation directory record that precedes the first text information record. The first record does not modify a preceding text record but rather defines the current PSECT and location. Relocation directory records contain 15 types of entries. These entries are classified as relocation or location modification entries. Table 7-2 lists the defined types.

Table B–2: Types of Entries for Relocation Directory Records

Type$_8$	Definition
1	Internal Relocation
2	Global Relocation
3	Internal Displaced Relocation
4	Global Displaced Relocation
5	Global Additive Relocation
6	Global Additive Displaced Relocation
7	Location Counter Definition
10	Location Counter Modification
11	Program Limits
12	PSECT Relocation
13	Not used
14	PSECT Displaced Relocation
15	PSECT Additive Relocation
16	PSECT Additive Displaced Relocation
17	Complex Relocation
20	Additive Relocation

Each type of entry is represented by a command byte (specifies type of entry and word/byte modification), followed by a displacement byte, and then by the information required for the particular type of entry. The displacement byte, when added to the value calculated from the load address of the preceding text information record (see Section B.3), yields the virtual address in the image that is to be modified. The command byte of each entry has the following bit assignments:

Bit 0 - 6

> Specify the type of entry. Potentially, the 128 command types can be specified, although only 15_{10} are implemented.

Bit 7 - Modification

> 0 = The command modifies an entire word.
>
> 1 = The command modifies only one byte. The Task Builder checks for truncation errors in byte modification commands. If truncation is detected, that is, if the modification value has a magnitude greater than 255, an error occurs.

Figure B–14:   Relocation Directory Record Format

0	4
DISP	CMD
INFO	INFO
INFO	INFO
~	~
CMD	INFO
INFO	DISP
INFO	INFO
''	''
''	''
''	''
INFO	INFO
DISP	CMD
INFO	INFO
INFO	INFO
INFO	INFO

MK–01062–00

B.4.1   Internal Relocation

The internal relocation entry illustrated in Figure B–15 relocates a direct pointer to an address within a module. The current PSECT base address is added to a specified constant, and the result is written into the task image file at the calculated address. (That is, a displacement byte is added to the value calculated from the load address of the preceding text block.)

For example:

```
A:      MOV         #A,RO

        or

        .WORD       A
```

Figure B–15: Internal Relocation Entry Format

DISP	B	1
CONSTANT		
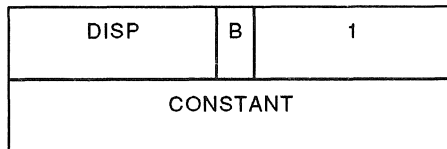
MK–01063–00

B.4.2 Global Relocation

The global relocation entry in Figure B–16 relocates a direct pointer to a global symbol. The definition of the global symbol is obtained and the result is written into the task image file at the calculated address.

For example:

```
        MOV         #GLOBAL,RO

        or

        .WORD       GLOBAL
```

Figure B–16: Global Relocation Entry Format

DISP	B	2
SYMBOL		
NAME		
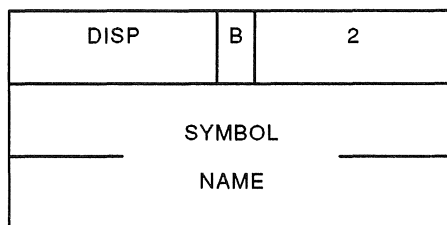
MK–01063–01

B.4.3 Internal Displaced Relocation

The internal displaced relocation entry in Figure B–17 relocates a relative reference to an absolute address from within a relocatable control section. The address plus 2 that the relocated value is to be written into is subtracted from the specified constant. The result is then written into the task image file at the calculated address.

For example:

```
        CLR              177550
    or
        MOV              177550,RO
```

Figure B-17: Internal Displaced Relocation Entry Format

DISP	B	3
CONSTANT		
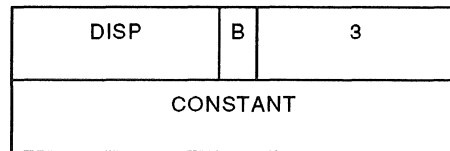
MK-01063-02

B.4.4 Global Displaced Relocation

The global displaced relocation entry in Figure B-18 relocates a relative reference to a global symbol. The definition of the global symbol is obtained, and the address plus 2 that the relocated value is to be written into is subtracted from the definition value. The result is then written into the task image file at the calculated address.

For example:

```
        CLR              GLOBAL
    or
        MOV              GLOBAL,RO
```

Figure B-18: Global Displaced Relocation Entry Format

DISP	B	4
SYMBOL		
NAME		
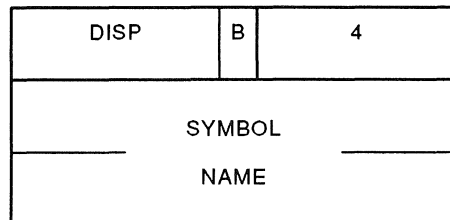
MK-01063-03

B.4.5 Global Additive Relocation

The global additive relocation entry in Figure B–19 relocates a direct pointer to a global symbol with an additive constant. The definition of the global symbol is obtained, the specified constant is added, and the resultant value is then written into the task image file at the calculated address.

For example:

```
        MOV             #GLOBAL+2,RO

    or

        .WORD           GLOBAL-4
```

Figure B–19: Global Additive Relocation Entry Format
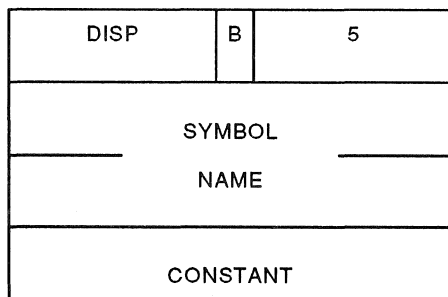

```
MK-01064-00
```

B.4.6   Global Additive Displaced Relocation

The global additive displaced relocation entry in Figure B–20 relocates a relative reference to a global symbol with an additive constant. The definition of the global symbol is obtained, and the specified constant is added to the definition value. The address plus 2 that the relocated value is to be written into is subtracted from the resultant additive value. The result is then written into the task image file at the calculated address.

For example:

```
        CLR             GLOBAL+2

    or

        MOV             GLOBAL-5,RO
```

Figure B-20:   Global Additive Displaced Relocation Entry Format

```
+----------------+---+-----------------+
|      DISP      | B |        6        |
+----------------+---+-----------------+
|                                      |
|              SYMBOL                  |
|                                      |
|              NAME                    |
|                                      |
+--------------------------------------+
|                                      |
|            CONSTANT                  |
+--------------------------------------+
```

MK-01064-01

B.4.7   Location Counter Definition

The location counter definition in Figure B-21 declares a current PSECT and location counter value. The control base is stored as the current control section, and the current control section base is added to the specified constant and stored as the current location counter value.

Figure B-21:   Location Counter Definition

```
+----------------+---+-----------------+
|       0        | B |        7        |
+----------------+---+-----------------+
|                                      |
|              PSECT                   |
|                                      |
|              NAME                    |
|                                      |
+--------------------------------------+
|                                      |
|            CONSTANT                  |
+--------------------------------------+
```

MK-01064-02

B.4.8 Location Counter Modification

The location counter modification entry in Figure B-22 modifies the current location counter. The current PSECT base is added to the specified constant and the result is stored as the current location counter.

For example:

```
.=.+N
```

    or

```
.BLKB          N
```

Figure B-22: Location Counter Modification

0	B	10
CONSTANT		
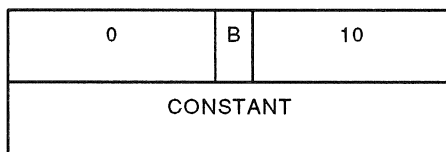
MK-01065-00

B.4.9 Program Limits

The program limits entry in Figure B-23 is generated by the .LIMIT assembler directive. The first address above the header (normally the beginning of the stack) and the highest address allocated to the task are obtained and written into the task image file at the calculated address and at the calculated address plus 2 respectively.

For example:

```
.LIMIT
```

Figure B-23: Program Limits Entry Format

DISP	B	11
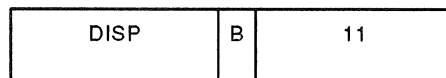
MK-01065-01

B.4.10 PSECT Relocation

The PSECT relocation entry in Figure B–24 relocates a direct pointer to the beginning address of another PSECT (other than the PSECT in which the reference is made) within a module. The current base address of the specified PSECT is obtained and written into the task image file at the calculated address.

