MicroPower/Pascal–RSX/VMS System User's Guide

Order No. AA–AK13C–TK

digital software

MicroPower/Pascal–RSX/VMS System User's Guide

Order No. AA–AK13C–TK

June 1987

This manual contains information you will need for using MicroPower/Pascal application development tools on either an RSX host system or a VAX/VMS host system. This manual includes directions for using MicroPower/Pascal utility programs, the MicroPower/Pascal compiler, and the MPBUILD automated build procedure.

This manual supersedes the *MicroPower/Pascal–RSX/VMS System User's Guide*, AA–AK13B–TK.

Operating System and Version: Micro/RSX Version 3.0
RSX–11M Version 4.2
RSX–11M–PLUS Version 3.0
VAX/VMS Version 4.0

Software Version: MicroPower/Pascal–Micro/RSX Version 2.4
MicroPower/Pascal–RSX Version 2.4
MicroPower/Pascal–VMS Version 2.4

Digital Equipment Corporation Maynard, Massachusetts

First Printing, February 1984
Revised, June 1985
Updated, April 1986
Updated, October 1986
Revised, June 1987

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

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

Contents

Preface xi

Chapter 1 Introduction

1.1 Overview of MicroPower/Pascal Components 1-2
1.1.1 Application Development Tools 1-2
1.1.2 Run-Time System Software 1-4
1.2 Overview of the Development Process 1-5
1.2.1 The Build Cycle 1-5
1.2.2 Relationship of MPBUILD to the Build Cycle 1-7
1.2.3 Loading and Debugging 1-7
1.3 Logical Devices for MicroPower/Pascal Files 1-8
1.3.1 Logical Device MP: for MicroPower/Pascal–RSX 1-8
1.3.2 Logical Device MICROPOWER$LIB for MicroPower/Pascal–VMS 1-8
1.4 User Set-Up Procedure for RSX and VAX/VMS 1-9

Chapter 2 MPBUILD Application-Building Procedure

2.1 Capabilities and Limitations of MPBUILD 2-1
2.2 The MPBUILD Dialog 2-3
2.2.1 Dialog Structure 2-3
2.2.2 Options, Usage Rules, and Defaults 2-4
2.2.3 Dialog Description 2-5
2.2.3.1 Kernel and Global–Information Section 2-5
2.2.3.2 System-Process Section 2-9
2.2.3.3 Beginning of User-Process Build Phase 2-10
2.2.3.4 User-Process Section 2-10
2.2.3.5 Bootstrap Section 2-11
2.2.3.6 End of Dialog 2-11
2.3 Error Messages 2-12

Chapter 3 Building the Kernel

3.1 Creating the Configuration File 3-2
3.2 Assembling the Configuration File 3-3
3.3 Merge Configuration Object File with Kernel Library 3-4
3.3.1 Merging the Kernel for a Mapped Target 3-4
3.3.2 Merging the Kernel for an Unmapped Target 3-5
3.3.3 Merging the Kernel for Debugging 3-5
3.4 Relocate Kernel Module and Create Kernel Symbol Table 3-5
3.4.1 Relocating a Mapped Kernel 3-5
3.4.2 Relocating an Unmapped Kernel 3-6
3.4.3 Relocating the Kernel for Debugging 3-7
3.5 Create Memory Image (.MIM) File 3-7
3.5.1 Creating a Memory Image for Debugging or Down-Line Loading 3-8
3.5.2 Creating a Memory Image for Booting 3-9
3.5.3 Creating a Memory Image for a ROM/RAM Environment 3-10
3.6 Optimizing the Kernel 3-11

Chapter 4 Building System Processes

4.1 Edit System Prefix Module 4-2
4.2 Assembling System Prefix Modules 4-4
4.3 Merging System Prefix Modules with the System Process Library 4-5
4.4 Relocating and Installing System Processes 4-5
4.4.1 Relocating a Mapped System Process 4-6
4.4.2 Relocating an Unmapped System Process 4-6
4.4.3 Installing System Processes in Memory Image 4-7
4.5 Repeating the System Process Build 4-7

Chapter 5 Building User Processes

5.1 Compiling or Assembling Static-Process Source Files 5-3
5.1.1 Compiling 5-3
5.1.2 Assembling 5-3
5.2 Merging Static Processes 5-4
5.3 Relocating and Installing Static Processes 5-5
5.3.1 The RELOC Command Line 5-6
5.3.2 The MIB Command Line 5-7
5.4 Repeating the User Process Build 5-8
5.5 Debugging and Rebuilding the Application 5-8

Chapter 6 Separation of Instruction and Data Space, and Shared Library Files

6.1 Separation of Instruction and Data Space 6-2
6.1.1 Restrictions on I/D Separation 6-2
6.1.2 Building a Process with I/D Separation 6-3
6.2 Shared Libraries 6-3
6.2.1 Types of Shared Libraries 6-4
6.2.2 Restrictions on Shared Libraries 6-5
6.2.3 Building Shared Libraries 6-6
6.2.4 Building a Supervisor-Mode Shared Library 6-6
6.2.5 Building a User-Mode Shared Library 6-7
6.2.5.1 Unmapped User-Mode Shared Libraries 6-7
6.2.5.2 Mapped User-Mode Shared Libraries 6-8
6.2.5.3 Multiple User-Mode Libraries in an Application 6-12
6.2.5.4 Building a Process to Reference a Shared Library 6-13
6.2.6 Building a Process to Reference a Supervisor-Mode Library 6-13
6.2.7 Building a Process to Reference One or More User-Mode Libraries 6-14
6.2.8 Debugging New Processes in Applications Having Shared Libraries 6-14

Chapter 7 Methods of Application Loading

7.1 Down-Line Loading the Application 7-1
7.2 Bootstrapping the Application from a Storage Device 7-3
7.3 Placing Your Application in PROM 7-5

Chapter 8 Using the MicroPower/Pascal Compiler

8.1 File Space Requirements 8-2
8.2 Compiler Invocation and Command Line Format 8-2
8.2.1 RSX Development System 8-2
8.2.2 VAX Development System 8-4
8.2.3 Command Line Usage Rules 8-7
8.3 Compilation Options 8-7
8.3.1 Compilation Options in Source Program 8-8
8.3.2 Command Line Options 8-9
8.3.2.1 Run-Time Checking Code (/CHeck:xxx) 8-9
8.3.2.2 Debug Symbol Information (/DEbug) 8-9
8.3.2.3 Extended Statistics (/EXtra) 8-10
8.3.2.4 Filter Unused Declaration (/FIlter-decls) 8-10
8.3.2.5 Instruction Set (/INstr:xxx) 8-11
8.3.2.6 Compilation Listing (/[NO]List=file-spec) 8-12

8.3.2.7 MACRO–11 Output Code (/MAcro) 8-12
8.3.2.8 No Real-Time Predefinitions (/NOpred) 8-12
8.3.2.9 Output Object File (/[NO]Object=file-spec) 8-12
8.3.2.10 Listing Page Size (/PAge_size=page-size) 8-12
8.3.2.11 Generate Warning and Informational Errors (/[NO]WArnings) 8-12
8.3.2.12 Standard Pascal Only (/STandard) 8-13
8.4 Compilation Listing 8-13
8.5 Compiling Large Programs 8-16
8.6 P-sect Generation 8-19

Chapter 9 The Merge Utility Program

9.1 Functions of MERGE 9-3
9.1.1 Resolving Intermodule Global References 9-3
9.1.2 Updating Relocation Records 9-4
9.1.3 Resolving Object Library References 9-4
9.1.3.1 Ordering of Multiple Object Libraries 9-5
9.1.3.2 Ordering of All MERGE Input Files 9-5
9.2 Role of MERGE in the Build Cycle 9-6
9.2.1 Merging the System Configuration File (Kernel) 9-6
9.2.2 Merging Each Static Process 9-7
9.2.3 Merging a Shared Library 9-7
9.3 Invocation and Use of MERGE 9-7
9.4 Section Map 9-10
9.5 Merge Options 9-12
9.5.1 Debug Symbols (/DE) 9-13
9.5.2 Include Module from Any Library (/IN) 9-14
9.5.3 Library File Identification (/LB) 9-15
9.5.4 Extract Modules from Specific Library (/LB:module:...) 9-15
9.5.5 Module Name (/NM) 9-15
9.5.6 Supervisor-Mode Shared Library .STB File (/SL) 9-16
9.5.7 User-Mode Shared Library .STB File (/UL) 9-16
9.5.8 Version Number (/VR) 9-16

Chapter 10 The RELOC Utility Program

10.1 Functions of RELOC 10-2
10.2 Role of RELOC in the Build Cycle 10-3
10.3 Invocation and Use of RELOC 10-4
10.4 Relocation Map 10-7
10.5 RELOC Options 10-9
10.5.1 Alphabetical Symbol Listing (/AB) 10-12
10.5.2 Align First RW Section at 4K-Word Boundary (/AL) 10-12
10.5.3 Debug Symbols (/DE) 10-14
10.5.4 Starting Address of Read-Only Data Space (/DR:n) 10-14
10.5.5 Disable Section Sort (/DS) 10-14
10.5.6 Starting Address of Read/Write Data Space (/DW:n) 10-17
10.5.7 Extend Section to Specified Size (/EX) 10-17
10.5.8 Separate Instruction and Data (I/D) Space (/ID) 10-17
10.5.9 Define User Library Base Address (/LS:name:addr) 10-18
10.5.10 Program/Process Name (/NM) 10-18
10.5.11 Base Address for Specified Program Section (/QB) 10-18
10.5.12 First RO Section at Specified Address (/RO) 10-19
10.5.13 First RW Section at Specified Address (/RW) 10-19
10.5.14 Short Map (/SH) 10-19
10.5.15 Supervisor-Mode Shared Library (/SL) 10-20
10.5.16 Build User-Mode Shared Library (/UL[:addr]) 10-20
10.5.17 Round Up Section Size (/UP) 10-20
10.5.18 Program Version Number (/VR:xxx) 10-20
10.5.19 Wide Map (/WI) 10-20
10.5.20 Value of Undefined Locations (/ZR:nnn) 10-21

Chapter 11 The MIB Utility Program

11.1 Functions of MIB 11-2
11.1.1 Creating a Memory Image File 11-2
11.1.2 PASDBG Load Format 11-3
11.1.3 Bootstrap Load Format 11-3
11.1.4 PROM Programmer Format 11-4
11.1.5 Installing Static Processes 11-5
11.1.6 Installing Shared Libraries 11-6
11.1.7 Installing a Bootstrap 11-6
11.1.8 Removing a Bootstrap 11-6
11.1.9 Creating a Map File 11-6
11.1.10 Initializing the Debug Symbol File 11-6
11.1.11 Installing Debug Symbols for a Static Process or Shared Library 11-7

11.2 Role of MIB in the Build Cycle 11-7
11.3 Invocation and Use of MIB 11-8
11.4 MIB Options 11-11
11.4.1 Install Bootstrap (/BS) 11-11
11.4.2 Exception Group Code (/GC) 11-12
11.4.3 Kernel Installation (/KI) 11-12
11.4.4 Process Priority (/PR) 11-13
11.4.5 Align Specified Program Section (/QB) 11-13
11.4.6 Remove Bootstrap (/RB) 11-14
11.4.7 Small Output Memory Image (/SM) 11-14
11.5 MIB Memory Map 11-14

Chapter 12 Making a Volume Bootable on the Target

12.1 Functions of COPYB 12-1
12.2 Invoking COPYB 12-2
12.3 The Copy-Boot Program (COPBOT) 12-4

Chapter 13 DECNET Down-Line Loading (RSX or VMS Only)

13.1 DECNET/Ethernet Down-Line Loading 13-2
13.2 DECNET/DDCMP Down-Line Loading 13-4

Appendix A Interaction of RELOC and MIB

A.1 Relocating Mapped Static Processes A-1
A.2 Relocating Unmapped Static Processes A-5

Appendix B Extended Disk (XD) and DRV11 (YA) Drivers

B.1 Extended Disk (XD) Driver B-1
B.2 DRV11 (YA) Driver B-2

Appendix C .MIM File Format

Index

Figures

1–1 Kernel Build Phase 1-5
1–2 System-Process Build Phase 1-6
1–3 User-Process Build Phase 1-6
1–4 Application Image Loading 1-8
3–1 Kernel Build Phase 3-1
3–2 Build the Kernel 3-2
4–1 System-Process Build Phase 4-2
4–2 Build DIGITAL–Supplied System Processes 4-4
5–1 User-Process Build Phase 5-2
5–2 Build User-Written Pascal Static Processes 5-2
6–1 Mapped Application, Relocatable User-Mode Shared Library 6-10
6–2 Mapped Application, Absolute User-Mode Shared Library 6-11
7–1 PASDBG or Bootstrap Load Format .MIM File 7-3
7–2 PROM Programmer Format .MIM File 7-5
8–1 Compilation Listing: Program with Errors, No /DE Option 8-14
8–2 Compilation Listing: Errors Removed, /DE Option Used to Show Statement Numbers 8-15
8–3 Pascal Code and Heap Usage 8-17
9–1 MERGE Utility Input and Output Files 9-2
9–2 MERGE's Part in the Build Cycle 9-6
9–3 Sample MERGE Section Map with No Referenced Shared Libraries 9-11
9–4 Sample MERGE with a Referenced Shared Library 9-12
10–1 RELOC Utility Input and Output 10-1
10–2 RELOC's Part in the Build Cycle 10-3
10–3 Sample RELOC Map of a Process Without I/D Separation or Shared Libraries . 10-8
10–4 Sample RELOC Map of a Process with I/D Separation 10-11
10–5 Sample RELOC Map of a Shared Library 10-13
10–6 Sample RELOC Map of a Process That References a Shared Library 10-16
11–1 MIB Utility Input and Output 11-1
11–2 PASDBG or Bootstrap Load Format .MIM File 11-4
11–3 PROM Programmer Format .MIM File 11-5
11–4 MIB's Part in the Build Cycle 11-8
11–5 Sample Unmapped MIB Memory Map with No I/D Separation or Shared Libraries 11-15
11–6 Sample Mapped MIB Memory Map with No I/D Separation or Shared Libraries 11-16
11–7 Sample MIB Memory Map with I/D Separation 11-17
11–8 Sample MIB Memory Map with a Shared Library 11-18
13–1 Application Image Loading 13-1
C–1 .MIM File Format C-2

Tables

4–1 Processes and Prefix Modules for All Targets 4-3
8–1 Compilation Options for RSX–11 Host 8-3
8–2 Compilation Options for VAX Host 8-5
9–1 MERGE Options 9-12
10–1 RELOC Options 10-9
11–1 MIB Options 11-11

Preface

Structure of This Document

This manual describes the MicroPower/Pascal–RSX and MicroPower/Pascal–VMS software packages and tells you how to use the MicroPower/Pascal tools for application development. Where required, this manual describes differences in operating procedures for the two host systems, RSX (Micro/RSX and RSX–11M/M–PLUS) and VAX/VMS.

Chapter 1 provides an overview of the MicroPower/Pascal development tools and runtime software and of the MicroPower/Pascal development process.

Chapter 2 describes the operation of MPBUILD, a question-and-answer procedure that greatly simplifies the task of building MicroPower/Pascal applications. You can use MPBUILD for most applications.

Chapter 3 explains how to use the individual utility programs to build a MicroPower/Pascal kernel image and briefly discusses how to optimize the kernel image so that it is as small as possible for a given application.

Chapter 4 explains how to use the individual utility programs to build system processes (typically device drivers).

Chapter 5 explains how to use the individual utility programs to build user processes.

Chapter 6 discusses the use of supervisor-mode and user-mode shared libraries and explains how to build applications with instruction- and data-space separation.

Chapter 7 describes methods of loading application images.

Chapter 8 describes the MicroPower/Pascal compiler and its options.

Chapters 9, 10, and 11 contain detailed descriptions of the MERGE, RELOC, and MIB utility programs and provide reference information on all build utility options. You may need some of these options, not covered in earlier chapters, when building applications for unusual target configurations that cannot be handled by the MPBUILD automated build procedure.

Chapter 12 describes the COPYB utility and COPBOT's use.

Chapter 13 discusses DECNET/Ethernet and DECNET/DDCMP down-line loading, for RSX or VMS only, as alternatives to the methods of application loading described in Chapter 7.

Appendix A contains many examples of RELOC and MIB command lines, showing the proper commands to use for several classes of application builds. You may want to refer to these examples if you cannot use MPBUILD.

Appendix B briefly discusses the DRV11 (YA) device driver.

Appendix C illustrates and discusses the .MIM file format.

Conventions Used in This Document

The following conventions are used in this manual:

- In interactive examples, your input appears in **boldface** type to differentiate it from system output.
- You terminate all your input, other than control characters, by pressing the RETURN key (carriage return). The symbol used in this manual to represent a carriage return is <RET> .
- To produce certain needed control characters, you use a combination of the CTRL key and an alphabetic key simultaneously. For example, holding down the CTRL key and typing Z produces the CTRL/Z character. Such key combinations are represented in command lines in the format <CTRL/x> —for example, <CTRL/Z> or <CTRL/C> .
- In descriptions of command syntax, uppercase letters represent fixed elements, such as command names and options, which you must type as shown. Lowercase letters represent a variable, such as a file specification, for which you must supply a value. Brackets ([ ]) enclose optional elements of a command; you can include an item in brackets or omit it as appropriate. The ellipsis (...) represents repetition; you can repeat the item that precedes the ellipsis.
- The capital letter O and the number 0 are so represented.
- Numeric values other than addresses are expressed in decimal, unless otherwise indicated.
- Address values are specified in octal, in conformance with standard PDP–11 practice.

Chapter 1
Introduction

The MicroPower/Pascal layered product is a set of software tools for developing real-time applications for low-end, 16-bit target systems. The target system can be any PDP–11 Q-bus microcomputer. MicroPower/Pascal–RSX software provides the development tools on Micro/RSX and RSX–11M/M–PLUS; MicroPower/Pascal–VMS software, on VAX/VMS. The MicroPower/Pascal–RSX and MicroPower/Pascal–VMS products have common or equivalent components (Section 1.1) and are closely related to MicroPower/Pascal–RT, which provides functionally equivalent capabilities on a single-user PDP–11 host system. This manual primarily describes the development tools—compiler and build utilities—used in building an application memory image for a target system.

MicroPower/Pascal is aimed at dedicated microprocessor or microcomputer applications in such areas as process control, instrumentation, control logic, intelligent subsystems, and robotics. You use a high-level language to develop the application code as a set of cooperating concurrent processes that employ semaphore-based constructs for synchronization and interprocess communication. The MicroPower/Pascal language is a superset of ISO Pascal, with extensions for both real-time programming and modular implementation techniques. You can also use the MACRO–11 assembly language for coding part or all of an application.

A MicroPower/Pascal application does not require a conventional operating system for its target run-time environment. Instead, user-coded processes are combined, in the target memory image, with the MicroPower/Pascal kernel—a set of executive modules capable of being tailored and ROMed—and with selected system processes such as device drivers. Thus, the application does not incur the overhead costs associated with execution under a generalized operating system. Using MicroPower/Pascal, you can achieve smaller and higher-performance solutions for dedicated real-time applications than would be possible with conventional implementation methods. In addition, applications developed with MicroPower/Pascal tools are ROMable.

Significant features described in other MicroPower/Pascal manuals are the following:

- Extended Pascal as the primary implementation language
- Optimizing Pascal compiler that generates efficient, ROMable code and supports modular compilation
- Compact and modular run-time executive (kernel)

- Modern semaphore-based architecture for concurrent processing, permitting efficient multi-tasking and fast real-time response
- DIGITAL-supplied system processes for device handling, clock service, and file system support
- Flexible set of utilities for building and loading the application memory image
- High-level symbolic debugging of target system from the host system
- Optional target file system capability, compatible with RT–11

1.1 Overview of MicroPower/Pascal Components

MicroPower/Pascal software components comprise both host development tools and run-time system software. The development tools are programs and command procedures that execute on the host system and enable you to compile or assemble your target application programs, build a target memory image, and load and debug your applications in the target system. Most of the development tools used to build the memory image are RSX programs common to both MicroPower/Pascal–RSX and MicroPower/Pascal–VMS.

The run-time software is the collection of DIGITAL-supplied system modules and processes that form part of an application image and execute on the 16-bit target system. This software provides the run-time executive support and device handling needed by user-written processes. MicroPower/Pascal run-time support is included in the target memory image, as needed by a given application, during the application build cycle.

The run-time system software is specific to the target system and is identical across all host versions of the product. The commonality of run-time software, together with compatible host development tools, means that an application image developed from a given set of sources will execute the same whether developed in an RSX, VMS, or RT–11 host environment.

1.1.1 Application Development Tools

The major MicroPower/Pascal development tools are the following:

- The MicroPower/Pascal Extended Pascal compiler, which is either a PDP–11 program running under RSX on a PDP–11 host or a VAX program running under VMS on a VAX host. The Pascal source language supported on an RSX or VAX host is the same as that supported by a MicroPower/Pascal–RT host system, and the object code generated by the compiler on one host system is compatible with the code generated by the compiler on any other host system.

  The extended Pascal source language is described in the *MicroPower/Pascal Language Guide*. The compiler command interface is described in Chapter 8 of this manual.

- The PASDBG symbolic debugger, which is implemented in the mode native to the host system on which it executes. PASDBG allows you to down-line load and debug an application over a host-to-target serial line, using Pascal-like interactive debugging commands. PASDBG recognizes processes, Pascal data types, user-defined data types, kernel structures, and target system states. It permits debugging source identifiers and statements and displays data in its proper type. PASDBG also permits you to down-line load a memory image for independent execution with no host/target interaction.

The command structure and use of PASDBG are described in the *MicroPower/Pascal Debugger User's Guide*. Minor differences in host/target line-setup requirements for each host system are described in Chapter 1 of the installation guide for your host system.

- The MicroPower/Pascal application-build utilities MERGE, RELOC, and MIB, which together allow you to build an application memory image for the target system. The build utility programs transform object modules produced by either the MicroPower/Pascal compiler or the PDP–11 MACRO–11 assembler into a loadable memory image and optionally provide symbol information for high-level debugging.

  Each execution of the three build utilities constitutes one step of the multistep, multiphase application-building process referred to as the build cycle (Section 1.2.1). (Preceding steps are editing and compiling/assembling.) In each phase, one component of the application memory image—the kernel, a system process, or a user process—is merged, relocated as required for the target environment, and installed in the memory image file. Each utility provides options that permit combinations of the following target system characteristics:

  Mapped or unmapped
  RAM only or ROM/RAM memory
  Debugging, down-line loading, or stand-alone configuration, relative to the host system

  Depending on the options used in the build cycle, the application can be down-line loaded from the host to the target for debugging or for independent execution, bootstrapped from a target storage device, or blasted into PROM for permanent installation in the target. (Software support for PROM blasting is not a component of the MicroPower/Pascal product but is available separately for VMS.)

  The direct use of the build utilities is described in Chapters 9, 10, and 11. As a convenient (and likely) alternative to direct use, the build utilities can be run indirectly by means of the MPBUILD command procedure.

- The MPBUILD command procedure, which integrates the use of the compiler, assembler, and build utilities, automating the entire application-building process. MPBUILD conducts a question-and-answer dialog and generates a customized build-command file based on your responses. The generated command file executes the compiler or assembler and the three build utilities as needed for each component of an application image.

  The MPBUILD command procedure is described in detail in Chapter 2.

- The COPYB utility, which produces a bootable storage volume to be used for loading an application from a target system device. This utility is described in Chapter 12.

1.1.2 Run-Time System Software

MicroPower/Pascal run-time software is supplied primarily as object library modules that are selectively included in the target memory image as needed for a particular application. This software falls into three main categories: the kernel, system processes, and file system support routines that are merged into user processes. The major run-time system components are the following:

- The kernel, the processor-specific, executive portion of the MicroPower/Pascal run-time system, manages the processor state, handles interrupts and traps, and schedules process execution. On request from individual processes, the kernel provides the interprocess synchronization and communication services, called primitives, which allow processes to interact in a real-time environment. The kernel also manages the memory area in which dynamic system data structures such as semaphores and ring buffers are created on behalf of requesting processes.

  The kernel is modular and can be tailored to the requirements of a specific application. The kernel is supplied as a library of object modules. The two kernel object libraries are PAXU.OLB for unmapped target systems and PAXM.OLB for mapped targets. You build a kernel by first assembling a user-prepared configuration source file, containing a set of configuration macro calls, together with one of the two MicroPower/Pascal system macro libraries—COMU.MLB for unmapped applications or COMM.MLB for mapped applications. You then merge the system configuration file with the appropriate kernel library. The result of the merge is a configured kernel object module, which you then relocate and install as the first element of a memory image file. Successive phases of the build cycle add system and user processes to the same memory image.

  Chapter 3 describes the steps involved in building a kernel. The *MicroPower/Pascal Run-Time Services Manual* provides a functional description of the kernel and configuration files.

- MicroPower/Pascal device drivers are system processes that provide hardware-level support for target I/O devices and device interfaces. The driver processes are supplied in object form in two driver object libraries—DRVU.OLB for unmapped target systems and DRVM.OLB for mapped targets.

  You include in the target memory image only those drivers required for the application. A driver process is built by merging a DIGITAL-supplied prefix module file—DLPFX.MAC for the DL driver, for example—with the appropriate driver library. The prefix module pulls the corresponding driver object module(s) from the library and configures it with application-dependent hardware parameters such as vector and CSR addresses, and number of units and controllers. The driver prefix modules contain user-modifiable default values for those parameters. After the merged driver module is relocated, the process image is installed in the memory image file.

  Chapter 4 describes the steps involved in building a driver process. The *MicroPower/Pascal I/O Services Manual* provides a functional description of the device drivers and prefix files.

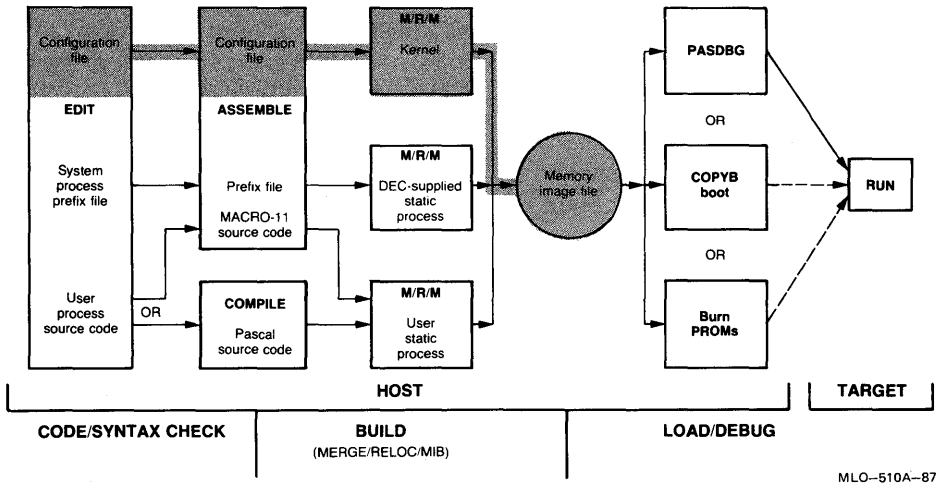
1.2 Overview of the Development Process

Figures 1–1 through 1–4 illustrate the MicroPower/Pascal development process. The shaded path in each figure represents a major phase of the process and indicates the principal development tools involved. The development process is by nature highly iterative, since each time an application memory image is built and a problem is discovered through debugging, some or all phases of the process must be repeated to correct the problem and retest the modified image.

1.2.1 The Build Cycle

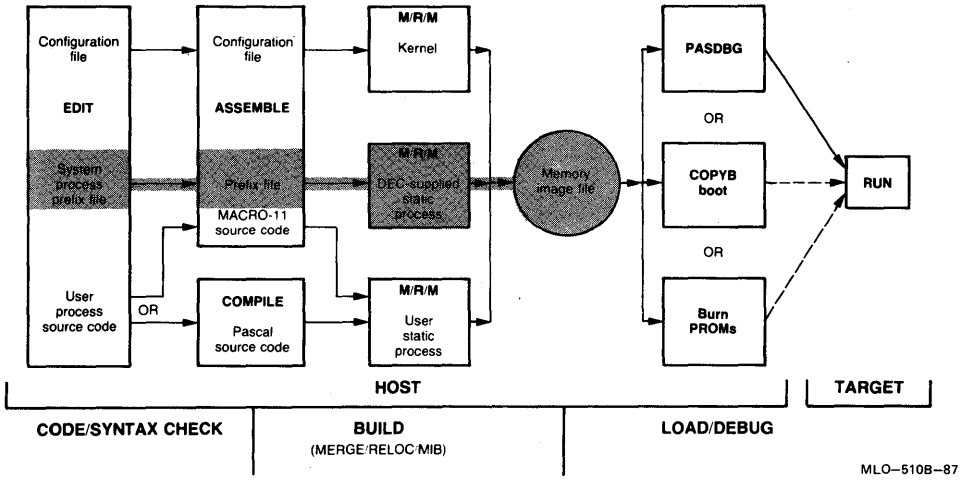
Taken together, Figures 1–1 through 1–3 represent the application build cycle, the sequence of steps and phases required to build or rebuild an application memory image. The kernel build phase, shown in Figure 1–1, builds and installs a kernel in a new memory image file. This phase is performed only once in any total build cycle, since just one component of the memory image is involved. This phase can be bypassed in subsequent partial rebuild cycles in which the kernel configuration need not be changed for the rebuilt image. Any time the kernel is modified in any way, however—to reflect a target hardware configuration change, for example—the application image file must be totally rebuilt, beginning with the kernel.

Figure 1–1: Kernel Build Phase



The system-process build phase, shown in Figure 1–2, builds and installs DIGITAL-supplied static processes in a memory image file initially containing only the kernel. This phase is iterative; it is performed once for a system process in any total build cycle or any partial build cycle starting with this phase. For example, if the application requires two system processes—two standard I/O device drivers—the system-process build phase consists of a 2-component "loop." The system-process build phase can be bypassed in subsequent partial rebuild cycles in which neither the kernel configuration nor the system processes need be modified for the rebuilt image—a rebuild in which only a user process bug is being fixed, for example.

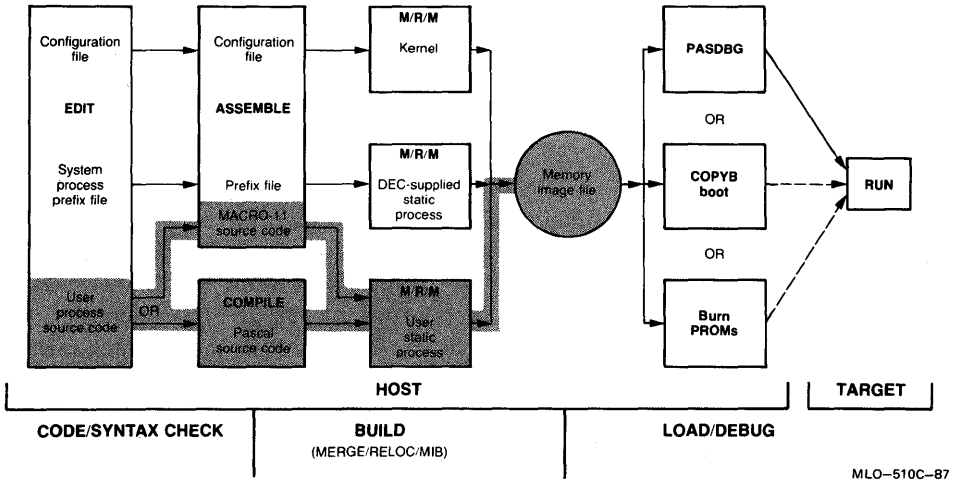
Figure 1–2: System-Process Build Phase



The user-process build phase, shown in Figure 1–3, builds and installs the user-written static processes in a memory image file initially containing the kernel and all required system processes. This phase is iterative; it is performed once for a user process in any total or partial build cycle. If the application requires five user processes, for example, the user-process build phase consists of a 5-component "loop."

Omission of a logically prior build phase implies that certain files have been preserved at an intermediate point in a previous build cycle.

Figure 1–3: User-Process Build Phase



1.2.2 Relationship of MPBUILD to the Build Cycle

As shown in Figures 1–1 through 1–3, each phase of the build cycle entails execution of the MERGE, RELOC, and MIB utilities in that order, possibly preceded by a compilation or an assembly. In combination, the three build utilities are similar to a linker or a task builder used in developing a program for execution in an operating system environment. Although you should understand the function of the individual build utilities, described in Chapters 9 through 11, ordinarily you will not need to run them directly. Instead, you can use the MPBUILD command procedure to automate most of the application-building process.

The MPBUILD procedure conducts an interactive question/answer dialog. Based on your answers to the questions during the dialog, MPBUILD creates an intermediate command file that, when executed, runs the compiler, assembler, and MicroPower/Pascal utility programs, as required, to perform the many operations necessary for a full or a partial build. The generated build-command files not only simplify the development process and eliminate the considerable clerical burden otherwise involved but also serve a useful tutorial purpose as examples of application building.

During the MPBUILD dialog, you specify the input to the various phases of the cycle; appropriate system software libraries are supplied automatically. User input for a full build cycle consists of the following:

- A system configuration file for the kernel build phase
- One or more device driver process prefix modules for the system-process build phase
- User program modules implementing user static processes, for the user-process build phase

The configuration, prefix, and program module files can be in either source or object form; MPBUILD includes or omits the corresponding compilation or assembly step for any given input module.

The several DIGITAL-supplied libraries required for building any given component of the image are automatically included in the command lines generated by MPBUILD. You do not need to specify any standard system input in the MPBUILD dialog.

MPBUILD is versatile enough to be used for building most applications, with very few exceptions. Chapter 2 provides a detailed description of MPBUILD.

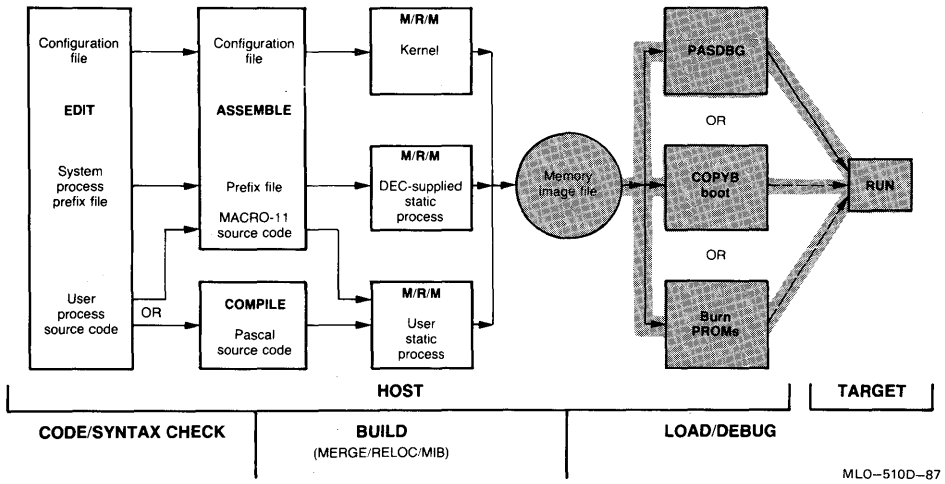
1.2.3 Loading and Debugging

Figure 1–4 shows the several ways that an application can be loaded into a target system, as appropriate to its stage of development. Starting with a complete memory image file built with symbolic debugging support, you can use PASDBG to load the target system over a host/target serial line and perform high-level debugging from a host terminal. Alternatively, you can use PASDBG for down-line loading only—that is, to load from a memory image file built without debug support and initiate target execution, with no subsequent host/target interaction.

Also, starting with a complete memory image file built without debug support, you can use the MicroPower/Pascal COPYB utility to prepare a bootable storage volume on a suitable host system I/O device. You can then move the volume to an identical or compatible device attached to the target system and boot the application from there. Optionally, you can use the console ODT capabilities of the target system, where available, for limited stand-alone debugging.

Finally, for an application in a late stage of development and built for a final ROM/RAM target environment, you can "burn" the read-only portion of the memory image into PROM or EPROM chips, which are then installed in the target system. (The hardware and software required for PROM programming are not parts of the MicroPower/Pascal product but are available separately for VMS only.) Since only very limited possibilities exist for debugging an application executing in a ROM/RAM environment, this form of application loading is generally preceded by a considerable amount of development done on a comparable RAM-only target system.

Figure 1–4: Application Image Loading



1.3 Logical Devices for MicroPower/Pascal Files

1.3.1 Logical Device MP: for MicroPower/Pascal–RSX

On a Micro/RSX or RSX–11M/M–PLUS host system, the logical device name MP: is defined to identify the default disk storage device on which all MicroPower/Pascal–RSX files are installed. By installation default, all user-relevant files, such as the configuration and prefix source files and macro and object libraries, reside in MP:[2,10]. (The system manager can override the standard [2,10] UFD at software installation time, however.)

Several files that are accessed automatically by MPPASCAL or PASDBG, including PREDFL.PAS and the TD bootstraps, reside in MP1:[1,1]. The MicroPower/Pascal-specific .TSK files are normally installed in MP1:[1,54]. By installation default, MP: and MP1: point to the same physical device.

1.3.2 Logical Device MICROPOWER$LIB for MicroPower/Pascal–VMS

On a VAX/VMS host system, the system logical device name MICROPOWER$LIB identifies the physical device and directory on which all MicroPower/Pascal–VMS files reside. By installation default, MICROPOWER$LIB is defined as a top-level directory on the system disk.

1.4 User Set-Up Procedure for RSX and VAX/VMS

Before using any of the MicroPower/Pascal facilities under RSX or VAX/VMS, you should execute the command file LB:[1,2]MPSTART.CMD (if RSX) or MICROPOWER$LIB:MPSETUP.COM (if VMS). The MPSETUP command procedure establishes logical command definitions such as MPPASCAL, MPBUILD, and MPMERGE, described in later chapters, and for PASDBG. These definitions allow you to invoke, in a convenient, shorthand fashion, the MicroPower/Pascal–VMS executable components that reside in MICROPOWER$LIB. In addition, the compiler DCL commands are installed.

Since you must execute MPSETUP once each terminal session in order to use the symbols defined by it, you should place the following command line in your LOGIN.COM file:

```
$ @MICROPOWER$LIB:MPSETUP
```

Chapter 2
MPBUILD Application-Building Procedure

MPBUILD is a comprehensive, flexible command procedure that you can use to build typical MicroPower/Pascal application images with a minimum of effort. Before using MPBUILD, you should read Chapter 1 for a general understanding of the application build cycle.

2.1 Capabilities and Limitations of MPBUILD

MPBUILD allows you to perform either an entire application build cycle or selected portions of a build cycle by responding to a series of questions. You do not specify any language-processor or utility-program commands when using MPBUILD; the procedure synthesizes all required build commands for you from your responses. MPBUILD produces as its output an indirect command file that contains all the commands necessary to perform the requested build cycle. MPBUILD allows for the following possibilities:

- A total build cycle to construct a complete application image starting with the kernel-building phase and proceeding through the system-process and user-process build phases. MPBUILD separates a total build cycle into two partial build cycles—one that produces, primarily, a kernel image file containing the kernel and selected system processes, and one that produces an application image file consisting of the contents of the kernel image file plus user processes. Secondary output of both cycles consists of the corresponding symbol table and debug files, as applicable.
- A partial build cycle, which builds just the kernel and, optionally, some device driver system processes. The primary output is a kernel image file that can be used as input to a subsequent partial build cycle.
- A partial build or rebuild cycle, which builds only the user processes using an existing kernel image file as input. Optionally, you can add system processes to the output image file at the same time. The primary output is an application image file. You can also add one or more user processes to an existing application image. You use a previously built application image file as input, as if it were a kernel image file. The primary output is a new application image file. Section 2.2.3.1 gives directions for performing this form of build operation, which involves special interpretation of certain MPBUILD questions.

Note

You should be aware that during the MPBUILD process, MPBUILD deletes intermediate files and purges .MIM, .STB, and .DBG utility-program files.

The MPBUILD command procedure leads you through a question-and-answer dialog and then generates a command file based on information gained both directly and by inference from your responses. When executed, the generated command file initiates the required sequence of build operations without further interaction on your part. The generated command file repetitively invokes the compiler and/or assembler and the MERGE, RELOC, and MIB utilities, as needed, to accomplish all steps and phases implied by the MPBUILD dialog. Any error or warning messages issued by those programs are displayed at your terminal.

MPBUILD automatically provides the most widely needed optional capabilities of the build utility programs, allowing you to build most applications using MPBUILD alone. For some applications, however, you may need to use the individual MERGE, RELOC, and MIB utilities, either independently or in conjunction with MPBUILD. MERGE, RELOC, and MIB can be used to add a static process to a memory image file already produced by MPBUILD—for example, to add a specially relocated process. MERGE, RELOC, and MIB are described in Chapters 9, 10, and 11. Alternatively, you may be able to edit certain commands in the command file generated by MPBUILD to overcome a specific limitation.

The significant limitations of MPBUILD are the following:

- You cannot specify a user-supplied macro library for a MACRO–11 assembly step. You can, however, use the MACRO–11 assembler to assemble the process in question prior to using MPBUILD. The MPBUILD procedure automatically includes the DIGITAL–supplied system macro libraries needed to assemble any user process.
- You cannot build a process that needs special, user-controlled address relocation.
- You cannot explicitly request any MACRO, MERGE, RELOC, or MIB options. MPBUILD automatically generates all commonly required options as needed, however.
- For unmapped target systems with interspersed ROM and RAM memory, you cannot split the RO or RW segment so that part of it is in one portion of memory and part in another.
- You cannot build multiple user-mode shared libraries.
- You cannot build a shared library without Pascal OTS included in it.

These limitations are not likely to prove troublesome in common practice.

Note

For Version 4.0 or later of VMS, the file name field of a file specification must not exceed nine characters, and the file type field must not exceed three characters. For Version 4.0 or later of VMS, underscores (_) are not valid in file specifications.

For all versions of VMS, dollar signs ($) are not valid in file specifications.

Also for VMS, any file names specified in response to any of the questions from MPBUILD should not match any logical names defined in the process, group, or system logical name tables. If such a match exists, VMS interprets the file name as the logical. For example, VMS interprets a file name of "TT" as the terminal. To resolve the conflict, include the file type with the file name. For example, specify "TT.MAC" rather than "TT".