For example:

```
        .PSECT    A
B:
        .
        .
        .
        .PSECT    C
        MOV       #B,RO
```

or

```
        .WORD     B
```

Figure B–24: PSECT Relocation Entry Format


```
+-------------+---+-------------+
|    DISP     | B |     12      |
+-------------+---+-------------+
|          PSECT              |
|          NAME               |
+-----------------------------+
```
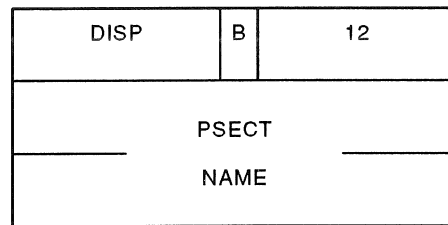
MK–01065–02

B.4.11 PSECT Displaced Relocation

The PSECT displaced relocation entry in Figure B–25 relocates a relative reference to the beginning address of another PSECT within a module. The current base address of the specified PSECT is obtained and the address plus 2 that the relocated value is to be written into is subtracted from the base value. The result is then written into the task image file at the calculated address.

For example:

```
        .PSECT    A
B:
        .
        .
        .
        .PSECT    C
        MOV       B,RO
```

Figure B-25: PSECT Displaced Relocation Entry Format

DISP	B	14
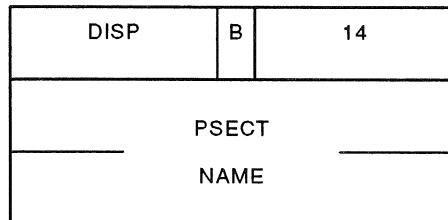
PSECT
NAME

MK-01066-00

B.4.12 PSECT Additive Relocation

The PSECT additive relocation entry in Figure B-26 relocates a direct pointer to an address in another PSECT within a module. The current base address of the specified PSECT is obtained and added to the specified constant. The result is written into the task image file at the calculated address.

For example:

```
        .PSECT   A
B:

        .
        .
        .
C:

        .
        .

        .
        .PSECT   D
        MOV      #B+10,RO
        MOV      #C,RO

    or

        .WORD    B+10
        .WORD    C
```

Figure B-26:   PSECT Additive Relocation Entry Format

```
┌─────────────────┬───┬───────────────┐
│      DISP       │ B │      15       │
├─────────────────┴───┴───────────────┤
│               PSECT                  │
│               NAME                   │
├──────────────────────────────────────┤
│             CONSTANT                 │
└──────────────────────────────────────┘
```
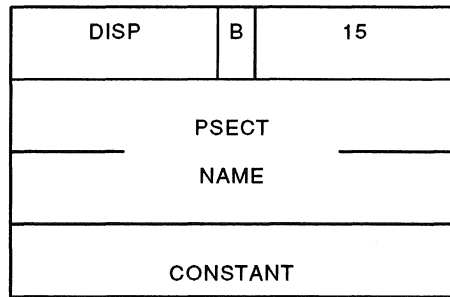
MK-01066-01

B.4.13   PSECT Additive Displaced Relocation

The PSECT additive displaced relocation entry in Figure B-27 relocates a relative reference to an address in another PSECT within a module. The current base address of the specified PSECT is obtained and added to the specified constant. The address plus 2 that the relocated value is to be written into is subtracted from the resultant additive value. The result is then written into the task image file at the calculated address.

For example:

```
        .PSECT   A
B:
        .
        .
        .
C:
        .
        .
        .
        .PSECT   D
        MOV      B+10,RO
        MOV      C,RO
```

```
+------------------+---+-----------+
|     DISP         | B |    16     |
+------------------+---+-----------+
|                              |
|            PSECT             |
+------                  ------+
|            NAME              |
|                              |
+------------------------------+
|          CONSTANT            |
+------------------------------+
```
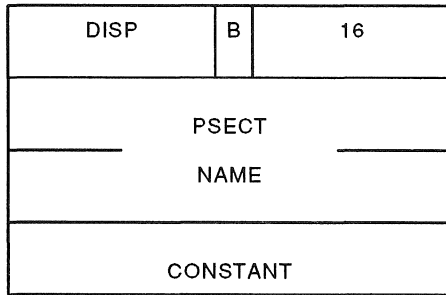
MK-01066-02

B.4.14   Complex Relocation

The complex relocation entry in Figure B–28 resolves a complex relocation expression. In such an expression, any of the MACRO-11 binary or unary operations are permitted. Any type of argument is permitted, regardless of whether the argument is unresolved global, relocatable to any PSECT base, absolute, or a complex relocatable subexpression.

The RLD command word is followed by a string of numerically-specified operation codes and arguments. Each operation code occupies one byte. The entire RLD command must fit in a single record. Table B-3 lists the defined operation codes.

Table B–3:   Defined Operation Codes for the RLD Command Word

0	No operation.
1	Addition (+).
2	Subtraction (-).
3	Multiplication (*).
4	Division (/).
5	Logical AND (&).
6	Logical inclusive OR (!).
10	Negation (-).
11	Complement ^C.
12	Store result (command termination).
13	Store result with displaced relocation (command termination).
16	Fetch global symbol. It is followed by four bytes containing the symbol name in Radix-50 representation.
17	Fetch relocatable value. It is followed by one byte containing the sector number and two bytes containing the offset within the sector.
20	Fetch constant. It is followed by two bytes containing the constant.

(continued on next page)

21	Fetch resident library base address. If the file is a resident library .STB file, the library base address is obtained; otherwise, the base address of the Task Image is fetched.

The STORE commands indicate that the value is to be written into the task image file at the calculated address.

All operands are evaluated as 16-bit signed quantities using two's complement arithmetic. The results are equivalent to expressions that are evaluated internally by the assembler. Note the following rules:

1. An attempt to divide by zero yields a zero result. The Task Builder issues a nonfatal diagnostic message.

2. All results are truncated from the left in order to fit into 16 bits. No diagnostic message is issued if the number was too large. If the result modifies a byte, the Task Builder checks for truncation errors as described in Section B.4.

3. All operations are performed on relocated (additive) or absolute 16-bit quantities. PC displacement is applied to the result only.

For example:

```
        .PSECT ALPHA
A:
        .
        .
        .
        .PSECT BETA
B:
        .
        .
        .
        MOV     #A+B -<G1/G2&^C<177120!G3>>,R1
```

Figure B–28: Complex Relocation Entry Format


```
┌──────────────┬───┬──────────┐
│    DISP      │ B │    17    │
├──────────────┴───┴──────────┤
│          COMPLEX            │
├─────────────────────────────┤
│                             │
│                             │
├─────────────────────────────┤
│          STRING             │
├──────────────┬──────────────┤
│  12 or  13   │              │
└──────────────┴──────────────┘
```
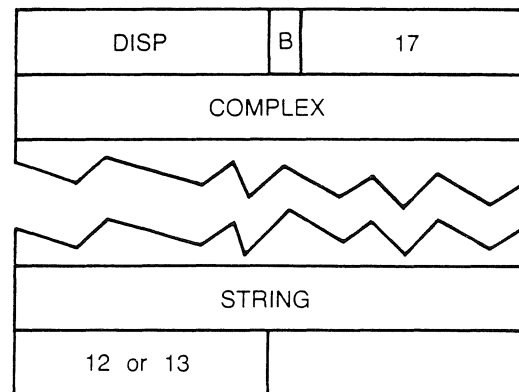
MK–01067–00

B.4.15 Additive Relocation

The shared run-time system (SRTS) additive relocation entry in Figure B-29 relocates a direct pointer to an address within an SRTS.

If the current file is a symbol table file (STB), the base address of the SRTS is obtained and added to the specified constant. The result is written into the task image file at the calculated address. If the file is not associated with an SRTS, the task base address is used.

Figure B-29: Additive Relocation Entry Format

DISP	B	20
CONSTANT		
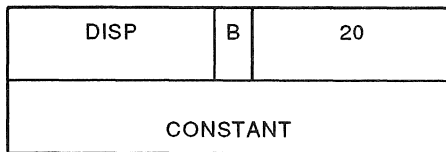
MK-01067-01

B.5   Internal Symbol Directory Record

The Internal Symbol Directory (ISD) record declares definitions of all symbols that are defined in the module. In addition to looking for global symbol definitions in the input object modules, TKB must look for ISD records. Some of these require no relocation and TKB can copy them directly into the .STB file. Others will require modification; after being modified, ISD records can be written to the .STB file. In addition, TKB may need to generate some ISD records of its own in the .STB file.

Except for autoloadable library entry points, TKB puts ISD records into the .STB file only if the /DA switch is used in the TKB command line. When TKB outputs the .STB file, it writes one of three major types of ISD records:

* Type 1 records, where TKB generates ISDs in language-independent form.

* Type 3 records, written for any type 2 records in an input object module. TKB does this after adding data and then changing the ISD record type to language-dependent and independent sections. Language processors generate these records and TKB modifies them. They contain information that can be used to find the absolute task image address of source program entities (for example, variables, program statements, and so on).

* Type 4 records, written to the .STB file without modification. Type 4 records are literal records that contain language-dependent information. Apart from the first few bytes, TKB ignores the rest of the record.

The following sections describe the record formats.

B.5.1 Overall Record Format

ISD records have the same basic structure as all object language records. Because of the variety of different types, the skeleton structure must include additional fields that are common to all ISD record types. The general format of all ISD records is shown in Figure B–30.

Figure B–30: General Format of All ISD Records

MUST BE 0	RECORD TYPE = 5
RESERVED (0)	ISD RECORD TYPE
RECORD TYPE DEPENDENT	

MK–01068–00

ISD record types fall into these general categories:

0	Illegal.
1	TKB-generated.
2	Compiler-generated relocatable.
3	Relocated (type 2 after TKB processing).
4-127	Not defined, reserved for future use.
128-255	Literal records. (The type code identifies the generating language processor, and thus, the internal structure.)

B.5.2 TKB-Generated Records (Type 1)

The content of this record type is a string of individual items, each with its own format. The items are either start-of-segment items, task identification items, or autoloadable entry point items. The TKB-generated record is similar to the structure of an RLD or GSD record. The general format is shown in Figure B–31.

Figure B–31:   General Format of a TKB-Generated Record

```
┌─────────────────────┬───────────────┐
│   LENGTH (BYTES)     │   ITEM TYPE   │
├─────────────────────┴───────────────┤
│                                      │
│    CONTENT DEPENDS ON ITEM TYPE      │
│                                      │
└──────────────────────────────────────┘
```

MK–01069–00

B.5.2.1   Start-of-Segment Item (Type 1)

The format of the start-of-segment item type is shown in Figure B–32.

Figure B–32:   Format of TKB-Generated Start-of-Segment Item (1)

```
┌─────────────────────┬───────────────┐
│    LENGTH = 8        │  ITEM TYPE = 1│
├─────────────────────┴───────────────┤
│                                      │
│ ──────── SEGMENT NAME ───────        │
│                                      │
├──────────────────────────────────────┤
│   SEGMENT DESCRIPTOR ADDRESS          │
└──────────────────────────────────────┘
```

MK–01070–00

B.5.2.2 Task Identification Item (Type 2)

The task identification item type ensures that an .STB file and the task image being debugged were generated at the same time. Otherwise, symbols that are found may not correspond to the actual task.

The task identification item type exists to make the correlation between the .STB file and its related task possible. The contents of this item type correspond exactly to the first ten words of an area in a task image file, which is in the TKB-created PSECT called $$DBTS.

The format of the task identification item type is shown in Figure B–33.

Figure B–33: Format of TKB-Generated Task Identification Item (2)

LENGTH = 22	ITEM TYPE = 2
EIGHT–WORD TIME STAMP (1)	
TWO–WORD NUMBER (2)	

(1) Its form is that which is returned by RSX directive GTM$.

(2) TKB generates this number as an additional check on correspondence. Currently always zero.

MK–01071–00

B.5.2.3 Autoloadable Library Entry Point Item (Type 3)

TKB outputs the autoloadable library entry point item into an .STB file when building overlaid resident libraries. The ISD record contains the information needed by TKB to dynamically generate autoload vectors for entry points in the library. Autoload vectors appear for only those entry points that are referenced by the task. Unlike the other items, the autoloadable library entry point item is not for use by debuggers.

The format of the autoloadable entry point item is shown in Figure B–34.

Figure B–34: Format of an Autoloadable Library Entry Point Item (3)

```
+---------------------------+---------------------------+
|      LENGTH = 12          |      ITEM TYPE = 3        |
+---------------------------+---------------------------+
|                      SYMBOL                           |
|                                                       |
|                       NAME                            |
+---------------------------+---------------------------+
|           0               |      FLAGS BYTE           |
+---------------------------+---------------------------+
|      ENTRY POINT OFFSET FROM LIBRARY BASE             |
+-------------------------------------------------------+
|      SEGMENT DESCRIPTOR OFFSET IN $$SGD1              |
+-------------------------------------------------------+
```

MK–01072–00

B.5.3 Relocatable/Relocated Records (Type 2)

These records are the central part of TKB's involvement in debugger communication. Every item type in these records must be standardized, and only standard items can appear. The general format is the same as that shown in Figure B–30.

A language processor outputs these record types as type 2. When TKB processes them, it changes the type to type 3. It also fills in or modifies some fields. In the following descriptions, fields that are filled in by TKB are marked with an asterisk (*). They should be left as zero in language processor output.

B.5.3.1 Module Name Item (Type 1)

A module name item should be the first ISD entry of each object module. A debugger can assume that all following ISD information up to the next module name item relates to this module.

The language code is included so that a debugger for a specific language can determine whether to ignore a module if it is written for another language. The language code has the same range of values as that of a language-dependent ISD record (128-255) and has the same meanings.

The format of the module name item type is shown in Figure B–35.

Figure B–35: Format of a Module Name Item (Type 1)

LENGTH	ITEM TYPE = 1
MUST BE 0	LANGUAGE CODE
MODULE NAME (1)	

(1) A counted ASCII string of the required length. (A counted ASCII string is a byte in which the first byte indicates the number of bytes to follow.)

MK–01073–00

B.5.3.2 Global Symbol Item (Type 2)

One type 2 item should appear for each global symbol definition that the language processor wants the debugger to understand. It need not, for example, include definitions generated for the language processor run-time system.

The format of the global symbol item type is shown in Figure B–36.

Figure B–36:   Format of a Global Symbol Item (Type 2)

```
+----------------------+----------------------+
|        LENGTH        |    ITEM TYPE = 2     |
+----------------------+----------------------+
|                                             |
|              SYMBOL NAME                    |
|                                             |
|              (RADIX–50)                     |
+---------------------------------------------+
|                 VALUE*                      |
+---------------------------------------------+
|    DESCRIPTOR ADDRESS FOR CONTAINING        |
|           OVERLAY SEGMENT*                   |
+----------------------+----------------------+
|    MUST BE ZERO      |       FLAGS          |
+----------------------+----------------------+
|          FULL SYMBOL NAME (1)               |
|                                             |
+---------------------------------------------+
```

(1) Counted ASCII string of the required length.
(A counted ASCII string is a byte string in
which the first byte indicates the number of
bytes to follow.)

MK–01074–00

B.5.3.3 PSECT Item (Type 3)

A concatenated PSECT has two base addresses: one for the whole PSECT, and the other for the part of it that belongs to this module. It is the base address for the part that belongs to this module that may be used by a debugger to convert local symbol values to absolute addresses.

The segment descriptor address is necessary because a PSECT may move to segments other than the one in which it is placed. This address is relevant to languages that provide semi-automatic overlay generation, like COBOL-11. This word may be zero if the PSECT has not moved to another segment.

The flag word is a copy of the flag word built by TKB. It allows for identification of VSECTs.

Some languages may need the full PSECT name.

The format of a PSECT item type is shown in Figure B-37.

Figure B-37: Format of a PSECT Item (Type 3)

LENGTH	ITEM TYPE = 3
PSECT NAME	
BASE ADDRESS OF PSECT IN THIS SEGMENT*	
BASE ADDRESS OF PSECT FOR THIS MODULE*	
LENGTH OF PSECT FOR THIS MODULE*	
DESCRIPTOR ADDRESS FOR CONTAINING SEGMENT*	
FLAG WORD*	
FULL PSECT NAME (1)	

(1) Counted ASCII string of the required length.
(A counted ASCII string is a byte string in
which the first byte indicates the number of
bytes to follow.)

MK-01075-00

B.5.3.4 Line-number or PC Correlation Item (Type 4)

This item provides the information needed to translate a source line-number into a task image address, or a task image address into a source line-number.

The format of a line-number of PC correlation item type is shown in Figure B–38.

Figure B–38: Format of a Line-Number or PC Correlation Item (Type 4)

LENGTH	ITEM TYPE = 4
PSECT NAME	
START PC (1)	
DESCRIPTOR ADDRESS OF CONTAINING OVERLAY SEGMENT*	
START PAGE NUMBER	
START LINE NUMBER	
STRING OF ONE-BYTE ITEMS	

(1) Offset into PSECT in type 2 records;
absolute address in type 3 records.

MK–01076–00

B.5.3.5 Internal Symbol Name Item (Type 5)

It is necessary to allow for the fact that a name may have more than one associated address. For example, a COBOL variable may have three associated addresses: the address of the area that actually contains the data, the address of a CIS descriptor, and the address of a picture string.

The internal symbol name item, which meets these requirements, is shown in Figure B–39.