2.2 The MPBUILD Dialog

On an RSX development system, you invoke the MPBUILD.CMD command procedure as follows:

```
>@dev:[dir]MPBUILD
```

In this command format, dev:[dir] is normally MP:[2,10] by installation default. Your CLI mode must be MCR when you execute MPBUILD.CMD and also when you execute the generated command file.

On a VAX development system, you invoke MPBUILD.COM by entering its name at system level, as follows:

```
$ MPBUILD
```

Use of the logical command symbol MPBUILD assumes that you have executed the MPSETUP.COM symbol-definition file, as described in Section 1.4.

For both development systems, MPBUILD issues an identifying message and then begins its question-and-answer dialog. The formats of the dialog produced by the two versions of MPBUILD differ in minor respects; such incidental, system-dependent format differences are mostly ignored in the following description.

2.2.1 Dialog Structure

The MPBUILD dialog consists of a variable sequence of questions. Many of the questions are conditional, either asked or bypassed depending on your responses to previous questions. The dialog is divisible into five logical sections:

1. Kernel and global information
2. System processes
3. Shared library
4. User processes
5. Optional bootstrap

The questions in the first section have implications for the entire MPBUILD procedure and affect the remainder of the dialog. The questions in that section elicit the following kinds of information:

- The type of build to be generated—kernel/driver image only, application image only, or both—and whether any system processes are to be added to an existing kernel image file
- File specifications for the kernel image file, system configuration file, application image file, and output command file, as applicable
- General information about the target system and the application that has a global effect on the command procedure to be generated, such as mapped or unmapped target, ROM/RAM or RAM-only target, optimized kernel, hardware instruction options, and inclusion of debugging support

Sections 2 through 5 of the dialog are conditional on the responses to certain questions asked in the first section.

The system-process section determines the device drivers that are to be built and installed in the kernel image file.

The shared library section requests any module or library you may wish to be added to the shared library.

The user-process section requests file specifications for the Pascal PROGRAM and MODULE files, and MACRO–11 processes, if any, representing user processes to be built and installed in the application image. The files may be in either source or object form.

The bootstrap section of the MPBUILD dialog determines whether a device bootstrap is to be installed in the application image and, if so, for which device. The bootstrap section is entered only for an all-RAM application that does not include PASDBG support.

2.2.2 Options, Usage Rules, and Defaults

Many MPBUILD questions require a file specification as a response. The options /LIB, /LIST, /OBJ, /MAP, /MAC, and /IDS can be appended to many user-input file specifications, as appropriate to the context. The detailed dialog description in Section 2.2.3 indicates where the options are applicable.

The meanings and effects of the options are as follows:

- The /LIB option indicates that a given input file is a user-supplied object library to be used in a process merge step. If /LIB is specified, the file type default is .OLB instead of .PAS. Alternatively, you can include the .OLB file type in the file specification.
- The /LIST option requests, for a user-process source (.PAS or .MAC) file only, that a listing file be produced in the corresponding compilation or assembly step of the build cycle. The listing file is generated in the user's default directory. The /LIST and /OBJ options are mutually exclusive. The /LIST option name can be abbreviated to /LIS.
- The /MAC option indicates that the specified file is a MACRO source module. The module will be assembled to produce an object module before used in the merge step. If /MAC is specified, the file type default is .MAC instead of .PAS. Alternatively, you can include the .MAC file type in the file specification.
- The /OBJ option indicates that the specified file is an object file rather than a source (.PAS or .MAC) file, in contexts where a source file specification is expected in the dialog. This option bypasses the implied compilation or assembly step for the file in the generated build cycle. If /OBJ is used, the file type default is .OBJ instead of .PAS. The /OBJ option can be used to suppress the implied compilation or assembly, regardless of file type specification. Alternatively, you can include the .OBJ file type in the file specification.
- The /MAP option causes the RELOC utility to produce a relocation map. RELOC maps can be generated for the kernel and for user processes, depending on where the /MAP option is used. The .MAP file is generated in the user's default directory. RELOC maps are of use primarily when when you debug a process implemented in MACRO–11.
- The /IDS option can be used for user static processes with J11-based targets to obtain instruction- and data-space separation.

General MPBUILD usage rules are as follows:

- In response to any question, you may request an explanation of the question by entering a question mark (?).
- Y or N is sufficient for an affirmative or a negative response, respectively.
- In descriptions of user responses, a RETURN-only response is indicated by <RET> .
- You can give your responses in either lowercase or uppercase; case is not significant. Literal responses are indicated in the dialog description in uppercase, however, for clarity.
- The MPBUILD command procedure does not verify the existence of any input file. Therefore, to preclude any nonexistent-file errors during the build cycle, ensure that all input files exist as specified in the dialog before executing the indirect command file produced by MPBUILD. You can verify all file specifications by inspecting the generated command file.

Many MPBUILD questions have default responses, which are indicated as such by appearing within brackets following the text of those questions. The format varies slightly, depending on your host system. For example:

```
    Do you wish to build a kernel? [yes]:      (VMS dialog)
or
    Do you wish to build a kernel? [S D:"yes"]:      (RSX dialog)
```

The default response to this question is indicated as yes. To accept the default response for any question, press the RETURN key.

A few questions accept only a limited set of alternative responses. In such cases, the alternatives are indicated within braces, with the default, if any, following within square brackets. For example:

```
    Instruction set hardware? {NHD,FPP,EIS,FIS} [EIS]:
or
    Instruction set hardware? {NHD,FPP,EIS,FIS} [S D"EIS"]:
```

The possible responses to this question are indicated as NHD, FPP, EIS, and FIS, with EIS as the default.

2.2.3 Dialog Description

2.2.3.1 Kernel and Global-Information Section

The following questions make up the kernel/global portion of the MPBUILD dialog.

Type "?" for help at any question.

Question 1: Do you want the long form of dialog? [no]:

Answer yes to get an explanation displayed before each question in the dialog. Alternatively, you can get explanations of selected questions by responding with ? to a given question.

Response: Y, N, or <RET>

Question 2: Do you wish to build a kernel? [yes]:

Indicate whether you want to create a new kernel image file (yes) or want to begin the build cycle with an existing, previously created input memory image (no). A no response implies that a new application image will be built based on an existing kernel or application image file. (If the existing input file contains an application image, you can add to existing processes.)

Response: Y, N, or <RET>

The answer to Question 2 determines whether Question 3 or 4 is asked.

Question 3: Kernel memory image file name? :

Specify a file name for the new kernel .MIM file. Your response names the new kernel/driver image, symbol (.STB), and debug (.DBG) files to be created in the build cycle. Do not specify a file type; directory or device information is optional. (You will be asked for the name of the application .MIM file in a later question.)

Response: File specification with no file type or version. (No file options are applicable.)

Question 4: Input memory image file name? :

Specify a file name for the existing kernel or application .MIM file. Your response names the existing input image, symbol (.STB) and debug (.DBG) files to be used in building the new application image. Do not specify a file type; directory or device information is optional. (You will be asked for the name of the output application .MIM file in a later question.)

Response: File specification with no file type or version.

Question 5: System config file spec? [default]:

Specify the system configuration file you will use to build the kernel image—for example, MP:CFDMAP.MAC (if RSX) or MICROPOWER$LIB:CFDMAP.MAC (if VMS). The default file type is MAC; that is, source input is expected. (If the file is a source file, you get a chance to edit it during the dialog.) Use the /OBJ option (or .OBJ extension), if appropriate, to indicate that the input is an object file. Use the /MAP option to produce a relocation map of the kernel (useful for MACRO process debugging).

Response: File specification; .MAC default. (/MAP and /OBJ are valid.)

Question 6: Do you wish to modify <file specified in 5> ? [no]:

You may want to edit the system configuration file if it is a source file. If the response is yes, you will enter EDT, with the specified file as input. (If the source of the file is other than your default directory, the new version of the file—the result of the edit—is written to your default directory and is accessed from there in the build cycle.) If the response is no, the file specified in (5) is used as is.

Response: Y, N, or <RET>

Question 7: Satisfied with edit? [yes]:

At this point, upon exit from the editor, you can either proceed to the next question (yes) or repeat the editing session (no). If the response is no, you are returned to EDT, with the edited file as input. (This question is asked each time you leave the editor.)

Response: Y, N, or <RET>

Question 8: Do you wish to build only the kernel/drivers? [no]:

Indicate whether you want to create the kernel image file only or want to create the application image file also (complete build cycle). If an application image is being built, the kernel/driver image will be copied from the kernel .MIM file to the application .MIM file before the user processes and shared library, if any, are installed.

Response: Y, N, or <RET>

Question 9: Application memory image file name? :

Specify a file name for the application .MIM output file. (The name applies to the application .DBG file, if any, as well.) Do not specify a file type; directory or device information is optional. On completion of the build cycle, the application .MIM file will contain a copy of the kernel or input application .MIM file contents plus the processes and shared library, if any, installed or added in the user-process build phase.

Response: File specification with no file type or version. (No file options are applicable.)

Question 10: Output command file spec? [default]:

Supply a specification for the command procedure (.CMD if RSX, .COM if VMS) file that MPBUILD will generate to invoke and control the actual build cycle. Following the MPBUILD dialog, you can execute the generated command file to initiate the build cycle.

Response: File specification; .CMD (if RSX) or .COM (if VMS) default. (No file options are applicable.)

Question 11: Mapped image? [no]:

Indicate whether the kernel and/or application images are to be built in mapped or unmapped form. A yes response implies that the target system configuration includes memory-management (mapping) hardware—for example, the KT-11 option on an LSI-11/23 processor. The response must match the specification for the MMU=YES/NO parameter of the PROCESSOR macro in the system configuration file.

Response: Y, N, or <RET>

Question 12: Debug support required? [yes]:

Indicate whether the kernel and/or application is to be built with PASDBG symbolic debugging support. If your response is no, no debug symbol (.DBG) files are created or used in the build cycle. Your response must match the specification for the DEBUG=YES/NO parameter in the SYSTEM configuration macro, which determines whether the debugger service module (DSM) is included in the kernel.

Response: Y, N, or <RET>

Question 13: Optimize the kernel? [no]:

Indicate whether the kernel should be automatically optimized to include only the primitives used by the application. If optimize is selected, the CONFIGURATION file SYSTEM macro should specify OPTIMIZE = YES, and the PRIMITIVES macro should not be used.

Response: Y, N, or <RET>

Question 14: Does this system contain any ROM? [no]:

Indicate whether the kernel and/or application is to be built for a ROM target environment or for a RAM-only target. Your response must agree with the target memory configuration described in the MEMORY macros of the configuration file used to build the kernel. The response is used to determine certain build-utility command options.

Response: Y, N, or <RET>

Question 15: What is the base of ROM (octal address)? [0]

Specify the base address of the first (low-order) ROM segment in the target memory configuration. This value will be used to relocate the kernel's initial read-only code section at the proper physical address. Zero is the proper value for most target configurations (exception: CMR21).

Response: Positive octal integer

Question 16: What is the base of RAM (octal address)? :

Specify the base address of the first (low-order) RAM segment in the target memory configuration. This value will be used to relocate the kernel's initial read/write data section at the proper physical address.

Response: Positive octal integer

Question 17: Instruction set hardware? {NHD,FPP,EIS,FIS} [NHD]:

Specify the instruction-set option to be used for compiling any Pascal user program. Your response determines the type of OTS library used to merge a Pascal process, in addition to supplying the compilation option.

Response: NHD, FPP, EIS, FIS, or <RET>

Question 18: LSI-11/2 mode on compilations? [no]:

Indicate whether the /IN:LS2 option is to be used for Pascal compilations. Answer yes only if the target is an LSI-11 or LSI-11/2 and your response to the previous question was EIS or FIS.

Response: Y, N, or <RET>

Question 19: Build a shared library? [no]:

Indicate whether a shared library containing the required Pascal OTS modules and optionally any user-specified modules should be included in the application build.

Response: Y, N, or <RET>

Question 20: Supervisor-Mode Library? [no]:

Indicate whether the shared library should execute in supervisor mode or user mode. Supervisor mode is available only on J11-based processors.

Response: Y, N, or <RET>

2.2.3.2 System-Process Section

This section consists of questions about device drivers and the system process. The three questions specific to drivers are asked repeatedly in a loop. Providing a null response to the first question terminates the section.

The dialog is as follows:

Beginning system-process section.

Question 21: Driver prefix file spec? :

Specify a prefix file for a DIGITAL-supplied system process (for example, a device driver) to be built and installed in the kernel .MIM file or added to the output application .MIM file. The prefix files, supplied in MP (RSX) or MICROPOWER$LIB (VMS), include:

ACPPFX.MAC	Ancillary control process
DDPFX.MAC	TU58
DLPFX.MAC	RL01/RL02
DUPFX.MAC	MSCP-class disk
DYPFX.MAC	RX02
KKPFX.MAC	Interface to Q-bus arbiter
KXPFX.MAC	KXT11-CA and KXJ11-CA slave interface
NSPPFX.MAC	Network services process
QNPFX.MAC	DEQNA Ethernet driver
TTPFX.MAC	Terminal driver (debug)
YFPFX.MAC	FALCON PIO port

A source file (MAC) is assumed unless you use the /OBJ option or .OBJ extension. A null response— <RET> only—signifies "no more drivers" and terminates the prefix-file question loop.

Response: File specification with a .MAC default or <RET> . (/OBJ is valid. If it is used, the type default becomes .OBJ.)

Question 22: Do you wish to modify <file specified in 21> ? [no]:

Indicate whether you want to edit the prefix file. If you answer yes, you will automatically enter EDIT/EDT, with a copy of the file as input. (When you exit, the output of the editing session is placed in your current default directory, regardless of the location of the original file, which might be MICROPOWER$LIB, for example.) If you answer no, the prefix file is assembled as is.

Response: Y, N, or <RET>

Question 23: Satisfied with edit? [yes]:

You now have the option of repeating the editing session (primarily in case you terminated the previous editing with the QUIT command). If you answer yes, the dialog continues. If you answer no, you will return to EDT, with the modified prefix file as input.

Response: Y, N, or <RET>

2.2.3.3 Beginning of User-Process Build Phase

The user-process build phase comprises the user-process sections of the dialog. This phase is bypassed if an application image is not being built. At this transition point in the generated build cycle, the kernel .MIM, .STB, and .DBG files are copied to the corresponding application files preparatory to installing the user processes.

User Process Build Phase.

Beginning shared library section.

Question 24: Additional module or library?:

Specify a module or library to be added to the shared library; a Pascal source module is assumed (.PAS default). The following must be appended: /MAC for a MACRO source module, /OBJ for an object module, /LIB for an object library file, and /LIST and /MAP are valid where applicable.

Response: File specification; .PAS default. (/MAC, /OBJ, /LIB, and /LIST and /MAP may be specified.)

2.2.3.4 User-Process Section

This section consists of four questions, which are asked repeatedly in two loops. The section is bypassed if only a kernel/driver image is being built.

The dialog is as follows:

Beginning user-process section.

Question 25: User process file spec? [default, if any]:

Supply a specification for a user-written static process file (Pascal program or MACRO module containing the DFSPC$ macro). The default file type is .PAS unless the /MAC or /OBJ option is used. The /IDS option may be used with J11–based targets to obtain instruction- and data-space separation in the static process. The /LIST and /MAP options are applicable; /LIST produces a compilation or assembly listing (.LST) file, and /MAP produces a relocation map (.MAP) file for the process. A null response— <RET> only—signifies "no more processes," terminating the user-process question loop.

Response: File specification; .PAS default. (/LIST, /MAC, or /OBJ, and /MAP are valid. If /MAC or /OBJ is used, the default type is .MAC or .OBJ, respectively.)

Question 26: Additional module or library? :

Here you may specify an additional source, object, or library module to be merged with the static-process file. The default file type is .PAS unless the /MAC, /OBJ, or /LIB option is used. The /LIST option may be used on a source file to produce a compilation or assembly listing file (.LIS type if VMS; .LST if RSX). A null response signifies "no more input," terminating the single-question loop.

Response: File specification; .PAS default. (/LIST, /LIB, /MAC, or /OBJ may be specified.)

Question 27: Is this process a device driver? [no]:

Indicate whether the process needs to be built with driver mapping. Driver mapping is necessary for any process that establishes an interrupt service routine (CONNECT_INTERRUPT or CINT$ primitive) in a mapped environment. A yes response implies that the DRIVER attribute was specified in the PROGRAM heading or that PT$DRV was specified in the DFSPC$ macro.

Response: Y, N, or <RET>

Question 28: Build this process with the shared library? [yes]:

Indicate whether this user process is to be built with the shared library.

Response: Y, N, or <RET>

2.2.3.5 Bootstrap Section

This section consists of two questions about installation of a device bootstrap in the application image file. The section is bypassed if debug support is included, the target system includes ROM, or you are not creating an application image file. The dialog is as follows:

Question 29: Include bootstrap? [yes]:

You may want a bootstrap installed in the application .MIM file in order to permit booting of the application from a target system device. The COPYB utility must subsequently be used to prepare the storage volume from which the application will be loaded. Do not include a bootstrap if you intend to use PASDBG to down-line "load and go" with the LOAD/EXIT command.

Response: Y, N, or <RET>

Question 30: Bootstrap device? {DY,DD,DL,DU}:

Here you supply the bootstrap device name—DY for an RX02 device, DD for a TU58 device, DL for an RL02 device, or DU for an RX50 or RDxx device.

Response: DY, DD, DL, or DU

2.2.3.6 End of Dialog

When you answer the last question in the dialog, MPBUILD constructs the required command procedure to build your application and issues the following informational message:

```
MPBUILD-S-Command procedure generated - <command-file-spec>
```

Control returns to system level. You can then inspect the generated procedure or execute it with the usual "@file-name" command.

Successful execution of the generated command procedure leaves you with the application .MIM, .STB, and optional .DBG files, if any, in the location specified by your response to question 9. All intermediate files created during the build cycle are deleted except for the kernel/driver image .MIM and associated .STB and .DBG files. Their location is determined by your response to question 3.

Any errors encountered during the build cycle result in an error message from the appropriate utility, followed by an exit from the generated command procedure, which also issues an error message. An error in the build cycle sometimes causes some intermediate files that otherwise would be deleted to be left in your directory.

2.3 Error Messages

The MPBUILD error messages are listed below in alphabetical order. Fatal errors are indicated by the letter F in the message heading; warnings, by the letter W.

Messages issued by MPBUILD–RSX are prefixed by the character "?"; messages issued by MPBUILD–VMS are prefixed by the character "%". The prefix character is shown in the following listing only for messages that are not common to both versions of MPBUILD.

%MPBUILD–F–Build dialog aborted

Explanation: The MPBUILD procedure terminated because of an unrecoverable error condition encountered during the build dialog.

MPBUILD–W–Conflicting options

Explanation: You specified two mutually exclusive options: /LIS and /OBJ, or /LIB and any other option. Enter your response again.

%MPBUILD–W–Illegal file specification

Explanation: Your response was not a syntactically valid file specification. You may have inadvertently used a predefined logical name, for example. Enter your response again.

MPBUILD–W–Invalid boot device; Enter DD, DY, DL or DU

Explanation: Self-explanatory; enter your response again.

?MPBUILD–W–Invalid character in < filespec>

Explanation: Your response was not a syntactically valid file specification. Enter your response again.

?MPBUILD–W–Invalid device spec

Explanation: Your response was not a syntactically valid file specification. Enter your response again.

?MPBUILD–W–Invalid file name

Explanation: Your response was not a syntactically valid file specification. Enter your response again.

?MPBUILD–W–Invalid file type

Explanation: Your response was not a syntactically valid file specification. Enter your response again.

MPBUILD–W–Invalid reply; Please enter NHD, FPP, EIS, or FIS

Explanation: Self-explanatory; enter your response again.

MPBUILD–W–Invalid reply; Please respond Yes or No

Explanation: Self-explanatory; enter your response again (RETURN only for default).

MPBUILD-W-Invalid or ambiguous option

Explanation: You used either an undefined option—something other than /LIB, /LIS, /OBJ, or /MAP—or an option that is invalid for the context. Enter your response again.

?MPBUILD-W-Invalid UFD

Explanation: Your response was not a syntactically valid file specification. Enter your response again.

?MPBUILD-W-Invalid version number

Explanation: Your response was not a syntactically valid file specification. Enter your response again.

%MPBUILD-W-Merge command line may need editing

Explanation: Because of the number of additional modules or libraries specified for a given process build, the length of the MERGE command line for that process may exceed the limit set by MERGE (255 characters). You may be able to edit the line in the generated command file to reduce its length, by deleting unneeded logical symbols, for example. Alternatively, before running MPBUILD, you could use the MERGE utility to "premerge" several object modules into one in order to reduce the number of individual modules input to MPBUILD for the process in question.

%MPBUILD-W-Must specify positive octal value

Explanation: The value for the base of RAM must be a valid nonzero octal address; enter your response again.

%MPBUILD-W-No application memory image file specified

Explanation: You responded with either RETURN only or nonprinting characters to the request for a required file specification. Enter your response again.

%MPBUILD-W-No kernel memory image file specified

Explanation: You responded with either RETURN only or nonprinting characters to the request for a required file specification. Enter your response again.

?MPBUILD-W-Options not allowed on file spec

Explanation: You specified an option on a file for which no options are valid.

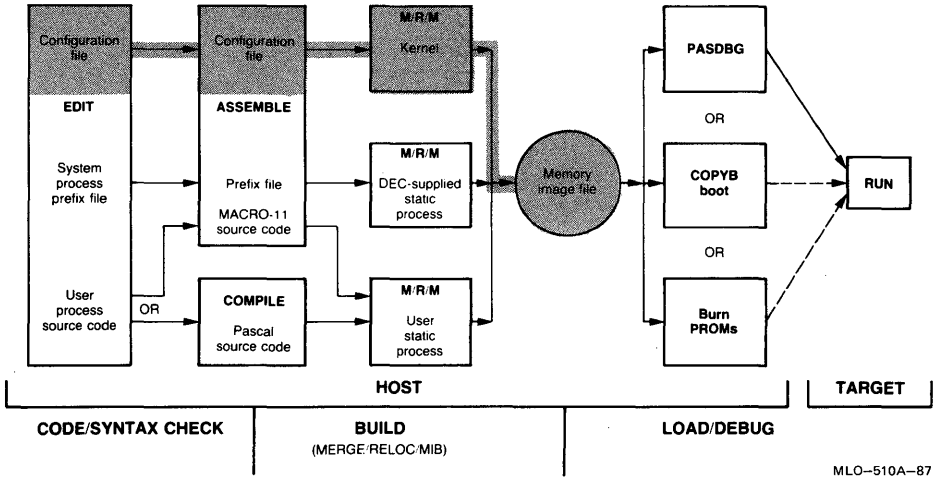
Chapter 3
Building the Kernel

To build a kernel, you must perform the following steps:

1. Create or edit a system configuration file.
2. Assemble the configuration file.
3. Build the memory image file:
   a. Merge the configuration file with the kernel library.
   b. Relocate the kernel and create the kernel symbol table.
   c. Construct, using MIB, a memory image file (.MIM) with the kernel installed.

These steps are shown in Figure 3–1.

Figure 3–1: Kernel Build Phase

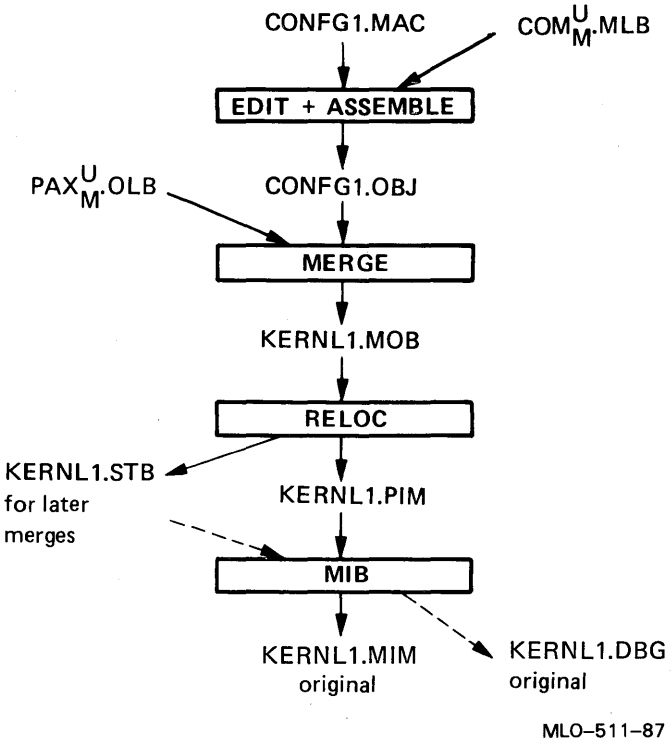


The steps involved in building the kernel image are described in detail below. You can use MPBUILD to create a command procedure containing the proper sequence of commands to build the kernel of your choice. Alternatively, you can run the editor, assembler, and MERGE, RELOC, and MIB utilities yourself and type in the required commands.

Using MPBUILD is certainly the more convenient method. If your application requires special features of MERGE, RELOC, or MIB that MPBUILD cannot handle, however, you may need to edit the command file created by MPBUILD before running it, create your own command file, or perform the entire operation manually.

Figure 3–2 illustrates the primary input and output files involved in each step. The user-specified file names in the figure are arbitrary, matching the sample file names used in the descriptions below.

Figure 3–2: Build the Kernel



3.1 Creating the Configuration File

A system configuration file consists of a series of macro calls written in MACRO–11 assembly language. The macros specify kernel software requirements such as free-memory resources, primitive modules, and trap processors and describe the target hardware configuration. Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual* describes the system configuration macros. The configuration file determines the contents of the kernel object module (.MOB) file produced

in the MERGE step and supplies the kernel, and indirectly the MIB utility, with information about the target hardware.

In particular, the DEBUG argument of the SYSTEM configuration macro specifies whether the debugger service module (DSM) is to be included in the kernel. You must specify DEBUG=YES when building an application with debug support and DEBUG=NO when rebuilding the application for testing or use without debug support.

You can create a configuration file for your application by modifying one of the sample configuration files included in the distribution kit. These files are:

CFDCMR.MAC	CMR21, with debug support
CFDFAL.MAC	FALCON target, with debug support
CFDFPL.MAC	FALCON–PLUS target, with debug support
CFDKJU.MAC	KXJ11–CA target, with debug support
CFDKTC.MAC	KXT11–CA target, with debug support
CFDMAP.MAC	Mapped LSI–11 target, with debug support
CFDUNM.MAC	Unmapped LSI–11 target, with debug support

You can use KED, EDT, or another editor on your host system to create a configuration file from scratch, but you will probably find it easier to modify one of the sample configuration files, listed above and supplied as part of the MicroPower/Pascal distribution kit, to reflect your target system's particular configuration. Choose the sample configuration file that is closest to your target system, and use an editor to make any changes that your application requires.

3.2 Assembling the Configuration File

After you have tailored a configuration file to reflect your target hardware, assemble the configuration source file together with one of the two MicroPower/Pascal system macro libraries: COMU.MLB for unmapped applications or COMM.MLB for mapped applications.

If you do not use MPBUILD, assemble the configuration file with the following form of MACRO–11 command:

```
>[MCR] MAC
MAC>CONFG1=mpp-lib:COMx.MLB/ML,user-dir:CONFG1
```

In the command line above, x is U for an unmapped application or M for a mapped application. The MACRO–11 assembler uses the specified macro library, COMM.MLB or COMU.MLB, to satisfy the macro references in CONFG1.MAC. The /ML option tells MACRO–11 that that file COMx is a macro library file. The example assumes that your edited configuration file is named CONFG1.MAC and that it resides in your default directory. In this example, the output of assembling the configuration file is an object file called CONFG1.OBJ.

Note

This example and subsequent examples include in the command form the symbol "mpp-lib:", which stands for a device/directory specification for a MicroPower/Pascal library or other DIGITAL-supplied file. In your own command lines, substitute a specification that is appropriate for your development system. For a PDP–11/RSX development system, the appropriate substitution

for mpp-lib: would likely be "MP:[2,10]". For a VAX/VMS development system, the standard substitution for mpp-lib: would be "MICROPOWER$LIB:". Those locations reflect the standard installation defaults. Check with your system manager if the MicroPower/Pascal development-software files—primarily libraries and prefix modules—are not installed in those locations on your system.

In addition, the symbol "user-dir:" stands for an explicit user device/directory specification where one is required because of the "stickiness" of device/directory specifications on input files.

3.3 Merge Configuration Object File with Kernel Library

You must merge the configuration object file with the version of the kernel module library that matches your target system. The two versions of the kernel object library are PAXU.OLB for unmapped targets and PAXM.OLB for mapped targets.

If you do not use MPBUILD, run MERGE as described below to merge the configuration object file with either PAXM.OLB or PAXU.OLB. By merging the configuration object module with the kernel library, you extract and configure the kernel library modules needed for your application. The output of the merge is a customized kernel object module. Chapter 9 describes the MERGE utility in greater detail.

General Command Format

```
>[MCR] MRG     (RSX)
or
$ MPMERGE      (VMS)
MRG>[mobfile][,mapfile][,auxfile]=infile[,infile2,...]
```

Note

Throughout this chapter, command formats and examples assume that all MicroPower/Pascal utility programs have been installed as RSX multiuser tasks with the standard task names determined by installation procedure defaults. The VAX/VMS program invocation names, such as MPMERGE, also assume that you have executed the MPSETUP.COM procedure (Section 1.4). See Chapters 9 through 11 for further details about running the MicroPower/Pascal utility programs.

3.3.1 Merging the Kernel for a Mapped Target

Use the kernel object library PAXM.OLB for building a mapped kernel. The following command example for building a mapped kernel assumes that your edited and assembled configuration file is named CONFG1.OBJ and that the merged kernel object file is to be named KERNL1.MOB. The inclusion of the /LB option with PAXM identifies it as a library file. The CTRL/Z response to the second utility prompt shown in this example causes an exit from the utility. (The second prompt will be omitted from subsequent utility command examples.)

```
>[MCR] MRG
MRG>KERNL1=CONFG1,mpp-lib:PAXM/LB
MRG><CTRL/Z>
```

3.3.2 Merging the Kernel for an Unmapped Target

Use the kernel object library PAXU.OLB for building an unmapped kernel. The following command example assumes that your edited and assembled configuration file is named CONFG1.OBJ and that the merged kernel object file is to be named KERNL1.MOB. The name of the kernel object library is the only difference from the mapped example shown in Section 3.3.1. The /LB option identifies PAXU.OLB as a library file.

```
>[MCR] MRG
MRG>KERNL1=CONFG1,mpp-lib:PAXU/LB
```

3.3.3 Merging the Kernel for Debugging

You must include the /DE option in the command line if you will be debugging the application with the PASDBG symbolic debugger. The debugger needs special kernel symbol information that the /DE option of MERGE includes in the output object module.

```
>[MCR] MRG
MRG>KERNL1=CONFG1/DE,mpp-lib:PAXx/LB/DE
```

In the command line above, x is M for a mapped application or U for an unmapped application.

If you are planning to use PASDBG with your application, remember that PASDBG can be used only with a RAM-only target. This restriction does not affect the way that you merge the kernel, but you must keep the restriction in mind as you go on to use RELOC and MIB.

3.4 Relocate Kernel Module and Create Kernel Symbol Table

RELOC must subsequently process the kernel merged object (.MOB) file to create the kernel image (.PIM) file and the kernel symbol table (.STB) file.

Use the RELOC utility to relocate the kernel object module and to produce a kernel image (.PIM) file and kernel symbol table (.STB) file. You will need the .STB file for all subsequent process MERGE steps and, if you are debugging, for producing the debug symbol (.DBG) file in the following MIB step.

General Command Format

```
>[MCR] RLC     (RSX)
```

or

```
$ MPRELOC      (VMS)
```

```
RLC>[pimfile][,mapfile][,stbfile]=mobfile[,mimfile][/options]
```

3.4.1 Relocating a Mapped Kernel

In a mapped application, the kernel's read/write memory (data) segment must be mapped by page address registers (PARs) 4, 5, and 6. In the command line example shown below, the relocation option /RW:100000 ensures correct mapping of the kernel's data space. This option forces the kernel's first read/write program section—the beginning of the read/write segment—to start at virtual address 100000(octal), which corresponds to the base of PAR 4. Memory segmentation and mapping conventions are described in Sections 2.1.6 and 2.1.7 of the *MicroPower/Pascal Run-Time Services Manual.*

```
>[MCR] RLC
RLC>KERNL1,,KERNL1=KERNL1/RW:100000
```

The first KERNL1 in the command line names the output image file, KERNL1.PIM; the second names the output symbol table file, KERNL1.STB. The third KERNL1, to the right of the equal sign, specifies the input object file, KERNL1.MOB.

RELOC physically combines and reorders program sections so that all read-only sections precede all read/write sections prior to their relocation.

When you relocate a mapped application, no distinction needs to be made between a RAM-only target and a ROM/RAM target. MIB will map the read-only and read/write sections to the appropriate physical addresses.

3.4.2 Relocating an Unmapped Kernel

When you relocate an unmapped kernel, the approach you need for a RAM-only target system differs slightly from the one you need for a ROM/RAM target system. The sections below illustrate the differences.

Unmapped RAM-Only Target System

```
>[MCR] RLC
RLC>KERNL1,,KERNL1=KERNL1
```

The first KERNL1 in the command line names the output image file, KERNL1.PIM; the second names the output symbol table file, KERNL1.STB. The third KERNL1, to the right of the equal sign, specifies the input object file, KERNL1.MOB. No RELOC options are required, since all memory is RAM-only, and unmapped applications have no special addressing requirements. The RW segment of the kernel can immediately follow the RO segment—both in RAM memory.

Unmapped ROM/RAM Target System

For an unmapped ROM/RAM application, you must provide a physical RAM address at which RELOC is to begin the kernel's read/write data segment. In the unmapped ROM/RAM command line example shown below, the relocation option /RW:ram-base supplies that address; ram-base represents an octal address value. This option forces the kernel's first read/write program section to begin at the specified RAM address. Ordinarily, you would specify the lowest RAM address in your target memory. The only constraint on the address, however, is that it must be the base of a contiguous RAM storage area large enough to contain the entire kernel data segment.

```
>[MCR] RLC
RLC>KERNL1,,KERNL1=KERNL1/RW:ram-base
```

The first KERNL1 in the command line names the output image file, KERNL1.PIM; the second names the output symbol table file, KERNL1.STB. The third KERNL1, to the right of the equal sign, specifies the input object file, KERNL1.MOB.

3.4.3 Relocating the Kernel for Debugging

The /DE option of RELOC processes the debug symbol information, called internal symbol directory (ISD) records, contained in the .MOB file and includes that information in the .STB file along with the kernel's global-symbol definitions. If you plan to use PASDBG, be sure to include the /DE option in the RELOC command line.

```
>[MCR] RLC
RLC>KERNL1,,KERNL1=KERNL1/DE
```

The /DE option is not appropriate for a ROM/RAM target, since PASDBG can be used only with an application image built for a RAM-only target.

3.5 Create Memory Image (.MIM) File

MIB must subsequently process the relocated kernel (.PIM) file and the kernel symbol table (.STB) file to create a memory image (.MIM) file. MIB can create a memory image (.MIM) file in one of three formats:

- PASDBG load format: RAM-only memory image, no bootstrap in .MIM file, debugger service module (DSM) included in the kernel for "load and debug" or not included for "load and go" (LOAD/EXIT)
- Bootstrap load format: RAM-only memory image, appropriate bootstrap in the .MIM file, no DSM in the kernel
- PROM programmer format: ROM/RAM memory image, no bootstrap in .MIM file, no DSM in the kernel

The parameters specified in the MEMORY and SYSTEM macros of the configuration file used to build the kernel define the type of memory image finally constructed.

You need a memory image file in PASDBG load format if you intend to use PASDBG either to load and debug your application or to load it for independent execution. In either event, you do not install a bootstrap in the memory image file. A bootstrap is unnecessary because PASDBG uses the host-resident TD bootstrap to down-line load the image.

You need a memory image file in bootstrap load format if you intend to boot and load the application image from a target system disk or TU58 DECtape II.

You need a memory image file in PROM programmer format if you intend to place the application in PROM chips.

Use the MIB utility program and specify the /KI option to create and initialize a memory image file containing the kernel image. If you intend to debug the application with PASDBG, you must also specify a .STB file as input and a debug symbol (.DBG) file as output.

The /SM (SMall image) option is generally recommended. It limits the size of the .MIM file created in this step to the minimum needed for installing the kernel image. Then, in subsequent MIB steps, you use the "small" .MIM file as input, and MIB creates a new, larger copy as output. At any time, the .MIM file is only as large as it needs to be to contain the information in it. If you do not specify /SM, MIB creates a .MIM file that corresponds in size to the total amount of target memory specified in the configuration file.

The /SM option allows you to conserve file space while retaining interim versions of the memory image in a compact form, either for backup or for use in a later partial-rebuild cycle. In addition, there is no point in ending up with a final .MIM file that is larger than the file space required by the installed program components.

If you have limited disk space, however, creating an initial full-size .MIM file into which you install components may be more efficient. By creating the full-size file, you do not need space for both an input and an output .MIM file when running MIB; although /SM may produce a smaller file, the disk space savings may be more than offset by the need to have both an input and an output file during the .MIM file creation process.

General Command Format

```
>[MCR] MIB     (RSX)
or
$ MPMIB     (VMS)
MIB>[outmim][,mapfile][,dbgfile]=[pimfile][,inmim][,stbfile][/options]
```

3.5.1 Creating a Memory Image for Debugging or Down-Line Loading

Debugging

If you specify a .DBG output file when you create the .MIM file containing the kernel, MIB creates and initializes the debug symbol file and places the kernel symbols in it.

The debug symbol (.DBG) file is an image-mode file in a special tree-structured format. The symbolic debugger, PASDBG, uses the information in this file to find and interpret the locations and structures you specify symbolically during debugging operations. When you invoke PASDBG to down-line load and debug a memory image from the host development system, the debugger loads all or part of the debug file into host memory as needed.

If you want .DBG file output from MIB, you must include a kernel symbol table file (.STB) as input in the kernel build phase. The kernel .STB file must contain debug symbol information (ISD records) as well as the normal global symbol definitions (GSD records) for the kernel. RELOC produces the kernel .STB file, from which MIB produces the initial portion of the .DBG file. The /DE option must be used in both the MERGE and RELOC steps for the kernel.

The following example shows the basic form of a MIB command line used to create a memory image file for loading and symbolic debugging on a RAM-only target. Presumably, the debugger service module (DSM) has been included in the kernel image; this action is controlled by the configuration file (Section 3.1). The example is applicable to either a mapped or an unmapped application.

```
>[MCR] MIB
MIB>KERNL1,,KERNL1=KERNL1,,KERNL1/KI/SM
```

In this example, the first KERNL1 names the output memory image file, KERNL1.MIM; the second KERNL1 names the output debug symbol file, KERNL1.DBG. The third KERNL1 in the command line specifies the input kernel image file, KERNL1.PIM; the fourth specifies the input symbol table file, KERNL1.STB. The .STB file must be included as input in order to produce the output .DBG file. In this step, you must use the /KI (kernel installation) option; it indicates to MIB that the .PIM file contains a kernel image rather than a process image. MIB will create and initialize a new .MIM file and .DBG file instead of looking for existing files with the specified name(s).

In this step, MIB copies the kernel debug records from the .STB file to the new debug file, in a special tree-structured format. In subsequent MIB steps, debug records for successive user processes are added to the existing debug file; that is, the same file is used for output throughout the build cycle and is updated in place.

Down-Line Loading

To create a memory image file for down-line loading only, using PASDBG's "load and go" capability (LOAD/EXIT command), you must have configured the kernel without the DSM when editing the configuration file. (If the kernel included the DSM, the application would load into the target but would not execute.) Also, you can omit the .DBG and .STB files in the current MIB operation and in all subsequent MIB steps, since a debug symbol file would be of no use. In this case, the MIB command line would be:

```
>[MCR] MIB
MIB>KERNL1=KERNL1/KI/SM
```

3.5.2 Creating a Memory Image for Booting

The following example shows the basic form of a MIB command line used to create a memory image file with a bootstrap installed. You can use this type of .MIM file for booting the application from a target TU58, RL01/RL02, RX50, RDxx, or RX02 device, after processing the completed file with the COPYB utility. See Chapter 7 for information on methods of loading the application.

```
>[MCR] MIB
MIB>KERNL1=KERNL1/KI/SM/BS:"bootstrap-filespec"
```

The quote signs enclosing the bootstrap file specification in the /BS (bootstrap) option are required if the file specification contains device/directory information—implying an embedded colon (:) or comma (,)—as, for example, in the specification MP:[2,10]DYBOTU.BOT or MICROPOWER$LIB:DYBOTU.BOT.

The DIGITAL-supplied bootstrap files are as follows:

DDBOTU	Unmapped TU58 bootstrap
DLBOTU	Unmapped RL01/RL02 bootstrap
DYBOTU	Unmapped RX02 bootstrap
DUBOTU	Unmapped MSCP-class disk bootstrap (RX50, RDxx)
DDBOTM	Mapped TU58 bootstrap
DLBOTM	Mapped RL01/RL02 bootstrap
DYBOTM	Mapped RX02 bootstrap
DUBOTM	Mapped MSCP-class disk bootstrap (RX50, RDxx)

These files have the file type .BOT, which is the default for the /BS option.

In this example, the first KERNL1 names the output memory image file, KERNL1.MIM; the second KERNL1 specifies the input kernel image file, KERNL1.PIM. The /KI option must be used in this step, as explained in the preceding section. The /BS option of MIB installs the bootstrap contained in the specified file at the beginning of the .MIM file before installing the kernel. DDBOTx is the TU58 (radial/serial protocol) bootstrap, used for booting from a DECtape

II cartridge. DYBOTx is the RX02 bootstrap, used for booting from a diskette, DLBOTx is the RL01/RL02 bootstrap, and DUBOTx is the MSCP-class disk bootstrap (RX50, RDxx).

Note

PASDBG uses the host-resident TD bootstrap for down-line loading an application image. You must not install any bootstrap in the .MIM file if you intend to use it with PASDBG for either debugging or load and go.

This example assumes that the configuration file used in the kernel MERGE step describes the target memory as RAM-only and specifies DEBUG=NO in the SYSTEM macro, so that the kernel image does not contain the debugger service module (DSM).

A bootstrap can be added to a memory image at any point in the development process; the addition does not have to be made in the kernel installation step. (See sections on the /BS option in Chapter 11.)

If you install a bootstrap in your application, you must be sure that certain memory requirements are met. First, an unmapped target system must have at least 3584 (7000 octal) contiguous bytes of memory, starting at location 0. A mapped target system must have at least 4096 (10000 octal) contiguous bytes, starting at location 0. Normally, that requirement should never be a problem. Second, the highest contiguous 512 (1000 octal) bytes of memory on the target system must not be initially loaded by your application. Unless you deliberately place part of your program at the very top of memory or you use almost all the memory on the target, that requirement should never be a problem in normal use either. You should nonetheless be aware of these restrictions.

3.5.3 Creating a Memory Image for a ROM/RAM Environment

The following example shows the basic form of MIB command line for creating a memory image file to be used for "programming" PROM chips.

```
>[MCR] MIB
MIB>KERNL1=KERNL1/KI/SM
```

In this example, the first KERNL1 names the output memory image file, KERNL1.MIM, and the second KERNL1 specifies the input kernel image file, KERNL1.PIM. MIB creates the .MIM file and installs the kernel image in it. The /KI option must be used in this step, as explained in Section 3.5.1.

This example assumes that the configuration file used in the kernel macro step describes the target memory as a mixture of ROM and RAM—with at least enough zero-based ROM to accommodate the kernel code and pure-data (read-only) segment. The example also assumes that the kernel image does not contain the debugger service module (DSM).

3.6 Optimizing the Kernel

The most convenient method for optimizing the kernel is to use MPBUILD and respond to the "Optimize the kernel?" question by typing YES. By this means, the execution of MPBUILD's resulting command file will automatically optimize the kernel by including only the primitives used by the application. The CONFIGURATION file SYSTEM macro should specify OPTIMIZE=YES, and the PRIMITIVES macro should not be used. Should you choose not to use the automatic method of optimizing the kernel, please read the remainder of this chapter for alternative means.

You can use MERGE's optional auxiliary output file to build a kernel that includes only the primitive service routines that your application uses, thus minimizing the size of the kernel. Using the MERGE auxiliary file method to optimize the kernel primitives is an alternative to using the PRIMITIVES macro in the system configuration file.

Developing an application generally involves several build/debug/rebuild cycles, in which you must repeatedly debug and rebuild various static processes, possibly add or delete processes, and modify certain kernel configuration parameters such as PACKETS and STRUCTURES. During the early development phases, you might accept the kernel configuration default of "all" primitive routines, or you might achieve an approximate optimization by means of the PRIMITIVES macro. After the application is complete and largely debugged, you may want to rebuild the image to exclude from the kernel all primitives unused by any process in the application. This involves a special use of MERGE to produce object files that contain only unsatisfied process references to kernel entry-point symbols. The MERGE auxiliary output capability allows you to produce such a file.

The general procedure is as follows:

1. Perform a merge operation for each static process in the application, but do not merge the kernel .STB file with the processes. In each of those merges, request only an auxiliary output (.AUX) file, which will contain the unresolved global references that would otherwise have been satisfied by the kernel .STB file. (You do not need to generate any output .MOB files.) Together, the auxiliary files for all static processes will contain references to all the primitive service modules needed in your application's kernel.
2. Next, merge a specially modified configuration file together with all of the auxiliary files and the kernel library to create the optimized kernel .MOB file, relocate the .MOB file to obtain an optimized kernel .PIM and .STB file, and install the optimized kernel image with MIB.
3. Finally, rebuild and install all the static processes, using the optimized kernel symbol table (.STB) file in the MERGE step for each process.

In more detail, the steps involved in building an optimized kernel by this method are as follows:

1. Repeat the merge operation for each static process in the application, but do not merge the kernel .STB file with the processes, and omit the output .MOB file. Instead, request an auxiliary output (.AUX) file as shown in the following sample command lines:

```
MRG>,,xxDRV=xxDRV,mpp-lib:DRVx/LB
MRG>,,APASX=APASX,mpp-lib:FILSYS/LB,LIBxxx/LB
MRG>,,BPASX=BPASX,mpp-lib:FILSYS/LB,LIBxxx/LB
MRG>,,CPASX=CPASX,mpp-lib:FILSYS/LB,LIBxxx/LB
MRG> ...
```

The auxiliary output files will contain all references to kernel routines in the application.

2. Edit your system configuration file, if necessary, as follows:

   a. Change the arguments in the SYSTEM macro to OPTIMIZE=YES and DEBUG=NO.

   b. Include the RESOURCES macro and optionally the TRAPS macro.

   c. Delete the PRIMITIVES macros, if any.

   Specifying DEBUG=NO excludes the debugger service module from the kernel, although you can still optimize even with DEBUG=YES. Specifying OPTIMIZE=YES causes kernel optimization by means of the RESOURCES, TRAPS, and PRIMITIVES macros, but omission of the PRIMITIVES macros results in no primitive modules being referenced by the configuration file. (See Section 4.1 of the *MicroPower/Pascal Run-Time Services Manual* for more information on editing the configuration file.) Assemble the configuration file. Merge the configuration file with the auxiliary files and the kernel module library (PAXU or PAXM) to create the optimized kernel .MOB file, as in the following example command line:

```
MRG>OPTKRN=OPTCFG,xxDRV.AUX,APASX.AUX,BPASX.AUX,CPASX.AUX, ... PAXx/LB
```

   In the kernel merge, the primitive references in the .AUX files will cause only those primitive modules called on by your application's processes to be included from PAXU.OLB or PAXM.OLB.

3. Complete the build cycle, starting with the kernel relocation step (Section 3.4), using OPTKRN.MOB.

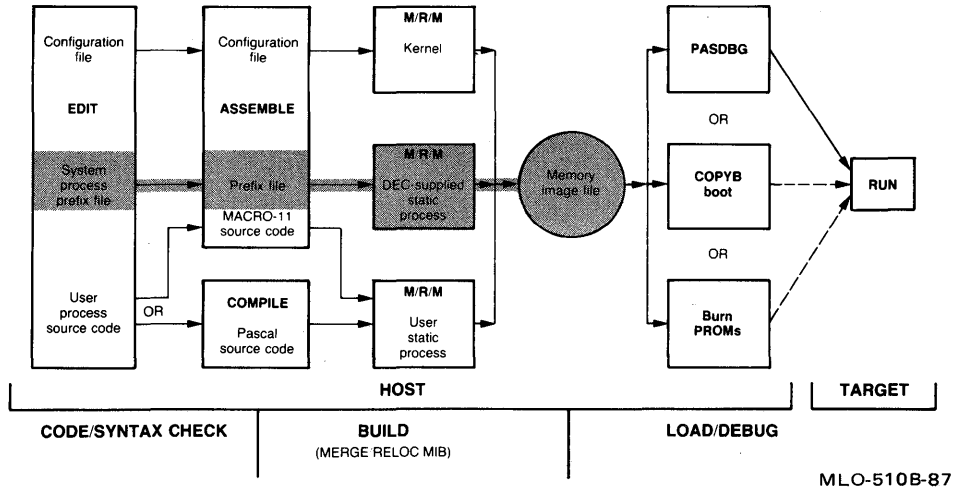
Chapter 4
Building System Processes

After building the kernel, you are ready to build the DIGITAL-supplied system processes your application needs. To build a DIGITAL-supplied system process, you must perform the following steps:

1. Edit the prefix file for the required driver or system process.
2. Assemble the edited prefix file with COMU/ML.
3. Build the memory image file.
    a. Merge the prefix file with the correct driver library and with the kernel symbol table generated when you built the kernel (see Chapter 3).
    b. Relocate the system process.
    c. Install, using MIB, the system process in the memory image (.MIM) file containing your configured kernel.
    d. Repeat for each additional system process required.

The steps involved for each system process are described below. Normally, debug support is not included, since the system processes are already debugged. Figure 4–1 illustrates the system-process build phase.

Figure 4–1: System-Process Build Phase



4.1 Edit System Prefix Module

The DIGITAL-supplied system I/O device driver processes are listed in Table 4–1. These processes are supplied in object form in two object module libraries:

```
DRVM.OLB for mapped targets
DRVU.OLB for unmapped targets
```

As shown in Table 4–1, each system process has a corresponding prefix module file in MACRO–11 (.MAC) source form. The prefix module has two functions in a system-process MERGE step—to select the required object module from the DRVx.OLB driver library and to supply that module with device-specific parameters, such as CSR/vector addresses.

For each system process to be included in your application, you must inspect and possibly modify the matching prefix module and then assemble it. For example, to build the RX02 (DY) device driver into your application, edit the DY prefix module DYPFX.MAC as needed to reflect your target hardware, and assemble the module. The *MicroPower/Pascal I/O Services Manual* describes each prefix module in detail and explains the default parameters that you may need to modify.

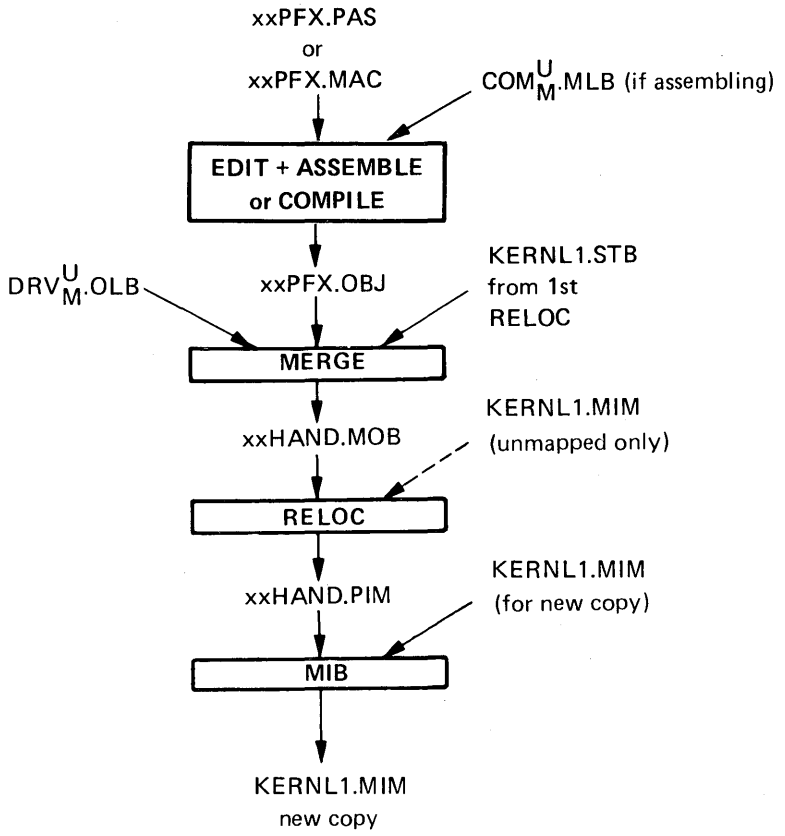
Table 4–1 shows the DIGITAL-supplied system processes for mapped and unmapped PDP–11 targets and the prefix modules that should be merged with them. Figure 4–2 shows the build phase for DIGITAL-supplied system processes.

Table 4–1: Processes and Prefix Modules for All Targets

Process	Device Name	Prefix Module
ADV11 or AXV11 A–to–D converter	AD	ADPFX.MAC
Asynchronous DDCMP	CS	CSPFX.MAC
TU58 DECtape II	DD	DDPFX.MAC
RL02 disk	DL	DLPFX.MAC
MSCP disk	DU	DUPFX.MAC
RX02 diskette	DY	DYPFX.MAC
KXT11–CA and KXJ11–CA arbiter/slave protocol	KK	KKPFX.MAC
KWV11 real-time clock	KW	KWPFX.MAC
KXT11–CA and KXJ11–CA arbiter/slave protocol	KX	KXPFX.MAC
TMSCP tape	MU	MUPFX.MAC
KXT11–CA and KXJ11–CA DMA transfer controller	QD	QDPFX.MAC
DEQNA Ethernet interface	QN	QNPFX.MAC
DZ and DHV–11 serial line interface	TT	TTPFX.MAC
Virtual memory	VM	VMPFX.MAC
DRV11–J (four ports) parallel line interface	XA	XAPFX.MAC
IEQ11–A instrument bus interface	XE	XEPFX.MAC
DPV11 synchronous serial line	XP	XPPFX.MAC
KXT11–CA/KXJ11–CA synchronous serial line interface	XS	XSPFX.MAC
DRV11–B DMA interface	YB	YBDRV.MAC
SBC–11/21 parallel line interface	YF	YFPFX.MAC
KXT11–CA and KXJ11–CA parallel port timer/counter	YK	YKPFX.MAC

See Appendix B for information on building the Pascal extended disk (XD) driver and the Pascal DRV11 (YA) driver.

Figure 4–2: Build DIGITAL–Supplied System Processes



MLO–512–87

4.2 Assembling System Prefix Modules

A prefix module written in MACRO–11 must be assembled with either the COMU.MLB or the COMM.MLB system macro library.

You can use the following form of MACRO–11 command to assemble the DYPFX.MAC file, for example:

```
>[MCR] MAC
MAC>DYPFX=mpp-lib:COMx.MLB/ML,user-dir:DYPFX
```

In the command line above, x is U for an unmapped application or M for a mapped application. As explained previously, the symbols mpp-lib: and user-dir: denote appropriate device/directory specifications as required for the input files. The example assumes that the output file, DYPFX.OBJ, is to be created in your default directory.

4.3 Merging System Prefix Modules with the System Process Library

After editing and assembling or compiling a system prefix module for a driver, merge the prefix object module with the kernel symbol table that you created during the kernel merge step (see Chapter 3) and with the appropriate device driver object library. You must include the kernel symbol table file to resolve references to primitive services in the kernel. The DRVx library supplies the required driver object module(s). The two device driver object libraries are as follows:

- DRVU.OLB for an unmapped LSI, FALCON, KXT11–CA, or KXJ11–CA target system
- DRVM.OLB for any mapped target system

General Command Format

```
>[MCR] MRG     (RSX)
```

or

```
$ MPMERGE      (VMS)
```

```
MRG>[mobfile][,mapfile][,auxfile]=infile[,infile2,...]
```

The examples below merge the DY (RX02) driver prefix object module, DYPFX.OBJ, with the DRVU or DRVM library and a kernel symbol table, KERNL1.STB, created when you relocated the kernel (see Chapter 3). The output is the merged device driver object module DYHAND.MOB.

Mapped System

```
>[MCR] MRG
MRG>DYHAND=DYPFX,KERNL1.STB,mpp-lib:DRVM/LB
```

Unmapped System

```
>[MCR] MRG
MRG>DYHAND=DYPFX,KERNL1.STB,mpp-lib:DRVU/LB
```

System process symbols are not ordinarily used in user-level debugging, so you typically do not specify the /DE option when merging system processes.

4.4 Relocating and Installing System Processes

After merging the system-process prefix file with the appropriate library files and kernel symbol table file, use RELOC to relocate the resulting merged object module (.MOB) file. This operation produces a device driver process image (.PIM) file to install in the application memory image (.MIM) file that contains your configured kernel.

RELOC and MIB each play a role in assigning addresses to a static process. The two utilities have a certain amount of interaction, and it is helpful to see the combinations of RELOC and MIB commands that you use to build various types of target applications.

For a mapped application, RELOC assigns only virtual addresses. By default, the RO segment has a zero origin, and the RW segment addressing is contiguous with the RO segment. For mapped drivers, special RELOC options must be used to satisfy the addressing requirements of driver mapped processes. MIB assigns actual physical addresses when you install the static process in the memory image.

If the application is unmapped, however, RELOC needs physical start addresses to which it can relocate read-only (RO) and read/write (RW) segments. Unlike RELOC for a mapped application, RELOC for an unmapped application assigns actual physical addresses; MIB merely installs the static-process image in the .MIM file.

After you have used the /KI option of MIB to create a memory image file, you use MIB to install any system processes that your application requires, one at a time, in that memory image.

For an unmapped memory image, the placement of the new process in the image is predetermined by the physical addresses already assigned by RELOC. For a mapped memory image, however, MIB determines the placement of the new process, based on first available space in the image, and sets up the process's virtual-to-physical memory mapping (PAR values) accordingly.

General Command Format

```
>[MCR] RLC     (RSX)
```

or

```
$ MPRELOC      (VMS)
```

```
RLC>[pimfile][,mapfile][,stbfile]=mobfile[,mimfile][/options]
```

In both examples given in the following sections, the first DYHAND names the output process image (.PIM) file DYHAND.PIM; the second specifies the input merged object file DYHAND.MOB created in the previous merge step.

4.4.1 Relocating a Mapped System Process

```
>[MCR] RLC
RLC>DYHAND=DYHAND/RO:40000/RW:60000
```

The special relocation options /RO:40000/RW:60000 are needed to ensure correct mapping of the driver's code and data segments according to the requirements for driver mapped processes. These two options force the driver's code (read-only) segment to be mapped by PAR 2 and its data (read/write) segment to be mapped by PAR 3. The /RO:40000 option forces the driver's first read-only program section—the beginning of the code segment—to start at virtual address 40000, corresponding to PAR 2. The /RW:60000 option forces the driver's first read/write program section to start at virtual address 60000, corresponding to PAR 3.

The size of a device driver's code segment is limited to 4K words, the address range of a single PAR. The same is true for the data segment. See Chapter 2 of the *MicroPower/Pascal Run-Time Services Manual* for information about driver/ISR mapping.

4.4.2 Relocating an Unmapped System Process

```
>[MCR] RLC
RLC>DYHAND=DYHAND,KERNL1
```

In this example, KERNL1 specifies the memory image file created in the previous merge phase, KERNL1.MIM, as input. Specification of this file allows RELOC to obtain the physical starting address(es) it needs for relocating the unmapped process. RELOC searches the existing .MIM file—the one to be used in the subsequent MIB step—to find the next available memory locations in which the process can be installed. That is the normal, "automatic" method of using RELOC when you are building an unmapped process; it works for both RAM-only and ROM/RAM targets.

4.4.3 Installing System Processes in Memory Image

After merging and relocating the system process, use MIB to add the process to the memory image (.MIM) file containing your configured kernel and any previously installed system processes.

General Command Format

```
>[MCR] MIB     (RSX)
or
$ MPMIB     (VMS)
MIB>[outmim][,mapfile][,dbgfile]=[pimfile][,inmim][,stbfile][/options]
```

In the following example, the DY driver process image is installed in a new copy of the .MIM file created in the initial MIB step. The example is applicable to either a mapped or an unmapped application.

```
>[MCR] MIB
MIB>KERNL1=DYHAND,KERNL1/SM
```

The first KERNL1 in the command line names the output .MIM file; the second specifies the existing KERNL1.MIM file as input. Because an input .MIM file is specified, MIB creates a new output file, copies the contents of the existing .MIM file into it, and then installs the DYHAND.PIM process image.

4.5 Repeating the System Process Build

Your output .MIM file now contains a configured kernel and one system process. Repeat the steps outlined in this chapter to install as many additional system processes in the .MIM file as your application requires. You can save a copy of the .MIM file at any point to use in other build cycles. For example, you may want to build two applications that are the same except for one or two processes. Build a .MIM file containing all the processes that are the same for both applications; then you can use that .MIM file as a base for adding the processes that are different in the two applications.

If you give the output .MIM file the same name as the input .MIM file, MIB creates a new higher-numbered version. Unless you want to keep the input .MIM file for input to another partial-rebuild cycle starting at the build point reflected in that .MIM file, using the same name is probably the most straightforward approach. If you do want to save the input .MIM file for use in other builds, however, you will probably want to give the output file a different name, such as APPLC2.

Chapter 5
Building User Processes

After building the kernel and system processes, you are ready to build the user-written processes your application needs. Building user-written processes is much the same as building system processes, described in Chapter 4. To build a user-written process, perform the following steps:

1. Create your own static-process source file(s).
2. Compile or assemble the source file(s).
3. Build the memory image file:
    a. Merge the resulting .OBJ file(s) with the correct OTS library, if necessary, and with the kernel symbol table you created while building the kernel (see Chapter 3).
    b. Relocate the user static process.
    c. Install, using MIB, the user static process in the memory image file containing your configured kernel and other system and user processes.
4. Repeat for each additional user process.

These steps are shown in Figure 5–1.

The steps involved for each user process are described below. Figure 5–2 illustrates the build phase of user-written Pascal static processes.

Figure 5–1: User-Process Build Phase

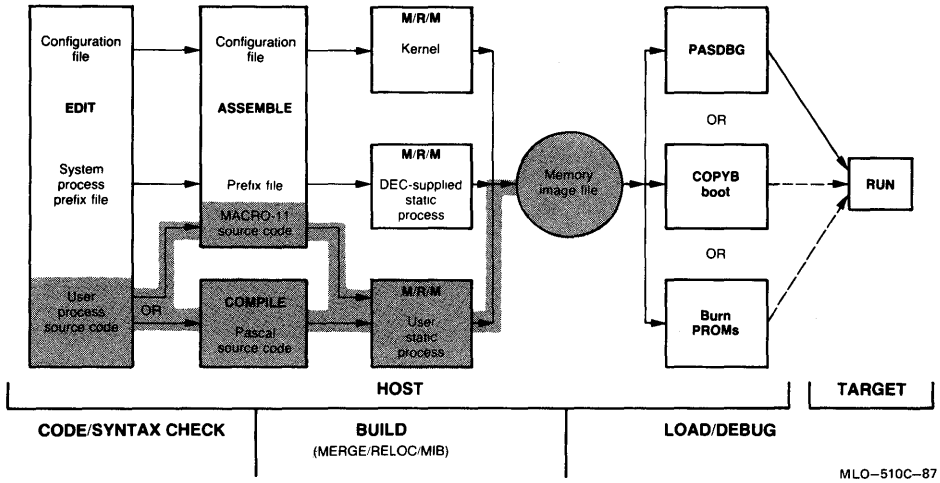
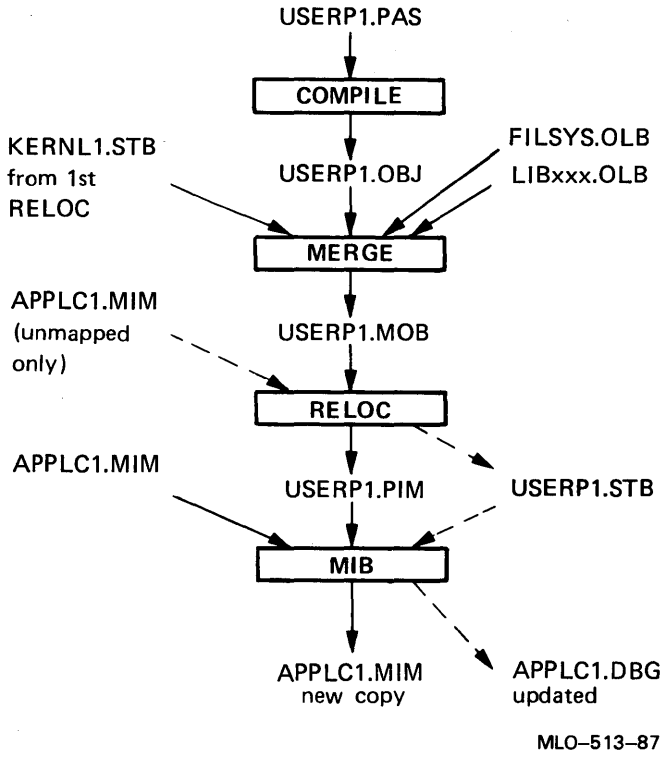


Figure 5–2: Build User-Written Pascal Static Processes



5.1 Compiling or Assembling Static-Process Source Files

5.1.1 Compiling

Use the MicroPower/Pascal compiler to compile a user process written in Pascal. Chapter 8 provides a complete description of compiler operation.

General Command Format

```
>[MCR] MPP     (RSX)
or
$ MPPASCAL     (VMS)

MPP>[objfile][,listfile]=sourcefile[/options] (RSX)
or
MPP>MPPASCAL/OBJECT=B/LIST=C A      (VMS)
```

In the following example, the source file USERP1.PAS is compiled for a target system with Extended Instruction Set (EIS) hardware, and debug symbol information is included in the object file for eventual use by the PASDBG symbolic debugger. The example is applicable to either a mapped or an unmapped application.

```
>[MCR] MPP
MPP>USERP1=USERP1/DE/IN:EIS  (RSX)
or
MPP>MPPASCAL/OBJECT=USERP1/DEBUG/INSTRUCTIONS=(EIS) USERP1  (VMS)
```

The compiler /DE option includes debug symbol information for both local and global source symbols in the object file. The /I:EIS option specifies the target system instruction set; other values for /IN: include NHD (default), FIS, and FPP. No list file is requested in the example.

5.1.2 Assembling

Use MACRO–11 to assemble a user process written in MACRO–11 assembly language. Unlike the MicroPower/Pascal compiler, the MACRO–11 assembler does not generate the type of debug symbol information needed by PASDBG. The /EN:DBG option of MACRO–11 produces a different type of debug symbol records and should not be used.

User processes must be assembled with the standard MicroPower/Pascal COMU.MLB or COMM.MLB macro library. In the following example, the source file USERP2.MAC is assembled, producing the object file USERP2.OBJ.

```
>[MCR] MAC
MAC>USERP2=mpp-lib:COMx.MLB/ML,user-dir:USERP2
```

In the command line above, COMx is COMM for a mapped application or COMU for an unmapped application.

5.2 Merging Static Processes

Each static-process .OBJ file must be merged with the kernel symbol table created by RELOC in the kernel build phase. Static processes written in Pascal must also be merged with a Pascal object-time system (OTS) library. The seven MicroPower/Pascal OTS libraries supplied with your MicroPower/Pascal kit consist of the following:

- LIBFPP.OLB, FP-11 floating-point hardware
- LIBFIS.OLB, FIS floating-point hardware
- LIBEIS.OLB, EIS hardware
- LIBNHD.OLB, no special hardware
- SUPEIS.OLB, EIS supervisor mode
- SUPFPP.OLB, FPP supervisor mode
- FILSYS.OLB, file system interface

Use the OTS library that matches your target hardware instruction set. In addition, you must specify the file system library FILSYS.OLB just before the OTS library in the MERGE command line.

General Command Format

```
>[MCR] MRG    (RSX)
or
$ MPMERGE     (VMS)

MRG>[mobfile][,mapfile][,auxfile]=infile[,infile2,...][/options]
```

In the following example, the USERP1.OBJ module, compiled from Pascal source code in Section 5.1.1, is merged with the files needed to satisfy its external references:

```
>[MCR] MRG
MRG>USERP1=USERP1/DE,KERNL1.STB,mpp-lib:FILSYS/LB,LIBEIS/LB
```

The KERNL1.STB file satisfies references to kernel primitive service entry points. FILSYS.OLB contains the appropriate file I/O routines. The LIBEIS.OLB library file satisfies references to OTS modules; MERGE includes the corresponding OTS routines in the output file. (The USERP1 source file was compiled with the /IN:EIS option in the previous step.) The /DE option is used only on the USERP1.OBJ module. Ordinarily, you would not want debug symbols for the globals of the OTS routines contained in LIBxxx.OLB or FILSYS.OLB. You should not use the /DE option on an .STB file in any event, since those symbols are normally already in the .DBG file.

The following example shows the analogous command line for merging the USERP2.OBJ module, which was assembled from MACRO-11 source code in Section 5.1.2:

```
>[MCR] MRG
MRG>USERP2=USERP2/DE,KERNL1.STB
```

If MERGE finds no valid debug records in an input module, as it will not with a MACRO-11 module, MERGE generates them for the global symbols of that module. The only difference between this example and the preceding, Pascal-oriented, one is that no object-time or file system library is required for user processes written in assembly language.

You can also merge your own user object library with the other required object files, as in the following example:

```
>[MCR] MRG
MRG>USERP3=USERP3/DE,KERN.STB,USERLIB/LB/DE,mpp-lib:FILSYS/LB,LIBxxx/LB
```

If you use the /DE option on an object library, as in this example, MERGE includes DBG records from the object library on a module-by-module basis, as required by the application.

5.3 Relocating and Installing Static Processes

RELOC and MIB each play a role in assigning addresses to a static process. The two utilities have a certain amount of interaction, and it is helpful to see the combinations of RELOC and MIB commands that you use to build various types of target applications.

For a mapped application, RELOC assigns only virtual addresses. By default, the RO segment has a zero origin, and the RW segment addressing is contiguous with the RO segment. MIB assigns actual physical addresses when you install the static process in the memory image.

If the application is unmapped, however, RELOC needs physical start addresses to which it can relocate read-only (RO) and read/write (RW) segments. Unlike RELOC for a mapped application, RELOC for an unmapped application assigns actual physical addresses; MIB merely installs the static-process image in the .MIM file.

After you have used the /KI option of MIB to create a memory image file containing a kernel (Chapter 3) and used MIB to install any system processes that your application requires (Chapter 4), use MIB to install static-process images, one at a time, in that memory image. MIB links the static processes into the kernel's static-process list, updates the memory allocation table, and removes the memory used by the process from the kernel's free-memory list.

For an unmapped memory image, the placement of the new process in the image is predetermined by the physical addresses already assigned by RELOC. For a mapped memory image, however, MIB determines the placement of the new process, based on first available space in the image, and sets up the process's virtual-to-physical memory mapping (PAR values) accordingly.

MIB can also install, in the optional debug symbol (.DBG) file, debug symbol information for a given process. MIB processes the optional .STB file generated by RELOC for each relocated process to format the debug symbol information specific to that process and add it to the .DBG file. Here again, use the /DE option in the MERGE and RELOC steps for the process in question and also in the compilation step in the case of a Pascal process.

The following sections summarize the RELOC and MIB command lines, then give a series of command examples for most build situations that you are likely to encounter. If you use MPBUILD, of course, the proper commands for RELOC and MIB are generated automatically.

5.3.1 The RELOC Command Line

Use RELOC to relocate each static process before you install it in the memory image file.

General Command Format

```
>[MCR] RLC     (RSX)
```

or

```
$ MPRELOC      (VMS)
```

```
RLC>[pimfile][,mapfile][,stbfile]=mobfile[,mimfile][/options]
```

If the application is unmapped, RELOC needs physical starting addresses to which it can relocate read-only (RO) and read/write (RW) segments. (Normally, the RO and RW segments can be placed contiguously in memory except in the ROM/RAM case.) RELOC can obtain the needed address information by inspecting the existing .MIM file in order to find the next available memory location(s) in the current memory image. To allow RELOC to do this for an unmapped process, you must include the name of the existing memory image file—USERP1, for example—as an input in the command line.

For a mapped application, RELOC assigns only virtual addresses. By default, the RO segment has a zero origin, and the RW segment addressing is contiguous with the RO segment. (The /RO and/or /RW options can be used to override the default virtual addressing if needed, as for a driver mapped process.) If the mapped process is to be used in a mixed ROM/RAM configuration, however, the RELOC command requires the /AL option, which starts the RW (RAM) segment on a 4K-word virtual address boundary. The examples below relocate the USERP1.MOB file for differing environments. The mapped examples assume a process with general, privileged, or device-access mapping. (Section 4.4.1 shows how to relocate a driver mapped process.)

Mapped - RAM-only

```
RLC>USERP1,,USERP1=USERP1/DE
```

Mapped - ROM/RAM

```
RLC>USERP1=USERP1/AL
```

Unmapped - RAM-only

```
RLC>USERP1=USERP1,USERP1/DE
```

Unmapped - ROM/RAM

```
RLC>USERP1=USERP1,USERP1
```

The command line in the two RAM-only examples requests a symbol table output file, USERP1.STB, as well as the process-image output file, USERP1.PIM. You need to specify a symbol table file as output if you want debug symbol information for the static process. MIB requires the symbol table input in the following step in order to update the debug symbol file. You must also specify the /DE option to cause RELOC to place the debug information in the symbol table file. You omit both the .STB file and /DE in the ROM/RAM cases, since they are not needed; PASDBG cannot be used with an application in ROM, of course.

The effect of the /AL option in the mapped ROM/RAM case is to begin the read/write segment of the process at the first available 4K-word address boundary following the last virtual address assigned to the read-only segment. (The read-only segment is automatically origined at virtual address 0 by default.) This ensures that the end of the process's RO segment and the beginning of its RW segment are mapped by different page address registers, which allows MIB to allocate

the segments in ROM and RAM, respectively. For example, if PAR 3 is the highest-numbered PAR used for the process's code and pure data, the /AL option forces the mapping of the process's impure data to begin with PAR 4.

(The RELOC command line for a process written in MACRO–11 is the same as for one written in Pascal.)

5.3.2 The MIB Command Line

Use MIB to install static processes in the memory image after relocating them with RELOC.

General Command Format

```
>[MCR] MIB     (RSX)
```

or

```
$ MPMIB     (VMS)
```

```
MIB>[outmim][,mapfile][,dbgfile]=[pimfile][,inmim][,stbfile][/options]
```

MIB adds the current process to the memory image. You can specify the debug file as output in the MIB command line to add this static process's symbols to the debug file you created when you built the kernel. You can use PASDBG to symbolically debug only those processes for which you include debug information in the application debug file.

The example below creates a new copy of the existing memory image file APPLC1.MIM, containing the "old" memory image with the new static process USERP1 added. The example does not produce a map file but does include debug information for the current process in the debug symbol file (APPLIC.DBG). The result is an updated application memory image in which the input static process is installed along with the kernel and the previously installed processes. (The example is applicable to either a mapped or an unmapped application.)

```
>[MCR] MIB
MIB>APPLC1,,APPLC1=USERP1,APPLC1,USERP1/SM
```

On the left-hand, or output, side of the equal sign, the first APPLC1 names the output .MIM file; the second APPLC1 specifies the existing APPLC1.DBG file, which will be updated in place with additional debug symbols. On the right-hand side of the equal sign, the first USERP1 specifies the input USERP1.PIM file, APPLC1 specifies the existing APPLC1.MIM file, and the second USERP1 specifies the USERP1.STB symbol table file created in the preceding RELOC step. (You must specify an input .STB file if you specify an output .DBG file.) You can omit both the .DBG and the .STB files when building an application without debug support, as for a ROM/RAM target system.

For an unmapped memory image, the placement of the new process in the image is predetermined by the physical addresses already assigned by RELOC. For a mapped memory image, however, MIB determines the placement of the new process, based on first available space in the image, and sets up the process's virtual-to-physical memory mapping (PAR values) accordingly. MIB creates a new output .MIM file, copies the contents of the existing .MIM file into it, and then installs the USERP1.PIM process image.

Since the output .MIM file was given the same name as the input .MIM file in the example, MIB creates a new higher-numbered version. If you want to save the original .MIM file, you must either specify a different name for the output .MIM file or copy the original .MIM file before issuing the MIB command line. In addition, because a .DBG file is specified as output, MIB processes the debug records contained in the USERP1.STB file and adds the modified records

to those already in the existing KERNL1.DBG file. If you want to save the original .DBG file, you must copy it before issuing the MIB command line. This is necessary because the .DBG file is updated in place.

If you are continuing the build discussed in Chapters 3 and 4 and this is the first user process, this example assumes that you have already copied the kernel/system-process .MIM file, KERNL1.MIM, to APPLC1.MIM and the .DBG file, KERNL1.DBG, to APPLC1.DBG.

5.4 Repeating the User Process Build

Your output .MIM file now contains a configured kernel, the required system processes, and one user static process. Repeat the steps outlined in this chapter to install as many additional user static processes in the .MIM file as your application requires.

5.5 Debugging and Rebuilding the Application

You must specify certain options when building your application if you plan to use PASDBG to debug it. Regardless of the method used to build the application, you must specify DEBUG=YES in the kernel configuration file when building an application for debugging. You can then use MPBUILD to build the application, or you can run MERGE, RELOC, and MIB yourself. Be sure to build the .MIM file in PASDBG load format.

The debug phase of application development generally involves iterative debugging and rebuilding operations. You will probably find that you have to debug and rebuild various static processes several times—some processes more times than others. Each time you modify a static process to fix a bug and then rebuild the application, you will no doubt want to use the debugger again to retest the modified application image. Therefore, you would rebuild "from scratch," without PASDBG support, only when the application is fully debugged in its host-dependent form and you are ready to down-line load and test a stand-alone version.

When you replace one or more user processes with modified versions, you do not need to perform a complete image rebuild. You can use the previously built kernel/driver .MIM, .DBG, and .STB files, assuming that they were saved, and start the rebuild at the user process phase. You then rebuild only the user processes and add them to a copy of the original kernel/driver image. The MPBUILD procedure facilitates that by subdividing the entire build cycle into two partial build cycles, one for building a kernel/driver image and one for building a complete application image. These partial cycles can be performed separately or together.

At some point in the development of an application, you may need to modify the original kernel software configuration. For example, your application may require a larger kernel common-memory pool than was originally estimated. You may also choose to optimize the kernel's primitive service modules in order to reduce the size of the kernel. Any change to the kernel implies the need to rebuild all system and user processes as well—that is, to perform an entire build cycle.

If you modify the target hardware in any way, you will also need to do a complete rebuild. For example, if you add to target memory, change from an unmapped to a mapped system or vice versa, change the number of devices supported, or change interrupt vector locations, you need to change the configuration file—and possibly some system-process prefix files—and rebuild the kernel and all processes.

When you build the final kernel and will no longer be debugging with PASDBG, you must specify DEBUG=NO in the SYSTEM macro of the configuration file. This action excludes the DSM from the kernel. All processes will again have to be rebuilt. At this point, you would probably want to recompile all Pascal-implemented processes without the /DE option to allow the compiler to perform full optimization. This action can reduce the size of some processes significantly, as well as increase their execution speed. If you use the individual build utilities rather than MPBUILD, you no longer need to specify the MERGE and RELOC /DE options in any phase. A debug symbol table is not required, so no .STB files need be input to MIB, and no .DBG file need be specified as output. The only .STB file needed at this point is the one RELOC creates for the kernel in the final kernel build phase. This file is needed for all process MERGE steps to resolve the process-to-kernel references with the latest kernel address values.

When you perform the final rebuild, you may install a bootstrap in the application .MIM file, for subsequent processing by the COPYB utility, if the stand-alone version of the application is to be loaded from a target system boot device. You will not install a bootstrap if the image is to be down-line loaded using the PASDBG LOAD/EXIT command, for initial stand-alone testing.

Chapter 6

Separation of Instruction and Data Space, and Shared Library Files

You can build your MicroPower/Pascal applications with several space-saving and virtual-space-expanding options. These include user-mode shared libraries, a supervisor-mode shared library running in supervisor mode on J11-based processors, and use of hardware separation of instruction (I) and data (D) space on J11-based processors.

This chapter discusses these techniques and their limitations and interaction. For purposes of focusing on the topics of I/D separation and shared libraries, the examples in this chapter do not show debug support. The method of specifying debug support in these examples, however, parallels the method in the Chapter 5 examples.

A shared library is a single copy of a group of subroutines in memory, capable of being shared by a number of processes. Shared libraries were added to MicroPower/Pascal primarily to permit the sharing of OTS routines among Pascal-implemented processes; in the most general case, however, a shared library can include other subroutines that you may wish to share among processes as well, subject to certain restrictions outlined later in this chapter.

Special considerations are involved in configuring your kernel for a J11-based processor if you want to use I/D separation or supervisor-mode shared libraries. You must properly define the "type" and "mmu" parameters in the PROCESSOR configuration macro to enable support for these features. In addition, you must understand certain terminology. In particular, you must be careful to distinguish between "running on a J11 processor" and "configured for a J11 processor." The latter phrase means that you have specified parameters in the configuration file to enable support for I/D separation and supervisor-mode libraries and implies that the resulting application must run on a J11 or equivalent processor. The former phrase, on the other hand, does not necessarily mean that the application has separation of instruction and data space or a supervisor-mode library; nor does it imply that the application is mapped, only that the application is running on a J11 processor.

PROCESSOR macro arguments have four relevant combinations, as follows:

- type=J11, mmu=yes—Enables full J11-based processor features, including supervisor mode and I/D separation. Use this combination when you want I/D separation and supervisor

mode. The terms "configured for a J11 processor" and "J11 mapping" refer to this choice of parameters.

- type=L1123, mmu=yes—Treats a J11-based processor as though it were a mapped LSI-11/23, without supervisor mode or I/D separation. Use this combination when you want to run on a J11 and do not want I/D separation or supervisor mode, but you do want memory mapping.
- type=J11, mmu=no—Treats a J11-based processor as though it were an unmapped LSI-11/2 or LSI-11/23 without supervisor mode or I/D separation. Use this combination when you want to run on a J11 processor but do not want mapping, I/D separation; or supervisor mode.
- type=L1123 or L112, mmu=no—Equivalent to type=J11, mmu=no.

Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual* describes the PROCESSOR macro in detail.

6.1 Separation of Instruction and Data Space

If your target processor supports hardware separation of instruction and data space, you can use this hardware feature by specifying the J11 processor type and memory mapping in your configuration file and requesting the separation when you build a particular user process. Separation of I/D space gives each general mapped user static process in your application a 32K-word virtual address space for instructions and a separate 32K-word virtual address space for data, thereby doubling the potential size of most processes. (The virtual space available for processes of other mapping types is increased proportionately.)

I/D separation incurs a small amount of overhead in both space and time. The process-context save area must be somewhat larger, and the time required to do a context option between processes is slightly longer. Except as a way to increase the virtual address space available to a program, I/D separation offers no advantages unless the program (static process) is built with a supervisor-mode shared library. Use I/D separation when you are faced with addressing limitations that I/D separation can solve for you; otherwise, ordinary mapped format is just as good.

You can mix processes with and without I/D separation in the same application.

6.1.1 Restrictions on I/D Separation

A process built with I/D separation can run only with a kernel configured for a mapped J11-type processor in the configuration file PROCESSOR macro. You can, however, include processes built without I/D separation as part of an application containing processes that do have I/D separation. The kernel itself is always built without I/D separation.

You cannot build a user-mode shared library for a target configured for I/D separation; you can only build a supervisor-mode shared library.

Building a driver with I/D separation is not recommended.

6.1.2 Building a Process with I/D Separation

Assuming that your target system hardware supports I/D separation, two requirements must be met for building a process having I/D separation:

- Specify J11 as the processor type and mmu=yes in the PROCESSOR macro of the system configuration file.
- Use the /ID option of RELOC to build the static process with I/D separation; if you use MPBUILD and include /IDS when you specify the user-process file name, MPBUILD will generate the proper options for RELOC automatically.