Figure B–39:   Format of an Internal Symbol Name Item (Type 5)

LENGTH	ITEM TYPE = 5
OFFSET TO NAME	OFFSET TO DATA
MUST BE ZERO	NUMBER OF ADDRESSES

ADDRESS 1:

PSECT
NAME
TASK IMAGE ADDRESS/OFFSET (1)
SEGMENT DESCRIPTOR ADDRESS*

ADDRESS 2:

PSECT
NAME
TASK IMAGE ADDRESS/OFFSET (1)
SEGMENT DESCRIPTOR ADDRESS*

ADDRESS n: .

LANGUAGE–DEPENDENT DATA
SYMBOL NAME (2)
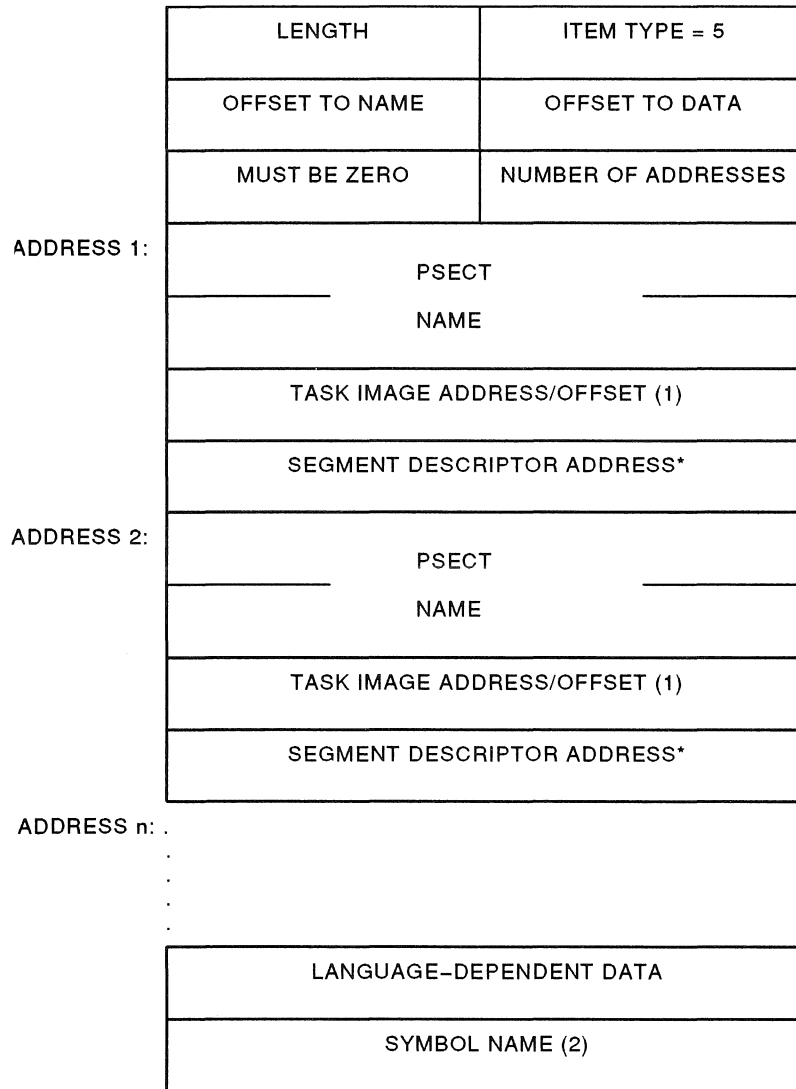
(1) Modified by TKB.

(2) A counted ASCII string of the required length. (A counted ASCII string is a byte string in which the first byte indicates the number of bytes to follow.

MK–01077–00

B.5.4 Literal Records (Type 4)

Literal records may take any form except for the two-byte header shown in Figure B–40.

Figure B–40: Format of a Literal Record Type

RESERVED (0)	ISD RECORD TYPE 4

MK–01078–00
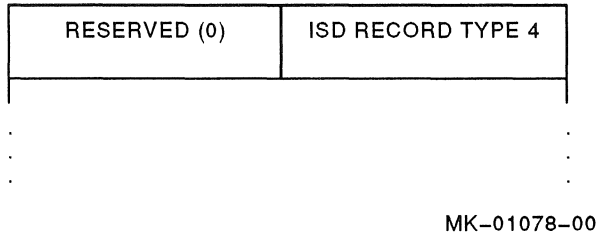
B.6 End of Module

The end-of-module record in Figure B–41 declares the end of an object module. Exactly one end-of-module record must appear in each object module. It is one word in length.

Figure B–41: End-of-Module Record Format

0	6

MK–01067–02
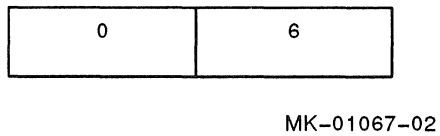
Executable File Structure

This appendix describes the elements of an executable file.

The executable file as it is recorded on the disk appears in Figure C–1.

Figure C–1: Task Image on Disk

AUTOLOAD VECTORS CO–TREE OVERLAY	BLOCK
AUTOLOAD VECTORS CO–TREE ROOT	BLOCK
AUTOLOAD VECTORS	
MAIN TREE OVERLAY	BLOCK
AUTOLOAD VECTORS SEGMENT TABLES	
ROOT SEGMENT CODE & DATA	
STACK FP/EA SAVE AREA HEADER	BLOCK
CHECKPOINT AREA	
LABEL (Block 0 of the file)	BLOCK

MK–01079–00

C.1 Label Block Group

The label block group shown in Figure C–2 precedes the task on the disk and contains data that need not be resident during task execution. This group is composed of two elements:

- Task and resident library data (Label Block 0)

- Table of LUN assignments (Label Block 1) which contains the name and logical unit number of each device assigned

Figure C–2: Label Block Group

Label	Non–I&D Tasks/ID Tasks	Offset *		
L$BTSK		0	Task	
		2	Name	
L$BPAR		4	Task	
		6	Partition	
L$BSA		10	Base address of task	
L$BHGV		12	Highest window 0 virtual address	
L$BMXV		14	Highest virtual address in task	
L$BLDZ		16	Load size in 64–byte blocks	
L$BMXZ		20	Maximum size in 64–byte blocks	
L$BOFF		22	Task offset into partition	
L$BWND/L$BSYS		24	System ID | Number of window blocks*	
L$BSEG		26	Size of overlay segment descriptors	
L$BFLG		30	Task flag word	
L$BDAT		32	Task creation date –Year	
		34	– Month	
		36	– Day	
L$BLIB		40	Library/common	
		42	Name	R$LNAM
		44	Base address of library	R$LSA
		46	Highest address in first library window	R$LHGV
		50	Highest address in library	R$LMXV
		52	Library load size (64–byte blocks)	R$LLDZ
		54	Library maximum size (64–byte blocks)	R$MXZ
		56	Library offset into region	R$LOFF
		60	Number of library window blocks	R$LWND
		62	Size of library segment descriptors	R$LSEG
		64	Library flag word	R$LFLG
		66	Library creation date – Year	R$LDAT
		70	– Month	
		72	– Day	
		:	:	
		344/704	0	
L$BPRI		346/706	Task priority	
L$BXFR		350/710	Task transfer address	
L$BEXT		352/712	Task extension (64–byte blocks)	
L$BSGL		354/714	Block number of segment load list	
L$BHRB		356/716	Block number of header	
L$BBLK		360/720	Number of blocks in label	
L$BLUN		362/722	Number of logical units	
L$BROB		364/724	Relative block of R–O image	
L$BROL		366/726	R/O load size	
L$BRDL		370/730	R/O data size in 32–word blocks	
L$BHDB		372/732	Relative block number of data header	
L$BDHV		374/734	High virtual address of data window 1	
L$BDMV		376/736	High virtual address of data	
L$BDLZ		400/740	Load size of data	
L$BDMZ		402/742	Maximum size of data	
L$BAPR		404/744	APR mask word	
L$DAPR		404/772	Second task flag word	
L$BFLZ		404/774	Label block revision number	
L$BLRL		404/776	AME (must be 0)	

(Library Request (maximum of 7 or 15 14–word entries))

* Less library window blocks.

* If the image is a SIL (output of MAKSIL program), add 1000 (octal) to these values. Also, location 776 (no 1000 added)=SIL in Radix–50.

Task and resident library data are described in Table C-1.