For example, a typical RELOC command line might be:

```
>[MCR] RLC
RLC>PROC1.PIM=PROC1/ID
```

That will build the specified static process with I/D separation. The first RO program section of instruction space will begin at virtual address 0 of I-space mapping, and the first RO program section of data space will begin at virtual address 0 of D-space mapping; their RW segments will immediately follow their RO segments in memory.

You can use other options together with the /ID option to achieve other results. If you have a ROM/RAM environment, you will want to use the /AL option to start the RW instruction segment and the RW data segment on the next available 4K-word boundaries following their RO segments.

In addition, if you have special addressing requirements, you can use the /DR, /DW, /RO, and /RW RELOC options, singly or in combination, to specify particular base addresses for RO and RW data space and for RO and RW instruction space. You can also use the /QB option to achieve the same results.

6.2 Shared Libraries

When you build a MicroPower/Pascal application, each static process is normally a self-sufficient unit. That is, all generated code, Pascal OTS routines, and other support that the process requires, with the exception of kernel primitives, are part of the process itself. For example, if two separate processes require the same routines from a Pascal OTS library (LIBxxx), the processes will contain separate copies of the modules. If you have written subroutines that are used by several of your static processes, they too will be built into each process that uses them.

Obviously, if this duplication is extensive, a large amount of physical memory could be saved if a single copy of each OTS routine were shared among all the processes that use it. The situation is less critical in a mapped application, since each process has its own 32K-word virtual address space, but even a mapped application may encounter space constraints. In an unmapped application, the 28K- or 30K-word memory limit for all processes is even more constraining.

As noted above, processes do share the kernel primitives; a single copy of the configured kernel is used by all processes in the application. References by processes to kernel primitives are resolved when each process is built, by merging each process with the kernel symbol table (.STB) file created at kernel build time.

Shared libraries work in a similar manner. A single copy of the subroutines in a library is shared by any number of processes. When you build a shared library, a library .STB file is produced that reflects the contents of that shared library. Then, when you build a process that references routines in the shared library, you merge the process with the library .STB file; the .STB file resolves the references to the routines in the shared library.

6.2.1 Types of Shared Libraries

MicroPower/Pascal supports two types of shared libraries: supervisor mode and user mode. A supervisor-mode library requires supervisor-mode memory mapping support on the target system, as found on J11-based processors, whereas user-mode libraries are applicable to any unmapped target system and any mapped system without J11 mapping.

Note

Be careful not to assume that a "user-mode" library necessarily contains any user-written routines. A user-mode shared library may well contain the same routines that a supervisor-mode library contains. "User mode" refers to the processor mode used by the library, not to the contents of the library. Both types of shared library typically contain only the Pascal OTS routines, although either type of shared library can contain user-written routines as well.

A supervisor-mode library uses the supervisor-mode mapping registers and resides almost entirely outside a process's address space. A supervisor-mode library thereby saves both physical and virtual address space for a process. Physical space is saved because processes can share routines in the library, and each process does not need to have its own copy of a routine; virtual space is saved because the supervisor-mode library resides in a virtual address space that is separate from that occupied by the process itself.

The referencing process can have I/D separation or not. The data-space active page registers (APRs) for the supervisor-mode library, which are not used to map data within the library, map back to the referencing process. This gives the library access to data within the referencing process. See the *MicroPower/Pascal Run-Time Services Manual* for more information on mapping in supervisor-mode libraries.

User-mode libraries save on physical address space by sharing routines among processes, but the entire library is mapped in the virtual address space of each referencing process. User-mode libraries do not save on virtual address space for mapped processes, therefore, since they use the same virtual address space as the processes themselves. (For an unmapped application, a shared library offers a clear advantage for the application as a whole, since there is no physical/virtual space distinction.)

See Section 2.1.7 of the *MicroPower/Pascal Run-Time Services Manual* for a description of process mappings and supervisor-mode library mapping.

6.2.2 Restrictions on Shared Libraries

You can build your application with either a single supervisor-mode library or one or more user-mode libraries, but not both.

Interlibrary references are not allowed; that is, the routines in one library cannot reference routines in another library. If such references are required, you must build a new, combined library containing all the needed modules. The routines within a library, however, can reference each other without restriction.

Routines in a shared library cannot call routines in a process that references the library. Therefore, if you put any Pascal routines in a shared library, all Pascal OTS routines required to support them must also be in that library. Moreover, you cannot put the following kinds of Pascal routines in a shared library:

- Any routine that does exception handling
- Any initialization or termination routine

All user-written routines in a shared library must be reentrant. All code and data must be in read-only program sections. If you have any read/write sections of nonzero length, you will get the RELOC error message "Libraries must be reentrant—read/write section not allowed." A routine in a shared library can save values only in the registers, on the stack, or in a parameter block in the main calling program, through a pointer passed to the library routine in a register or on the stack.

A supervisor-mode shared library can have, at most, 4K words of data.

A supervisor-mode library must always contain CTS support for any process that references it. It cannot contain just user-coded MACRO or Pascal modules. A supervisor-mode library can contain user-coded MACRO or Pascal modules in addition to the OTS support. Moreover, a process written in MACRO–11 cannot reference a supervisor-mode library.

An interrupt service routine (ISR) cannot reference a shared library. In general, therefore, device drivers cannot reference shared libraries.

When you are building a user-mode shared library for a mapped application, you can choose to build it relocatable or absolute. If you build a relocatable shared library—that is, if you do not fix the library routines at particular addresses with the /UL:addr option—the code must be position independent (PIC). If you build an absolute library, on the other hand, by using the /UL:addr option when you build the library, the PIC requirement does not apply. (The compiler output is not position-independent code.) Any mapped user-mode library with Pascal code in it must be built absolute.

All instructions and data in a user-mode shared library must be contiguous; you cannot use the /QB option of RELOC to break it into segments. Similarly, all instructions in a supervisor-mode shared library must be in a contiguous segment, and all data must be in another contiguous segment (the normal result when you create a process with I/D separation). Note that an application can have only one supervisor-mode shared library.

For user-mode, having more than one shared library is normally not useful. In a few complex cases for mapped applications, the size of a single library and the varying needs of the processes that reference it dictate splitting the library into two smaller libraries to overcome virtual-addressing limitations.

6.2.3 Building Shared Libraries

The basic procedure for building a shared library is as follows:

1. Create a merged object module with the routines in the shared library.
2. Use RELOC to produce a .PIM file and a .STB file.
3. Use MIB to install the library.

Always build and install the library before building any process that references the library. If you have included debug support for a shared library and a referencing process, you can access the library with the debugger through the process.

6.2.4 Building a Supervisor-Mode Shared Library

If your target processor implements supervisor mode, you can use a supervisor-mode shared library when building your application. Supervisor mode is implemented on a J11 or equivalent processor.

If you are including the Pascal OTS in the shared library, you can use only two Pascal OTS object libraries when you build a supervisor-mode shared library—SUPEIS.OLB and SUPFPP.OLB, which assume EIS or FPP, respectively. Choose the object library that is appropriate for your application and target system. Do not attempt to use any of the LIBxxx object libraries when building a supervisor-mode shared library; they will not execute correctly.

To build a supervisor-mode library with the Pascal OTS in it, first compile the Pascal processes that will make up your finished application. Then use MERGE to create a .AUX file for each process. That file is an object module containing a list of all the global references that remain unresolved after the merge.

Include a .AUX file output file name in the MERGE command line and include your kernel .STB file as input, but do not specify any Pascal OTS object library file name or the FILSYS object library. In a later step, MERGE uses the .AUX files you create to extract the required Pascal OTS routines (which you would not know otherwise) from the proper supervisor-mode OTS library and from the FILSYS library for inclusion in the supervisor-mode shared library. If you have compiled the two Pascal processes PROC1 and PROC2, the MERGE commands to create the .AUX files for them would be as follows:

```
MRG>,,PROC1.AUX=PROC1.OBJ,KERNEL.STB
MRG>,,PROC2.AUX=PROC2.OBJ,KERNEL.STB
```

After creating .AUX files for all processes that will reference the shared library, run MERGE again. This time, specify the .AUX files, the FILSYS, and a Pascal OTS library as input and a name for the shared library .MOB file as output.

The following example uses the two .AUX files created in the previous step to extract the necessary support routines from the libraries FILSYS.OLB and SUPEIS.OLB for inclusion in the supervisor-mode shared library .MOB file SUPLIB:

```
MRG>SUPLIB.MOB=PROC1.AUX,PROC2.AUX,mpp-lib:FILSYS/LB,mpplib:SUPEIS/LB
```

Next, use RELOC to create shared library .PIM and .STB files from the .MOB file:

```
>RLC
RLC>SUPLIB.PIM,,SUPLIB.STB=SUPLIB.MOB/SL
```

During creation of the supervisor-mode library, RELOC looks up the file MP1:[1,1]LIBSUP.OBJ on an RSX host or MICROPOWER$LIB:LIBSUP.OBJ on a VMS host to read in the library list element. Be sure that file is available.

Finally, use MIB to create the final .MIM file:

```
>MIB
MIB>APPLIC.MIM=SUPLIB.PIM,APPLIC.MIM/SM
```

If you want to include MACRO and/or Pascal modules in the supervisor-mode library in addition to the OTS, merge the modules along with the .AUX files when creating the shared-library merged object (.MOD) file. For example, if you want to include a Pascal module PASMOD.PAS and a MACRO module MACMOD.MAC in the example above, first compile PASMOD and assemble MACMOD.

As mentioned above, compile the Pascal processes that will make up your finished application. Then use MERGE to create a .AUX file for each process.

After creating the .AUX files for all processes that reference the shared library, run MERGE again. This time, specify the .AUX files and the additional modules:

```
*SUPLIB.MOB=PROC1.AUX,PROC2.AUX,PASMOD.OBJ,MACMOD.OBJ,KERNEL.STB,MPPLIB:FILSYS.OLB,SUPEIS.OLB
```

Any additional OTS support necessary for PASMOD will be placed in the .MOB file, besides the support necessary for PROC1 and PROC2.

6.2.5 Building a User-Mode Shared Library

If your target processor does not support supervisor mode and I/D separation or if you do not want to use supervisor mode or I/D separation, you can build a user-mode shared library.

You can build three kinds of user-mode shared libraries:

- Unmapped
- Mapped relocatable
- Mapped absolute

Regardless of which kind of user-mode shared library you build, you use one of the following OTS object libraries, if needed: LIBNHD, LIBFIS, LIBEIS, or LIBFPP.

6.2.5.1 Unmapped User-Mode Shared Libraries

During the RELOC step, either specify /UL and let RELOC look up the starting address in the input .MIM file, or use /UL:addr to specify a starting address.

To build an unmapped user-mode library with the Pascal OTS in it, compile the Pascal processes that will make up your completed application. Then, use MERGE to create a .AUX file for each process. Include a .AUX file output file name in the MERGE command line and include your kernel .STB file as input, but do not specify any Pascal OTS object library file name or FILSYS object library. In a later step, MERGE will use the .AUX files you create to select the required Pascal OTS routines from the proper user-mode OTS library for inclusion in the user-mode shared library. If you have compiled the two Pascal processes PROC1 and PROC2, the MERGE commands to create the .AUX files for them would be as follows:

```
MRG>,,PROC1.AUX=PROC1.OBJ,KERNEL.STB
MRG>,,PROC2.AUX=PROC2.OBJ,KERNEL.STB
```

After creating .AUX files for all processes that will reference the shared library, run MERGE again. This time, specify the .AUX files, the FILSYS library, and a Pascal OTS library (LIBxxx) as input and a name for the shared library .MOB file as output.

The following example uses the two .AUX files created in the previous step to extract the necessary support routines from FILSYS and LIBEIS for inclusion in the user-mode shared library USRLIB:

```
MRG>USRLIB.MOB=PROC1.AUX,PROC2.AUX,KERNEL.STB,mpp-lib:FILSYS/LB,mpplib:LIBEIS/LB
```

Next, use RELOC to create .PIM and .STB files from the .MOB file:

```
>RLC
RLC>USRLIB.PIM,,USRLIB.STB=USRLIB.MOB,APPLIC.MIM/UL
```

If you want, you can include a base address as part of the /UL:addr option of RELOC. Alternatively, if you omit the addr argument, as shown in the example above, RELOC assigns a base address by looking for the first free space in the input .MIM file that you specify.

During creation of the user-mode library, RELOC looks up the file MP1:[1,1]LIBUSR.OBJ on an RSX host or MICROPOWER$LIB:LIBUSR.OBJ on a VMS host to read in the library list element. Be sure that file is available.

Finally, use MIB to create the final .MIM file:

```
>MIB
MIB>APPLIC.MIM=USRLIB.PIM,APPLIC.MIM/SM
```

6.2.5.2 Mapped User-Mode Shared Libraries

You can build a mapped user-mode shared library either relocatable or absolute. If you build it relocatable, the library can be mapped anywhere in a given referencing process's virtual address space.

If you build the shared library absolute, you select the virtual addresses for it when you build it. Thus, the library is fixed in the virtual address space of all processes that reference it.

Figure 6–1 shows an example of a relocatable shared library. Three processes reference the relocatable shared region RELSHR: process A, process B, and process C. The shared region is 6K words long and therefore requires that much space in the virtual address space of the three processes. Process A is 16K words long and uses APRs 0 to 3 to map its code and data. Two APRs are needed to map the library, so any consecutive pair starting with APR 4 could be used to map the library. Here, APRs 6 and 7 are used.

Process B is 8K words long and uses APRs 0 and 1 to map its code and data. Process B uses the first available APRs after that—APRs 2 and 3—to map the library.

Process C is 21K words long and uses APRs 0 to 5 to map its code and data. Process C uses the only available APRs—APRs 6 and 7—to map the library.

If you do not specify which address—and which corresponding APR—is to be used to start mapping a relocatable shared library, RELOC selects the first available set in the referencing process's virtual address space. In Figure 6–1, for example, RELOC would have selected 100000(octal), or APR 4, to start mapping the library RELSHR. Both process B and process C

use the first available set of APRs. You can override this default selection by using the /LS RELOC option.

Figure 6–2 shows an example of an absolute shared library. Only two of the three processes can reference the absolute shared library ABSSHR: process D and process E. The absolute shared library ABSSHR is 6K words long and is built to occupy virtual addresses 120000(octal) to 150000(octal). These addresses correspond to APRs 5 and 6. Processes D and E can reference the library because APRs 5 and 6 are not used for process code and data. However, process F is 24K words long; even though it has 8K words of virtual address space available to map the shared library, APR 5, which corresponds to virtual address 120000(octal), has been allocated to the code and data of the process. If the shared library ABSSHR were built relocatable, task F could reference it.

In general, you should build a shared library relocatable whenever possible. However, the code in the library must be position independent (PIC) in order to do this. That is, the code must execute correctly regardless of its location in the address space of the referencing process. In addition, all data in the library must be address independent. The Pascal compiler does not produce PIC code. Therefore, you must build any shared library that contains Pascal modules absolute.

Figure 6–1: Mapped Application, Relocatable User-Mode Shared Library

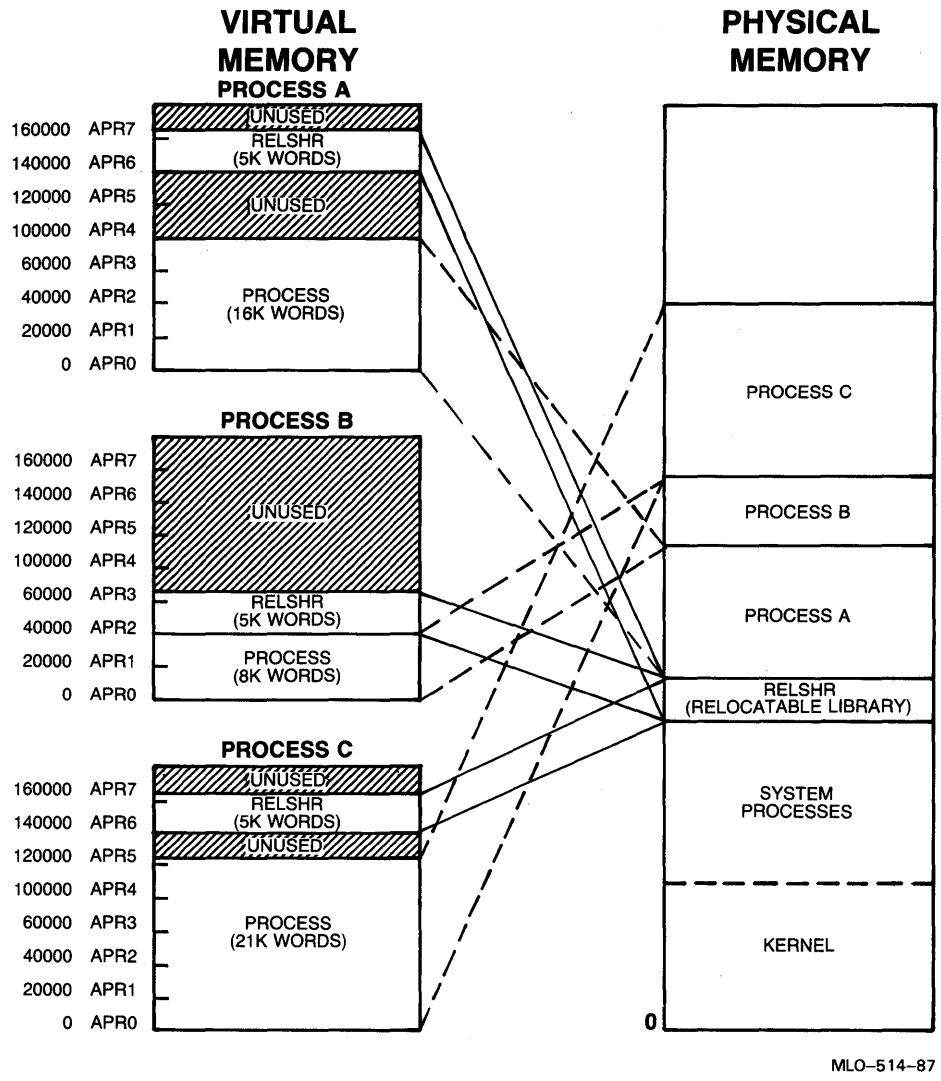
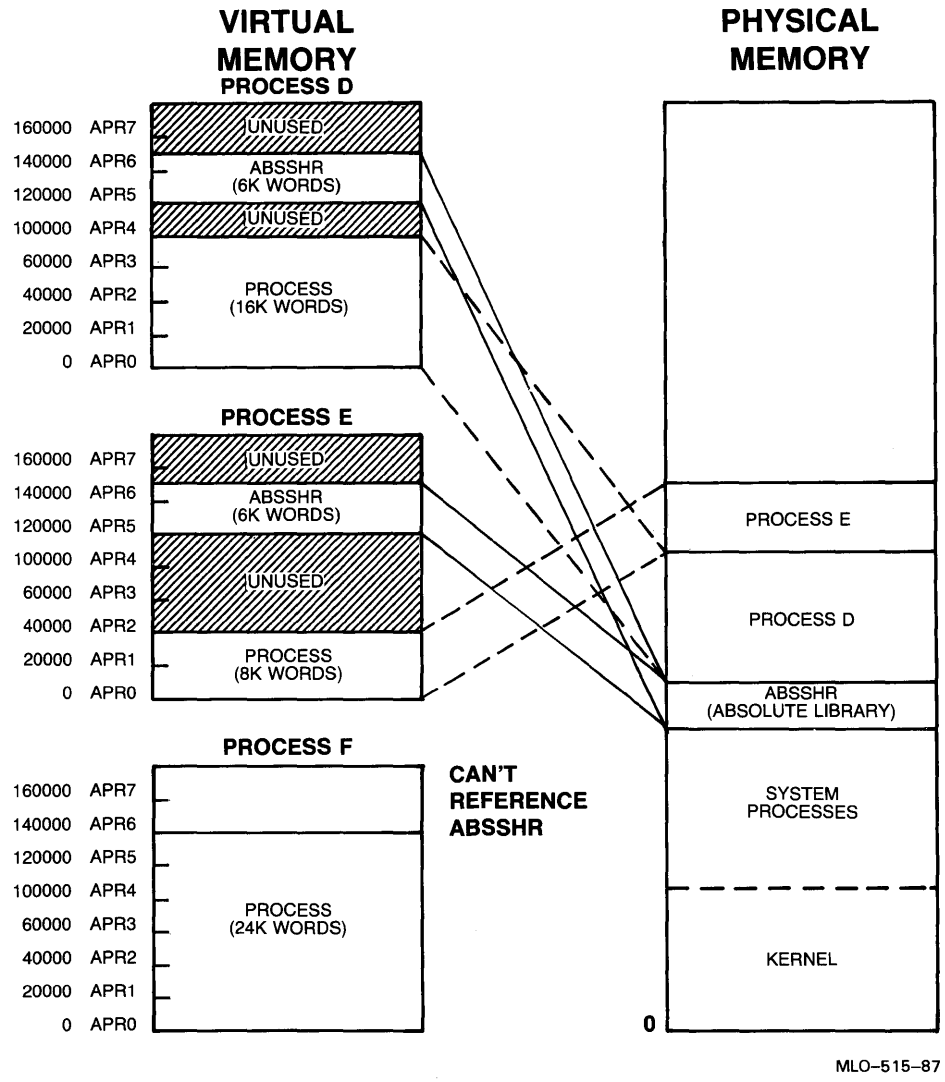


Figure 6–2: Mapped Application, Absolute User-Mode Shared Library



MLO-515-87

To build a mapped user-mode library with the Pascal OTS in it, compile the Pascal processes that will make up your completed application. Then, use MERGE to create a .AUX file for each process. Include a .AUX file output file name in the MERGE command line and include your kernel .STB file as input, but do not specify any Pascal OTS or FILSYS object library file name. In a later step, MERGE will use the .AUX files you create to select the required Pascal OTS routines from the proper user-mode OTS library for inclusion in the user-mode shared library. If you have compiled the two Pascal processes PROC1 and PROC2, the MERGE commands to create the .AUX files for them would be as follows:

```
MRG>,,PROC1.AUX=PROC1.OBJ,KERNEL.STB
MRG>,,PROC2.AUX=PROC2.OBJ,KERNEL.STB
```

After creating .AUX files for all processes that will reference the shared library, run MERGE again. This time, specify the .AUX files, the FILSYS object library, and a Pascal OTS object library as input and a name for the shared library .MOB file as output.

The following example uses the two .AUX files created in the previous step to extract the necessary support routines from FILSYS and LIBEIS for inclusion in the user-mode shared library USRLIB:

```
MRG>USRLIB.MOB=PROC1.AUX,PROC2.AUX,KERNEL.STB,mpp-lib:FILSYS/LB,mpp-lib:LIBEIS/LB
```

Next, use RELOC to create .PIM and .STB files from the .MOB file:

```
>RLC
RLC>USRLIB.PIM,,USRLIB.STB=USRLIB.MOB/UL
```

This will build a relocatable user-mode shared library. If you want to build an absolute library, include a base virtual address as part of the /UL:addr option of RELOC. The address must be on a 4K-word virtual address boundary.

During creation of the user-mode library, RELOC looks up the file MP1:[1,1]LIBUSR.OBJ on an RSX host or MICROPOWER$LIB:LIBUSR.OBJ on a VMS host to read in the library list element. Be sure that file is available.

Finally, use MIB to create the final .MIM file:

```
>MIB
MIB>APPLIC.MIM=USRLIB.PIM,APPLIC.MIM/SM
```

If you are including your own Pascal or MACRO–11 routines in the library, in addition to the Pascal OTS, modify the procedure above as follows:

1. Compile or assemble the source modules to produce object modules.
2. Add these object modules to the MERGE command with the .AUX files, the kernel .STB file, and the FILSYS and LIBxxx object libraries.
3. If you are including any Pascal routines in the library, you must build the library absolute by specifying a starting virtual address with the /UL option.

6.2.5.3 Multiple User-Mode Libraries in an Application

In most cases, you need not use more than one user-mode shared library. If you have multiple user-mode libraries in an application, each library name must be unique. The default name taken from the .TITLE statement of the shared library object file is $USRLB for a user-mode library. For subsequent user-mode libraries, use the /NM option of RELOC to override the name $USRLB. If you do not do that, the error "Library with the same name, $USRLB, is already installed in—filespec" will be reported by MIB when you try to install the second library.

One possible use for having multiple shared libraries would be to have the Pascal OTS in one library and one or more user MACRO modules in another library. The following steps show how to build a user-mode shared library with user MACRO modules (and no Pascal OTS support):

1. Assemble the modules.

2. Merge to produce a shared library .MOB file.

```
MRG>MACLIB=MOD1,MOD2,MOD3,KERNEL.STB
```

No OTS (FILSYS or LIBxxx) is needed, since everything is in MACRO. After this, the build is the same as for any other shared library.

6.2.5.4 Building a Process to Reference a Shared Library

The basic procedure for building a process that references a shared library is as follows:

1. Build and install the shared library. Get a shared library .STB file during the RELOC step.
2. Use MERGE to merge the process's object modules with the kernel .STB file and the shared library .STB file. If the shared library contains Pascal OTS support for the process, omit the OTS object libraries (FILSYS, LIBxxx, and SUPxxx) from the command line.
3. Use RELOC to relocate the process normally, except for relocatable mapped libraries in which you wish to specify a starting virtual address for mapping the library, rather than having RELOC use the first available addresses.
4. Use MIB to install the process.

If you want to perform an iterative form of merge, you must specify any shared library .STB files as the last step. MERGE checks to see if /UL or /SL is specified in a file containing program sections from shared libraries and further checks that all program sections in that file are shared library program sections. The partial .MOB file may contain some library program sections and some nonlibrary program sections.

If you have included debug support for a shared library and a referencing process, you can access the library with the debugger through the process.

6.2.6 Building a Process to Reference a Supervisor-Mode Library

Except for including the /UL option, you follow the same steps that you do when you build a process to reference the kernel. You must include the library's .STB file in the MERGE command line. The following MERGE command line builds a .MOB file for a process that references a supervisor-mode library with Pascal OTS support in it:

```
>MRG
MRG>PROC1.MOB=PROC1.OBJ,KERNEL.STB,SUPLIB.STB/SL
```

The /SL option indicates that the .STB file is for a supervisor-mode shared library.

RELOC must in turn process the .MOB file. If you include the /ID option (separate instruction and data space) in the RELOC command line, RELOC will take one of two actions, depending on the content of the supervisor-mode library. If the supervisor-mode library has no RO data sections—recall that all supervisor-mode libraries are built with I/D separation—RELOC assigns a base virtual address of 0 to the process's data area. On the other hand, if the supervisor-mode library has a data area, the first 4K words of virtual data space are allocated to the supervisor-mode library's data area, and the default base address for the process's data area is 20000. Thus, APR 0 maps the supervisor-mode library's data area, and APR 1 through 7 map the user process's data area. You can override this default address assignment by including the /DR:addr option in the RELOC command line. Keep in mind that if the supervisor-mode library has data and you start the process's data at 0, the process's data mapped by APR 0 will not be accessible from the supervisor-mode library.

If you do not include the /ID option to separate instruction and data space in the process, similar conditions apply. If there is no supervisor-mode data area, the base virtual address for user RO program sections defaults to 0. If there is supervisor-mode data, RELOC begins the user process RO program sections at 20000. In this case, you can override the default address assignments by including the /RO:addr option in the RELOC command line. If the supervisor-mode library has data and you start the RO program section at 0, the process's instructions and data mapped by APR 0 are accessible from the supervisor-mode library.

You can also specify other starting addresses if necessary, whether or not you separate instruction and data space, by including the /RO, /RW, or /QB options. In addition, if you do separate instruction and data space, you can use the /DR and /DW options as well.

6.2.7 Building a Process to Reference One or More User-Mode Libraries

In general, you follow the same steps that you do when you build a process to reference the kernel. You must include the library's .STB file in the MERGE command line, just as you include the kernel .STB file. The following MERGE command line builds a .MOB file for a process that references the kernel and two user-mode libraries, assuming, for example, that ULIB1 is the Pascal OTS library:

```
>MRG
MRG>PROC1.MOB=PROC1.OBJ,KERN.STB,ULIB1.STB/UL,ULIB2.STB/UL
```

RELOC must in turn process the .MOB file. RELOC requires no special options to process referenced libraries. For a mapped application, RELOC by default allocates virtual address space first to the processes, then to absolute libraries, then to libraries with specified virtual base addresses, and finally to any relocatable libraries, assigning starting addresses for the relocatable libraries at the first available 4K-word virtual address boundary. In the unmapped case, RELOC allocates memory as requested and issues an error message if the specified pieces do not fit.

In either the mapped or the unmapped case, the process and all of its referenced user-mode libraries can access no more than eight noncontiguous memory segments. MIB reports an error if you exceed this limit. RELOC reports an error for an address between the process code and any referenced library.

You can override the default allocation algorithm determining starting addresses for relocatable mapped shared libraries, if necessary, using the /LS:name:addr option, where name is the user-mode library module name, and addr is the desired base virtual address; the address must be on a 4K-word address boundary. For example:

```
>RLC
RLC>PROC1.PIM=PROC1.MOB/LS:LIB1:100000:LIB2:140000
```

6.2.8 Debugging New Processes in Applications Having Shared Libraries

As an alternative to having a shared library, you can build all OTS modules into the process. Use this technique if you have an application with a shared library and one or more debugged processes and you want to add a new process and debug it. Use this technique to avoid having to rebuild the library and all of the debugged processes. In such an instance, omit the shared library .STB file and include LIBxxx (not SUPxxx) even for an application having a supervisor-mode library. Only LIBxxx routines work outside of a supervisor-mode library. Later, when the new process is debugged, rebuild the shared library and all the referencing processes.

Chapter 7
Methods of Application Loading

To execute the finished (debugged, then rebuilt without debug support) MicroPower/Pascal target application developed on the host development system, you must load the application into target memory. You do that in one of three ways. For a RAM-only target environment, you can either down-line load from the host system or bootstrap the application from a storage volume on a target system device (TU58 DECtape II; RL01/RL02, RD51, or RD52 disk; or RX02 or RX50 diskette). For a ROM/RAM target environment, you can—if you have suitable PROM-programmer hardware and control software—place the application in programmable read-only memory chips (PROM) and install the PROM in the target system.

7.1 Down-Line Loading the Application

The LOAD/EXIT form of the PASDBG LOAD command permits you to down-line load an application for independent execution—that is, for execution without further target/debugger interaction. LOAD/EXIT causes PASDBG to load a specified .MIM file into the target system, requirements for down-line loading of) start the application, and then EXIT, returning you to system level. To be loaded in this manner, the memory image must have been built without debugger support; specifically, the debugger service module (DSM) must not be included in the kernel image. (The SYSTEM configuration macro of the configuration file used to build the application must specify DEBUG=NO.) Otherwise, the application will load but will not execute.

For loading by means of LOAD/EXIT, the target must have the same hardware configuration as required for debugging. In particular, the host/target terminal line must be connected to the target's console terminal port as for debugging. In addition, the host-side line speed and other characteristics must be set, and the logical device TD: assigned, as described for debugging. (See Appendix A of the *MicroPower/Pascal Debugger User's Guide*.)

Bootstrap Load Format

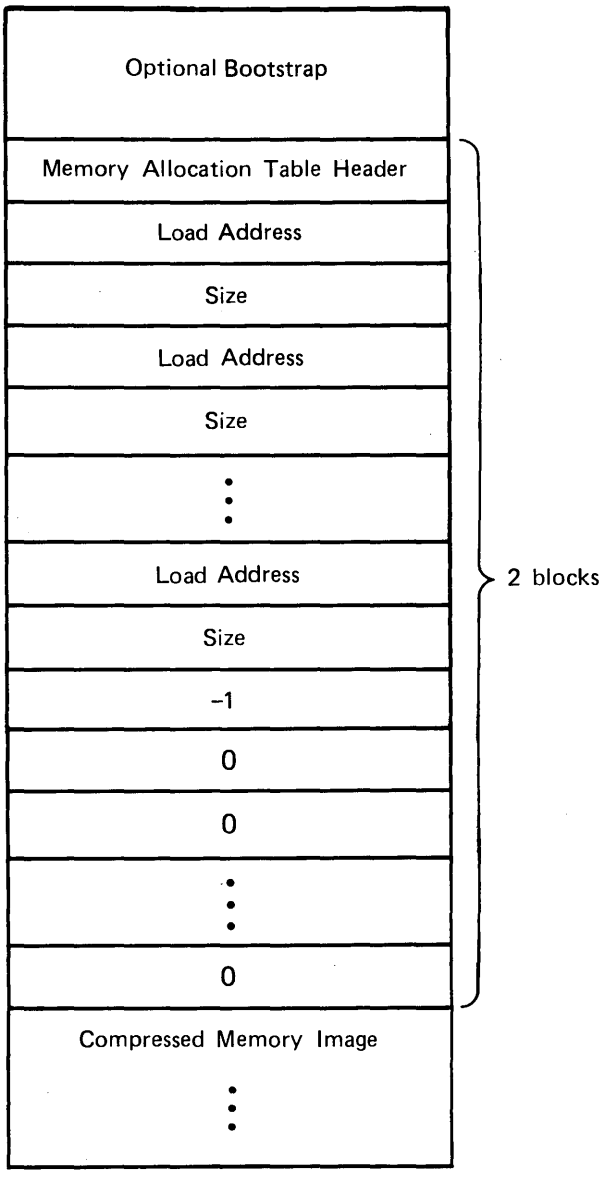
A .MIM file in bootstrap load format is identical to PASDBG load format, except that it contains a bootstrap at the beginning of the file. (See Figure 7–1 and Appendix C.) DIGITAL supplies bootstraps for all disk or disk-like devices supported on a target system. You specify the appropriate bootstrap file for your system with the MIB /BS option (Section 11.4.1), and MIB installs the bootstrap at the front of the .MIM file.

After you have built a complete memory image, you copy the .MIM file onto a suitable storage volume—one matching the type of bootstrap installed—using the RSX FLX utility or the VMS V4 EXCHANGE utility; then use the MicroPower/Pascal COPYB utility to make the volume bootable from a device on your target system. See Chapter 12 for further details.

You can install a bootstrap either when you create the .MIM file in the kernel build step or at the end of a build cycle. The latter strategy is convenient if you think you might want the same memory image to be bootable from several devices. If you build a complete .MIM file with no bootstrap installed, you can then create copies of it with different bootstraps, prefixing the bootstrap appropriate for the desired boot device.

Alternatively, you can use the /RB option to remove an installed bootstrap from the .MIM file so that you can install a different bootstrap in its place.

Figure 7–1: PASDBG or Bootstrap Load Format .MIM File



7.2 Bootstrapping the Application from a Storage Device

If the host- and target-system hardware configurations include mass-storage devices of the same type—for example, DECtape II or RX02 diskettes—you can prepare a bootable volume on the host system and bootstrap the application directly from the volume on the target.

The following steps are necessary to load the application in this way:

1. Build the application memory image without debug support—no DSM in the kernel—and install the appropriate bootstrap in the .MIM file. The bootstrap can be installed either when the application image is built, by means of MPBUILD, or as a separate MIB operation through use of the /BS option of the MIB utility program. (See Section 2.2.3.5, Chapter 3, or Chapter 11.)
2. Copy the memory image file to the storage volume that you will later transport to the target system. The volume must have, or be initialized to, the RT-11 file format and thus must be mounted as a foreign volume on the host system's device drive. You use the FLX file-transfer utility or the V4 VMS EXCHANGE utility to initialize the volume, if necessary, and to perform the copy operation.
3. Invoke the MicroPower/Pascal COPYB utility to make the volume bootable. COPYB modifies and moves the bootstrap contained in the memory image file to the bootblock of the volume. Chapter 12 describes the COPYB utility program.
4. Mount the bootable volume in the target system's device drive, and power up the target processor.

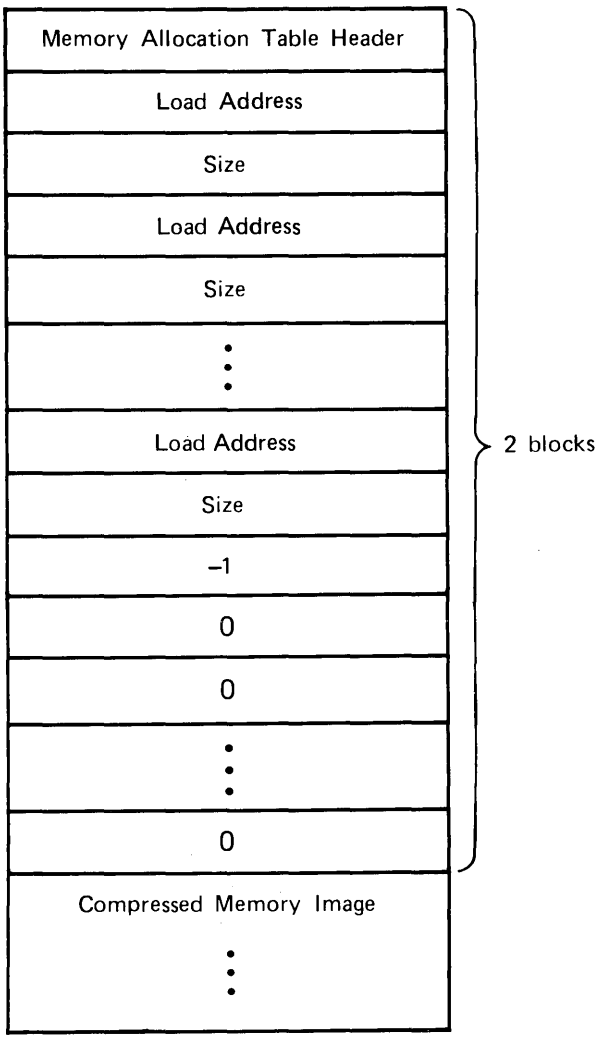
A suitable boot ROM must be included in the target hardware configuration. When you initiate the bootstrap procedure by powering up the target processor, the hardware bootstrap reads block 0 of the bootable volume into memory. Block 0 contains the primary software bootstrap, which initiates loading of the application image.

PROM Programmer Format

The PROM programmer (ROM/RAM image) .MIM file format differs from the PASDBG and bootstrap load format files in that the PROM file's memory image contains only ROM (read-only memory) segments. No space is allocated in the file for read/write segment text. The memory allocation table has entries only for the read-only segments. The file must not contain a bootstrap. The memory allocation table provides the information needed by the utility program that will subsequently be used to control the "PROM blasting" process (VAX DECprom on a VAX/VMS host system, for example

Figure 7—2 shows a PROM programmer format memory image file.

Figure 7–2: PROM Programmer Format .MIM File



7.3 Placing Your Application in PROM

If you have access to a suitable PROM programmer device—such as one of the DATA I/O family of PROM programmers—and to programmer-control software, available from DIGITAL, that recognizes MicroPower/Pascal .MIM file format, you can place an application in programmable read-only memory (PROM) chips or erasable PROM (EPROM) chips. Both the VAX DECprom utility program, which executes under VAX/VMS, and the PROM/RT–11 utility program, which executes under the RT–11 operating system, support MicroPower/Pascal .MIM files as PROM programmer input. These programs are optional software, not part of the MicroPower/Pascal

product. Refer to the documentation describing those programs for further information on "burning" PROM/EPROM chips.

Ordinarily, you will build the first versions of an application for a RAM-only target system even though the application is intended eventually for a ROM/RAM environment. You would do so in order to take advantage of the convenient facilities for initial development and debugging that exist for a RAM-only environment. Before you attempt to place the application in PROM or EPROM, the application memory image must be completely rebuilt in ROM/RAM form, as described in Chapter 3. In particular, the MEMORY macros of the configuration file used to build the kernel must accurately describe the configuration of ROM and RAM to be used in the target system. Any debug support built into earlier versions must be excluded. The hardware must be configured to enter the initialization procedure by means of the power/fail vector at location 24(octal).

Chapter 8
Using the MicroPower/Pascal Compiler

The MicroPower/Pascal compiler, MPP, generates 16-bit object code for the PDP-11 family of microcomputers supported as targets by MicroPower/Pascal software. The generated code is ROMable, that is, suitable for ROM/RAM target memory environments. The extended Pascal language implemented by MPP is described in the *MicroPower/Pascal Language Guide.*

The language provides special extensions designed to support development of efficient real-time control applications using concurrent programming techniques. The majority of these extensions facilitate the creation of parallel, interacting processes, synchronization of processes through a variety of semaphore operations, and communication of data between processes. (The PROCESS construct is central to MicroPower/Pascal programming.) Other extensions permit a high degree of control over the allocation of storage for variables, in terms of both specific storage locations/boundaries and data packing. MicroPower/Pascal also supports separate compilation of source code units, primarily through the MODULE compilation unit and the EXTERNAL/GLOBAL declaration attributes.

Standard MicroPower/Pascal object-time support is provided by the OTS libraries LIBNHD.OLB, LIBEIS.OLB, LIBFIS.OLB, and LIBFPP.OLB. The OTS libraries SUPEIS.OLB and SUPFPP.OLB provide object-time support when you build a supervisor-mode shared library. Each of these libraries corresponds to one of the instruction set options described in Table 8–1 or Table 8–2. All Pascal object modules must be merged with one of these libraries. In addition, FILSYS.OLB must be used for file system support.

In the VMS compiler, diagnostic messages are divided into three classes:

- Informational—Messages that flag the use of a nonstandard feature of MicroPower/Pascal or provide information about the source program. The generated object module is executable.
- Warning—Messages that flag an error that may cause unexpected results, but which do not prevent the object module from being linked and executed.
- Fatal—Messages that flag an error that prevents generation of the object code.

Object code generation is suppressed only if the compiler detects a fatal error. If the compiler generates at least one diagnostic message, and no internal compiler errors exist, compilation terminates with the following message:

```
%PASCAL-I-Informational errors diagnosed - [number]
         Warning errors diagnosed - [number]
         Fatal errors diagnosed - [number]
```

If any of the numbers is zero, the compiler suppresses the whole line.

8.1 File Space Requirements

During a compilation, the compiler opens temporary files for use as intermediate working space (four files for VAX/VMS, five for RSX). These temporary files are created in the user's default file directory. Therefore, sufficient free space must exist on the user's default storage device—or in the VAX/VMS user's file space quota—to accommodate these files in addition to the output object and/or listing files. The minimum amount of free space required depends to some extent on the size of the program to be compiled. As a rule of thumb, 300 blocks of free space should be adequate for compiling most programs, with a maximum requirement of 500 blocks for a very large program. The usual approximation includes the space needed for both an object and a listing file, but an unusually large listing might impose an additional requirement.

8.2 Compiler Invocation and Command Line Format

If you have an RSX host development system, see Section 8.2.1. If you have a VMS host development system, see Section 8.2.2.

8.2.1 RSX Development System

Assuming that MPPASCAL has been installed according to installation procedure defaults, you reference it by the task name MPP, as follows:

```
>MPP
MPP>
```

or

```
$ MCR MPP
```

Precede "MPP" with "MCR" only if your CLI mode is DCL.

Command Line Format

In response to the compiler's command line prompt, MPP> , enter a command line specifying the input and output files and any needed options, or specify a command file that contains the required command line(s). (See Section 8.2.3.) The command line requested by the MPP> prompt has the following syntax:

```
[object-file] [,listing-file] = source-file [option-list]
```

object-file

A file specification for the object code output file. The object file is optional, but you must specify at least one output file (object or listing). Except as noted in Section 8.2.3, the compiler accepts a standard RSX file specification. The compiler supplies standard defaults for device, directory, and version number. The file type default is .OBJ unless the /MA option is used, in which case the default is .MAC.

listing-file

A file specification for the listing output file. The listing file is optional, but you must specify at least one output file (object or listing). Except as noted in Section 8.2.3, the compiler accepts a standard RSX file specification. The compiler supplies standard defaults for device, directory, and version number. If you specify a device or a directory for the object file, however, the compiler applies the same value as a default for the corresponding field of the listing file specification. The listing file type default is .LST.

source-file

A file specification for the input source file. Except as noted in Section 8.2.3, the compiler accepts a standard RSX or VMS file specification. The compiler supplies standard defaults for device, directory, and version number. The file type default is .PAS.

option-list

One or more compilation options, specified in the following form:

```
/option-name[:option-value[:...]] [/...]
```

You can also choose to enter the command line on a single line immediately following MPP as follows:

```
> MPP A,A=A
```

or

```
$ MCR MPP A,A=A
```

Table 8–1 summarizes the command line option names and values and lists the corresponding source code option, if any. Section 8.3 describes each option in detail. Only the first two characters of an option name are significant in a command line. Option values are always three characters. Several examples of option specifications follow:

```
/DE                 Minimum form of /DEBUG
/IN:EIS             Minimum form of /INSTRUCTIONS:EIS
/CH:IND             Minimum form of /CHECK:INDexes
/DE/IN:EIS/CH:IND   All of the above
/CH:IND:RAN:STA     /CH:IND + /CH:RANge + /CH:STAck
```

Table 8–1: Compilation Options for RSX–11 Host

Option Name	Corresponding Source Option	Purpose
/CHeck:		Generate run-time checks for:
IND	INDEXCHECK	o Array index values
MAT	MATHCHECK	o Division by 0
POI	POINTERCHECK	o Null pointer value
RAN	RANGECHECK	o Variable value range
STA	STACKCHECK	o Stack overflow
/DEbug	(none)	Include symbol information for PASDBG debugger
/EXtra-stats	(none)	List extended compilation statistics

Table 8–1 (Cont.): Compilation Options for RSX–11 Host

Option Name	Corresponding Source Option	Purpose
/FIlter-dcls	(none)	Filter out (discard) unused source declarations
/INstructions: EIS FIS FPP LS2 NHD	(none)	Generate instructions for: o EIS hardware option o FIS hardware option o FPP hardware option o LSI–11/2 with ROM and EIS o No special hardware
(none)	NOLIST LIST	Selectively disable source listing Selectively enable source listing
/MAcro	(none)	Generate MACRO–11 output code
/NOpredfl	(none)	Disable real-time definition file PREDFL.PAS
/PAge_size=page-size	(none)	Specify number of lines per page for source listing; page-size must be an integer value in the range 40 to 128; default is 66
/STandard	STANDARD	Flag nonstandard features

The significant characters of an option name are shown in uppercase. Any additional characters are optional and are ignored by the compiler.

Syntax Examples

1. `MPP> B, C = A`
Compile source file A.PAS, producing object file B.OBJ and listing file C.LST.

2. `MPP> B = A`
Compile source file A.PAS and produce object file B.OBJ only.

3. `MPP> ,C.LST = A`
Compile source file A.PAS and produce listing file C.LST only.

4. `MPP> FOOB,FOOB=FOOB/IN:FPP/CH:POI/DE`
Compile FOOB.PAS and produce FOOB.OBJ and FOOB.LST, under control of the options IN:FPP, CH:POI, and DE. (See Table 8–1.)

8.2.2 VAX Development System

The MicroPower/Pascal compiler on a VAX development system uses a DCL interface. Section 1.4 describes the MPSETUP command procedure used to define the MPPASCAL DCL command. After executing the MPSETUP.COM procedure, you invoke the compiler on a VAX host by using the logical symbol MPPASCAL, as follows:

```
$ MPPASCAL[option-list] source-file[option-list]
```

source-file

The input source file specification. Except as noted in Section 8.2.3, the compiler accepts a standard VMS file specification. The compiler supplies standard defaults for device, directory, and version number. The default file type is .PAS.

option-list

One or more compilation options, specified in the following form:

```
/option-name[=(option-value[,...])]  [/...]
```

You can specify options either after the logical symbol MPPASCAL or after the source file name. You can specify as part of the option list one or more of the options listed in Table 8–2. Section 8.3 describes each option in detail. Only the first character of the option name is significant in a VMS command line; option values, however, are always three characters long. Several examples of valid VMS option specifications are given below:

```
/D          Minimum form of /DEBUG
/I=(EIS)    Minimum form of /INSTRUCTIONS=(EIS)
/C=(IND)    Minimum form of /CHECK=(INDexes)
/D/I=(EIS)/C=(IND) All of the above
/C=(IND,RAN,STA) /CHECK=(INDexes) + /CHECK=(RANge) + /CHECK=(STAck)
```

Table 8–2: Compilation Options for VAX Host

Option Name	Corresponding Source Option	Purpose
/[NO]CHECK=		[Do not] generate run-time checks for:
(IND)	INDEXCHECK	o Array index values
(RAN)	RANGECHECK	o Variable value range
(POI)	POINTERCHECK	o Null pointer value
(STA)	STACKCHECK	o Stack overflow
(MAT)	MATHCHECK	o Division by 0 Default is NOCHECK.
/[NO]DEBUG	(none)	[Do not] include symbol information for PASDBG debugger; default is NODEBUG.
/[NO]EXTRA_STATS	(none)	[Do not] list extended compilation statistics; default is is NOEXTRASTATS.
/[NO]FILTERDEC	(none)	[Do not] filter out (discard) unused source declarations; default is NOFILTERDEC.
/INSTRUCTIONS=		Generate instructions for:
(NHD)		o No special hardware
(EIS)	(none)	o EIS instruction set
(FIS)		o FIS instruction set
(FPP)		o FPP instruction set
(LS2)		o LSI–11/2 with ROM and EIS o Default is NHD.

Table 8–2 (Cont.): Compilation Options for VAX Host

Option Name	Corresponding Source Option	Purpose
/[NO]MACRO[=file-spec]	(none)	[Do not] generate MACRO–11 output code, write output to file-spec; default is NOMACRO.
/PAge_size=page-size	(none)	Specify number of lines per page for source listing; page-size must be an integer value in the range 40 to 128; default is 66.
/[NO]PREDFL	(none)	[Do not] use real-time definition file PREDFL.PAS; default is PREDFL.
/[NO]STANDARD	STANDARD	[Do not] flag nonstandard features; default is STANDARD.
/[NO]LIST[=file-spec]	(none)	[Do not] generate listing file, write output to file-spec; default is NOLIST.
	LIST/NOLIST	Selectively enable/disable listing of sections of source program.
/[NO]OBJECT[=file-spec]		[Do not] generate object file, write output to file-spec; default is OBJECT.
[NO]WARNINGS		[Do not] generate warning and informational errors.
/[NO]VERSION	NOVERSION	[Do not] display the compiler version number; default is NOVERSION.

The source, object, list, and macro file types default to .PAS, .OBJ, .LIS, and .MAC, respectively. The object, list, and macro file names default to the source file name with the appropriate extension. The object, list, and macro file specifications default to the device and directory of the current process rather than to those of the source file specification.

The specification of both /OBJECT and /MACRO is not allowed. As a convenience, the explicit specification of only the /MACRO qualifier is acceptable—you need not explicitly specify /NOOBJ with /MACRO.

If you specify no output files at all, the compiler just reports any diagnostics at your terminal.

Syntax Examples

1. **$ MPPASCAL/OBJECT=B/LIST=C A**

Compile source file A.PAS, producing object file B.OBJ and listing file C.LIS.

2. **$ MPPASCAL/OBJECT=B A**

Compile source file A.PAS and produce object file B.OBJ only.

3. **$ MPPASCAL/OBJECT=FOOB/LIST=FOOB/INSTRUCTIONS=-**
(FPP)/CHECK=(POI)/DEBUG FOOB

Compile FOOB.PAS and produce FOOB.OBJ and FOOB.LST, under control of the options INSTRUCTIONS=(FPP), CHECK=(POI), and DEBUG. (See Table 8–2.)

8.2.3 Command Line Usage Rules

The following general rules and restrictions apply to the compilation command line:

1. Spaces around delimiters are optional.
2. A node name (node::) is not valid in a file specification. All other fields of a file specification are recognized, including subdirectory names.
3. No wildcard characters are recognized in a file specification.
4. Any explicit device or directory information specified for the object file is "sticky" for the listing file if both files are specified. (RSX only)
5. One source file is accepted for each compilation; the file must contain one complete compilation unit, that is, one PROGRAM or MODULE.
6. If the /MA option is specified, the "object" output file contains MACRO–11 assembly code instead of binary object code, and the file type defaults to .MAC. (RSX only)
7. If the /FI option is specified, either an object or a listing file is valid as output but not both.
8. Continuation lines are accepted; command continuation is indicated by a hyphen (-) as the last character of an input line. That is, if a hyphen immediately precedes the carriage return terminating an input line, the compiler prompts for an additional line of input. The trailing hyphen can appear anywhere in the command string.
9. An indirect command file can be specified (@file-spec) in response to the prompt in place of a direct command line.

   The default file type for an indirect command file is .COM for VMS and .CMD for RSX. (Alternatively, the compilation command line can be contained, together with the compiler invocation command, in a command file specified at system level.)
10. The compiler returns control to system level after processing one complete compilation command; that is, the compiler performs only one compilation for each invocation.
11. A comment line is accepted in response to an MPP> prompt. A comment line is indicated by a leading exclamation point (!) for VMS and a leading semicolon (;) for RSX. The compiler ignores the entire line and prompts for more input. Comment lines are useful for documenting a command file.

8.3 Compilation Options

Section 8.3.1 describes in detail the source code compilation options. These options are applicable to either an RSX host or a VAX host system.

Section 8.3.2 describes in detail the command line compilation options. Although you must specify some command line options slightly differently according to whether you are using an RSX host or a VMS host, the effect of an option is the same for both host systems. Table 8–1 summarizes the compilation options for an RSX host system, and Table 8–2 summarizes the compilation options for a VAX host system.

8.3.1 Compilation Options in Source Program

Source program options have both a positive and a negative form. Thus, you can selectively enable and disable an optional compilation feature at various points in the compilation unit. For example, you might want the compiler to generate a given type of run-time checking code only for selected procedures. The source program option names are as follows:

Positive Form	Negative Form
INDEXCHECK	NOINDEXCHECK
MATHCHECK	NOMATHCHECK
POINTERCHECK	NOPOINTERCHECK
RANGECHECK	NORANGECHECK
STACKCHECK	NOSTACKCHECK
STANDARD	NOSTANDARD
LIST	NOLIST

Source program options are specified within a special form of the Pascal comment. The syntax of an option specification is as follows:

```
(*$option-name[,...] *)
```

The alternative form, using { } instead of (* *) comment delimiters, is as follows:

```
{$option-name[,...] }
```

No spaces are allowed in the option specification except after the last or only option name; any text following a space or other nonprinting character is treated as normal commentary. Valid examples are the following:

```
(*$POINTERCHECK*)
{$NOPOINTERCHECK}
(*$LIST *)
(*$RANGECHECK,MATHCHECK*)
(*$rangecheck,mathcheck*)
{$NOLIST the rest is just commentary}
```

Invalid examples are the following:

```
{ $INDEXCHECK}          $INDEXCHECK is ignored
(*$ NOSTANDARD*)        NOSTANDARD is ignored
(*$LIST, STANDARD*)     STANDARD is ignored
(*$LIST,$STANDARD*)     STANDARD is ignored
```

Note

The source program options do not act as "on/off switches"; instead, they increment (positive form) or decrement (negative form) a counter associated with an option. That is, the compiler keeps track of how many times each option is enabled and disabled in the source program. For example, if MATHCHECK appears twice without an intervening NOMATHCHECK, NOMATHCHECK must appear twice in order to disable the option.

8.3.2 Command Line Options

Options specified in the command line affect the entire compilation unit. Options specified in the source program affect only the portion of the compilation unit in which the option is enabled.

Note

For clarity, the following sections illustrate only RSX option syntax, although the descriptions apply equally to VAX options. For example, Section 8.3.2.1 talks about the /CHeck:xxx option; if you have a VAX host, mentally substitute the VMS option form—/Check=(xxx)—when reading the option descriptions.

8.3.2.1 Run-Time Checking Code (/CHeck:xxx)

The /CH:xxx options (source code xxxxCHECK) include code in the output object module to check for invalid values and other error conditions that can occur during program execution. The checks report an appropriate exception condition if the specified error is detected. You can request the following specific checks:

/CH:IND	Check all array indexes for out-of-range values. Verifies that computed array subscripts remain within the bounds specified in their type declarations. (Source code option INDEXCHECK.)
/CH:MAT	Check for integer or unsigned division by 0. Tests all divisors for a zero value before use. (Source code option MATHCHECK.)
/CH:RAN	Check all assignment expressions for out-of-range values. Verifies that computed values are within the range declared for the target variable. Use of this option does not cause a check for an INTEGER variable greater than 32767 or less than -32768. (Source code option RANGECHECK.)
/CH:POI	Check for the NIL, or undefined, pointer value. Detects any attempted use of a pointer with a NIL value in an address reference. This check does not preclude the use of NIL as a list-terminating pointer value. (Source code option POINTERCHECK.)
/CH:STA	Check for stack overflow on entrance to a procedure or a function. (Source code option STACKCHECK.)

The run-time checks are useful for program debugging. Use of any checking option makes the generated code larger than it would otherwise be. The checking options are disabled by default.

8.3.2.2 Debug Symbol Information (/DEbug)

The /DE option includes symbol-definition information in the object file for debugging purposes. The debug option allows symbolic debugging of the compiled program by means of the PASDBG symbolic debugger. (The build utilities place the symbol information in the debugger's symbol table file.) The debug option limits the degree of optimization performed by the compiler to the statement level and also provides statement numbers—in addition to line numbers—in the source listing for debugging purposes. Use of the debug option usually results in an increase in the size of the generated code because of the limited optimization performed. The debug option is disabled by default.

8.3.2.3 Extended Statistics (/EXtra)

The /EX option provides information about the generated object code in the source listing. The extended information consists of the amount of stack space used for each procedure, function, and process and the name and size of each program section (p-sect) in the generated object module. Both an object file and a listing file must be generated to allow the stack-space information and valid p-sect sizes to be reported; the compilation must be free of errors. The extended-statistics option is disabled by default.

8.3.2.4 Filter Unused Declaration (/FIlter-decls)

The /FI option filters out, or discards, all unused type, variable, constant, and subprogram declarations found in the source code during the input-scan phase. (Multipurpose INCLUDE files and the implicitly included PREDFL.PAS file may well contain many such unused declarations, which place an unnecessary burden on compiler resources, especially dynamically allocated memory.) Appendix H of the *MicroPower/Pascal Language Guide* describes the compiler's limits on the number of subprograms, unique identifiers, and types that can be processed in a compilation unit. If a program fails to compile because it exceeds any of those limits, use of the /FI option may permit the program to compile successfully.

The compiler uses the following rules in determining whether a given declaration is "used" or "unused":

- An identifier is considered used if it appears in a statement at the main program level or in a statement of a subprogram that is used.
- An identifier is considered used if it appears in the formal parameter list of a subprogram that is used.
- A subprogram is considered used if there is a possible program path from the main level or from a used subprogram to that subprogram.
- A type identifier is considered used if it is subordinate to a used constant, variable, or subprogram identifier.
- A type identifier is considered used if it is a scalar type and if one or more of its enumerated elements is used.
- A variable or a subprogram declared with the GLOBAL, INITIALIZE, or TERMINATE attribute is considered used.
- All outer-level variable declarations of the compilation unit are considered used if the compilation unit has the OVERLAID attribute.
- All constant, type, variable, and subprogram declarations that are not determined to be used according to the foregoing rules are considered to be unused.

The filtering mechanism is conservative in applying these rules and may sometimes treat an identifier as used when it is not, as in the case of duplicate identifiers appearing in nested scopes.

If you use the /FI option, you can specify either a code or a listing file as output but not both. That is, you cannot get both generated code and a listing from the same compilation under the filter option. If you specify an output code file (object-file) and if the program compiles correctly, the compiler generates object code—or macro code if /MA was also specified. The

compiler will have ignored all unused declarations in the source code as if you had physically removed them from the input(s).

If you specify /FI and a listing file, the compiler produces either of two kinds of listing, depending on the results of the compilation. If any lexical errors are encountered, the compiler produces an error listing identifying the error(s) detected. If no lexical errors occur, the compiler produces a listing in which all unused identifiers are indicated by the annotation "*** IDENTIFIER(S) NOT REFERENCED" in combination with circumflex characters (^). Be aware that other errors that are syntactic or semantic are not detected when you specify the combination of /FI and listing file.

Any unused declarations supplied by PREDFL.PAS (Section 8.3.2.8) are filtered but are not reported in the summary of unreferenced declarations, since declarations from PREDFL.PAS never appear in a listing.

8.3.2.5 Instruction Set (/INstr:xxx)

The /IN:xxx option indicates the class of optional instructions, if any, that the compiler can use in the generated code. Indirectly, the option identifies the kind of target processor(s) on which the code will be executed, in terms of minimum hardware capabilities. You can specify the following instruction set options:

/IN:EIS	Extended Instruction Set; optional on LSI-11 and LSI-11/2 processors, standard on LSI-11/23 processors.
/IN:FIS	Floating Instruction Set, applicable to LSI-11 and LSI-11/2 processors only. This option implies the EIS option; EIS capability is a subset of the FIS capability.
/IN:FPP	Floating Point Processor (FP-11) instruction set, applicable to LSI-11/23 processors only. (This option corresponds to either the KEF-11 or the FPF11 hardware option.) This option implies the EIS option; EIS capability is standard on the LSI-11/23 processor.
/IN:LS2	Used in combination with the EIS or the FIS option for LSI-11 or LSI-11/2 processors only. This option indicates that the target system has ROM storage. The effect of this option is to inhibit generation of the immediate-operand form of EIS instruction, which is not ROMmable in an LSI-11 or LSI-11/2 environment. (The combination of the EIS or FIS option and the LS2 option can be specified compactly as /IN:EIS:LS2 or /IN:FIS:LS2.)
/IN:NHD	Basic PDP-11 instruction set only; no special instructions. This option must be used for an SBC-11/21 (FALCON) or a KXT11-CA or KXJ11-CA processor or for an LSI-11 or LSI-11/2 without the EIS or FIS hardware option. This option can be used to generate "common" code that will execute on any supported target configuration.

The NHD option is enabled by default.

8.3.2.6 Compilation Listing (/[NO]List=file-spec)

The /L option is available only on VAX host systems. The /[NO]List=file-spec option lets you specify an output listing file name.

The /[NO]List=file-spec option is disabled by default.

8.3.2.7 MACRO-11 Output Code (/MAcro)

The /MA option instructs the compiler to produce MACRO-11 assembly code as output instead of binary object code. This form of compilation output is suitable for input to the MACRO-11 assembler.

If you have a VMS host, the option format is /[NO]MACRO=file-spec, which allows you to specify a file name for the .MAC output file.

The /MA option is disabled by default.

8.3.2.8 No Real-Time Predefinitions (/NOpred)

The /NO option suppresses the otherwise implicit and automatic inclusion of the PREDFL.PAS file into the compilation unit. The PREDFL.PAS file supplies the predefined procedures, functions, and data types needed for use of the real-time programming requests described in Part Two of the *MicroPower/Pascal Language Guide*. You can use this option to save a considerable amount of compilation time and space when compiling a program or module that does not use any real-time features.

If you have a VMS host, the option format is /[NO]PRedfl. The /NO option is disabled by default.

8.3.2.9 Output Object File (/[NO]Object=file-spec)

The /O option is available only on VAX host systems. The /[NO]Object option lets you specify the name of an output object file name. If the file name is omitted, the compiler uses the name of the input source file and the .OBJ file type.

The /[NO]Object option is enabled by default.

8.3.2.10 Listing Page Size (/PAge_size=page-size)

The /PA option allows you to specify the number of source code lines on a listing page. The page-size value comprises the page heading, code lines, error messages, and top and bottom margins—usually corresponding to the size of the paper on which the listing is to be printed. You must replace the symbol "page-size" with an integer in the range 40 to 128.

8.3.2.11 Generate Warning and Informational Errors (/[NO]WArnings)

The /WA option is available only on VAX host systems. The /[NO]WArnings option lets you specify whether or not the compiler should report any warning or informational class errors.

The /[NO]WArnings option is enabled by default.

8.3.2.12 Standard Pascal Only (/STandard)

The /ST option (source code STANDARD) issues an error message for any nonstandard Pascal language feature encountered in the compilation unit. (As used here, "standard Pascal" refers to only those language features described by the International Standards Organization specification for the Pascal language.) When this option is used, no object code is generated if any nonstandard feature is detected.

The /ST option is disabled by default.

8.4 Compilation Listing

The compiler produces an annotated source program listing if you include a listing file specification in the command line. If you use the /EX option in the command line and specify both an object and a listing file, the listing includes additional information about the compiled code—maximum stack depths for the main program and each subprogram and the names and sizes of the generated program sections. (If you include the /EX option but omit the listing file specification, the compiler displays the lines-per-minute compilation statistic at your terminal.)

Also, use of the /FI (filter declarations) option with a listing file specification affects the form of listing file produced, as described in Section 8.3.2.4.

Figures 8–1 and 8–2 show two standard listings illustrating the common characteristics of an MPPASCAL listing file. Figure 8–1 shows the listing of a program containing syntax errors that was compiled without the /DE (debug) option. Figure 8–2 shows a listing of the same program but without errors and compiled with the /DE option. The effect of /DE on the listing is to produce statement numbers as well as source line numbers; the statement numbers can be utilized in various PASDBG debugging commands. Circled numbers in the figures point out the following information:

1. Title of object module, from PROGRAM or MODULE name
2. Time and date of compilation
3. Name and version number of compiler
4. Name of source file
5. Line numbers
6. Include-file listing levels
7. Comment flag
8. Procedure level
9. Statement level
10. Statement numbers, if any (DEBUG compilations only)
11. Error diagnostics, if any, and summaries
12. Elapsed time of compilation and average speed
13. List of options selected

Figure 8–1: Compilation Listing: Program with Errors, No /DE Option

```
                    ERROR SUMMARY

⑪ Line     27 - Fatal - Undefined identifier
  Line     32 - Fatal - Undefined identifier

① SUGEXAMP ②15:52:27  14-Aug-87 Friday  ③ PASCAL V02.04                Page 1-1
④ File: DISK01:[EXAMPLES]SUGEX1.PAS

  LINE-IC-PL-SL-SOURCE
    ⑤       ⑧  ⑨
  0001      0  0 [ SYSTEM(MICROPOWER), DATA_SPACE(2000),
  0002      0  0   STACK_SIZE(200), PRIORITY (25) ] PROGRAM SUGEXAMP;
  0003      0  0
  0004      0  0 VAR
  0005      0  0   Play, Esc : CHAR;
  0006      0  0   I, J : INTEGER;
  0007      0  0
  0008 ⑥⑦   0  0 %INCLUDE 'change.pas'
  0001 2    0  0
  0002 2C   0  0 (* Include file with definition of Procedure Changecharacteristics
   *)
  0003 2    0  0
  0004 2    1  0 PROCEDURE Changecharacteristics;
  0005 2    1  1 BEGIN
  0006 2    1  1   WRITE (Esc, '<',                (* Enter ANSI mode *)
  0007 2    1  1          Esc, CHR (91), '?8l',    (* Turn off auto-repeat *)
  0008 2    1  1          Esc, CHR (91), '?2l');   (* Enter VT52 mode *)
  0009 2    0  0 END;
  0009      0  0
  0010      0  1 BEGIN
  0011      0  1   Esc := CHR (155);
  0012      0  1   Changecharacteristics;
  0013      0  1   Play := 'y';
  0014      0  1   WHILE (Play = 'Y') OR (Play = 'y') DO
  0015      0  2     BEGIN
  0016      0  2       WRITE (Esc, 'H', Esc, 'J');
  0017      0  2       WRITE ('Enter a character, please: ');
  0018      0  2       READLN (Let);
                              ^85
  ***  85: Fatal - Undefined identifier

  0019      0  2       WRITE (Esc, 'H');
  0020      0  2       FOR J := 1 TO 19 DO
  0021      0  3         BEGIN
  0022      0  3           FOR I := 1 TO 79 DO
  0023      0  3             WRITE (Let);
                                ^85
  ***  85: Fatal - Undefined identifier

  0024      0  3           WRITELN;
  0025      0  2         END;
  0026      0  2       WRITE ('Go again ? Y/N ');
  0027      0  2       READLN (Play);
  0028      0  1     END;
  0029      0  0 END.

⑪ There were 2 lines with errors diagnosed.

⑫ Compilation required 6 seconds.
  Average compilation speed was 380 lines/min.

⑬ Options Selected

  No Special Instructions                                    MLO-1018-87
  PREDFL.PAS Not Used
```

Figure 8–2: Compilation Listing: Errors Removed, /DE Option Used to Show Statement Numbers

```
①SUGEXAMP  ②6:16:58  14-Aug-87 Friday  ③PASCAL V02.04
④File: DISK01:[EXAMPLES]SUGEX2.PAS

LINE-STMT-IC-PL-SL-SOURCE
 ⑤          ⑧  ⑨
0001          0  0 [ SYSTEM(MICROPOWER), DATA_SPACE(2000),
0002          0  0   STACK_SIZE(200), PRIORITY (25) ] PROGRAM SUGEXAMP;
0003          0  0
0004          0  0 VAR
0005          0  0   Play, Esc, Let : CHAR;
0006          0  0   I, J : INTEGER;
0007          0  0
0008      ⑥⑦  0  0 %INCLUDE 'change.pas'
0001       2  0  0
0002       2C 0  0 (* Include file with definition of Procedure Changecharacteri
stics *)
0003       2  0  0
0004       2  1  0 PROCEDURE Changecharacteristics;
0005     ⑩ 2  1  1 BEGIN
0006     1 2  1  1   WRITE (Esc, '<',                (* Enter ANSI mode *)
0007       2  1  1           Esc, CHR (91), '?8l',   (* Turn off auto-repeat *)
0008       2  1  1           Esc, CHR (91), '?2l');  (* Enter VT52 mode *)
0009     2 2  0  0 END;
0009          0  0
0010          0  1 BEGIN
0011     1    0  1   Esc := CHR (155);
0012     2    0  1   Changecharacteristics;
0013     3    0  1   Play := 'y';
0014     4    0  1   WHILE (Play = 'Y') OR (Play = 'y') DO
0015          0  2     BEGIN
0016     5    0  2       WRITE (Esc, 'H', Esc, 'J');
0017     6    0  2       WRITE ('Enter a character, please: ');
0018     7    0  2       READLN (Let);
0019     8    0  2       WRITE (Esc, 'H');
0020     9    0  2       FOR J := 1 TO 19 DO
0021          0  3         BEGIN
0022    10    0  3           FOR I := 1 TO 79 DO
0023    11    0  3             WRITE (Let);
0024    12    0  3           WRITELN;
0025          0  2         END;
0026    13    0  2       WRITE ('Go again ? Y/N ');
0027    14    0  2       READLN (Play);
0028          0  1     END;
0029    15    0  0 END.

⑪There were no lines with errors diagnosed.

⑫Compilation required 12 seconds.
  Average compilation speed was 190 lines/min.

⑬Options Selected

   Debugging
   No Special Instructions
   PREDFL.PAS Not Used                                          MLO-1019-87
```

8.5 Compiling Large Programs

The MicroPower/Pascal compiler cannot compile some large programs, because of compiler capacity overflow. Usually, code size alone is not the problem. Appendix H of the *MicroPower/Pascal Language Guide*, MicroPower/Pascal Compiler Limitations, discusses compiler capacity. The information given here is intended as a supplement; you should already be familiar with the appendix material.

In most cases when programs exceed compiler capacity, the compiler's "heap" (free-memory pool) has no more room to place new "type" entries. The prevalence of type definitions in the program is the single characteristic of source code that is most likely to cause a compilation to fail because of compiler heap capacity limitations. Type definitions appearing in the Pascal TYPE declaration section and the implicit type definitions used in the VAR and CONST sections both contribute to the problem.

Programs that use the modular features of the language and that limit declarations to the modules and procedure scopes where they are needed are less likely to run into heap space problems. Large applications, however, may have many declarations at the program level. This condition may occur if many modules share data declarations by means of one or more generalized "include" files. Such applications are more subject to capacity overflow.

If a program exceeds compiler capacity, first try compiling the program with the /FI option to filter out unused declarations. If the program uses the standard PREDFL declarations or if the program uses any large "include" files, /FI may permit compilation by filtering out unused declarations.

If /FI does not solve the problem, consider restructuring the program. The section of Appendix H on the "TYPE TABLE" refers to anything that goes on the heap. Following the suggestions in Appendix H, try putting declarations in disjoint scopes.

If you cannot put declarations in disjoint scopes or if that does not solve the capacity problem, try rewriting the declarations. Different sorts of declarations impact the compiler heap space differently. You may be able to replace some declarations with equivalent code that uses the compiler heap more efficiently. The examples below illustrate how different constructs use the heap.

Figure 8–3 shows Pascal code and heap usage. Circumflexes (^) under a section of Pascal code indicate that a heap entry is made during compilation for that construct. The comments explain why the heap entry is made and what possible alternative coding styles might reduce heap use. Heap entries may result from VAR, TYPE, and CONST declarations. Declarations of structures may result in more than one heap entry. Character strings result in heap use no matter where they appear in the source code (declarations or statements). Recall from Appendix H that as a program or module is processed, type entries and implicit type entries occupy the heap for the duration of the Pascal scope in which they appear.

Figure 8–3: Pascal Code and Heap Usage

```
SOURCE CODE                          COMMENTS

VAR

   i : integer;                      "integer" is a predefined type, so no
                                     heap entry is made.

   j : ARRAY [1..2] OF integer       Every "ARRAY" causes one heap entry, as
       ^     ^                       does every ".." (subrange) construct.

TYPE
  colors = (red, white, blue)        Generally, every user defined type,
           ^                         in this case the simple enumerated
                                     scalar "colors", results in at least one
                                     heap entry.

  rect = RECORD
         ^                           Every "RECORD" symbol results in a heap
           wall, floor,              entry. If you define a type, then use
           ceiling : colors;         it again, no more heap entries
           CASE paint : colors OF    result. Each case variant label
        red: (glossy:boolean);       gets a heap entry.
        ^
        white,blue: (spray:boolean)
        ^     ^
         END;

  ptr = ^integer;                    Every "^" (pointer), "SET" and "FILE"
        ^                            results in a heap entry. Notice that
  alpha = SET OF char;               there is a heap entry for "0..255". If
          ^                          you repeat a subrange value in other
  file1t = FILE OF 0..255;           declarations you should probably give it
           ^       ^                 a type id.
  byterng = 0..255;
            ^

  file2t = FILE OF byterng;          Use of a type id for a subrange reduces
           ^                         heap use in subsequent occurrences of
  byteptr = ^byterng;                that subrange.
            ^

  arrayrange = 1..10;                That rule applies to any frequently-used
               ^                     variable or type component.
  arrayt1 = ARRAY[arrayrange,arrayrange,arrayrange] OF byterng;
            ^

VAR
  a1,a2,a3 : arrayt1;                To reduce heap use, collapse commonly
                                     used structure components into single
                                     type definitions that are reused.
                                     Compare this simple declaration with
                                     the equivalent:
  a1 : ARRAY [1..10,1..10,1..10] OF 0..255;
       ^     ^     ^     ^          ^
  a2 : ARRAY [1..10,1..10,1..10] OF 0..255;
       ^     ^     ^     ^          ^
  a3 : ARRAY [1..10,1..10,1..10] OF 0..255;
       ^     ^     ^     ^          ^
```

```
CONST
  one = 1;                          As with VAR or TYPE declarations, if the
                                    type of a CONST value is a standard type
                                    (in this case "integer"), the compiler
                                    makes no heap entry.

  ch = 'a';                         Standard type "char".

  s1 = 'abc';                       If a character string appears anyplace
       ^^                            in a program, either in a declaration or
                                    statement, it will consume two heap
                                    entries. In this case, declare 'abc' as
                                    a constant id if it is used more than
                                    once in statement code.

TYPE
   tablet = ARRAY[arrayrange] OF PACKED ARRAY[1..5] OF char;
            ^                       ^               ^

CONST
   commands = tablet('start','stop ','left ','right','help ',
                     ^^      ^^      ^^      ^^      ^^
                     'revrs','fast ','slow ','signl','beep ');
                     ^^      ^^      ^^      ^^      ^^
                                    Note that generous use of strings causes
                                    extensive use of the heap.

VAR
  command : tablet;                 The same problem occurs when strings
                                    appear in statement code.
  { statement code }
BEGIN
   command := tablet('start','stop ','left ','right','beep ',
                     ^^      ^^      ^^      ^^      ^^
                     'revrs','fast ','slow ','signl','help ');
                     ^^      ^^      ^^      ^^      ^^
   IF command[1] = 'beep ' THEN { ... } ;
                   ^^

END;
```

If a single procedure contains many strings, it is best to divide the procedure into two procedures. The disjoint scopes of the procedures will reduce the number of string heap entries active at any one time. If code containing strings is localized and appears at the top level of the program, such as when tables of strings are initialized, this code should be moved into a separate procedure. If the capacity problem persists, the procedure may optionally be placed in a separate module with an abbreviated declaration section and called externally.

The capacity problem of the compiler, which limits the number of type definitions in a compilation unit, should never force you to avoid the beneficial features of the language or to otherwise sacrifice good software design for the sake of implementation. If logically distinct segments of applications are partitioned into modules and if programs are prepared in accordance with the suggestions in Appendix H and those given above, you should avoid encountering the capacity problem.

8.6 P-sect Generation

The compiler generates the following p-sects:

```
.PSECT .alst.  i,ro,con,gbl,rel   Contains the static process list element.

.PSECT .cdat.  d,rw,con,gbl,rel   Contains program level data when OVERLAID is
                                   not used.

.PSECT .code.  i,ro,con,lcl,rel   Contains the generated code.

.PSECT .idat.  d,ro,con,gbl,rel   Contains initialization data for the OTS.

.PSECT .ini1.  d,ro,con,gbl,rel   Contains the addresses of INITIALIZE
                                   procedures.

.PSECT .ini0.  d,ro,con,gbl,rel   Contains the label $binit:

.PSECT .ini2.  d,ro,con,gbl,rel   Contains the label $einit:
                                   Used with .ini0. to determine the size of
                                   the .ini1. psect.

.PSECT .odat.  d,rw,ovr,gbl,rel   Contains the file variables INPUT and OUTPUT
                                   If OVERLAID is used, contains all program
                                   level data.

.PSECT .pbit.  d,ro,ovr,gbl,rel   Contains bit masks.

.PSECT .pcon.  d,ro,con,lcl,rel   Contains constants.

.PSECT .peis.  i,ro,ovr,gbl,rel   Contains a table of ASL and ASR instructions
                                   used to shift values in NHD mode.

.PSECT .sdat.  d,rw,con,gbl,rel   Contains the heap and stack space.  The size
                                   is determined by DATA_SPACE.
```

Chapter 9
The Merge Utility Program

The MERGE utility program combines input object modules and modules from object libraries into a single merged object module. In so doing, MERGE resolves address references between input modules and satisfies references to library routines. The input modules are segments of code and data that you have compiled or assembled into binary object code. These modules generally need to be merged with one or more DIGITAL-supplied object libraries. You can either run MERGE yourself or, for most applications, use the automated build procedure, MPBUILD.CMD (for RSX) or MPBUILD.COM (for VMS), described in Chapter 2. The indirect command file generated by MPBUILD will contain all the MERGE commands necessary to build your application.

The contents of an object module are organized into formatted binary data blocks. MERGE performs operations on four types of data block records: global symbol directory (GSD), internal symbol directory (ISD), relocation symbol directory (RLD), and text (TXT).

Global symbols are labels and identifiers that are declared in one object module and can be referenced from another object module. GSD records hold information needed to resolve those intermodule references.

ISD records contain information on all symbols, including global and local (Pascal only), used in the object module. PASDBG uses this information when debugging to determine the structure of a program and to find kernel data structures and user-defined variables. ISDs are present only in those input modules compiled with the debug compiler option (/DE).

MERGE can generate ISD records (for global symbols only) for input modules that do not already contain ISDs. These modules are normally created by the MACRO–11 assembler. MERGE can also generate ISD records for Pascal programs compiled without the /DE option, but these ISD records will not be as complete as the ISD records produced with the /DE compiler option.

Note

The MACRO–11 assembler supports an option (/EN:DBG) to create ISD records. Their format is not the same as MicroPower/Pascal's ISD records, however, and they are incompatible with PASDBG. Do not use /EN:DBG when assembling modules; allow MERGE to create ISD records for modules written in MACRO. If you happen to use a module containing ISD records created by MACRO–11's

/EN:DBG option, MERGE will process the module correctly, but MERGE will issue a warning message and then create ISD records of the proper format.

RLDs store information on how the addresses in each TXT record must be modified to place them correctly in the application. RLD records are also used, together with GSD records, for resolving symbol references and linking the code so that it executes correctly after relocation.

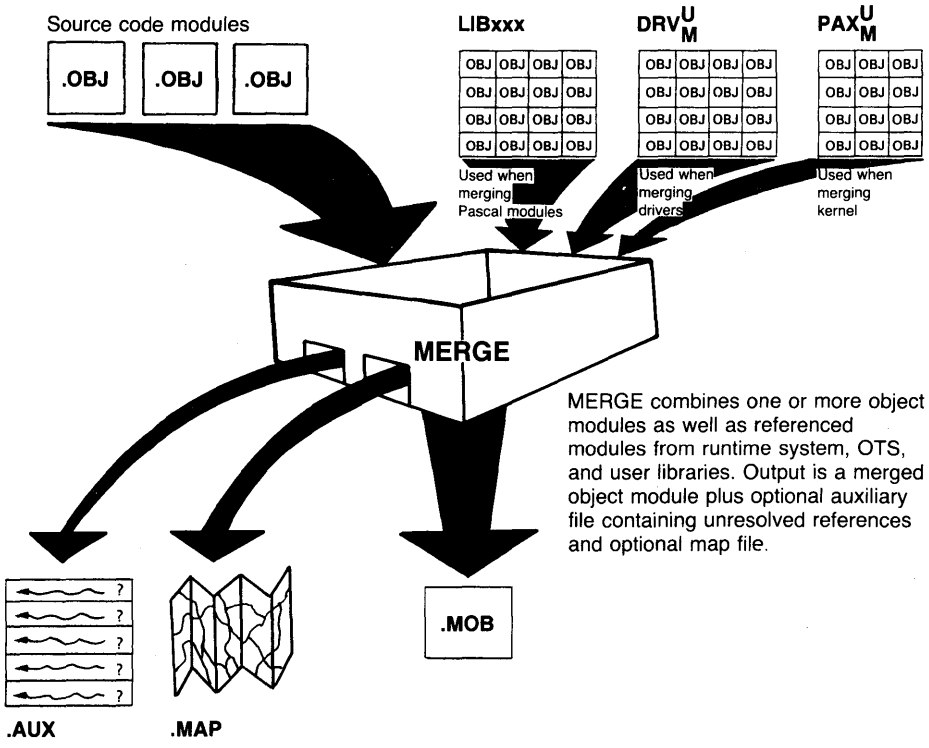
TXT records contain the binary code and data of the module.

This chapter discusses the following topics:

- MERGE's functions
- MERGE's role in the build cycle
- The invocation and use of MERGE
- The section map that MERGE optionally produces
- The options that you use with MERGE

Figure 9–1 shows the MERGE utility's input and output files.

Figure 9–1: MERGE Utility Input and Output Files



9.1 Functions of MERGE

The MERGE utility resolves intermodule references, using GSD records to match each global reference in the input modules with a global definition. MERGE also combines program section (p-sect) contributions having the same name and updates GSD, ISD, RLD, and TXT records to reflect the new relative positions of p-sect contributions in the merged object module.

A special form of intermodule reference resolution involves object libraries. MERGE satisfies any reference to a global symbol defined in an object library module by including that module in the merged object module.

Use MERGE also with a kernel .STB file to resolve references to kernel primitive service entry points and with a shared-library .STB file to resolve references to shared-library routine entry points.

The output of a merge operation can be input to a further merge operation (iterative merging) or can be input to the RELOC utility. All modules making up a static process must be merged into one module before they are input to RELOC.

MERGE can optionally provide a section map. The section map supplies information about program section names, lengths, and attributes, as well as any unresolved global references and the names of the files included in the merge.

MERGE can also optimize the kernel primitive service routines. You can use MERGE's optional auxiliary output file capability to build a kernel that includes only the primitives that the application uses or to build an optimized shared library with the OTS or other common code. (See Section 3.6, Optimizing the Kernel.)

9.1.1 Resolving Intermodule Global References

MERGE scans the input modules you specify to resolve intermodule global references. Global symbols are communication links between object modules. You create global symbols when you use the GLOBAL attribute in Pascal or the :: label terminator, == operator, or .GLOBL directive in MACRO-11. MERGE looks through the input object modules to find and flag, in its symbol table, all global-symbol definitions and external references. MERGE tries to find a global definition to satisfy each external reference. When MERGE matches a global reference with a global definition, MERGE deletes the reference but retains the definition.