Table C-1: Task and Resident Library Data

L$BTSK	Task name consisting of two words in Radix-50 format. This parameter is set by the TASK keyword.
L$BPAR	Partition name consisting of two words in Radix-50 format. This parameter is set by the PAR keyword.
L$BSA	Starting address of task. Marks the lowest task virtual address. This parameter is set by the PAR keyword.
L$BHGV	Highest virtual address mapped by address window 0.
L$BMXV	Highest task virtual address. This value is set to L$BHGV.
L$BLDZ	Task load size in units of 64-byte blocks. This value represents the size of the root segment.
L$BMXZ	Task minimum size in units of 64-byte blocks. This value represents the size of the root segment plus any additional physical memory needed to contain task overlays.
L$BOFF	Task offset into partition in units of 64-byte blocks.
L$BWND	Number of task windows (less windows of declared libraries (SRTS))- Low byte.
L$BSYS	System I.D.- High byte 1 = RSX-11M. 4 = RSX-11M-PLUS.
L$BSEG	Size of overlay segment descriptors (in bytes).
L$BFLG	Task flags word. The following flags are defined:

Bit	Flag	Meaning When Bit = 1
15	TS$PIC	Task contains position-independent code (PIC).
14	TS$NHD	Task has no header.
12	TS$PMD	Task generates Postmortem Dump.
7	TS$CMP	Task is built in compatibility mode.
6	TS$CHK	Task is not check-pointable (not supported on RSTS/E).
5	TS$RES	Task has memory-resident overlays.
3	TS$SUP	Image linked as supervisor-mode library.
0	TS$NEW	New header format L$BFL2 and after are valid.

L$BDAT	Three words containing the task creation date as two-digit integer values: Year (since 1900) Month of year Day of month
L$BLIB	Resident library entries.
L$BPRI	Task priority set by the PRI keyword. Ignored by RSTS/E.
L$BXFR	Task transfer address. (Not used by RSTS/E.)
L$BEXT	Task extension size in units of 64-byte blocks. This parameter is set by the EXTTSK keyword.
L$BSGL	Relative block number of segment load list. Set to zero if no list is allocated.

(continued on next page)

Table C–1 (Cont.):   Task and Resident Library Data

L$BHRB	Relative block number of header.
L$BBLK	Number of blocks in label block group.
L$BLUN	Number of logical units.
L$BROB	Relative block number of R/O image.
L$BROL	R/O load size in 32-word blocks.
L$BRDL	Size of R/O data in 32-word blocks.
L$BHDB	Relative block number of data header.
L$BDHV	High virtual address of data window 1 of D-space task.
L$BDMV	High virtual address of data.
L$BDLZ	Load size of data.
L$BDMZ	Maximum size of data.
L$BAPR	The APR mask word.
L$BFL2	Second task flag word. The following flags are defined:

Bit	Flag	Meaning When Bit = 1
1	TZ$FMP	Task uses fast map facility.

L$BLRL	Label block revision level.
L$AME	Always 0 for AME compatibility.

The contents of the SRTS/common name block are listed in Table C-2. This block is constructed by referencing the disk image of the SRTS/common block. The format is identical to words 3 through 16 of the label block.

Table C–2: Contents of SRTS/Common Name Block

R$LNAM	Library/command name consisting of two words in Radix-50 format.
R$LSA	Base virtual address of library or common.
R$LHGV	Highest address mapped by first library window.
R$LMXV	Highest virtual address in library or common.
R$LLDZ	Library/common block load size in 64-byte blocks.
R$LMXZ	Library maximum size in units of 64-byte blocks.
R$LOFF	Size of mapped array allocated by the resident library.
R$LWND	Number of window blocks required by library.
R$LSEG	Size of library overlay segment descriptors in bytes.
R$LFLG	Library flags word. The following flags are defined:

Bit	Meaning
15	LD$ACC - Access intent (1=read/write, 0=read-only)
14	LD$RSV - APR was reserved
13	LD$CLS - Library is part of a cluster
7	Default member of cluster (or HISEG).
5	LD$RES-library has memory resident overlays.
3	LD$SUP - Supervisor-mode library (1=yes)
2	LD$REL - Position-independent code (PIC) flag (1=PIC)
1	LD$TYP-shared region type (1=common, 0=library).

R$LDAT	Three words containing the library/common block creation date in the following format:

WORD 0:	Year since 1900
WORD 1:	Month of year
WORD 2:	Day of month

C.2 Header

The task header starts on a block boundary and is immediately followed by the task image. The task is read into memory starting at the base of the root segment. Because the root segment is a set of contiguous disk blocks, it is loaded with a single disk access.

The header is divided into two parts: a fixed part, as shown in Figure C–3, and a variable part, as shown in Figure C–4.

Figure C–3: Task Header Fixed Part

Label	Offset	Description
H.CSP	0	Current Stack Pointer (R6)
H.HDLN	2	Header length
H.EFLM	4	Event flag mask
	6	Event flag address
H.CUIC	10	Current UIC
H.DUIC	12	Default UIC
H.IPS	14	Initial PS
H.IPC	16	Initial PC (R7)
H.ISP	20	Initial Stack Pointer (R6)
H.ODVA	22	ODT SST vector address
H.ODVL	24	ODT SST vector length
H.TKVA	26	Task SST vector address
H.TKVL	30	Task SST vector length
H.PFVA	32	Power fail AST control block
H.FPVA	34	Floating Point AST control block
H.RCVA	36	Receive AST control block
H.EFSV	40	Address of event flag context
H.FPSA	42	Address of floating point context
H.WND	44	Pointer to number of window blocks
H.DSW	46	Directive Status Word
H.FCS	50	Address of FCS impure storage
H.FORT	52	Address of language impure storage
H.OVLY	54	Address of overlay impure storage
H.VEXT	56	Address of impure vectors
H.SPRI	60	Swapping priority
H.NML	61	Mailbox LUN
H.RRVA	62	Receive by reference AST control block
	64	Reserved
	66	Reserved
	70	Reserved
H.GARD	72	Header guard word pointer
H.NLUN	74	Number of LUNs

MK–01081–00

Figure C–4: Task Header Variable Part

H.LUN	LUN Table (2 words/LUN)

	Offsets
Number of Window Blocks	
Partition Control Block Address	W.BPCB
Low Virtual Address Limit	W.BLVR
High Virtual Address Limit	W.BHVR
Address of Attachment Descriptor	W.BATT
Window Size (in 32–word blocks)	W.BSIZ
Offset into Partition (in 32–word blocks)	W.BOFF
First PDR Address	W.BFPD
Number of PDRs to Map	W.BNPD
Contents of Last PDR	W.BLPD

	INITIAL VALUES
Current PS	
Current PC	
Current R5	relative block number of header
Current R4	indent word #2
Current R3	indent word #1
Current R2	task name word #2
Current R1	task name word #1
Current R0	program transfer address
Header Guard Word	

MK–01082–00

The variable part of the header contains window blocks that describe the following:

- The task's virtual-to-physical mapping

- Logical unit data

- Task context

The task header is used by RSTS/E mainly for setting the initial conditions of the task. Only locations 46 through 56 have identical meanings as in RSX.

NOTE

To save the identification, the initial value set by the Task Builder should be moved to local storage. When the program is fixed in memory and being restarted without reloading, the reserved program words must be tested for their initial values to determine whether the contents of R3 and R4 should be saved.

The contents of R0, R1, and R2 are set only when a debugging aid is present in the task image.

C.2.1 Low Core Context

The low core context for a task consists of the Directive Status Word and the Impure Area vectors. The Task Builder recognizes the following global names:

.FSRPT	File Control Services work area and buffer pool vector
$OTSV	Language OTS work area vector
N.OVPT	Overlay Runtime System work area vector
$VEXT	Vector extension area pointer

The only proper reference to these pointers is by symbolic name.

The Impure Area pointers contain the addresses of storage used by the reentrant library routines described above.

The address contained in the vector extension pointer locates an area of memory that can contain additional impure area pointers.

The format of the vector extension area is shown in Figure C–5. Each location within this region contains the address of an impure storage area that is referenced by subroutines that must be reentrant. Addresses below $VEXTA, referenced by negative offsets, are reserved for Digital applications. Addresses above this symbol, referenced by positive offsets, are allocated for user applications.

.PSECTs $$VEX0 and $$VEX1 have the attributes D, GBL, RW, REL, and OVR.