The MicroPower/Pascal compiler divides a module into program sections, or collections of instructions, data, or both, that can be relocated as a unit. MERGE consolidates global-symbol definitions belonging to a given program section. When MERGE finds several contributions to a program section in different object modules, it creates a single program section name entry in the merged object module. MERGE places all global-symbol definitions for that program section after the section name entry. If MERGE finds conflicting program section attributes, it issues a warning message and uses the first section's attributes. The *MicroPower/Pascal-RSX/VMS Messages Manual* lists and explains all MERGE error messages.

9.1.2 Updating Relocation Records

Each program section in an input module contains instructions and data that assume that the section starts at location 0. Relocation directory records describe how to modify those instructions and data when the starting location of the program section contribution changes. When MERGE combines program section portions, thereby changing the starting locations of the various pieces, it updates the RLD records to reflect the amount of text already included in the section from other modules.

9.1.3 Resolving Object Library References

One of MERGE's most important functions is to resolve references to object library modules. Object libraries are specially formatted files that contain more than one object module—usually many. If any references remain unresolved after MERGE processes the input object module(s), MERGE searches any object library files specified in the MERGE command line. During the search phase, when MERGE finds a global definition that matches an unresolved reference, it extracts the library object module that contains the matching definition and merges that module with the input object module(s). MERGE thereby resolves references to library routines and data structures and includes any referenced routines from libraries in the output .MOB file.

For building Pascal processes that do not reference any shared libraries and for building user-mode shared libraries containing the OTS, the MicroPower/Pascal software package supplies four versions of the Pascal object-time system (OTS) library, each supporting a different type of math hardware:

- LIBNHD.OLB, No special hardware
- LIBEIS.OLB, EIS instruction set
- LIBFIS.OLB, FIS instruction set
- LIBFPP.OLB, FPP instruction set

For supervisor-mode shared libraries containing the OTS, the MicroPower/Pascal software package supplies two versions of the Pascal OTS library, each supporting a different type of math hardware:

- SUPEIS.OLB
- SUPFPP.OLB

When merging object modules created by the MicroPower/Pascal compiler, you include the appropriate OTS library for your target system—conventionally referred to in the MERGE command line as LIBxxx.OLB, or SUPxxx.OLB for a supervisor-mode library (EIS and FPP only). In addition, you must include the file system library FILSYS.OLB.

If MERGE cannot resolve all symbol references within the input object modules it is merging, it searches FILSYS.OLB, LIBxxx.OLB, and any other object libraries you might specify to find matching global definitions. If any unresolved references remain after MERGE scans the libraries, MERGE issues the warning message "Undefined globals:" followed by a list of the unresolved symbols.

MERGE searches object libraries in the order in which they appear in the MERGE command and satisfies references on a "first found" basis. In addition, some object library modules contain references to other libraries, as discussed below. The order in which multiple libraries are specified is therefore often important. In general, always specify the common Pascal OTS library (LIBxxx) last when performing a MERGE operation.

The MicroPower/Pascal software package also includes a number of more specialized object libraries:

- PAXM.OLB and PAXU.OLB—mapped and unmapped versions of the kernel module library, used for merging a system configuration file to build a kernel
- DRVM.OLB and DRVU.OLB—mapped and unmapped versions of the device driver library, used for merging a driver prefix module to build a device driver process
- FILSYS.OLB—the Pascal file system support library, used for merging a Pascal-implemented user process

Chapters 3 and 4 discuss and give examples of using the PAXx and DRVx libraries.

9.1.3.1 Ordering of Multiple Object Libraries

MERGE attempts to resolve any unsatisfied references found within a module that it extracts from an object library; one module may refer to another in the same library or in a different library. MERGE attempts such resolution by further iterative searching of the current library and, if necessary, by searching any additional libraries that you specify. MERGE does not return, however, to a previously searched library in an attempt to resolve a reference. Thus, the symbol-resolution process can be affected by the order in which you specify input files—especially libraries.

9.1.3.2 Ordering of All MERGE Input Files

In general, the recommended safe ordering of input files in the MERGE command line is as follows:

1. Specify all object modules (.OBJs).
2. Specify the kernel .STB file, if any is used.
3. Specify any shared library .STB files.
4. Specify any mutually unrelated object libraries.
5. Specify FILSYS.OLB, if used, just before LIBxxx.
6. Specify LIBxxx.OLB, if used, last.

In addition, if there are multiple input object modules, the module containing the static process definition—the Pascal PROGRAM heading or the MACRO-11 DFSPC$ macro call—should be the first input specified in the MERGE command. This positioning is mandatory when you are building for debugging.

Consider, for example, a modularly implemented Pascal user process that consists, at source level, of three compilation units: the PROGRAM unit, PROCSX, and two modules, MODULX1 and MODULX2. PROCSX.OBJ should be the first input object file specified in the MERGE command for that process.

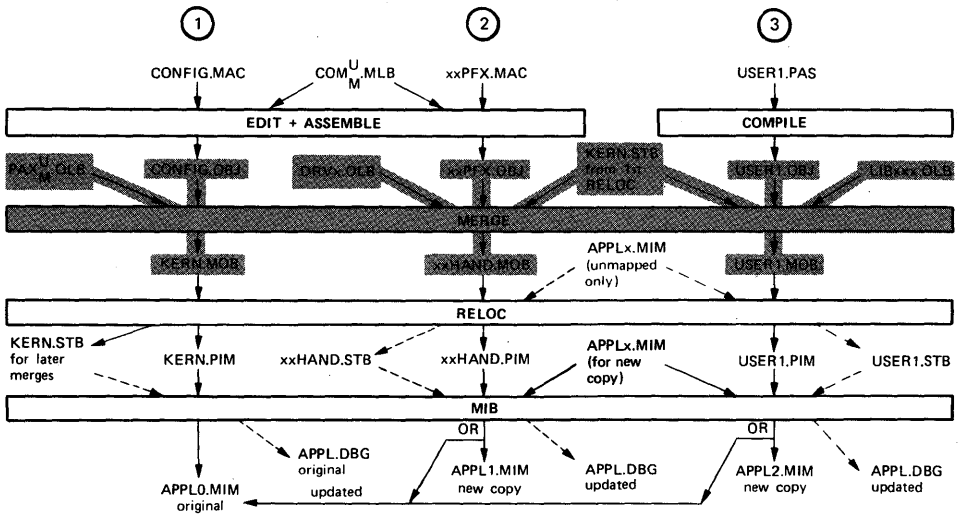
9.2 Role of MERGE in the Build Cycle

You use MERGE at several stages in the application build cycle (see Figure 9–2):

1. You merge the assembled system configuration file with the kernel library (PAXU or PAXM) to create the kernel object module (Chapter 3).
2. You then merge the DIGITAL-supplied system processes required by your target hardware devices—primarily device drivers (Chapter 4).
3. You then use MERGE to create a complete merged object module for each user static process to be included in the application image (Chapter 5).
4. If you are using a shared library, you merge it before you merge any process that references it (Chapter 6).
5. Optionally, you can use MERGE to optimize the kernel primitive modules after a complete application has been developed and tested (Chapter 3).

See Chapter 1 for an overview of the application build cycle.

Figure 9–2: MERGE's Part in the Build Cycle



9.2.1 Merging the System Configuration File (Kernel)

By merging the configuration object module with the kernel object library, you extract and configure the kernel library modules needed for your application. The output of the merge is a customized kernel object module. Chapter 4 of the *MicroPower/Pascal Run-Time Services Manual* describes the system configuration macros and explains how to create or modify a configuration source file for assembly.

RELOC must subsequently process the kernel merged object (.MOB) file to create the kernel image (.PIM) file and the kernel symbol table (.STB) file. You input the kernel image file to MIB to create the initial memory image file with only the kernel installed in it.

9.2.2 Merging Each Static Process

You use MERGE to create a complete object module for each static process in the application, merging the input module(s) for a given process with the kernel symbol table and any needed object libraries. The kernel symbol table is one of the output files produced when you relocate the kernel merged object file. That symbol table file contains global symbol definitions for the kernel, giving the relocated address (or other) value of all global symbols defined in the kernel. MERGE uses the kernel symbol table to resolve run-time references within a static process to kernel primitive service routines. The referenced routines from the specified object libraries are merged into the output .MOB file.

As an alternative to merging the object modules into each static process, a group of routines—typically, the Pascal OTS—can be placed in a shared library in the application. In such an instance, the shared library is built first, along with a library symbol table file. The symbol table file is specified as an input file to MERGE instead of the corresponding object modules or object library (or libraries). MERGE uses the library symbol table file to resolve static-process references to the routines in the shared library. The routines themselves are not placed in the output .MOB file.

9.2.3 Merging a Shared Library

You can use MERGE to create a complete object module for each shared library in the application. The Pascal OTS is typically placed in a shared library. Use MERGE to merge the modules needed in the library with the kernel symbol table. The library modules can be the needed modules from the appropriate OTS object library, your own object modules, or both.

9.3 Invocation and Use of MERGE

For an RSX Development System:

Assuming that MERGE has been installed according to installation procedure defaults, you invoke it by the task name MRG, as follows:

```
>[MCR] MRG
```

(Precede MRG with MCR only if your CLI mode is DCL.) The following three standard RSX forms of direct invocation can be used:

```
>[MCR] MRG command-line
>[MCR] MRG @command-file
```

In this command line, the specified .CMD file contains one or more MERGE command lines.

```
>[MCR] MRG
MRG>command-line  or  @command-file
MRG>
```

The format of the MERGE command line is described below. Only the first two forms of MERGE invocation can be used within a command file. The first form limits the line to 80 characters and precludes the use of continuation lines. The second and third forms can be used to issue several MERGE command lines within one invocation or to avoid the 80-character limit. Type CTRL/Z in response to the MRG> prompt to exit.

For a VAX Development System:

If you have executed the MPSETUP.COM procedure (Section 1.4), you can invoke MERGE by the logical symbol MPMERGE, as follows:

```
$ MPMERGE
MRG>command-line or @command-file
MRG>
```

The format of the MERGE command line is described below. The default type for an indirect command file is .COM; the file may contain one or more MERGE command lines. Type CTRL/Z in response to the MRG> prompt to exit.

Command Line Format

MERGE accepts a command line in the form shown below. All file specifications are in standard RSX format with respect to device and directory (UFD) information if you are using an RSX host system. The standard location for DIGITAL-supplied MicroPower/Pascal files, such as libraries and prefix modules, is MP:[2,10] for an RSX system.

If you are using a VAX/VMS host, however, all file specifications are in standard VMS format with respect to device and directory information. The logical symbol MICROPOWER$LIB defines the VMS device/directory for DIGITAL-supplied MicroPower/Pascal files.

Output files appear on the left side of the equal sign (=), and input files appear on the right. Brackets ([ ]) indicate optional fields of the command. When you omit an optional output file specification, indicate the null field with consecutive commas to hold its place, except in the case of trailing fields. Trailing commas can be omitted. You must specify at least one input file. If no output file is specified, MERGE performs an error check only.

Note

For Version 4.0 or later of VMS, the file name field of a file specification must not exceed nine characters, and the file type field must not exceed three characters.

For Version 4.0 or later of VMS, underscores (_) are not valid in file specifications.

For all versions of VMS, dollar signs ($) are not valid in file specifications.

```
[mobfile],[mapfile][,auxfile]=infile1[,infile2,...][/options]
```

mobfile

The file specification for the output merged object module. The default file type is .MOB. If this field is omitted, no output object module is produced. The debug option, /DE, can be appended to the output file specification; if /DE is used on the output side of the command line, the effect is the same as if /DE were specified for each input file. Use of the /DE option on individual input files, however, is recommended practice.

mapfile

The file specification for the program section map. The default file type is .MAP. If this field is omitted, no section map is generated.

auxfile

The file specification for the auxiliary output file, which will contain a copy of any global references that remain unresolved after the merge. The default file type is .AUX. If you specify an auxiliary file—typically, in building an optimized kernel or a shared library—MERGE assumes that unsatisfied global references are expected, and no warning message is issued if any occur. If this field is omitted, no auxiliary file is produced, and a warning message is issued for any unsatisfied references. The unresolved references will be listed in the section map if one is specified. An auxiliary output file may be thought of as a limited form of object module file, containing only GSDs for undefined symbols.

infile-i

A file specification for an input file, which may be an object module file (.OBJ or .MOB), a symbol table file (.STB), or an object library file (.OLB). The library option, /LB, must be appended to a library file specification. The default file type is .OBJ unless the /LB option is specified; the type default for a library file is .OLB. The debug option, /DE, can be appended to any input file specification. The /DE option is normally used on object module files and may be specified for an object library. In general, you should not specify the /DE option for a .STB file.

/options

Any of the position-independent options summarized in Table 9–1. Multiple options can be specified in a list of the following form:

/option1/option2/...

The /DE, /LB, /LB:xxx, /UL, and /SL options are position dependent; they are associated with the file specification that they follow. The meaning of /DE also depends on which side of the equal sign (=) it appears.

You can extend a single MERGE command line to multiple input lines by using the standard command-continuation symbol, "-", immediately preceding a carriage return anywhere in a partially completed command. The maximum length of a MERGE command line is 355 characters.

The following examples illustrate basic MERGE command line syntax.

Example 1

```
>[MCR] MRG  or  $ MPMERGE
MRG>MYPROG.MOB,MYPROG.MAP=MOD1.OBJ,MOD2.OBJ,MOD3.OBJ
MRG><CTRL/Z>
```

This command merges three object modules—MOD1, MOD2, and MOD3—and produces the merged object file MYPROG.MOB and the section map file MYPROG.MAP. No auxiliary output file is produced, and no debug symbol information is propagated from or generated for the input modules. The CTRL/Z response to the second MRG> prompt causes an exit from MERGE. Since only standard file types are used, you could omit the file types and let MERGE assume the defaults, as shown below:

```
MRG>MYPROG,MYPROG=MOD1,MOD2,MOD3
```

Example 2

```
MRG>,MYPROG=MOD1,MOD2,MOD3,KERNL5.STB
```

This command merges the three object files MOD1, MOD2, and MOD3 and the symbol table file KERNL5.STB but produces only the section map file MYPROG.MAP as output.

Example 3

```
MRG>PROCS1=PROCS1/DE,KRNDBG.STB,mpp-lib:FILSYS/LB,LIBFIS/LB
```

This command merges the object module PROCS1, compiled from Pascal source code using the /DE and /IN:FIS compilation options with the kernel symbol table, KRNDBG.STB, the file system object library, FILSYS.OLB, and the OTS object library, LIBFIS.OLB. KRNDBG.STB resolves references to kernel primitive entry points. FILSYS.OLB resolves references to the file system for doing I/O. LIBFIS.OLB resolves references to OTS modules, and MERGE includes the required file system and OTS modules from the library in the merged object module PROCS1.MOB. The /DE option propagates all debug symbol information (ISD records) that MERGE finds in PROCS1.OBJ to the merged object module.

If MERGE did not find any valid ISD records in the input module, as would be the case if PROCS1.PAS were compiled without /DE, MERGE would generate such records for all global symbol definitions in the module.

Example 4

```
MRG>PROCS1=PROCS1/DE,KRNDBG.STB,SHRLIB.STB/UL
```

9.4 Section Map

If you specify a map file in the command line, MERGE creates a section map that includes the following information:

1. MERGE version identification
2. Date and time of merge
3. Program section information for each referenced shared library—section name, section length, and section attributes
4. Unresolved global references, if any
5. List of files included in the merge

Figures 9–3 and 9–4 show sample MERGE section maps.

Figure 9–3: Sample MERGE Section Map with No Referenced Shared Libraries

```
①MICROPOWER MERGE V02.00 VMS    Load Map    ②Fri 15-Mar-85 10:13:11
                    Title:EXAMPLIdent:062153
③Section  Size   Attributes

    . ABS.     000000 (RW,I,GBL,ABS,OVR)
    .SDAT.     002424 (RW,D,GBL,REL,CON)
    .CODE.     006020 (RO,I,LCL,REL,CON)
    .ODAT.     000022 (RW,D,GBL,REL,OVR)
    .CDAT.     000074 (RW,D,GBL,REL,CON)
    .PBIT.     000050 (RO,D,GBL,REL,OVR)
    .PEIS.     000200 (RO,I,GBL,REL,OVR)
    .PCON.     001516 (RO,D,LCL,REL,CON)
    .ALST.     000076 (RO,I,GBL,REL,CON)
    .INIO.     000000 (RO,D,GBL,REL,CON)
    .INI2.     000000 (RO,D,GBL,REL,CON)
    .IDAT.     000022 (RO,D,GBL,REL,CON)
    .DEBG.     000032 (RO,I,LCL,REL,OVR)
    .INI1.     000002 (RO,D,GBL,REL,CON)
    .SNDF.     177766 (RO,I,LCL,ABS,OVR)
    .HDDF.     177776 (RO,I,LCL,ABS,OVR)
    .STDF.     000012 (RO,I,LCL,ABS,OVR)
    .HDRA.     000400 (RO,I,LCL,ABS,OVR)
    .BSEM.     000006 (RO,I,LCL,ABS,OVR)
    .CSEM.     000006 (RO,I,LCL,ABS,OVR)
    .QSEM.     000012 (RO,I,LCL,ABS,OVR)
    .RBUF.     000036 (RO,I,LCL,ABS,OVR)
    .SREG.     000010 (RO,I,LCL,ABS,OVR)
    .LGNM.     000004 (RO,I,LCL,ABS,OVR)
    .LNAT.     000400 (RO,I,LCL,ABS,OVR)
    .SQUE.     000010 (RO,I,LCL,ABS,OVR)
    .SCDF.     000010 (RO,I,LCL,ABS,OVR)
    .SMDF.     000400 (RO,I,LCL,ABS,OVR)
    .SEDF.     000052 (RO,I,LCL,ABS,OVR)
    .SDDF.     000016 (RO,I,LCL,ABS,OVR)
    .EXTB.     000042 (RO,I,LCL,ABS,OVR)
    .CHND.     000022 (RO,I,LCL,ABS,OVR)
    .AIMP.     000070 (RO,I,LCL,ABS,OVR)
    .OTS.      004764 (RO,I,GBL,REL,CON)
    .EMSK.     100000 (RO,I,LCL,ABS,OVR)
    .EMSC.     000022 (RO,I,LCL,ABS,OVR)
    .ESUB.     100002 (RO,I,LCL,ABS,OVR)
    .AQIO.     000052 (RO,I,LCL,ABS,OVR)
    .AIFN.     000000 (RO,I,LCL,ABS,OVR)
    .AIFM.     100000 (RO,I,LCL,ABS,OVR)
    .AICS.     000012 (RO,I,LCL,ABS,OVR)
    .AIDK.     000006 (RO,I,LCL,ABS,OVR)
    .AITP.     000000 (RO,I,LCL,ABS,OVR)
    .AICM.     000006 (RO,I,LCL,ABS,OVR)
    .AICD.     000000 (RO,I,LCL,ABS,OVR)
    .AITT.     000010 (RO,I,LCL,ABS,OVR)
    .AILP.     000000 (RO,I,LCL,ABS,OVR)
    .AIRT.     000014 (RO,I,LCL,ABS,OVR)
    .AIPR.     000002 (RO,I,LCL,ABS,OVR)
    .AIYK.     000006 (RO,I,LCL,ABS,OVR)
    .AISS.     000006 (RO,I,LCL,ABS,OVR)
    .FDB.      000112 (RO,I,LCL,ABS,OVR)
    .FST0.     100000 (RO,I,LCL,ABS,OVR)
    .FST1.     100000 (RO,I,LCL,ABS,OVR)
    .FSYS.     002410 (RO,I,GBL,REL,CON)
    .FDAT.     000224 (RO,D,GBL,REL,CON)

⑤Files included:

   EXAMPD.OBJ;4
   TE001.STB;1
   FILSYS.OLB;3
   LIBNHD.OLB;3
```

Figure 9–4: Sample MERGE with a Referenced Shared Library

```
①MICROPOWER MERGE V02.00 VMS    Load Map     ②Fri 15-Mar-85 10:16:44
                Title:EXAMPLIdent:062153

  Section  Size    Attributes

    . ABS.     000000 (RW,I,GBL,ABS,OVR)
    .SDAT.     002424 (RW,D,GBL,REL,CON)
    .CODE.     006102 (RO,I,LCL,REL,CON)
    .ODAT.     000022 (RW,D,GBL,REL,OVR)
    .CDAT.     000074 (RW,D,GBL,REL,CON)
    .PBIT.     000050 (RO,D,GBL,REL,OVR)
    .PCON.     001516 (RO,D,LCL,REL,CON)
    .ALST.     000076 (RO,I,GBL,REL,CON)
    .INI0.     000000 (RO,D,GBL,REL,CON)
    .INI2.     000000 (RO,D,GBL,REL,CON)
    .IDAT.     000022 (RO,D,GBL,REL,CON)
    .DEBG.     000032 (RO,I,LCL,REL,OVR)
    .INI1.     000002 (RO,D,GBL,REL,CON)
    $USRLB     007342 (RO,I,LCL,ABS,CON) -- Referenced user-mode library

⑤Files included:

    EXAMED.OBJ;4
    TE101.STB;2
    TE701.STB;1
```

9.5 Merge Options

Sections 9.5.1 through 9.5.8 describe the MERGE options that are summarized in Table 9–1.

Table 9–1: MERGE Options

Option	Meaning
/DE	Includes debug symbol information (ISD records) in the output object file. Kernel and user-process symbol information must be propagated throughout the MERGE, RELOC, and MIB steps to permit the use of PASDBG for symbolic debugging of an application.
/IN:symbol1[:symbol2:...]	Specifies global symbols to be treated as references to any library named in the MERGE command. During its library search, MERGE includes the library modules that satisfy those symbols on a first-found basis.
/LB	Identifies a file as an object library to include in the general library search.
/LB:mod1[:mod2 :...]	Identifies a file as an object library and specifies one or more modules to be extracted from that library. The option arguments mod1, mod2, ..., modn must be object module names, not global symbol references. A file so identified does not take part in the general library search unless it is also separately specified with the /LB option having no arguments.
/NM:name	Specifies the name to assign to the object module created during the MERGE operation. This option overrides the effect of a MACRO–11 .TITLE statement, if any, or a Pascal PROGRAM or MODULE name.

Table 9–1 (Cont.): MERGE Options

Option	Meaning
/SL	Identifies a file as a supervisor-mode shared library .STB file for resolving references to the library entry points.
/UL	Identifies a file as a user-mode shared library .STB file for resolving references to the library entry points.
/VR:xxx	Specifies a program version number or other "ident" value for the output object module. This option overrides the effect of a MACRO–11 .IDENT statement, if any, or the time and date of compilation supplied as an "ident" value by the Pascal compiler.

9.5.1 Debug Symbols (/DE)

The /DE option propagates, or passes along, information about source program symbols for eventual use in symbolic debugging with PASDBG. The debug symbol information is represented by a special form of object file record called an internal symbol directory (ISD) record, formatted for the specific needs of the PASDBG symbolic debugger. Both the MicroPower/Pascal compiler and the MERGE utility can produce the MicroPower/Pascal-specific form of ISD record.

The compiler creates ISD records for all identifiers declared in a program, both global and local, when the /DE option is used.

MERGE creates ISDs when necessary, but only for modules, program sections, and global program symbols. If you do not specify the debug option for the compiler but specify /DE for MERGE, MERGE creates ISD records from any GSD symbol definition records in the .OBJ module produced by the compiler. The resulting ISD records, however, are far less complete than those produced by the compiler's debug option.

Ordinarily, you would use the /DE option of MERGE to create rather than propagate ISDs only for input modules generated by the MACRO–11 assembler. The /EN:DBG option of the MACRO–11 assembler produces ISD records, but those ISDs are not in the format expected by MicroPower/Pascal. For MACRO–11 modules, do not use the /EN:DBG option of MACRO–11; allow the /DE option of MERGE to create ISD records for the module instead. If MERGE encounters an ISD record created by /EN:DBG in an input module, MERGE processes the module correctly, but MERGE issues a warning message and then produces ISD records in the format expected by PASDBG.

In summary, if MERGE finds any valid ISDs in an input module or an extracted library module, MERGE includes them in the output .MOB file. If MERGE finds no ISDs or finds foreign ISDs, it creates ISD records for all global symbols defined by the module in question and includes them in the output .MOB file. Thus, the /DE option includes debug symbol information in the output object module in either case, regardless of whether that information was generated initially by the Pascal compiler or in the merge step.

If you specify /DE on the output .MOB file, MERGE propagates or creates ISD records as described above for all input files. The effect is exactly as if /DE had been appended to each input file specification. Normally, you will not want to use /DE on the output side of the command line, because MERGE will probably generate many ISDs you will have no use for and may duplicate symbols already in the debug file.

Use of /DE on the output side of the command line is not recommended as a general practice. You might do so for convenience, however, where you would otherwise intentionally specify /DE on all of the input files, as in the following example of merging several user object modules into one .OBJ file, perhaps for convenience in later using that single file as input in several other command lines.

In example 1, MERGE includes ISD records in the output module for each of the input modules—PRGXYZ, MODX1, MODX2, and MODX3. In example 2, MERGE includes ISD records only for PRGXYZ.OBJ.

Example 1

```
MRG>PRGXYZ.OBJ/DE=PRGXYZ,MODX1,MODX2,MODX3
```

Example 2

```
MRG>PRGXYZ=PRGXYZ/DE,KRNDBG.STB,mpp-lib:FILSYS/LB,LIBNHD/LB
```

When you build an application for debugging, the module containing the PROGRAM compilation unit in Pascal must be the first input file to merge. If it is not and you do not use the /NM option, the SET PROGRAM command in PASDBG will fail. In example 1, for instance, PRGXYZ.OBJ must be the first input module specified in the command line, assuming that PRGXYZ represents a Pascal PROGRAM unit. See the /NM option description for further information.

When building the kernel for an application with debugging support, you must specify /DE on the kernel library file (PAXU or PAXM), since PASDBG needs kernel global symbols for the debugging of any process in the application. When building a user process for debugging, specify /DE on the object library file(s) only if you specifically want global symbols from object library modules for debugging.

Generally, you should not specify the /DE option for an .STB file. Doing so in the standard case of the kernel .STB file results only in unneeded replication of kernel symbol information and consequent wasted space in the resulting .DBG file. Worse, if the .STB file already contains ISD records, as it frequently does, you get the error message "Bad ISD in—filespec." Similarly, do not use /DE on the output side of a command line that specifies an .STB file as input.

9.5.2 Include Module from Any Library (/IN)

The format of the /IN option is as follows:

```
/IN:symbol1[:symbol2:...]
```

In this format, each "symbol-i" is a name to be treated by MERGE as an undefined global symbol (presumably an object library reference). The /IN option of MERGE extracts and includes in the output module any object library modules MERGE finds that resolve those symbols, during its general search of the libraries named in the MERGE command. This option permits you to force inclusion of a module from a library, even if the module is not needed to satisfy a direct reference in one of the other input object modules, or before the module is needed to satisfy a reference in a subsequent library module. The /IN option may be useful in solving certain forms of interlibrary reference conflict that cannot be solved solely by the order in which the several libraries are specified.

9.5.3 Library File Identification (/LB)

The /LB option with no arguments indicates that the file to which it is appended is an object library that is to take part in the general library search for resolution of undefined global symbols. This option has a distinctly different meaning if specified with arguments, as described in Section 9.5.4.

9.5.4 Extract Modules from Specific Library (/LB:module:...)

The /LB option with module-name arguments indicates that a file is an object library from which only the specified modules are to be extracted. MERGE extracts and processes the named modules as if they were input from individual object files. The format of this form of the /LB option is as follows:

```
/LB:mod1[:mod2 ...:modn]
```

The arguments mod1, mod2, ..., modn are module names, not global-symbol references. A library file so identified does not take part in the general library search for global-symbol definitions unless it is also separately specified with the /LB option (no arguments). For example, consider the following command:

```
MRG>OUTMOD=INPUT1,MYLIB/LB:MODULX,MYLIB/LB,GENLIB/LB
```

The first specification of MYLIB extracts the library module MODULX from the library MYLIB.OLB and merges MODULX with INPUT1.OBJ. The second specification of MYLIB allows MERGE to search MYLIB, prior to GENLIB, if required to satisfy any remaining references. If the second MYLIB/LB, without arguments, did not appear in the command, MERGE would use only GENLIB in its general library search.

9.5.5 Module Name (/NM)

The /NM:name option allows you to specify the name of the merged object module to be created during the merge. The name argument can consist of up to six RAD50 characters. The name you specify supersedes the output module name that MERGE would otherwise choose during its processing of input modules. Module and file names are separate and distinct; the module name is defined in a special form of GSD entry found at the beginning of every object module.

Normally, MERGE uses the module name of the first object module that it encounters in the input stream as the output module name. The module name GSD comes from the .TITLE directive, if any, in a MACRO–11 source module or from the PROGRAM or MODULE name declared in a Pascal compilation unit. The Pascal program name implicitly establishes, therefore, both the run-time process-id and the module name for a static process. If a MACRO–11 source module does not contain a .TITLE directive, the assembler generates the conventional default module name .MAIN.

Ordinarily, you would use the /NM option only in the case of a static process implemented in MACRO–11 to remedy the lack of a .TITLE directive in the first or only input object module, for example, or to override the effect of an incorrect .TITLE directive in the first or only input object module. In addition, as noted above, you would use it to override a default library name.

During debugging, however, the output module name must match the run-time name defined for the static process represented by a merged object module. For a static process implemented in Pascal, the module and static-process name correspondence necessary for debugging is automatically established by the compiler and MERGE, provided that the PROGRAM compilation unit is the first or only object module input to a merge.

After debug-support processing by MERGE, RELOC, and MIB, the module name contained in the .MOB file and "passed through" the related .PIM and .STB files becomes a program node name in the resulting .DBG file, where it identifies the set of debug symbols associated with a given static process, or "program." In response to a SET PROGRAM xxx command, PASDBG looks for a matching node name in the .DBG file in order to locate the debug symbols for the specified static process.

RELOC provides a corresponding option affecting its output .STB file, which allows you in turn to override the module name determined in the MERGE step.

9.5.6 Supervisor-Mode Shared Library .STB File (/SL)

The /SL option identifies the input file as a supervisor-mode library .STB file. This option is needed to resolve references to the library entry points when you are building a process that references a supervisor-mode shared library. The /SL option is position dependent. MERGE checks the file to ensure that only library records are in it. If verification is not successful, the error message "Incompatible use of /UL or /SL on file—filespec" is displayed. If /SL is omitted on a supervisor-mode shared library .STB file, MERGE issues the error message "/UL or /SL missing on referenced library .STB file—filespec."

9.5.7 User-Mode Shared Library .STB File (/UL)

The /UL option identifies the input file as a user-mode library .STB file. This option is needed to resolve references to the library entry points when you are building a process that references a user-mode shared library. The /UL option is position dependent. MERGE checks the file to ensure that only library records are in it. If verification is not successful, the error message "Incompatible use of /UL or /SL on file—filespec" is displayed. If /UL is omitted on a user-mode shared library .STB file, MERGE issues the error message "/UL or /SL missing on referenced library .STB file—filespec."

9.5.8 Version Number (/VR)

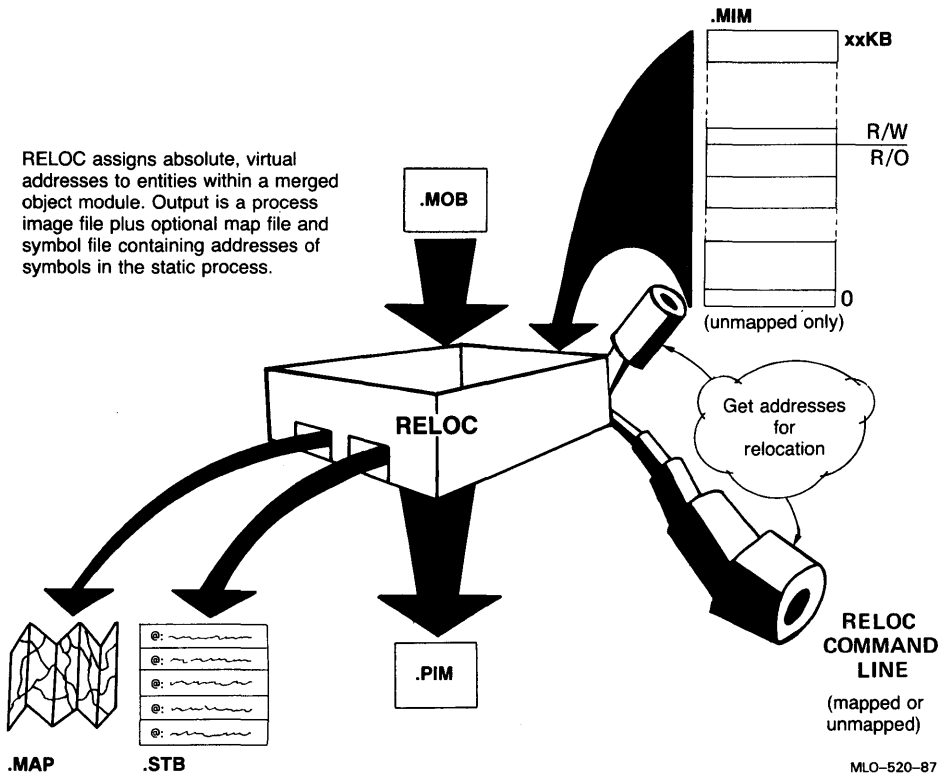
The /VR:xxx option allows you to specify a program version number or other identifier value for the output object module. The option argument can consist of up to six RAD50 characters. This option overrides the effect of a MACRO–11 .IDENT statement, if any, or the time and date of compilation supplied as an "ident" value by the Pascal compiler. The identifier appears in the maps produced by MERGE, RELOC, and MIB.

If you do not specify the /VR option, MERGE uses the first nonblank version number—program identification GSD record—it encounters in the input modules as the version number of the output module. A program identification GSD record comes from the .IDENT statement, if any, in a MACRO–11 module or from the date and time of compilation for a Pascal module.

Chapter 10
The RELOC Utility Program

The RELOC utility program allocates memory for each program section (p-sect), assigns base virtual addresses to the p-sects, sorts p-sects by read-only or read/write attribute alphabetically within each category, relocates the p-sects, and produces as output a process image (.PIM) file. RELOC also produces symbol table (.STB) files. See Figure 10–1.

Figure 10–1: RELOC Utility Input and Output



This chapter discusses the following topics:

- RELOC's functions
- RELOC's role in the build cycle
- The invocation and use of RELOC
- The optional RELOC memory map
- The options you use with RELOC

You can run RELOC yourself, as described herein, but for most applications, you can use MPBUILD.CMD (for RSX) or MPBUILD.COM (for VMS), automated procedures described in Chapter 2.

10.1 Functions of RELOC

After MERGE combines program sections (p-sects), RELOC allocates memory for them and adjusts symbol reference addresses in all p-sects. Prior to relocation, the addresses of all symbols in each p-sect are offsets from a base address of 0, as recorded in the relocation directory (RLD) records of each p-sect. RELOC assigns physical memory addresses for unmapped applications and virtual addresses for mapped applications, allocating all p-sects contiguously by default. RELOC changes the 0-base offset value of each symbol reference so that it becomes the actual address reference required, based on the symbol's position within the relocated program image (.PIM) file. RELOC then adjusts all text records, using relocation directory (RLD) information to assign new addresses.

RELOC normally sorts all p-sects by the read-only (RO), read/write (RW) attributes, collecting all p-sect contributions having the same name, before it relocates them and puts them in the output module. The sorting of p-sects by RO/RW attribute separates code and pure data, which can be loaded in read-only memory (ROM), from impure data, which must be loaded in read/write memory (RAM). In addition, RELOC sorts the p-sects alphabetically by p-sect name within the read-only and read/write segments. You can disable the RELOC sorting of p-sects, but you would not ordinarily do so for MicroPower/Pascal applications.

RELOC also offers some optional capabilities:

- RELOC can begin ROM or RAM segments at specific addresses.
- RELOC can begin RAM segments on the next available 4K-word virtual address boundary.
- You can specify the starting location of a given p-sect in memory; you supply either a physical or virtual address, as required by the application image.
- RELOC can extend a p-sect to a specific size.
- RELOC can begin a p-sect at an address that is a multiple of a specified power of 2.
- You can request a symbol table (.STB) output file, which contains relocated global symbol information. In addition, RELOC can include ISD records in the .STB file. The ISDs are necessary if you want to use PASDBG with the application.
- RELOC can separate instruction and data program segments to take advantage of I/D space hardware on a J11 target.

- RELOC can build user-mode shared libraries and, for target systems having supervisor-mode hardware support, supervisor-mode shared libraries.
- RELOC can provide a relocation map that contains p-sect names, sizes, starting addresses, and global symbols. RELOC can alphabetize the map symbols and create a shortened map.

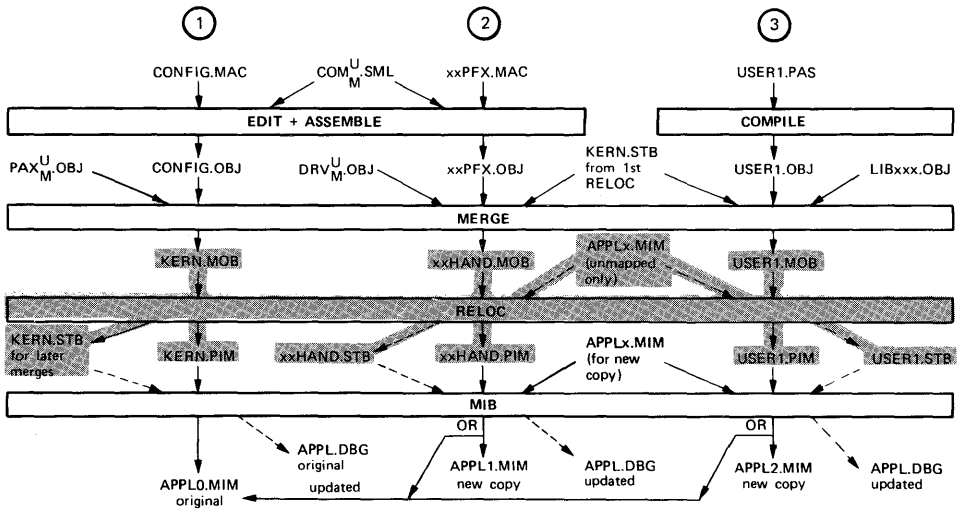
Table 10–1, in Section 10.5, summarizes the RELOC options.

10.2 Role of RELOC in the Build Cycle

You use RELOC at several stages in the build cycle (see Figure 10–2). After you merge the configuration file and the kernel library, you use RELOC to produce the kernel symbol table (.STB) file and the kernel image (.PIM) file. Later in the build cycle, you merge and relocate each static process and each shared library, producing for each its own .PIM file and optionally its own .STB file for debugging. The .STB file for a shared library is needed as input when you merge a static process that references the library. Then you use the MIB utility to insert each process in the memory image file. See Section 4.4 for a general discussion of how RELOC and MIB interact in determining the placement of a process in the memory image, particularly in the case of an unmapped image.

The primary output of RELOC is a process image file that contains relocated p-sects—p-sects that have been allocated physical or virtual addresses. See Chapter 1 for an overview of the application build cycle.

Figure 10–2: RELOC's Part in the Build Cycle



MLO–519B–87

10.3 Invocation and Use of RELOC

For an RSX Development System:

Assuming that RELOC has been installed according to installation procedure defaults, you invoke it by the task name RLC, as follows:

```
>[MCR] RLC
```

(Precede "> RLC" with "MCR" only if your CLI mode is DCL.) The following three standard RSX forms of direct invocation may be used:

- > **[MCR] RLC command-line**
- > **[MCR] RLC @command-file**

  In both instances, the specified .CMD file contains one or more RELOC command lines.

- > **[MCR] RLC** RLC> **command-line** or **@command-file** RLC>

The format of the RELOC command line is described below. Only the first two forms of RELOC invocation can be used within a command file. The first form limits the line to 80 characters and precludes the use of continuation lines. You can use the second and third forms to issue several RELOC command lines within one invocation or to avoid the 80-character limit. Type CTRL/Z in response to the RLC> prompt to exit.

For a VAX Development System:

If you have executed the MPSETUP.COM procedure (Section 1.4), you can invoke RELOC by the logical symbol MPRELOC, as follows:

```
$ MPRELOC
RLC>command-line  or  @command-file
RLC>
```

The format of the RELOC command line is described below. The default type for an indirect command file is .COM; the file can contain one or more RELOC command lines. To exit, type CTRL/Z in response to the RLC> prompt.

Command Line Format

RELOC accepts a command line in the form shown below. In the command line, all file specifications are in the standard RSX format with respect to device and directory (UFD) information if you are using an RSX host system. If you are using a VAX/VMS host, however, all file specifications are in standard VMS format with respect to device and directory information.

Output files appear on the left side of the equal sign (=); input files, on the right. Brackets ([ ]) in the sample command lines indicate optional fields of the command. When you omit an optional output file specification, indicate the null field with consecutive commas to "hold its place," except in the case of trailing fields. Trailing commas can be omitted. You must specify at least one input file.

Note

For Version 4.0 or later of VMS, the file name field of a file specification must not exceed nine characters, and the file type field must not exceed three characters.

For Version 4.0 or later of VMS, underscores (_) are not valid in file specifications.

For all versions of VMS, dollar signs ($) are not valid in file specifications.

```
RLC>[pimfile],[mapfile][,stbfile]=mobfile[,mimfile][/options]
```

pimfile

The file specification for the output image file containing the relocated, executable program text that RELOC produces. The default file type is .PIM (for "process image"). If no output file is specified, RELOC performs an error check only, and no image file is produced.

mapfile

The file specification for the RELOC map file. The default file type is .MAP. If this field is omitted, no relocation map is generated.

stbfile

The file specification for the symbol table file. The default file type is .STB. The symbol table consists of a global symbol name GSD entry, under the p-sect name . ABS., for each global symbol defined in the input .MOB file. The entries contain absolute, relocated addresses for the symbols, reflecting either physical or virtual addresses in the application memory image. These entries are needed for a kernel .STB file for subsequent merging with processes.

For a shared library, the GSDs for all symbols from the . ABS. p-sect are under the . ABS. p-sect. The GSDs for all symbols from other program sections are under a p-sect having the same name as the library. These entries are needed for subsequent merging with any processes that reference the library.

In addition, if the /DE option is specified, the .STB file also contains all the ISD records found in the input .MOB file, following their relocation. Thus, the ISDs also contain absolute symbol values, reflecting physical or virtual addresses in the memory image, depending on the type of target system. You need the ISD records when building either a kernel or a user process for debugging, since the .STB file is the link for passing debug symbol information between the RELOC and MIB steps for inclusion in MIB's .DBG file. A process .STB file, unlike the kernel or shared library .STB file, is generally of no further use following the MIB step for that process. If this field is omitted, no symbol table file is produced.

mobfile

The file specification of the merged object module to be relocated. The default file type is .MOB. You must specify an input mobfile.

mimfile

The memory image (.MIM) file in which the process image will be installed in the succeeding MIB step (unmapped applications only). The default file type is .MIM.

- Always specify an input .MIM file if you are relocating a static process or shared library for an unmapped memory image.
- Never specify an input .MIM file when you are building the kernel for an unmapped application.
- Never specify an input .MIM file for a mapped application.

If you do not specify any special addressing for the process, RELOC reads the memory image file to obtain the next available physical starting address(es) in the image for use as relocation base values and to verify that the .MIM file is built unmapped. RELOC does not look up start addresses in the .MIM file if you use any of the following options; RELOC checks only to ensure that the file is an unmapped application:

```
/RO     /EX     /UL:addr
/RW     /UP     /DS
/QB     /AL
```

/options

Any of the options summarized in Table 10–1. All RELOC options are position independent. Multiple options can be specified in a list of the following form:

/option1/option2/...

The following examples illustrate basic RELOC command line syntax.

Example 1

```
>[MCR] RLC  or  $ MPRELOC
RLC>PROCXZ.PIM,PROCXZ.MAP,PROCXZ.STB=PROCXZ.MOB/DE
RLC><CTRL/Z>
```

This command relocates the merged object module PROCXZ.MOB for a mapped application and produces the process image file (PROCXZ.PIM), the map file (PROCXZ.MAP), and the symbol table file (PROCXZ.STB). The /DE option includes in the .STB file any debug symbol information (ISDs) contained in the .MOB file. Since only the standard file types are used, you could omit the file types and let RELOC assume the defaults, as follows:

```
RLC>PROCXZ,PROCXZ,PROCXZ=PROCXZ/DE
```

Example 2

```
RLC>PRCABC,PRCABC=PRCABC,APPLUM
```

This command relocates the merged object file PRCABC.MOB based on physical starting addresses obtained by inspection of the unmapped APPLUM.MIM memory image file. The output specified on the left side of the equal sign consists of the process image file PRCABC.PIM and the map file PRCABC.MAP. The command does not request a symbol table file; therefore, no /DE option is applicable. This form of RELOC command, specifying an input memory image file, is valid only for unmapped applications. Since no special addressing options are specified, RELOC will look in the .MIM file for the next available physical starting address.

Example 3

```
RLC>,PRCABC=PRCABC,APPLUM
```

This command relocates the merged object file as described in Example 2 but produces only a relocation map file.

10.4 Relocation Map

If you specify a map file in the command line, RELOC creates a "load" map that contains the following information:

1. RELOC version identification
2. Date and time of relocation
3. The .PIM file specification and input module name (Title) and version (Ident)
4. Information about each p-sect processed by RELOC during the relocation, including p-sect name, p-sect attributes, p-sect base address, p-sect length, and global symbol names and values
5. Nominal transfer address; not significant, see the MIB memory map
6. Total ROM (RO segment) size
7. Total RAM (RW segment) size

For an I/D process or a supervisor-mode shared library, separate information is listed about instruction space and data space and separate I–RO, I–RW, D–RO, and D–RW information. For a process that references a supervisor-mode shared library or one or more user-mode shared libraries, separate information on each shared library is given. All symbols are listed under a single p-sect, with the name of the library for a user-mode library or a supervisor-mode shared library with no data. Two p-sects are listed, having the same name (one instruction space, the other data space) for a supervisor-mode shared library with data. The map also reports any undefined global symbols encountered during relocation.

By default, the map is three columns wide for convenient display on a video terminal. You can, however, use the /WI option to produce a 6-column map listing. Figure 10–3 shows a sample 3-column RELOC map, with circled numbers keyed to the items listed above. An extensive midsection of the map has been deleted from the sample reproduction, as indicated by the vertical line of dots.

Figure 10–3: Sample RELOC Map of a Process Without I/D Separation or Shared Libraries

```
(1)MICROPOWER RELOC V02.00 VMS     Load Map      (2)Fri 15-Mar-85 10:20:07
(3)TE003.PIM       Title:  EXAMPL  Ident:  062130

(4)Section   Address  Size          Global  Value   Global  Value   Global Value

. ABS.     000000 000000 (RW,I,GBL,ABS,OVR)

                                   HA$CMR 000000   HA$FIS 000000   HA$ROM 000000
                                   HA$FPP 000000   HA$T11 000000   HA$J11 000000
                                   HA$IOP 000000   HA$Q22 000000   SA$RIR 000000

                                          .

                                          .

                                          .


.OTS.      011030 004760 (RO,I,GBL,REL,CON)

                                   $B75   011030   $SAV6  011030   $B77   011054
                                   $RES6  011054   $RDC   011072   $B0    011072
                                   $B2    011072   $B20   011130   $B22   011130
                                   $WRC   011130   $B32   011244   $B34   011244
                                   $WRS   011244   $B36   011424   $B38   011424
                                   $WRL   011424   $CRPCI 011522   $B200  011522
                                   $CRBSI 012322   $SGNLI 012420   $WAITI 012456
                                   $STPCI 012514   $DLSTI 012552   $PTERM 012610
                                   $STERM 012610   $B63   012610   $START 013114
                                   $CLRTN 013440   $B24   013440   $B26   013440
                                   $WRI   013440   $WRIR  013450   $IMOD  013674
                                   $B82   013674   $BGCHK 013774   $SV05  014012
                                   $SV03  014052   $SV02  014102   $NEWC  014126
                                   $NEW   014130   $B70   014130   $DSPC  014414
                                   $B72   014416   $DSP   014416   $KRPTR 014664
                                   $RPTER 014674   $SAV5  015016   $RES5  015040
                                   $DIVU  015054   $GETCH 015104   $STSAV 015134
                                   $STCLR 015164   $POP8  015200   $POP12 015214
                                   $COM1  015230   $COM2  015354   $COM3  015474
                                   $PSTRT 015572   $LDFPP 015604   $B120  015606
                                   $UMOD  015606   $UDIV  015644   $B118  015644
                                   $SAV4  015740   $RES4  015760   $COP3  015772

.PBIT.     016010 000050 (RO,D,GBL,REL,OVR)

.PCON.     016060 001516 (RO,D,LCL,REL,CON)

.PEIS.     017576 000200 (RO,I,GBL,REL,OVR)

.QSEM.     000000 000000 (RO,I,LCL,ABS,OVR)

.RBUF.     000000 000000 (RO,I,LCL,ABS,OVR)

.SCDF.     000000 000000 (RO,I,LCL,ABS,OVR)

.SDDF.     000000 000000 (RO,I,LCL,ABS,OVR)

.SEDF.     000000 000000 (RO,I,LCL,ABS,OVR)
```

Figure 10–3 (Cont.): Sample RELOC Map of a Process Without I/D Separation or Shared Libraries

```
.SMDF.    000000 000000 (RO,I,LCL,ABS,OVR)
.SNDF.    000000 000000 (RO,I,LCL,ABS,OVR)
.SQUE.    000000 000000 (RO,I,LCL,ABS,OVR)
.SREG.    000000 000000 (RO,I,LCL,ABS,OVR)
.STDF.    000000 000000 (RO,I,LCL,ABS,OVR)
.CDAT.    017776 000074 (RW,D,GBL,REL,CON)
.ODAT.    020072 000022 (RW,D,GBL,REL,OVR)
.SDAT.    020114 002424 (RW,D,GBL,REL,CON)

(5)Transfer address =  000001

(6)Total ROM size =  017776

(7)Total RAM size =  002542
```

10.5 RELOC Options

This section describes the RELOC options summarized in Table 10–1.

Note

Any numeric value specified in a RELOC option is assumed to be expressed in octal unless the number is terminated by a decimal point. Thus, the expressions 100 and 64. represent the same value, and 108 is an invalid expression.

Table 10–1: RELOC Options

Option	Meaning
/AB	Lists symbol names alphabetically within p-sects in the load map.
/AL	Aligns the first read/write p-sect on the next available 4K-word address boundary; intended specifically for mapped ROM/RAM build operations.
/DE	Includes debug symbol information (ISD records) as well as global symbol directory records (GSDs) in the symbol table file. If you specify a symbol table file in the RELOC command but do not specify /DE, RELOC puts only GSDs in the .STB file.
/DR:n	Specifies starting address n of read-only D-space section. Use with /ID option for separate instruction and data (I/D) space on J11 target.
/DS	Leaves p-sects in the order of their occurrence in the input module, disabling RELOC's normal sorting of p-sects first by read-only and read/write attribute and then by alphabetical order of p-sect names within the read-only and read/write segments. (This option is used only in a few special cases.)

Table 10–1 (Cont.): RELOC Options

Option	Meaning
/DW:n	Specifies starting address n of read/write D-space section. Use with /ID option for separate instruction and data (I/D) space on J11 target.
/EX:name:size	Extends a p-sect to a specified size, in bytes.
/ID	Builds application for target with separate instruction and data (I/D) spaces. This option is applicable only to a mapped J11 target system with I/D hardware support.
/LS:name:addr	Overrides the default memory allocation algorithm for relocatable libraries in a mapped system.
/NM:name	Specifies, for the output .STB file, the name of the module to be relocated; overrides actual input module name from the .MOB file. Overrides the PASDBG "program name" in only the .STB file and later the debug file for Pascal processes. For shared libraries, this name overrides the library name (written in the library list element in the .PIM file and in the .STB file and later in the debug file).
/QB:name:addr[:...]	Sets the base address for the named p-sect; multiple pairs of p-sect names and base addresses can be specified.
/RO:addr	Starts the first read-only p-sect at the specified physical or /RO:addr virtual address. If used with /DS, the base of the program is unconditionally set to addr. If used with /ID, the /RO option specifies the starting address of read-only I-space.
/RW:addr	Starts the first read/write p-sect at the specified physical or virtual address. If used with /ID, the /RW option specifies the starting address of read/write I-space.
/SH	Produces a shortened load map; symbols defined in the . ABS. p-sect are omitted (kernel symbols from the kernel .STB file).
/SL	Builds a supervisor-mode shared library.
/UL[:addr]	Builds a user-mode shared library. If the option includes an address, RELOC builds an absolute rather than a relocatable library; addr specifies the base virtual address for the absolute library.
/UP:name:n	Rounds up the length of the named p-sect so that the next free address is a whole-number multiple of n; n must be a power of 2.
/VR:xxx	Specifies a program version number or other "ident" to appear in the load map for the module to be relocated.
/WI	Produces a map listing six columns wide rather than the usual three; useful for line printer listings.
/ZR:n	Sets the value of unaccessed locations in the image to n; the default value is 0.

Figure 10–4 shows a sample RELOC map for a process with I/D separation.

Figure 10–4: Sample RELOC Map of a Process with I/D Separation

```
①MICROPOWER RELOC V02.00 VMS     Load Map      ②Fri 15-Mar-85 10:24:17
③TE403.PIM       Title:  EXAMPL  Ident:  062135

④INSTRUCTION SPACE ALLOCATION

  Section   Address  Size          Global Value Global  Value  Global Value

    . ABS.    000000 000000 (RW,I,GBL,ABS,OVR)

                                  HA$CMR 000000 HA$FIS 000000 HA$ROM 000000
                                  HA$FPP 000000 HA$T11 000000 HA$F11 000000
                                  HA$IOP 000000 HA$Q22 000000 SA$RIR 000000

                                         .
                                         .
                                         .

    .OTS.     010702 004424 (RO,I,GBL,REL,CON)

                                  $B75   010702 $SAV6  010702 $B77   010726
                                  $RES6  010726 $RDC   010744 $B0    010744
                                  $B2    010744 $B20   011002 $B22   011002

                                         .
                                         .
                                         .

     .STDF.     000000 000000 (RO,I,LCL,ABS,OVR)

④DATA SPACE ALLOCATION

  Section   Address  Size          Global Value Global  Value  Global Value

     .FDAT.    000000 000224 (RO,D,GBL,REL,CON)
     .IDAT.    000224 000022 (RO,D,GBL,REL,CON)
     .INI0.    000246 000000 (RO,D,GBL,REL,CON)
     .INI1.    000246 000002 (RO,D,GBL,REL,CON)
     .INI2.    000250 000000 (RO,D,GBL,REL,CON)
     .PBIT.    000250 000050 (RO,D,GBL,REL,OVR)
     .PCON.    000320 001516 (RO,D,LCL,REL,CON)
     .CDAT.    002036 000074 (RW,D,GBL,REL,CON)
     .ODAT.    002132 000022 (RW,D,GBL,REL,OVR)
     .SDAT.    002154 002424 (RW,D,GBL,REL,CON)

⑤Transfer address =  000001

⑥Total INSTRUCTION SPACE ROM size =  015326

⑦Total INSTRUCTION SPACE RAM size =  000000

⑥Total DATA SPACE ROM size =  002036

⑦Total DATA SPACE RAM size =  002542
```

10.5.1 Alphabetical Symbol Listing (/AB)

The /AB option lists symbol names in the load map alphabetically within a given p-sect. RELOC normally lists symbols in order of their value, which reflects the order of the corresponding locations in the image in the case of address symbols.

10.5.2 Align First RW Section at 4K-Word Boundary (/AL)

The /AL option starts the first read/write (RW) p-sect found in the input module (after the RELOC p-sect sort) on the next available 4K-word address boundary, presumably virtual. This option is intended specifically for mapped build operations. You ordinarily use it when relocating a process for a mapped ROM/RAM application, since each page address register (PAR) must start on a 4K-word boundary, and the RW (or RAM) segment of the static process is, in general, not contiguous with the RO (or ROM) segment. This option conveniently provides the virtual address adjustment required for the first RW p-sect that is not contiguous with the last RO p-sect. For I/D processes, /AL starts the first RW I-space section at the next available 4K-word virtual address boundary in instruction space and the first RW D-space section at the next available 4K-word virtual address boundary in data space.

For a RAM-only target environment, MIB treats the RW, or high-order, segment of a process as contiguous with the RO segment unless special relocation is specified in RELOC. That is, in the RAM-only case, MIB treats all p-sects as if they were read/write p-sects when positioning the p-sects in physical memory. Special relocation in a mapped RAM-only environment is ordinarily needed only for driver mapped device driver processes. (You use /RO and /RW in relocating driver-mapped processes for any mapped image.)

If you are building an unmapped application and use /AL—not common practice—RELOC does not look up start addresses in the input .MIM file. In addition, the /AL option and the /RW, /DW, or /DS option are mutually exclusive.

The /AL option is intended for mixed ROM/RAM applications, but you can use the option in other cases as well to get particular effects. If a mapped process is to be used in a mixed ROM/RAM configuration, you must use the /AL option of RELOC, which starts the RW (RAM) segment on a 4K-word virtual address boundary. The mapping hardware requires this separation, but the result is that up to 4K–1 words of virtual address space may be wasted between the RO and the RW sections; if the RO section is one word larger than a 4K boundary, all the addresses up to the next 4K boundary will be unused.

An application for a mapped RAM-only target does not require the /AL option, since all memory is the same and there is no need to start the RW section at any particular boundary. Since there is no gap between the RO and RW sections, no virtual address space is wasted. If you do not use /AL, however, RELOC includes both the RO and RW sections of the application in the output .MIM file, producing a larger disk image. Although there can be a loss of virtual address space, use of the /AL option reduces the size of the output .MIM file, thereby saving disk space. When you use the /AL option, RELOC places only the RO portion of the application in the output .MIM file. The RW portion, which is always initialized at run time, is left out of the .MIM file, making its disk image smaller. If you want, you can use /AL when building a RAM-only application; that will have the effect of possibly wasting almost 4K-words of virtual address space, but it will save on disk space.

Figure 10–5 shows a sample RELOC map of a shared library.

Figure 10–5: Sample RELOC Map of a Shared Library

```
①MICROPOWER RELOC V02.00 VMS      Load Map       ②Sun 03-Mar-85 13:25:36
③TE601.PIM        Title:  $USRLB  Ident:  062130

④                 USER-MODE LIBRARY

  Section   Address  Size          Global  Value  Global  Value  Global  Value

    .AICD.    000000 000000 (RO,I,LCL,ABS,OVR)
    .AICM.    000000 000000 (RO,I,LCL,ABS,OVR)
                                        .
                                        .
                                        .

    .OTS.     002716 004760 (RO,I,GBL,REL,CON)
                                $B75    002716 $SAV6   002716 $B77    002742
                                $RES6   002742 $RDC    002760 $B0     002760
                                $B2     002760 $B20    003016 $B22    003016
                                $WRC    003016 $B32    003132 $B34    003132
                                $WRS    003132 $B36    003312 $B38    003312
                                $WRL    003312 $CRPCI  003410 $B200   003410
                                $CRBSI  004210 $SGNLI  004306 $WAITI  004344
                                $STPCI  004402 $DLSTI  004440 $PTERM  004476
                                $STERM  004476 $B63    004476 $START  005002
                                $CLRTN  005326 $B24    005326 $B26    005326
                                $WRI    005326 $WRIR   005336 $IMOD   005562
                                $B82    005562 $BGCHK  005662 $SV05   005700
                                $SV03   005740 $SV02   005770 $NEWC   006014
                                $NEW    006016 $B70    006016 $DSPC   006302
                                $B72    006304 $DSP    006304 $KRPTR  006552
                                $RPTER  006562 $SAV5   006704 $RES5   006726
                                $DIVU   006742 $GETCH  006772 $STSAV  007022
                                $STCLR  007052 $POP8   007066 $POP12  007102
                                $COM1   007116 $COM2   007242 $COM3   007362
                                $PSTRT  007460 $LDFPP  007472 $B120   007474
                                $UMOD   007474 $UDIV   007532 $B118   007532
                                $SAV4   007626 $RES4   007646 $COP3   007660

    .PTDF.    000000 000000 (RO,I,LCL,ABS,OVR)
    .QSEM.    000000 000000 (RO,I,LCL,ABS,OVR)
    .RBUF.    000000 000000 (RO,I,LCL,ABS,OVR)
    .SCDF.    000000 000000 (RO,I,LCL,ABS,OVR)
    .SDDF.    000000 000000 (RO,I,LCL,ABS,OVR)
    .SEDF.    000000 000000 (RO,I,LCL,ABS,OVR)
    .SMDF.    000000 000000 (RO,I,LCL,ABS,OVR)
    .SNDF.    000000 000000 (RO,I,LCL,ABS,OVR)
    .SPMP.    000000 000000 (RO,I,LCL,ABS,OVR)
    .SQUE.    000000 000000 (RO,I,LCL,ABS,OVR)
    .SREG.    000000 000000 (RO,I,LCL,ABS,OVR)
    .STDF.    000000 000000 (RO,I,LCL,ABS,OVR)

⑥Total ROM size =  007676

⑦Total RAM size =  000000
```

10.5.3 Debug Symbols (/DE)

The /DE option relocates and includes in the output symbol table (.STB) file all internal symbol directory (ISD) records found in the input module. The ISDs contain symbol information that is subsequently installed in the .DBG file by MIB and used by PASDBG. The /DE option has no effect on the output .PIM file. You must specify /DE when relocating a kernel for an application to be built with debug support and when relocating any process or shared library that you will want to debug with PASDBG.

10.5.4 Starting Address of Read-Only Data Space (/DR:n)

The /DR:n option allows you to specify the starting address of the read-only D-space portion in an application built with I/D separation by means of the /ID option.

10.5.5 Disable Section Sort (/DS)

The /DS option prevents RELOC from performing its standard p-sect sort and instead relocates all p-sects in the process image according to the order of their occurrence in the input module file. Otherwise, RELOC first segregates the p-sects into two groups, or segments, by RO/RW attribute, with the RO segment preceding the RW segment, and then sorts the p-sects alphabetically by p-sect name within each segment.

If you use the /ID option with the /DS option, RELOC separates instruction and data space but does not sort the sections within those spaces.

Many intrinsic build-time and run-time MicroPower/Pascal mechanisms depend on RELOC's grouping and alphabetic sorting of p-sects for their proper functioning. In addition, the segregation of RO and RW p-sects is crucial for the physical memory segmentation required by an actual or simulated ROM/RAM target environment. The specific requirements for the .ALST. p-sect are described below. The /DS option, therefore, is not intended for general use. If you do use it, do so with great care in putting the p-sects in proper order.

You cannot use the /RW, /DW, or /AL options if you use the /DS option. Do not specify /DS for a kernel build. If you specify /DS for a process in an unmapped application, RELOC does not look up start addresses in the .MIM file. You must therefore specify start addresses as needed by using the /RO, /RW, or /QB option. In addition, if you use /DS, the load map produced by RELOC does not show the ROM (RO segment) and RAM (RW segment) total sizes.

Relocating the .ALST. P-sect

A static process for any kind of target system environment must meet one general requirement. The first relocatable p-sect for any static process must contain the static process list element in order for that element to be correctly positioned in the process image, allowing the kernel to "find" the process at initialization time. That is achieved through standard MicroPower/Pascal conventions and the normal RELOC sort by ensuring that the p-sect containing the list element, named .ALST., is the alphabetically "lowest" read-only p-sect in any static process .MOB file. Consequently, the text of that p-sect is placed properly at the beginning of the process.

In a MACRO–11 source program, the Define Static Process (DFSPC$) macro call establishes the .ALST. p-sect, which contains the static process list element generated by DFSPC$. The MACRO programmer, therefore, must not define any p-sect that will precede that section but should use only the standard MicroPower/Pascal macros PURE$, PDAT$, and IMPUR$ for program sectioning in a MACRO–11 process implementation.

The MicroPower/Pascal compiler automatically generates the .ALST. p-sect for a PROGRAM compilation unit.

Zero-length p-sects such as . ABS. are not a consideration in determining p-sect ordering.

Figure 10–6 shows a sample RELOC for a process that references a shared library.

Figure 10–6: Sample RELOC Map of a Process That References a Shared Library

```
(1)MICROPOWER RELOC V02.00 VMS     Load Map      (2)Sun 03-Mar-85 13:26:26
(3)TE602.PIM        Title:  EXAMPL  Ident:  062130
(4)Section   Address  Size         Global  Value   Global  Value   Global  Value

    .ALST.    000000 000076 (RO,I,GBL,REL,CON)
    .CODE.    000076 006020 (RO,I,LCL,REL,CON)
                                $BEGIN 001736
    .DEBG.    006116 000032 (RO,I,LCL,REL,OVR)
    .IDAT.    006150 000022 (RO,D,GBL,REL,CON)
    .INI0.    006172 000000 (RO,D,GBL,REL,CON)
    .INI1.    006172 000002 (RO,D,GBL,REL,CON)
    .INI2.    006174 000000 (RO,D,GBL,REL,CON)
    .PBIT.    006174 000050 (RO,D,GBL,REL,OVR)
    .PCON.    006244 001516 (RO,D,LCL,REL,CON)
    .PEIS.    007762 000200 (RO,I,GBL,REL,OVR)
    .CDAT.    010162 000074 (RW,D,GBL,REL,CON)
    .ODAT.    010256 000022 (RW,D,GBL,REL,OVR)
    .SDAT.    010300 002424 (RW,D,GBL,REL,CON)

(4)REFERENCED USER-MODE LIBRARIES
  Section   Address  Size         Global  Value   Global  Value   Global  Value
    $USRLB    020000 007676 (RO,I,LCL,REL,CON)
                                $B208   020306   $B205   020312   $B62    020320
                                $WRFV   020412   $B57    020412   $B59    020432
                                $RDFV   020432   $LOCKF  020452   $UNLKF  020472
                                $B206   020500   $IGET   020522   $B61    020522
                                $ILZY   020524   $B60    021302   $PUTCH  021304
                                $BRAKF  021504   $BRAKA  021514   $BRAKB  021520
                                $WTWRT  022000   $IOBEG  022062   $IOEND  022336
                                $FSWT   022340   $QIO    022356   $B75    022716
                                $SAV6   022716   $B77    022742   $RES6   022742
                                $RDC    022760   $B0     022760   $B2     022760
                                $B20    023016   $B22    023016   $WRC    023016
                                $B32    023132   $B34    023132   $WRS    023132
                                $B36    023312   $B38    023312   $WRL    023312
                                $CRPCI  023410   $B200   023410   $CRBSI  024210
                                $SGNLI  024306   $WAITI  024344   $STPCI  024402
                                $DLSTI  024440   $PTERM  024476   $STERM  024476
                                $B63    024476   $START  025002   $CLRTN  025326
                                $B24    025326   $B26    025326   $WRI    025326
                                $WRIR   025336   $IMOD   025562   $B82    025562
                                $BGCHK  025662   $SV05   025700   $SV03   025740
                                $SV02   025770   $NEWC   026014   $NEW    026016
                                $B70    026016   $DSPC   026302   $B72    026304
                                $DSP    026304   $KRPTR  026552   $RPTER  026562
                                $SAV5   026704   $RES5   026726   $DIVU   026742
                                $GETCH  026772   $STSAV  027022   $STCLR  027052
                                $POP8   027066   $POP12  027102   $COM1   027116
                                $COM2   027242   $COM3   027362   $PSTRT  027460
                                $LDFPP  027472   $B120   027474   $UMOD   027474
                                $UDIV   027532   $B118   027532   $SAV4   027626
                                $RES4   027646   $COP3   027660

(5)Transfer address =  000001
(6)Total ROM size =  010162
(7)Total RAM size =  002542
```

10.5.6 Starting Address of Read/Write Data Space (/DW:n)

The /DW:n option allows you to specify the starting address of the read/write D-space portion in an application built with I/D separation using the /ID option.

10.5.7 Extend Section to Specified Size (/EX)

The /EX:name:size option lets you extend a named p-sect to a specified size. The name argument to the option is a p-sect name, and the size argument is the total number of bytes, expressed in octal, you want allocated to the p-sect. The size value must be an even number greater than the actual p-sect size; if the value is less, RELOC ignores the option value and uses the actual p-sect size. In any case, RELOC starts the following p-sect at the next contiguous location.

You can use this option, for example, to extend a p-sect to increase the size of a stack or to allocate patch space at the end of the p-sect.

If you are building an unmapped application and use /EX, RELOC does not look up start addresses in the memory image (.MIM) file. (You should still specify the .MIM file, though.) That is, you must use the /RO option, and possibly /RW also, in addition to /EX in the unmapped case.

You can extend only one p-sect in a given static process.

10.5.8 Separate Instruction and Data (I/D) Space (/ID)

The /ID option builds a process with separate instruction and data space. This option is applicable only to applications being built for a J11 target system with memory mapping and I/D hardware support.

If you do not specify this option, RELOC allocates memory first to all RO program sections, sorted alphabetically by p-sect name, then to all RW program sections, sorted alphabetically, with no separation of instructions and data. If you specify /ID, RELOC separates instruction segments from data segments before allocating memory. Then, RELOC allocates memory separately in instruction space for instruction segments and in data space for data segments. In each address space, RO and RW sections are separated and sorted alphabetically.

Note

Except as a way to increase the virtual address space available to a program, I/D separation offers no advantages unless the program (static process) is built with a supervisor-mode shared library. Use I/D separation when you are faced with addressing limitations that I/D separation can solve for you; otherwise, ordinary mapped format is just as good.

10.5.9 Define User Library Base Address (/LS:name:addr)

The /LS option allows you to specify base virtual addresses for user-mode libraries in a mapped application instead of using RELOC's default memory allocation algorithm. Specify this option only on libraries that were built relocatable for which you want to change the default base address. You determine the default base address by first allocating memory for the process and then looking for the first segment beginning on a 4K-word virtual address boundary that is large enough to map the library. When you are using this option, the user-mode library module is name, and the desired virtual base address is addr, which must be a 4K-word boundary. You can specify multiple libraries by using multiple arguments. For example, a RELOC command might be:

```
>RLC
RLC>PROC1.PIM=PROC1.MOB/LS:LIB1:100000:LIB2:140000
```

You can use this option only for libraries that were built as relocatable. If you attempt to use the option on an absolute library, you will get an error message.

10.5.10 Program/Process Name (/NM)

The /NM:name option lets you specify a program (static-process) name for debugging, overriding the module name GSD/ISD entries contained in the input merged object module.

This option is provided as an alternative to the MERGE /NM option for Pascal processes. (See the description of /NM for the MERGE utility.) The name argument can consist of up to six RAD50 characters. The option affects only the output .STB file and, consequently, the .DBG file updated in the subsequent MIB step.

For shared libraries, using /NM lets you change the name of the library in the library list element of the application and in the .STB file and, consequently, in the debug file updated in the subsequent MIB step. If your application has multiple user-mode libraries, this option must be used to ensure that the names of the libraries are unique. If you do not use the /NM option, RELOC uses the name in the module name GSD in the shared-library object module (LIBSUP.OBJ for a supervisor-mode shared library or LIBUSR.OBJ for a user-mode shared library). The module name GSD comes from the .TITLE directive in the MACRO–11 source module.

10.5.11 Base Address for Specified Program Section (/QB)

The /QB option lets you specify a base address for any named p-sect in the input module. In the absence of any special relocation options, RELOC starts each p-sect at the next available physical or virtual memory address. If that memory location must be used for another purpose or is nonexistent, the /QB option allows you to override the normal "next contiguous address" relocation of any given p-sect to meet special application requirements or to satisfy special target-memory constraints. The format of the /QB option is:

```
/QB:name1:addr[:name2:addr ...]
```

In this format, each "name-i" is a p-sect name, and addr is the starting address for that p-sect. You can specify up to eight p-sect names and addresses. For unmapped applications, /QB specifies a physical address; for mapped applications, /QB specifies a virtual address. If you are building an unmapped application and use /QB, RELOC does not look up start addresses in the memory image (.MIM) file.

Note

In a mapped target system, any p-sect noncontiguous with other p-sects must start on a 4K-word virtual address boundary to satisfy mapping hardware requirements. In addition, for mapped applications, the .ALST. program section containing the static process or library list element must start on a 4K-word virtual address boundary.

10.5.12 First RO Section at Specified Address (/RO)

The /RO:addr option starts the first read-only (RO) p-sect found in the input module (after the RELOC p-sect sort) at the specified address. For unmapped applications, /RO specifies a physical address; for mapped applications, /RO specifies a virtual address. If you use /RO with the /DS option, the starting address of the first p-sect of the program is set to addr regardless of its access attribute. Note that /DS is not recommended for general use.

If you are building an unmapped application and use /RO, RELOC does not look up start addresses in the memory image (.MIM) file. If your target system is ROM/RAM, you must use the /RW or /QB option in addition to /RO.

Note

In a mapped target system, any p-sect noncontiguous with other p-sects must start on a 4K-word virtual address boundary to satisfy mapping hardware requirements. In addition, for mapped applications, the .ALST. program section containing the static process or library list element must start on a 4K-word virtual address boundary—normally the first read-only p-sect unless you use the /DS option.

10.5.13 First RW Section at Specified Address (/RW)

The /RW:addr option starts the first read/write (RW) p-sect found in the input module—after the RELOC p-sect sort—at the specified address. For unmapped applications, /RW specifies a physical address; for mapped applications, /RW specifies a virtual address.

If you are building an unmapped application and use /RW, RELOC does not look up starting addresses in the input memory image (.MIM) file. In addition, the /RW option and the /AL or /DS options are mutually exclusive.

Note

In a mapped target system, any p-sect noncontiguous with other p-sects must start on a 4K-word virtual address boundary to satisfy mapping hardware requirements. If /ID is specified, /RW refers to instruction space.

10.5.14 Short Map (/SH)

The /SH option produces a "short" load map by excluding the kernel symbol definitions contained in the . ABS. p-sect from the map listing. By default, the map file includes the kernel symbol definitions. The numerous kernel symbols (roughly 700) are ordinarily of no direct use for debugging. If you omit the map file specification, the /SH option is ignored.

10.5.15 Supervisor-Mode Shared Library (/SL)

The /SL option builds the output file as a supervisor-mode shared library. Your target system must have supervisor mode, I/D space hardware support, and a kernel built with J11 mapping to use a supervisor-mode shared library. When you use /SL, RELOC separates I- and D-space program sections, includes the library list element and supervisor-mode dispatcher in the output file, checks for read/write program sections, and creates a special .STB file if a .STB file is specified. RELOC reads in the library list element and the supervisor-mode dispatcher from LIBSUP.OBJ. (See Section 6.2.4 for the file specification.)

If you specify the /DE option with the /SL option, RELOC includes ISD records in the output .STB file so that PASDBG will recognize it as a supervisor-mode shared library.

You cannot use /DR, /DW, /QB, /RO, or /RW with the /SL option.

10.5.16 Build User-Mode Shared Library (/UL[:addr])

The /UL[:addr] option builds a user-mode shared library. If the option includes a value for addr, RELOC builds an absolute rather than a relocatable library; addr specifies the base virtual address for the absolute library. When you use /UL, RELOC includes the library list element in the output file, checks for read/write program sections, and creates a special .STB file if one is specified. RELOC reads in the library list element from LIBUSR.OBJ. (See Section 6.2.5 for the file specification.)

10.5.17 Round Up Section Size (/UP)

The /UP:name:n option rounds up the length of the named p-sect so that the next free address in the process image is a whole-number multiple of n. The following p-sect will normally start at that address boundary. The specified boundary value must be a power of 2, and you can round only one p-sect for each process. The /UP option is useful if you want to include patch space in the application or align a following program or data section on a 32(decimal)-word or other power-of-2 boundary.

If you are building an unmapped application and use /UP, RELOC does not look up starting addresses in the input memory image (.MIM) file.

10.5.18 Program Version Number (/VR:xxx)

The /VR:xxx option lets you override the program version number or other program identification value for the module being relocated. This option argument can consist of up to six RAD50 characters. The specified value appears in the load map.

If you do not use the /VR option, RELOC uses the version number found in the program identification GSD record of the input module.

10.5.19 Wide Map (/WI)

The /WI option produces a memory map with six columns rather than the default three columns of global symbol names and values. The default 3-column format is intended for video terminal display of the map file. The 6-column map may be more convenient for line printer listings. If you omit the map file specification, the /WI option is ignored.

10.5.20 Value of Undefined Locations (/ZR:nnn)

The /ZR:nnn option lets you set the value of undefined locations in the process image. The n argument specifies the octal value that you want each undefined location to have. RELOC sets the value of undefined locations to 0 by default. Use /ZR if you want to set those locations to some other value.

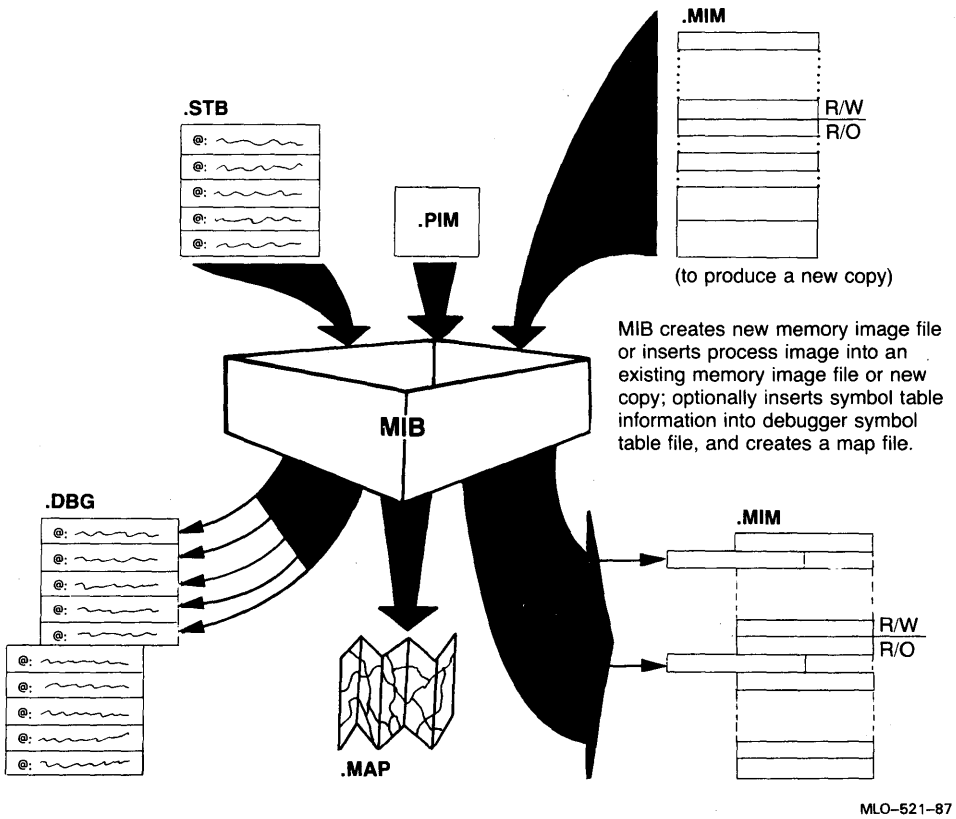
Note

The /ZR option sets only the values of undefined locations and does not modify locations defined by .WORD or .BYTE directives. Locations in the image reserved by .BLKW or .BLKB directives, however, and locations implied by use of the /EX or /UP option are affected by /ZR.

Chapter 11
The MIB Utility Program

The MIB utility program constructs a file that contains a memory image of a MicroPower/Pascal application. MIB can create and initialize an entirely new memory image file, or it can modify an existing memory image file. MIB can also create a new debug symbol file or modify an existing debug symbol file. You need the debug symbol file when you use the PASDBG symbolic debugger. See Figure 11–1.

Figure 11–1: MIB Utility Input and Output



This chapter discusses the following topics:

- MIB's functions
- MIB's role in the build cycle
- The invocation and use of MIB
- The interaction between RELOC and MIB
- The listing file that MIB optionally produces

You can run MIB yourself, but you will probably be able to use the automated build procedure MPBUILD instead (described in Chapter 2).

11.1 Functions of MIB

The MIB utility program creates the memory image file, installs and removes bootstraps, installs static processes and shared libraries, creates the debug symbol file and installs debug symbols, and creates map files. MIB can also perform some operations that let you avoid rebuilding the application. For example, you can use MIB when you need to change a process's start-up priority or change a process's exception group code.

11.1.1 Creating a Memory Image File

When you use the /KI option, the MIB utility creates and initializes a new memory image (.MIM) file and installs a kernel executable image. MIB constructs the memory image file in one of the following three formats:

- PASDBG load format: RAM-only memory image, no bootstrap in .MIM file, debugger service module (DSM) included in the kernel for "load and debug" or not included for "load and go" (LOAD/EXIT)
- Bootstrap load format: RAM-only memory image, appropriate bootstrap in the .MIM file, no DSM in the kernel
- PROM programmer format: ROM/RAM memory image, no bootstrap in .MIM file, no DSM in the kernel

The parameters specified in the MEMORY and SYSTEM macros of the configuration file used to build the kernel define the type of memory image finally constructed.

You need a memory image file in PASDBG load format if you intend to use PASDBG either to load and debug your application or to load your application for independent execution. In either case, do not install a bootstrap in the memory image file. A bootstrap is unnecessary because PASDBG uses the host-resident TD bootstrap to down-line load the image.

You need a memory image file in bootstrap load format if you intend to boot and load the application image from a target system disk or a TU58 device.

You need a memory image file in PROM programmer format if you intend to place the application in PROM chips.

11.1.2 PASDBG Load Format

A .MIM file in PASDBG load format (RAM-only) contains a memory allocation table and a compressed memory image. For all installed components, the image usually contains all read-only segments but, for mapped applications only, can contain some read/write segments. The file must not contain a bootstrap. The memory allocation table provides information that PASDBG uses to load program segments into target memory. The table is two blocks long; the used portion of the table ends with a –1, and the remainder is zero-filled. A bit in the attributes byte in the memory allocation table header indicates whether the memory image is for a mapped or an unmapped target; in addition, another bit indicates that the image is RAM-only (always so for PASDBG). The memory allocation table header also contains the address of the kernel/MIB communication area, a portion of the kernel containing information needed by both MIB and the kernel. Figure 11–2 illustrates both the PASDBG and bootstrap load formats.

11.1.3 Bootstrap Load Format

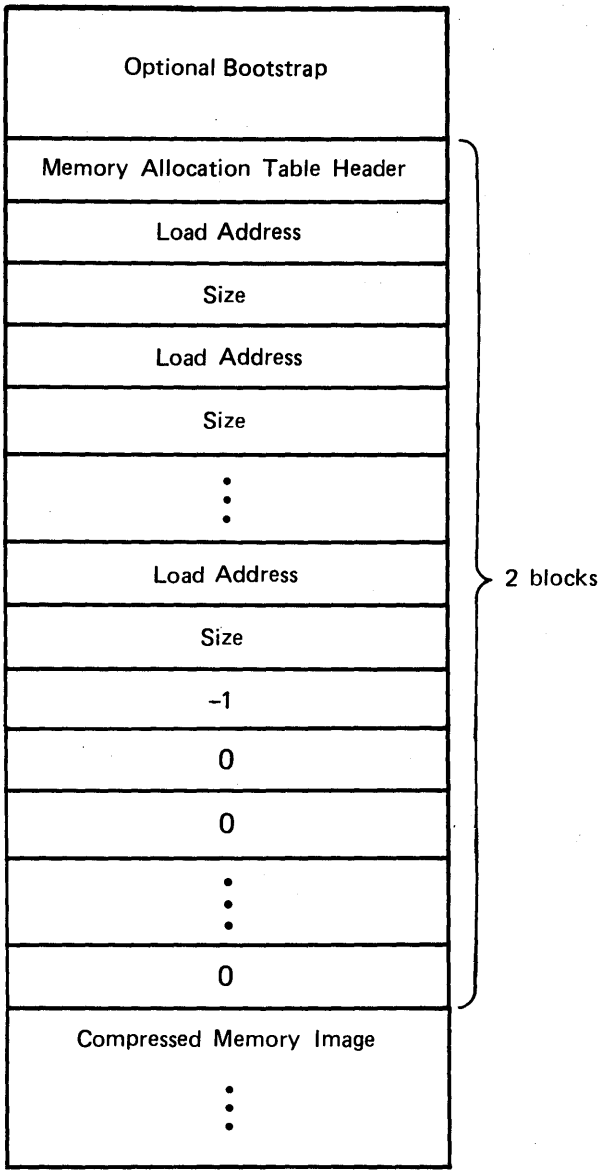
A .MIM file in bootstrap load format (RAM-only) is identical to PASDBG load format except that it contains a bootstrap at the beginning of the file. (See Figure 11–2.) DIGITAL supplies bootstraps for all disk or disk-like devices supported on a target system. When you specify the appropriate bootstrap file for your system with the MIB /BS option (Section 11.4.1), MIB installs the bootstrap at the front of the .MIM file.