Figure C-5: Vector Extension Area Format

```
            .PSECT
IMPURE:                                       ;
              .
              .
              .
            $VEXT
```

```
                                    .PSECT $$VEX0      ←   Reserved for
                                                       ←   DIGITAL use

            $VEXTA              →    .PSECT $$VEX1      ←   Reserved for
                                                       ←   user applications
```
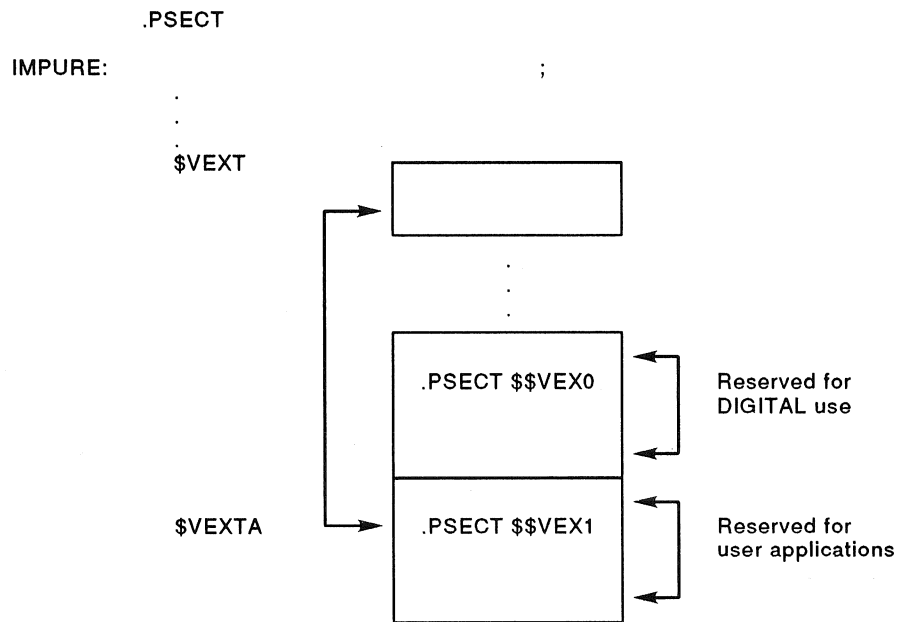
MK-01083-00

The .PSECT attribute OVR facilitates the definition of the offset to the vector and the initialization of the vector location at link time, as shown by the following example:

```
                .GLOBL      $VEXTA        ;MAKE SURE VECTOR AR A IS LINKED
                .PSECT      $$VEX1,D,GBL,RO,REL,OVR
BEG=.                                     ; POINT TO BASE OF POINTER TABLE

                .BLKW       N             ; OFFSET TO CORRECT LOCATION
                                          ; IN VECTOR AREA
LABEL:          .WORD       IMPURE        ; SET IMPURE AREA ADDRESS
                                          ; DEFINE OFFSET
OFFSET==LABEL-BEG
                .PSECT
IMPURE:
                    .
                    .
                    .
```

C.3 Overlay Data Structure

Figure C–6 illustrates the structure and principal components of the task-resident overlay data base.

Figure C–6: Task-Resident Overlay Data Base

Ⓐ Window descriptors are necessary for the windows that the overlay run–time system uses to map memory resident overlays. The overlay run–time system also needs window descriptors to map disk–resident overlays that are up–tree from memory–resident overlay segments.

Ⓑ The overlay run–time system uses region descriptors to map overlaid libraries.

Figure C–7 illustrates the task-resident overlay database for an I- and D-space overlaid task.

Figure C–7: Task-Resident Overlay Database for and I- and D-Space Overlaid Task

Ⓐ Window descriptors are necessary for the windows that the overlay run–time system uses to map memory resident overlays. The overlay run–time system also needs window descriptors to map disk–resident overlays that are up–tree from memory–resident overlay segments.

Ⓑ The overlay run–time system uses region descriptors to map overlaid libraries.

Autoload vectors are generated whenever a reference is made to an autoloadable entry point in a segment located farther away from the root than the referencing segment.

One segment descriptor is generated for each overlay segment in the task or shared region. The segment descriptor contains information on the size, virtual address, and location of the segment within the task image file. In addition, it contains a set of link words that point to other segments. The overlay structure determines the link word contents.

The following sections describe the composition of each element.

C.3.1 Autoload Vectors for Conventional Tasks

The autoload vector table consists of one entry per autoload entry point in the form shown in Figure C–8.

Figure C–8:   Autoload Vector Entry

Autoload Vector Entry

JSR @PC+2
Offset to pointer to autoload code
Segment descriptor address
Entry point address

MK–01055–00

The autoload vector contains a JSR to the autoload processor, $AUTO, followed by a pointer to the descriptor for the segment to be loaded and the real address of the entry point.

C.3.2 Segment Descriptor

The segment descriptor is composed of a six-word fixed-length portion. Segment descriptor contents are shown in Figure C–9.

Figure C–9: Segment Descriptor

TASK–RESIDENT SEGMENT DESCRIPTOR OFFSETS

15 12 11 0	BYTE
FLAGS | RELATIVE DISK BLOCK ADDRESS	0
VIRTUAL LOAD ADDRESS OF SEGMENT | F	2
LENGTH OF SEGMENT IN BYTES	4
LINK UP	6
LINK DOWN	10
LINK NEXT	12
——— SEGMENT NAME (2–WORD RADIX–50) ———	14
WINDOW DESCRIPTOR ADDRESS	20

FLAGS: 15–TASK RESIDENT FLAG (ALWAYS 1)
           14–SEGMENT HAS DISK ALLOCATION (1=NO)
           13–SEGMENT IS LOADED FROM DISK (1=YES)
           12–SEGMENT IS LOADED AND MAPPED (0=YES)

F: 0–SEGMENT FOR I– AND D–SPACE TASK (1=YES)

TASK–RESIDENT SEGMENT DESCRIPTOR EXTENSION
OFFSETS FOR I– AND D–SPACE TASKS ONLY

15 12 11 0	
UNUSED | D–SPACE DISK BLOCK ADDRESS	0
D–SPACE VIRTUAL LOAD ADDRESS	2
D–SPACE SEGMENT LENGTH IN BYTES*	4
D–SPACE WINDOW DESCRIPTOR ADDRESS	6

*0 IF ONLY I–SPACE SEGMENT

Word 0 contains the relative disk address in bits 0-11 and the segment status in bits 12-15. Each segment in the task image file begins on a disk block boundary. The relative disk address is the block number of the segment relative to the start of the root segment.

The segment flags are defined as follows:

Bit 15   Always set to 1.

Bit 14   0 = Segment loaded and mapped.
                1 = Segment is either not loaded or not mapped.

Bit 13   0 = Segment has disk allocation.
                1 = Segment does not have disk allocation.

Bit 12   0 = Segment not loaded from disk.
                1 = Segment loaded from disk.

Word 1 contains the load address of the segment. This address is the first virtual address of the area where the segment will be loaded.

Word 2 specifies the length of the segment in bytes.

The next three words point to the following segment descriptor:

Link Up   Points to the next segment away from the root. Link Up equals 0 of you are already at the leaf.

Link Down   Points to the next segment toward the root. Link Down equals 0 if you are already at the root.

Link Next   Points to the adjoining segment. Link Next equals the address of the current segment if there are no others on the same level with the same Link Down. Link Next links all segments on the same level that have the same Link Down in a circular fashion. Thus, in Figure C–10, Link Next in A3 points to A1, but Link Next in A11 points to A11 itself and Link Next in A0 points to A0 itself.

The segment descriptor for an I- and D-space task consists of a fixed part that is nine words long and an optional part that is four words long. The optional part is always present for task segments and never present for library segments. The bottom half of Figure C–9 illustrates the contents of the segment descriptor for I- and D-space tasks.

When a segment is loaded, the overlay run-time system follows the links to determine which segments are being overlaid and should therefore be marked out of memory.

Using the tree in Figure C–10 as an example:

Figure C–10: Sample Tree

MK–01088–00

The segment descriptors are linked as in Figure C–11.

If there is a co-tree, the Link Next for the root segment descriptor points to the co-tree root segment descriptor.

Words 6 and 7 contain the segment name in Radix-50 format.

Word 8 points to the window descriptor used to map the segment (0 = none).

Figure C–11: Segment Linkage Directives

MK–01089–00

C.3.2.1 Autoload Vectors for I- and D-Space Tasks

The autoload vector table consists of two entries (put into the task image for each autoload entry point) in the form shown in Figure C-12.

The I-space part of the autoload vector contains a move (MOV) instruction that places the address of the D-space part of the vector on the stack. The vector then executes an indirect jump (JMP) to $AUTO through .NAUTO. The D-space part of the vector contains the segment descriptor address of the required routine.

Figure C-12: Autoload Vector Entry for I- and D-Space Tasks

MOV (PC) +, – (SP)
ADDRESS OF PACKET (D–SPACE)
JMP @.NAUTO
PC RELATIVE OFFSET TO .NAUTO

I–SPACE PORTION

ADDRESS OF SEGMENT DESCRIPTOR
ENTRY POINT ADDRESS

D–SPACE PORTION

C.3.3 Window Descriptor

TKB allocates the window descriptors only if you define a structure containing memory-resident overlays. Figure C–13 illustrates the format of a window descriptor.

Figure C–13: Window Descriptor

```
 0 ┌─────────────────────────────┬─────────────────────────┐
   │ Base Active Page Register    │     Window ID           │
 1 ├─────────────────────────────┴─────────────────────────┤
   │              Virtual base address                      │
 2 ├────────────────────────────────────────────────────────┤
   │          Window size in 64–byte blocks                 │
 3 ├────────────────────────────────────────────────────────┤
   │                  Region ID                             │
 4 ├────────────────────────────────────────────────────────┤
   │              Offset in partition                       │
 5 ├────────────────────────────────────────────────────────┤
   │                Length to map                           │
 6 ├────────────────────────────────────────────────────────┤
   │                 Status word                            │
 7 ├────────────────────────────────────────────────────────┤
   │     Send/receive buffer address (always 0)             │
 8 ├────────────────────────────────────────────────────────┤
   │                  Flags word                            │
 9 ├────────────────────────────────────────────────────────┤
   │           Address of region descriptor                 │
   └────────────────────────────────────────────────────────┘
```

MK–01087–00

Words 0 through 7 constitute a window descriptor in the format required by the mapping directives (the Program Logical Address Space (.PLAS) Mapping Directives - see the *RSTS/E System Directives Manual* for more information). The overlay loading routine fills in the region ID at run time.

Words 8 and 9 contain additional data that the overlay routines refer to. Bit 15 of the flags word, if set, indicates that the window is currently mapped into the task's address space.

Word 9 contains the address of the associated region descriptor.

C.3.4 Region Descriptor

Figure C–14 illustrates the format of a region descriptor.

Figure C–14: Region Descriptor

```
0 ┌─────────────────────────┐
  │       Region ID         │
1 ├─────────────────────────┤
  │      Size of region     │
2 ├─────────────────────────┤
  │                         │
  │         Region          │
3 ├──────          ──────── │
  │          Name           │
  │                         │
4 ├─────────────────────────┤
  │                         │
  │         Region          │
5 ├──────          ──────── │
  │        Partition        │
  │                         │
6 ├─────────────────────────┤
  │      Region status      │
7 ├─────────────────────────┤
  │ Partition codes (always 0) │
8 ├─────────────────────────┤
  │          Flags          │
  └─────────────────────────┘
```

MK–01086–00

Words 0 through 7 constitute a region descriptor in the format required by the mapping directives. Word 8, the flags word, is referred to by the overlay load routine. Bit 15 of the flags word, if set, indicates that a valid region identification is in word 0.

C.4 Root Segment

The root segment is written as a contiguous group of blocks. The root segment is the first segment loaded and remains in memory for the entire life of the program execution.

C.5 Overlay Segments

Each overlay segment begins on a block boundary. The relative block number for the segment is placed in the segment table. Note that a given overlay segment occupies as many contiguous disk blocks as it needs to supply its space request. The maximum size for any nonroot segment is 28K words. The maximum size for the root segment is 32K words.

Appendix D

Reserved Symbols

All symbols and PSECT† names containing a period (.) or dollar sign ($) are reserved for Digital-supplied software. Several global symbols and PSECT† names are reserved for use by the Task Builder. 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 Task Builder. The relocated value of the symbol is taken as the modification address.

Table D–1 shows global symbols reserved by the Task Builder.

Table D–1: Task Builder Reserved Global Symbols

Global Symbol	Modification Value
$ALERR	OTS address of overlay load eroor handler.
$AUTO	OTS address of autoload routine.
$DBTS	Debugger time stamp.
.FSRPT	Address of file storage region work area (.FSRCB).
$FSTIN	OTS address of fast map overlay routine.
$MARKS	OTS address of MARK segment routine.
.MBLUN	Mailbox logical unit number.
.MOLUN	Error message output device.
.NALER	OTS entry point to overlay load error handler.
.NAUTO	OTS entry point to $AUTO or $LOAD.
.NDTDS	OTS highest displaced segment.
.NFAST	OTS AST supression control flags.
.NFMAP	OTS entry point to Fast Map initialization routine.
.NIOST	OTS common I/O status doubleword.
.NLUNS	The number of logical units used by the task, not including the message output and overlay units.
.NMARKS	OTS entry point to MARK segments.

(continued on next page)

---

† PSECTS are created by .ASECT, .CSECT, or .PSECT directives. The .PSECT directive eliminates the need for either the .ASECT or .CSECT directive, both of which are retained only for compatibility with other systems. In this document all sections are referred to as PSECTS unless the specific characteristics of .ASECT or .CSECT apply.

Table D–1 (Cont.): Task Builder Reserved Global Symbols

Global Symbol	Modification Value
.NOVLY	The overlay logical unit number.
N.OVPT	Address of overlay run-time system work area (.NOVLY).
.NRDSG	OTS entry point to READ segments.
.NSTBL	The address of the segment description tables. This location is modified only when the number of segments is greater than one.
.NSZSG	OTS size of resident segment descriptors.
.ODTL1	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).
.PTLUN	Logical unit number for plotter/graphics software.
$RDSEG	OTS address of READ segment routine.
.SUML1	P/OS standard utility module LUN.
.TRLUN	The trace subroutine output logical unit number.
.USLU1	Logical unit number for special purpose user software.
.USLU2	Logical unit number for special purpose user software.
$VEXT	Address of vector extension area ($VEXTA).

The following global symbols are reserved by TKB for tasks using disk-resident overlays:

Global Symbol	Modification Value
$MARDS	OTS entry point to I/D MARK segment routine.
$MAFKS	OTS entry point to optimized MARK segment routine.
$MAFDS	OTS entry point to optimized I/D MARK segment routine.

The following global symbols are reserved by TKB for tasks using memory-resident overlays:

Global Symbol	Modification Value
$MARKR	OTS entry point to MARK segment routine.
$MARDR	OTS entry point to I/D MARK segment routine.
$$MAFKR	OTS entry point to optimized MARK segment routine.
$MAFDR	OTS entry point to opitmized I/D MARK segment routine.

The following global symbols are reserved by TKB for tasks using cluster libraries:

Global Symbol	Modification Value
$MARKC	OTS entry point to MARK segment routine.
$MARDC	OTS entry point to I/D MARK segment routine.

Global Symbol	Modification Value
$MAFKC	OTS entry point to optimized MARK segment routine.
$MAFDC	OTS entry point ot optimized I/D MARK segment routine.

The PSECT names in Table D–2 are reserved by the Task Builder. In some cases, the definition of a reserved PSECT causes the PSECT to be extended if the appropriate option is specified.

Table D–2: PSECT Names Reserved by the Task Builder

Source Location	Section Name	Description
TKB	$$ALER	Contains code to process or trap Overlay Run-time system segment load errors. Provides named areas in the task for the FORTRAN Object Time System and the RSX Overlay Run-time System.
TKB	$$ALVC	Contains the segment autoload vectors for tasks without I- and D-space.
TKB	$$ALVD	Contains the D-space portions of the segment autoload vectors in an I- and D-space task.
TKB	$$ALVI	Contains the I-space portions of the segment autoload vectors in an I- and D-space task.
TKB	$$AUTO	Contains code to determine if a called subroutine in an overlay segment is already in memory or if that overlay segment should be read into memory before control is passed to the subroutine that is called.

Input Module	$$DBTS	This symbol should appear in the debugger input module with the symbol $DBTS as follows:
		
		`          .PSECT     $$DBTS`
		`.$DBTS::`
		`          .PSECT`
		
		The task builder extends $$DBTS and fills it with time stamp information followed by the filename information of the .STB file.

SYSLIB	$$DEVT	The extension length (in bytes) is calculated from the formula: EXT = (S.FDB+52)*UNITS The definition of S.FDB is obtained from the root segment symbol table, and UNITS is the number of logical units used by the task, excluding the message output, overlay, and ODT units.
SYSLIB	$$FSR1	The extension of this section is specified by the ACTFIL option.
SYSLIB	$$FTSM	Contains the code to map memory-resident overlays using the fast map facility instead of the standard executive mapping directive CRAW$.
SYSLIB	$$IOB1	The extension of this section is specified by the MAXBUF option.

Table D–2 (Cont.):   PSECT Names Reserved by the Task Builder

Source Location	Section Name	Description
TKB	$$IOB2	A zero length .PSECT containing a label, IOBFND, that is stored in the work area offset, W.BEND, representing the upper bound of the I/O buffer, $$IOB1. TKB uses $$IOB2 as a boundary value to determine whether the I/O buffer has overflowed.
TKB	$$LOAD	Overlay manual load routine.
TKB	$$MRKS	Contains code to properly mark those segments that are not needed any longer or have been overlaid by another segment as being out of memory. This ensures that a fresh copy of the overlay segment will be read in the next time the overlay segment is needed.
SYSLIB	$$OBF1	FORTRAN OTS uses this area to parse array type format specifications. This section can be extended by the FMTBUF keyword.
TKB	$$OBF2	A zero length .PSECT containing a label, OBFH, that is stored in the work area offset, W.OBFH, which represents the upper bound of the run-time format buffer, $$OBF1. TKB uses $$OBF2 to determine if the run-time format buffer has overflowed.
TKB	$$OVDT	The Overlay Run-time System impure data area. The symbol N.OVPT in low memory points to this area. This area defines the operational parameters with which the Overlay Run-time system operates on disk-resident and memory-resident overlay structures.
TKB	$$OVRS	The .ABS. program section that redefines the Overlay Run-time System impure data area with different symbols, defined as offsets and relative to zero. These offsets are necessary for proper linkages between the subroutines in the Overlay Run-time System. This program section is never included in the memory allocation of the task because of its absolute program section attribute.
TKB	$$PDLS	Cluster library service routine.
TKB	$$RDSG	Contains the code that reads into memory the overlay segment selected by the code contained in the programs section $$AUTO.
TKB	$$RGDS	Contains the region descriptors for resident libraries referred to by the task.
TKB	$$RTQ	Defines the PSECT used for selective enabling of AST recognition in the Overlay Run-time System. $$RTQ is 0 in length if $AUTOT is not included.
TKB	$$RTR	Defines the PSECT used for selective disabling of AST recognition in the Overlay Run-time System. $$RTR is 0 in length if $AUTOT is not included.
TKB	$$RTS	Contains the return instruction.

Source Location	Section Name	Description
TKB	$$SLVC	Supervisor-mode library transfer vectors (RSX-11M-PLUS only).
TKB	$$SGD0	Contains the program section adjoining the task segment descriptors.
TKB	$$SGD1	Contains the task segment descriptors.
TKB	$$SGD2	Contains a .WORD 0 following the task segment descriptors.
FORTRAN-77	$$TSKP	TKB fills in the following words in the PSECT (note that the word values are filled into the Section in order):

- APR bit map in word $APRMP
- Task offset into region in word $LBOFF
- Maximum physical read/write memory needed for task in word $MXLGH
- Maximum physical read-only memory needed for task in word $MXLGH+2
- Task extension in 32-word blocks in word BOK$LBEXT
- Total contribution of FORTRAN virtual arrays
- Maximum physical read-only D-space memory needed for task in word $MXLGH+4
- Maximum physical read/write D-space memory needed for task in word $MXLGH+6

Source Location	Section Name	Description
MACRO-11	$$TSKP	TKB fills in the first work of the PSECT called TSKP$. Refer to the *System Directives Manual* for more details.
TKB	$$WNDS	Contains task window descriptors.

Appendix E

Improving Task Builder Performance

This appendix contains procedures and suggestions to help you maximize Task Builder performance. Procedures are given for:

* Evaluating and improving Task Builder throughput

* Modifying command switch defaults to provide a more efficient user interface

E.1 Evaluating and Improving Task Builder Performance

Task Builder throughput is determined by these factors:

* The amount of memory available for table storage

* The amount of disk latency due to input file processing

The discussion in the following paragraphs outlines methods for improving throughput in each case. The methods approach their goals through judicious use of system resources and Task Builder features.

E.1.1 The Task Builder Work File

The largest factor affecting Task Builder performance is the amount of memory available for table storage. To reduce memory requirements, the Task Builder uses a work file to store symbol definitions and other tables. If the total size of these tables is within the limits of available memory, the work file is kept in core and not shunted to a disk. If the tables exceed the amount of memory available, some information must be moved to the disk, which degrades performance.

Work file performance can be gauged by consulting the statistics portion of the Task Builder map. The following parameters are displayed:

Number of work file references:

> Total number of times that work file data was referenced.

Work file reads:

> Number of work file references that resulted in disk accesses to read work file data.

NOTE

> If work file reads and writes equal zero and the number of work file references is greater than zero, you can be sure that the work file remained in memory.

Work file writes:

> Number of work file references that resulted in disk accesses to write work file data.

Size of Core Pool:

> Amount of in-core table storage in words. This value is also expressed in units of 256-word pages (information is read from and written to disk in blocks of 256 words.)

Size of Work File:

> Amount of work file storage in words. If this value is less than the core pool size, the number of work file reads and writes is zero. That is, no work file pages are removed to the disk. This value is also expressed in pages (256-word blocks).

Elapsed Time:

> Amount of time required to build the task image and produce the map. This value excludes ODL processing, option processing, and the time required to produce the global cross-reference.

The overhead for accessing the work file can be reduced in one or more of the following ways:

- By increasing the amount of memory available for table storage

- By placing the work file on the fastest random access device, such as the virtual disk (DV:)

- By decreasing system overhead required to access the file

- By reducing the number of work file references

The Task Builder automatically increases its size up to the maximum job size, which may be as large as 32K words. See the *RSTS/E System Manager's Guide* for information on how to change the maximum job size.

The size of the work file can be reduced by:

- Linking your task to a core-resident run-time system containing commonly used routines (for example, BASIC-PLUS-2 object time system) whenever possible

- Including common modules, such as components of an object time system, in the root segment of an overlaid task

- Using an object library file of concatenated object modules if many modules are to be linked

In the last two cases, system overhead is also significantly reduced because fewer files must be opened to process the same number of modules.

The number of work file references can be reduced by eliminating unneeded output files and cross-reference processing or by obtaining the short map. In addition, selected files, such as the default system object module library, can usually be excluded from the map. In this case, a full map can be obtained at less frequent intervals and retained.

Try the following procedures to improve work file performance:

*   Install I- and D version of TKB.

*   Decrease work file size by using resident run-time systems, concatenated object files, and object libraries.

*   Decrease work file size by moving common modules into the root segment of an overlaid task.

*   Decrease the number of work file references by eliminating the map and global cross-reference, obtaining the short map, or excluding files from the map.

*   Place the work file on the fastest possible device. If the system manager installs a system-wide logical "device:OV", the Task Builder uses a device other than SY: as the work file device.

    If the device is a private pack, all accounts of any user wishing to use the Task Builder must be entered on the private pack while the system-wide logical is in effect. Otherwise, a protection violation error occurs for those users without accounts when the Task Builder tries to create its work file.

    Again, make sure the device is mounted so users without access privileges will not obtain fatal errors when the Task Builder tries to create its work file.

*   Use the CCL/SI:## to increase size to maximum immediately. This may reduce swapping when TKB must increase in size.

E.1.2 Input File Processing

The suggestions for minimizing the size of the work file and number of work file accesses also drastically reduce the amount of input file processing.

A given module can be read up to three times when building the task:

1.   To build the symbol table
2.   To produce the task image
3.   To produce the long map

Files that are excluded from the long map are read only twice. The third pass is completely eliminated for all modules when a short map is requested. So, if you do not need the long map, use the /SH switch (described in Section 11.23) to eliminate the third pass.

Index

R

U

V

W

X

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	———	Local Digital subsidiary or approved distributor
Internal[1]	———	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page   Description

_____   _____

_____   _____

_____   _____

_____   _____

_____   _____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

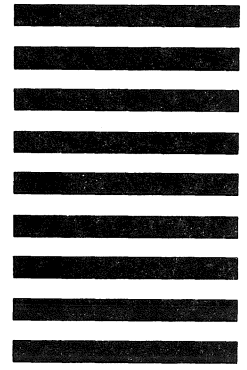I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

─ ─ ─ ─ ─ ─ Do Not Tear - Fold Here and Tape ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─

d i g i t a l ™
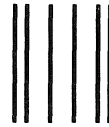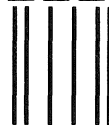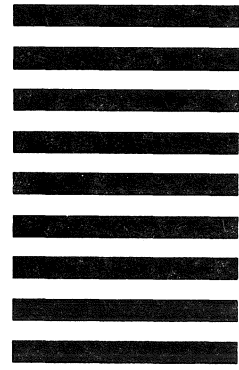
BUSINESS REPLY MAIL
FIRST- CLASS MAIL PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
CONTINENTAL BOULEVARD MKO1-2/E12
PO BOX 9501
MERRIMACK NH 03054-9982

─ ─ ─ ─ ─ ─ ─ Do Not Tear - Fold Here ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─

Cut Along Dotted Line

Reader's Comments

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page   Description

_____   _____

_____   _____

_____   _____

_____   _____

_____   _____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

digital ™

BUSINESS REPLY MAIL
FIRST- CLASS MAIL PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
CONTINENTAL BOULEVARD MKO1-2/E12
PO BOX 9501
MERRIMACK NH 03054-9982

digital

Printed in U.S.A.