After you have built a complete memory image, you copy the .MIM file onto a suitable storage volume—one matching the type of bootstrap installed—using the FLX or EXCHANGE utility program, and then use the MicroPower/Pascal COPYB utility to make the volume bootable from a device on your target system. See Chapter 12 for further details.

You can install a bootstrap either when you create the .MIM file in the kernel-build step or at the end of a build cycle. The latter strategy is convenient if you think you might want the same memory image to be bootable from several devices. If you build a complete .MIM file with no bootstrap installed, you can then create copies of it with different bootstraps, prefixing the bootstrap appropriate for the desired boot device.

Alternatively, you can delete an existing bootstrap from a .MIM file and install another bootstrap in its place. You can remove a bootstrap by means of the /RB option.

Figure 11–2: PASDBG or Bootstrap Load Format .MIM File



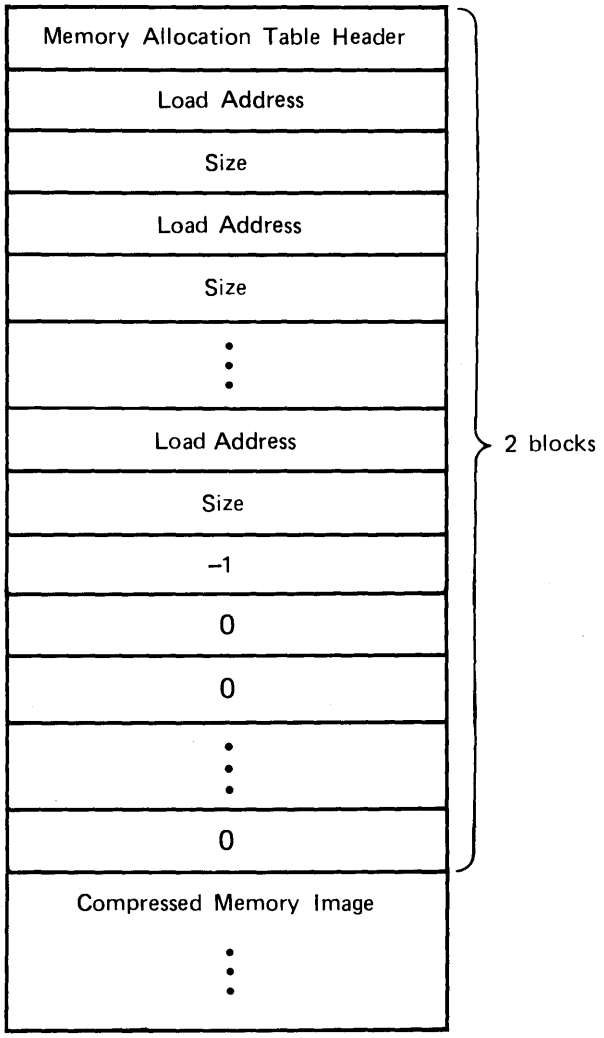
11.1.4 PROM Programmer Format

The PROM programmer .MIM file format (ROM/RAM) differs from the PASDBG and bootstrap load format files in that the PROM file's memory image contains only ROM (read-only) memory segments. (See Figure 11–3.) No space is allocated in the file for read/write segment text. A bit in the attributes byte in the memory allocation table header indicates that the image is ROM/RAM. The memory allocation table has entries only for the read-only segments. The file must not contain a bootstrap. The memory allocation table provides the information needed by

the utility program that will subsequently be used to control the "PROM blasting" process, such as VAX DECprom on a VAX/VMS host system.

Figure 11–3 shows a PROM programmer format memory image file.

Figure 11–3: PROM Programmer Format .MIM File



MLO–517–87

11.1.5 Installing Static Processes

After you use the /KI option to create a memory image file containing a kernel, you use MIB to install static-process images, one at a time, in that memory image. Ordinarily, you will probably want to use the /SM option and specify both an input and an output .MIM file in the successive MIB steps, as explained in the description of the /SM (small image) option in Sections 11.2 and

11.4.7. MIB links the static processes into the kernel's static-process list, updates the memory allocation table, and removes the memory used by the process from the kernel's free-memory list. If the process references any shared libraries, MIB ensures that the shared libraries are already installed in the image and links the process to the shared libraries. MIB can also install debug symbol information for a given process in the optional debug symbol file.

11.1.6 Installing Shared Libraries

If you use shared libraries in your application, you use MIB to install them, one at a time, in the memory image. You must install the shared library before you install any process that references the shared library. MIB links the shared library into the kernel's library list, updates the memory allocation table, and removes the memory used by the library from the kernel's free-memory list. MIB can also install debug information for a given shared library in the optional debug file.

11.1.7 Installing a Bootstrap

The MIB utility can install an appropriate bootstrap in the memory image file. You specify the desired bootstrap file with the /BS option. You must specify both an input and an output .MIM file unless you install the bootstrap in the kernel installation step using both the /KI and /BS options.

11.1.8 Removing a Bootstrap

Use the /RB option to remove a bootstrap from a .MIM file. You must specify both an input and an output .MIM file.

11.1.9 Creating a Map File

MIB can create a map file that shows the location and extent of the kernel, the installed process images and their attributes, and the layout of remaining available memory. The map also reports the type and extent(s) of target memory as it is described in the system configuration file. A memory map can be created as a separate MIB operation, if desired, or can be created in combination with another operation. Section 11.5 describes the MIB map.

11.1.10 Initializing the Debug Symbol File

If you specify a .DBG output file when you create the .MIM file containing the kernel, MIB creates and initializes the debug symbol file and places the kernel symbols in it.

The debug symbol (.DBG) file is an image-mode file in a special tree-structured format. The symbolic debugger, PASDBG, uses the information in this file to find and interpret the locations and structures you specify symbolically during debugging operations. When you invoke PASDBG to down-line load and debug a memory image from the host development system, the debugger loads all or part of the debug file into host memory as needed.

If you do want .DBG file output from MIB, you must include a kernel symbol table file (.STB) as input in the kernel build step. The kernel .STB file must contain debug symbol information (ISD records) as well as the normal global-symbol definitions (GSD records) for the kernel. RELOC produces the kernel .STB file, from which MIB produces the initial portion of the .DBG file.

The /DE option must be used in both the MERGE and RELOC steps for the kernel.

11.1.11 Installing Debug Symbols for a Static Process or Shared Library

MIB processes the optional .STB file generated by RELOC for each relocated process or shared library to format the debug symbol information specific to that process and add it to the .DBG file.

Here again, the /DE option must be used in the MERGE and RELOC steps for the process in question and also in the compilation step in the case of a Pascal process. In addition, a .STB file must be generated in the RELOC step.

11.2 Role of MIB in the Build Cycle

You use MIB repeatedly to install each component needed in your target memory image (see Figure 11–4). After merging the configuration file with PAXU.OLB or PAXM.OLB and relocating the kernel object file, you use MIB to initialize a memory image file and to install the kernel in it. Prior to these operations, you will have specified the hardware and software characteristics of the application in the source configuration file. Therefore, the kernel will contain the kernel primitives and other features appropriate for your application, and the size and type of memory recorded in the kernel's hardware configuration tables will be as described for your target system.

You use MIB again each time you add a system or user static process to your memory image file. In this case, the input to MIB is the process image (.PIM) file RELOC produces from the merged static-process object module, plus an input .MIM file if the /SM option was used in the previous MIB step. MIB installs the static process in the output .MIM file in the format necessary for the type of image being constructed—mapped or unmapped, and RAM-only or ROM/RAM. You also use MIB each time you add a shared library to your memory image file. A referenced library must be installed before you build any process that references that library.

You can begin the MIB sequence by using the small-image option (/SM) together with the /KI option to create a .MIM file that is only as large as needed to contain the kernel image. Alternatively, you can omit /SM and create a full-size .MIM file at the very beginning, in which case the size of the initial .MIM file reflects the full extent of target memory specified in the kernel's configuration tables. This allows you to modify the file in place—specifying only an output .MIM file—as you add the processes your application requires. In some instances, this strategy can be wasteful of file space, and you do not have the security of back-up versions of the .MIM file; MIB does not create a new generation of the .MIM file if you specify only an output .MIM file for updating in place.

If you choose to create a full-size .MIM file, omitting the /SM option, specify only the existing .MIM file as output in your subsequent commands to MIB. Each time you add a process, it will go into the existing file in space already available for it.

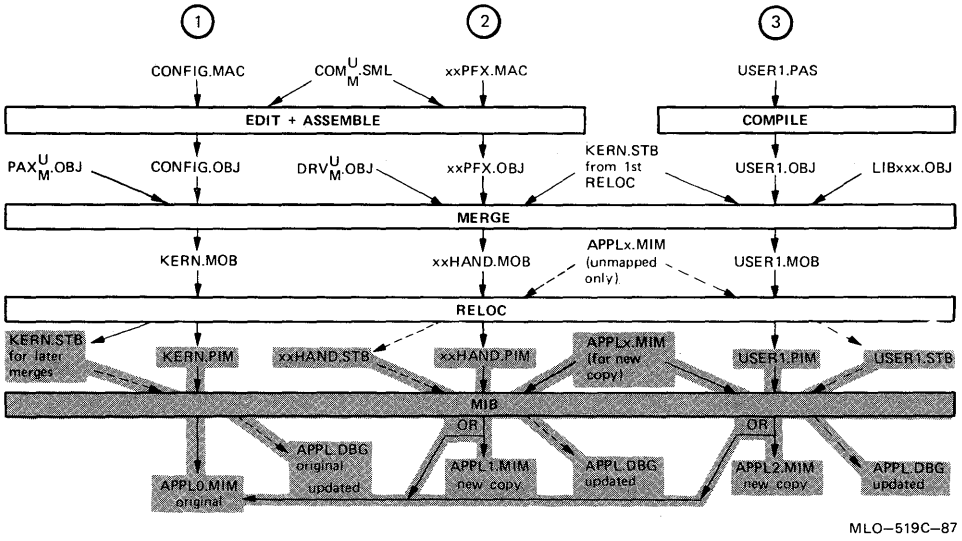
If you use /SM, however, MIB creates a new output .MIM file only as large as necessary to hold the kernel and any installed processes, thereby saving space on your development volume in the case of a large target memory. Each time you add a process and use /SM, you must specify both an input and an output .MIM file in your MIB command line. MIB copies the existing .MIM file contents to the new output file and extends the output file to accommodate the process being installed.

To avoid confusion, you should be consistent in using /SM—either use it all the time or omit it completely. Whether you use /SM or not, if you specify an input and an output .MIM file with identical file names, the system will generate a new version of the file. You can delete the older versions of the file as desired. The optional .DBG file, however, is always updated in place—the existing file is extended—and a new version is never created. That is, you can specify only an output .DBG file, which must be created in a kernel installation step. You may want to make a copy of the .DBG file yourself at an intermediate point in the cycle, for possible use in subsequent rebuilds. At any point in the build cycle, the .DBG file is only as large as required to hold the installed debug symbols.

Often it is desirable to save the .MIM file, .STB file, and .DBG file, which contain just the kernel and the system processes. In such a case, make a copy of those files before adding any user processes. That way, you can rebuild just from that point while debugging, instead of from the beginning.

Chapter 1 provides an overview of the entire build cycle.

Figure 11–4: MIB's Part in the Build Cycle



MLO–519C–87

11.3 Invocation and Use of MIB

For an RSX Development System:

Assuming that MIB has been installed according to installation procedure defaults, you invoke it by the task name MIB, as follows:

```
>[MCR] MIB
```

(Precede "MIB" with "MCR" only if your CLI mode is DCL.) The following three standard RSX forms of direct invocation may be used:

- **> [MCR] MIB command-line**
- **> [MCR] MIB @command-file**

The specified .CMD file contains one or more MIB command lines.

- **> [MCR] MIB**

  MIB> **command-line** or **@command-file**
  MIB>

The format of the MIB command line is described below. Only the first two forms of MIB invocation can be used within a command file. The first form limits the line to 80 characters and precludes the use of continuation lines. The second and third forms can be used to issue several MIB command lines within one invocation or to avoid the 80-character limit. Type CTRL/Z in response to the MIB> prompt to exit.

For a VAX Development System:

If you have executed the MPSETUP.COM procedure (Section 1.4), you can invoke MIB by the logical symbol MPMIB, as follows:

```
$ MPMIB
MIB>command-line  or  @command-file
MIB>
```

The format of the MIB command line is described below. The default type for an indirect command file is .CMD; the file may contain one or more MIB command lines. Type CTRL/Z in response to the MIB> prompt to exit.

Command Line Format

MIB accepts a command line in the form shown below. In the command line, all file specifications are in standard RSX format with respect to device and directory (UFD) information if you are using an RSX host system. If you are using a VAX/VMS host, however, all file specifications are in standard VMS format with respect to device and directory information.

Output files appear on the left side of the equal sign (=), and input files appear on the right. Brackets ([ ]) indicate optional fields of the command. When you omit an optional output file specification, indicate the null field with consecutive commas to "hold its place," except in the case of trailing fields. Trailing commas can be omitted. You must specify at least one output file for any given MIB operation.

Note

For Version 4.0 or later of VMS, the file name field of a file specification must not exceed nine characters, and the file type field must not exceed three characters.

For Version 4.0 or later of VMS, underscores (_) are not valid in file specifications.

For all versions of VMS, dollar signs ($) are not valid in file specifications.

```
[outmim],[mapfile][,dbgfile]=[pimfile],[inputmim][,stbfile][/options]
```

outmim

The file specification for the output memory image file to be created, extended, or modified by the MIB operation. The default file type is .MIM (memory image). If an input memory image file is also specified in the command line, MIB opens a new .MIM file for output. If neither an input memory image file nor the /KI option is specified, MIB attempts to open an existing file identified as "outmim" for updating. If you include the /SM option but not the /KI option, both

input and output .MIM files are required. The presence or absence of an outmim specification, along with option specifications, determines in part the type of MIB operation to be performed.

mapfile

The file specification of the optional memory image map file, which describes each installed component in the current memory image—the kernel and any processes—and describes the target memory layout. The default file type is .MAP. You can ask for a map file whenever you also specify an output or an input .MIM file.

dbgfile

The file specification for the optional debug symbol file, which contains the symbol tree that MIB links together from ISD-record input from individual .STB files. The default file type is .DBG. Specify this field whenever you want an operation performed on a .DBG file: file initialization or addition or deletion of symbols for a process. (Those operations can be performed either separately or in combination with the corresponding operations on the .MIM file.) If you do specify dbgfile and are either initializing the file or adding symbols to it, you must also specify the appropriate symbol table file (.STB) created by RELOC as input. MIB creates and initializes a new debug symbol file when it encounters both the dbgfile specification and the /KI (kernel installation) option; otherwise, it opens the existing dbgfile for updating—either addition or deletion of symbols as determined by other elements of the command line.

pimfile

The file specification for the .PIM file containing the kernel or process image to be installed in the memory image. The default file type is .PIM. If you specify a .PIM file, indicating a kernel/process installation, you must also specify at least one output .MIM file. The presence or absence of a pimfile specification, along with option specifications, determines in part the type of MIB operation to be performed.

This field is invalid if you also specify the /EX option.

inputmim

The file specification for the input memory image file. The default file type is .MIM. This field is required for a process installation if the current memory image file contains a "short image"—was constructed with the /SM option in the previous MIB step—or if you specify the /SM option for the current operation. Otherwise, this field is optional; see description of outmim field.

stbfile

The file specification for the symbol table file, which contains symbol information for the debugger—ISD records—if generated with the /DE option in the corresponding MERGE and RELOC steps. The default file type is .STB. Specify this field only if you are installing symbols in an output .DBG file; the dbgfile field must also be specified.

/options

Any of the options summarized in Table 11–1. All MIB options are position independent. Multiple options can be specified in a list of the following form:

```
/option1/option2/...
```

11.4 MIB Options

This section describes the MIB options that are summarized in Table 11–1.

Table 11–1: MIB Options

Option	Meaning
/BS:"file-spec"	Installs a bootstrap at the beginning of the memory image file; the bootstrap is copied from the specified file. Input and output .MIM files are required.
/GC:name:code	Changes the exception group code of a static process installed in the memory image.
/KI	Identifies the input .PIM file as a kernel image and signals MIB to create and initialize the output memory image file and, optionally, the output .DBG file. You cannot use this option together with /QB.
/PR:name:value	Changes the start-up priority of a static process installed in the memory image.
/QB:name:block[:...]	Aligns named p-sects at specified physical locations in the memory image during a process installation step; valid for mapped applications only. You cannot use this option together with /KI.
/RB	Removes the bootstrap from a .MIM file. Input and output .MIM files are required.
/SM	Limits the output memory image file to the exact size of the kernel, the installed static processes, and the bootstrap, if any.

11.4.1 Install Bootstrap (/BS)

The /BS:"file-spec" option installs a bootstrap at the beginning of the memory image file. The argument file-spec identifies the bootstrap file from which the bootstrap is to be copied. The file specification must be enclosed in double quotes if it contains either a colon (:) or a comma (,). The default type for the bootstrap file is .BOT.

The DIGITAL-supplied bootstrap files, normally residing in the RSX directory MP:[2,10] or the VMS directory defined by MICROPOWER$LIB, include the following:

File Name	Boot Medium/Target Type
DDBOTM.BOT	TU58 DECtape II, mapped target
DDBOTU.BOT	TU58 DECtape II, unmapped target
DLBOTM.BOT	RL01/RL02 disk, mapped target
DLBOTU.BOT	RL01/RL02 disk, unmapped target
DUBOTM.BOT	MSCP-class disk, mapped target
DUBOTU.BOT	MSCP-class disk, unmapped target
DYBOTM.BOT	RX01/RX02 diskette, mapped target
DUBOTU.BOT	RX01/RX02 diskette, unmapped target

(MSCP-class disks include the RX50 and RDxx devices.) The COPYB utility must process a .MIM file containing a bootstrap to make the volume bootable, as described in Chapter 12. See Chapter 7 for further discussion of bootable memory images.

You must specify both an input and an output .MIM file when you install a bootstrap unless you install it when you create the file in the kernel installation step.

If you install a bootstrap in your application image, you must be sure that certain target memory requirements are met. First, an unmapped target system must have at least 3584 (7000 octal) contiguous bytes of memory starting at location 0. A mapped target system must have at least 4096 (10000 octal) contiguous bytes starting at location 0. Normally, this requirement should never be a problem. Second, the highest 512 (1000 octal) contiguous bytes of memory on the target system must not be loaded by your application. Unless you deliberately place part of your application code or pure data at the very top of memory or you use almost all the memory on the target, this requirement should never be a problem either. You should, however, be aware of these bootstrap requirements.

11.4.2 Exception Group Code (/GC)

The /GC:name:nnn option changes the exception group code of a specified static process installed in the memory image. Every process has an exception group code that identifies the exception-handling group to which it belongs. The exception group establishes which exception handler—another process—should handle that process's exceptions. The /GC option allows you to change from one exception handler to another for a static process without rebuilding the application.

The value of n specifies the value to which the code should be changed. This value must be within the range 1 to 255 (1 to 377 octal). The specifed value is assumed by default to be expressed in octal. Terminate the number with a decimal point—for example, 40.—to indicate a decimal value.

The name can be up to six RAD50 characters. The name argument identifies the static process to be modified, which can be the process being installed by the operation or another, previously installed process.

The exception group of a dynamic process cannot be modified by this option.

11.4.3 Kernel Installation (/KI)

The /KI option indicates that the input .PIM file contains a kernel image rather than a process image. The /KI option signals MIB to create and initialize the output memory image (.MIM) file and, if requested, the debug symbol (.DBG) file also. An input .MIM file is invalid for a /KI operation, and a new output .MIM and an optional .DBG file are created whether or not a file by the same name already exists. Only the /SM and /BS options are valid or meaningful in combination with /KI. In particular, you cannot use this option together with /QB.

11.4.4 Process Priority (/PR)

The /PR:name:n option changes the start-up priority of a static process installed in the memory image to the value n. This option allows you to change process start-up priority without rebuilding the application. The /PR option affects only the INITIALIZE procedure of a static process implemented in Pascal; you must modify the source code to change the running priority of the main program.

The value of n must be within the range 1 to 255 (1 to 377 octal); priorities 248 through 255 are reserved for start-up of processes containing global initialization code. The specifed value is assumed by default to be expressed in octal. Terminate the number with a decimal point—for example, 150.—to indicate a decimal value.

The name argument identifies the static process to be modified, which can be the process being installed by the operation or another, previously installed process.

Note

If you change a Pascal process's start-up priority to less than 248 and the process contains an initialization procedure, that procedure may not execute in concert with other processes' initialization code at system start-up time. The INITIALIZE procedure attribute implies a start-up priority of 255 for that automatically called procedure, regardless of the running priority assigned to the main program (static process) by the PRIORITY attribute.

If the program does not contain an initialization procedure, the process starts up at its specified running priority, which is not affected by the /PR option.

11.4.5 Align Specified Program Section (/QB)

The /QB option aligns one or more named program sections at specified physical locations in the memory image for a mapped application. You cannot use the MIB /QB option for an unmapped image; the RELOC /QB option is its equivalent in the unmapped case. In addition, you cannot use this option together with /KI.

The format of the /QB option is as follows:

```
/QB:name1:block[:name2:block ...]
```

In this format, "name-i" is a p-sect name, and block is the physical starting location for that section in terms of 32-word—100(octal)-byte—increments from address 0.

You can specify up to eight section names and block offsets. Each section specified in the MIB /QB option must have been assigned a 4K-word virtual base address by RELOC. Thus, each specified section implies the beginning of a noncontiguous memory segment and the use of a new PAR. The starting address for the section must be a multiple of 100(octal) bytes. The block value in the option, effectively a "memory block" number, specifies this multiple. For example, to indicate the octal starting address 63000, you specify 630 as the block value (63000/100). The value is assumed to be expressed in octal unless terminated by a decimal point.

To load an entire static process starting at a desired physical address, specify a start location for the program section .ALST.; that section contains the static-process list element and is usually the first read-only section in the process. Thus, .ALST. indicates the beginning of the first or only RO segment of the process. To load the read/write portion of a Pascal static process at a desired physical address, specify a start location for .CDAT., the first read/write section for

static processes written in Pascal. Thus, .CDAT. indicates the beginning of the first or only RW segment of a Pascal-implemented process.

In a RAM-only memory image, the RO and RW segments of a process are not normally separated from each other, as they must be in a ROM/RAM image. That is, by default the end of the code and pure-data portion and the beginning of the impure-data portion of a process are physically contiguous and are mapped by the same PAR whenever possible in the RAM-only case.

11.4.6 Remove Bootstrap (/RB)

The /RB option removes the bootstrap from a .MIM file. Input and output .MIM files are required.

11.4.7 Small Output Memory Image (/SM)

The /SM option limits the output memory image file to the exact size of the kernel and the installed static processes. If you do not specify /SM, MIB constructs or updates a memory image file that is as large as the total amount of physical memory you specified for the target system in the configuration file used to build the kernel. (Information about the memory size and type is retained in a kernel/MIB communication area that is part of the kernel; therefore, this information is always available to MIB.) If you specify /SM for a process installation step (/KI not specified), you must specify an input .MIM file—either "small" or full size—as well as an output .MIM file. The /SM option creates an output .MIM file that is just big enough for the existing memory image from the input .MIM file plus the process image from the input .PIM file. If you use /SM in combination with /KI, MIB creates an output .MIM file just big enough to accommodate the kernel.

A full-size .MIM file can be updated in place for process installations. An existing "small" .MIM file cannot be so modified.

11.5 MIB Memory Map

If you specify mapfile in the command line, MIB creates a memory image map showing the following:

❶. MIB version number

❷. Date and time of operation

❸. Target memory type

❹. Boot device, if any

❺. Memory layout—physical memory present (base, size, and type) and available ROM and/or RAM in image (base and size)

❻. Kernel information—physical starting address and kernel segments (base and size)

❼. Shared-library information—library name and type, segment starting addresses and sizes

❽. Static-process information—process name and priority, process type and group, physical starting and termination addresses, stack address, segment starting addresses and sizes, and information about any referenced libraries

For unmapped applications, starting addresses are direct byte values in octal. For mapped applications, virtual and physical starting addresses are given. For both mapped and unmapped applications, sizes are given in decimal as well as in octal.

Figures 11–5 through 11–8 show various types of MIB memory maps.

Figure 11–5: Sample Unmapped MIB Memory Map with No I/D Separation or Shared Libraries

```
①MICROPOWER MIB V02.00 VMS        Application Map        ②Fri 17-May-85 15:41:29
③MIM Filename -- TE103.MIM;2      Unmapped Ram-Only System      ④Boot Device: DY

⑤Physical Memory Segments

         Start   Size    Type
         000000  100000  Ram

  Available Ram

         Start   Size
         064732  013046


⑥Kernel Segments

  Segment Physical Start: 000000 Size: 026156
  Segment Physical Start: 026156 Size: 006222
  Kernel Start Address: 020754

⑧Static Process Information

  Name  Priority      Type     Cntxsw    Group     Term Add. Stack  Start

  $TTDRV  250     Drv-Access              1         035666   042670 034476
          Segment Physical Start: 034400 Size: 005376
          Segment Physical Start: 041776 Size: 000674

  EXAMPL  248     General    MCX          1         050352   062614 044372
          Context Switch Location: 062706 Initial Value: 062616
          Segment Physical Start: 042672 Size: 017276
          Segment Physical Start: 062170 Size: 002542
```

Figure 11–6: Sample Mapped MIB Memory Map with No I/D Separation or Shared Libraries

```
(1)MICROPOWER MIB V02.00 VMS        Application Map        (2)Sun 03-Mar-85 13:24:28
(3)MIM Filename -- TE003.MIM;1      Mapped Ram-Only System

(5)Physical Memory Segments

          Start   Size    Type
          000000  002000  Ram

  Available Ram

          Start   Size
          000770  001010


(6)Kernel Segments

  Segment Virtual Start: 000000 Physical Start: 000000 Size: 000200
  Segment Virtual Start: 000200 Physical Start: 000200 Size: 000143
  Segment Virtual Start: 001000 Physical Start: 000343 Size: 000110
  Kernel Start Address: 026434

(8)Static Process Information

  Name  Priority      Type     Cntxsw    Group     Term Add. Stack  Start

  $TTDRV  250     Drv-Access             1         041266   060604 040076
          Segment Virtual Start: 000400 Physical Start: 000453 Size: 000060
          Segment Virtual Start: 000600 Physical Start: 000533 Size: 000007

  EXAMPL  248     General    MCX         1         006104   020422 001736
          Context Switch Location: 020514 Initial Value: 020424
          Segment Virtual Start: 000000 Physical Start: 000542 Size: 000200
          Segment Virtual Start: 000200 Physical Start: 000742 Size: 000026
```

Figure 11–7: Sample MIB Memory Map with I/D Separation

```
(1)MICROPOWER MIB V02.00 VMS        Application Map        (2)Sun 03-Mar-85 14:16:39
(3)MIM Filename -- TE403.MIM;2      Mapped Ram-Only System
                                    J11 Mapping

(5)Physical Memory Segments

          Start   Size    Type
          000000  002000  Ram

  Available Ram

          Start   Size
          000773  001005


(6)Kernel Segments

  Segment Virtual Start: 000000 Physical Start: 000000 Size: 000200
  Segment Virtual Start: 000200 Physical Start: 000200 Size: 000152
  Segment Virtual Start: 001000 Physical Start: 000352 Size: 000110
  Kernel Start Address: 027330

(8)Static Process Information

  Name  Priority      Type     Cntxsw    Group     Term Add. Stack  Start

  $TTDRV  250     Drv-Access              1         041266   060604 040076
          Segment Virtual Start: 000400 Physical Start: 000462 Size: 000060
          Segment Virtual Start: 000600 Physical Start: 000542 Size: 000007

  EXAMPL  248     General    MCX          1         006226   002462 001776
          Context Switch Location: 002554 Initial Value: 002464
          ID process

          I-Space
          Segment Virtual Start: 000000 Physical Start: 000551 Size: 000154

          D-Space
          Segment Virtual Start: 000000 Physical Start: 000725 Size: 000046
```

Figure 11–8: Sample MIB Memory Map with a Shared Library

```
①MICROPOWER MIB V02.00 VMS        Application Map        ②Sun 03-Mar-85 13:26:34
③MIM Filename -- TE602.MIM;1      Mapped Ram-Only System

⑤Physical Memory Segments

          Start   Size    Type
          000000  002000  Ram

  Available Ram

          Start   Size
          000771  001007


⑥Kernel Segments

  Segment Virtual Start: 000000 Physical Start: 000000 Size: 000200
  Segment Virtual Start: 000200 Physical Start: 000200 Size: 000143
  Segment Virtual Start: 001000 Physical Start: 000343 Size: 000110
  Kernel Start Address: 026434

⑦Library Information

  Name    Type

  $USRLB  User-mode
          Segment Virtual Start: 000000 Physical Start: 000542 Size: 000077

⑧Static Process Information

  Name  Priority      Type    Cntxsw    Group     Term Add. Stack  Start

  $TTDRV  250     Drv-Access             1         041266   060604 040076
          Segment Virtual Start: 000400 Physical Start: 000453 Size: 000060
          Segment Virtual Start: 000600 Physical Start: 000533 Size: 000007

  EXAMPL  248     General    MCX         1         006104   010606 001736
          Context Switch Location: 010700 Initial Value: 010610
          Segment Virtual Start: 000000 Physical Start: 000641 Size: 000130

          References user-mode library $USRLB
          Segment Virtual Start: 000200 Physical Start: 000542 Size: 000077
```

Chapter 12
Making a Volume Bootable on the Target

The COPYB utility program prepares an RT-11-format storage volume containing a MicroPower/Pascal application .MIM file with bootstrap for use on a target system boot device.

This chapter discusses:

- COPYB's functions
- Invoking and using COPYB
- COPBOT's use

12.1 Functions of COPYB

After you have developed and debugged a MicroPower/Pascal application, you can boot the application image into a target system from one of several types of mass-storage devices. The host system configuration must include the type of device intended to be used as the boot device on the target system. If both the host and target hardware configurations include a TU58 device, for example, the application can be bootstrapped from a DECtape II cartridge. Similarly, if they both include RX02 or RX50 diskette devices or RL01/RL02 disks, the application can be bootstrapped from a diskette or hard disk.

The boot-device volume is prepared on the host system by means of the COPYB utility. The volume must be in RT-11 file system format, which involves use of the RSXFLX utility or the VMS V4 EXCHANGE utility, as well as COPYB.

If you intend to use this method to load the application, you have to perform the following steps:

1. Build the application memory image (.MIM) file without debug support and with an appropriate bootstrap installed in the file, using either MPBUILD or the /BS option of the MIB utility program.
2. Copy the memory image file to the mass-storage volume that you want to be bootable. (Under Micro/RSX, RSX-11M-PLUS, or VAX/VMS, the volume is mounted as a foreign volume on the host system's device drive; under RSX-11M, the volume is "unmounted.") Use the FLX utility program or the VMS V4 EXCHANGE utility to initialize the volume in

RT–11 format, if necessary, and to perform the .MIM file copy operation in image mode. If the volume is already in RT–11 format and contains other files, it need not be reinitialized.

3. Invoke COPYB to process the bootstrap, making the volume bootable.
4. Mount the bootable volume in the target system's device drive and power up the target processor.

During its bootstrap processing, COPYB performs the following operations:

- Reads the first block of the bootstrap—called the primary software bootstrap—contained in the .MIM file
- Modifies a word in the primary software bootstrap to reflect the location of the second block of the .MIM file on the volume
- Writes the modified primary software bootstrap into logical block 0 (the boot block) of the volume

When you power up the target processor, the primary hardware bootstrap, located in ROM on the target processor, reads block 0 of the volume into memory. The bootstrap in block 0 initiates loading of the application memory image from the volume into the target memory.

12.2 Invoking COPYB

For an RSX Development System: Assuming that COPYB has been installed according to installation procedure defaults, you invoke it by the task name CPB, as follows:

```
>[MCR] CPB
CPB>
```

Precede "CPB" with "MCR" only if your CLI mode is DCL.

For a VMS Development System: If you have executed the MPSETUP.COM procedure (Section 1.4), you can invoke COPYB by the logical symbol MPCOPYB, as follows:

```
$ MPCOPYB
CPB>
```

(Use CTRL/Z to exit from COPYB.)

In response to its prompt for input, COPYB accepts a command line of the following form:

```
CPB>dev:filename[.typ][/CS:nnnnnn]
```

dev:

A device specification identifying the drive unit in which the RT–11-format volume is loaded. (The volume must be mounted as FOREIGN under Micro/RSX, RSX–11M–PLUS, or VAX/VMS or must be unmounted under RSX–11M.)

filename[.typ]

The name of the memory image file that was previously copied onto the volume and, optionally, the file type. The default file type is .MIM. The file name is limited to six characters.

/CS:nnnnnn

Optionally, the address of the boot-device CSR (control and status register) on the target system, if the CSR is not at the standard address for an LSI–11 processor. (This option may be needed for a FALCON or FALCON–PLUS target system, for example.)

When you use COPYB, the memory image file must already reside on the volume that is to be made bootable. This implies prior use of the FLX utility or the VMS V4 EXCHANGE utility to perform the necessary .MIM file transfer and format conversion.

The bootstrap contained in the .MIM file assumes that the CSR for the boot device on the target system is configured at the standard LSI–11 bus address. You can use the /CS option, if necessary, to cause COPYB to change that address in the modified primary bootstrap block before it is placed in block 0 of the volume.

The following examples illustrate use of the FLX or EXCHANGE utility to copy the .MIM file to an already initialized RT volume. The volume must be mounted as FOREIGN under Micro/RSX, RSX–11M–PLUS, or VAX/VMS or must be unmounted under RSX–11M.

Example 1: VMS V4

```
$EXCHANGE
EXCHANGE>COPY APPLC5.MIM DY0:/TRANSFERMODE:BLOCK
EXCHANGE>EXIT
```

Example 2: VMS prior to V4

```
$MCR FLX
FLX>DY0:/RT/IM=APPLC5.MIM
FLX><CTRL/Z>
```

Example 3: RSX MCR

```
>FLX
FLX>DY0:/RT/IM=APPLC5.MIM
FLX><CTRL/Z>
```

Example 4: RSX DCL

```
>MCR FLX
FLX>DY0:/RT/IM=APPLC5.MIM
FLX><CTRL/Z>
```

The FLX command line transfers the Files–11 APPL5.MIM file in your default device/directory to the RX02 diskette mounted in DY drive unit 0. The /RT option requests conversion of output to RT–11 file format. (The command assumes that the output volume is already in RT–11 format.) The /IM option requests an image-mode copy operation; that is, the file content is to be copied with no internal format modification.

The name of the .MIM file as copied to the volume by the FLX utility is restricted to six characters, the RT–11 limit for file names. Therefore, you may want to rename the host system copy of the .MIM file prior to the FLX operation. FLX truncates a longer file name to six characters without indicating that it has done so; COPYB also truncates a longer name but issues a warning message.

If you want to initialize the disk first, use one of the following commands before copying the file over:

```
EXCHANGE>INITIALIZE DY0:
(default volume format is RT11)
FLX>DY0:/RT/ZE

>[MCR] CPB  or  $ MPCOPYB
CPB>DY0:APPL5.MIM
```

The COPYB command line causes COPYB to look for file APPL5.MIM on the DY0: device. COPYB reads and modifies the primary software bootstrap from that file and writes it to block 0 of the diskette. (The standard target CSR address that the bootstrap uses for an RX02 device remains at 177170.)

```
>[MCR] CPB  or  $ MPCOPYB
CPB>DD1:APPL6/CS:176540
```

This command line causes COPYB to look for file APPL6.MIM on the DECtape II cartridge mounted in DD drive unit 1 and to copy the primary software bootstrap from that file to block 0 of the cartridge. The /CS option causes the target CSR address in the TU58 device bootstrap to be changed to 176540, which is correct for a FALCON or FALCON–PLUS SLU2 port; the LSI–11 default CSR address is 176500.

12.3 The Copy-Boot Program (COPBOT)

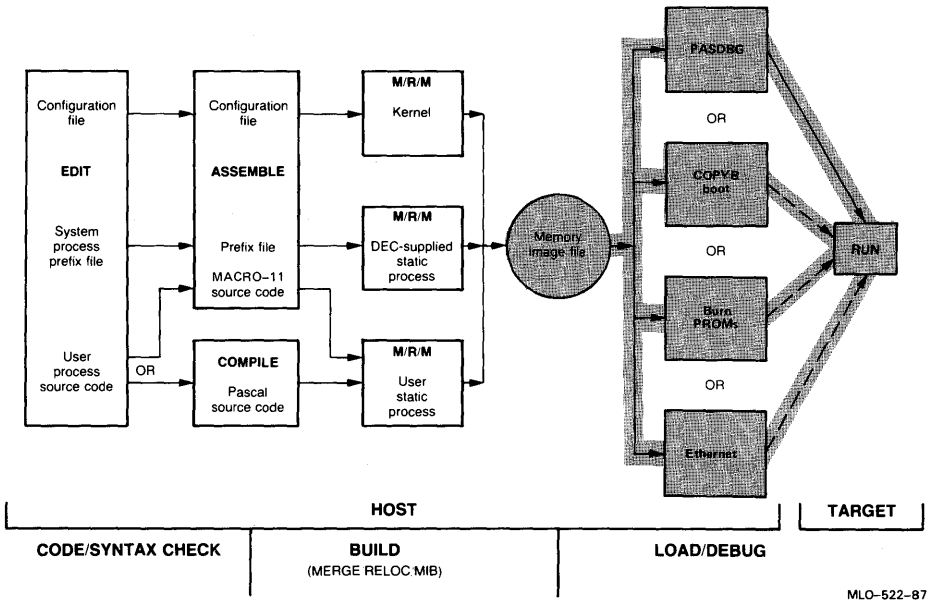
The source file, COPBOT.PAS, which is included in the kit, can be used for copying a .MIM file and performing the COPYB function directly on a target system. This technique is necessary for making a nonremovable target medium, such as an RD51 or RD52 Winchester disk, bootable. See the source file for a list of required components.

Chapter 13

DECNET Down-Line Loading (RSX or VMS Only)

Under MicroPower/Pascal–RSX or –VMS, an application image can be booted into a target system through use of the Ethernet or DDCMP, as an alternative to the methods of application loading described in Chapter 7. Figure 13–1 shows the several ways that an application can be loaded into a target system.

Figure 13–1: Application Image Loading



13.1 DECNET/Ethernet Down-Line Loading

The Ethernet down-line loading method pertains only to the final production image of the application. Loading for debugging is done with PASDBG.

DECnet/Ethernet down-line loading uses DECnet network facilities on the host system (RSX or VMS) and a down-line bootstrap loader residing on the target machine. Part of this loader resides in the firmware ROM on the target processor board and another part within the ROM of the DEQNA module.

Note

Refer to your target processor documentation on how to initiate the Ethernet down-line load procedure.

Together those software components use the Ethernet to copy a MicroPower/Pascal image file from the host system to the main memory of the target machine. Once the MicroPower/Pascal software is in the target memory, it gains control of the processor and begins executing. The MicroPower/Pascal image need not contain the Ethernet (QN) driver or the network service process (NSP) to be down-line loaded. You should, however, include those processes if the application requires network communication support at run time.

For the host operating system to provide the down-line load facility, it must be able to recognize a boot request message from the target system. Also, the MicroPower/Pascal target machine must be described in the host system's network node database.

The principal tool used to control the network for VMS or RSX is the Network Control Program (NCP). We recommend that you become familiar with the NCP utility. Please refer to the network manager's guide for your host system for information on NCP, including the way to invoke the utility and the privileges required.

Note

The Ethernet circuit is the name of the host system's hardware device controller, which is connected to the same Ethernet as the target machine. Depending on your configuration, the correct specification might be UNA–0 or QNA–0. Check your configuration to determine the proper circuit specification.

To enable the host to recognize a boot request from the Ethernet, the circuit must have service enabled:

```
NCP> SET CIRCUIT UNA-0 SERVICE ENABLED
```

To enter a MicroPower/Pascal target machine into the host system's network node database, you issue NCP commands to store the target machine's node name and address, Ethernet hardware address, and host circuit for loading. For example, for the node "UPOWER::":

```
NCP> SET NODE UPOWER ADDRESS 5.410
NCP> SET NODE UPOWER SERVICE CIRCUIT UNA-0
NCP> SET NODE UPOWER HARDWARE ADDRESS xx-xx-xx-xx-xx-xx
NCP> SET NODE UPOWER LOAD FILE file-specification    <- described below

NCP> SET NODE UPOWER SECONDARY LOADER MICROPOWER$LIB:SECQNA.SYS  <- VMS
NCP> SET NODE UPOWER TERTIARY LOADER MICROPOWER$LIB:TERQNA.SYS
  or
NCP> SET NODE UPOWER SECONDARY LOADER dev:[5,54]SECQNA.SYS       <- RSX
NCP> SET NODE UPOWER TERTIARY LOADER dev:[5,54]TERQNA.SYS
```

The node name and address may have already been specified when your network was installed. Make sure that each node in your network has a unique address and name.

The hardware address is required for down-line loading over the Ethernet. The hardware address is the physical address of the DEQNA on the target machine and is normally printed on the controller board. If the address is not printed on the board, you can determine the address with the following ODT sequence—assuming the DEQNA CSR is at location 774440, the factory configuration:

```
@774440/177652     (low 8 bits = 252 octal = AA hex)
@774442/177400     (              000          00    )
@774444/177403     (              003          03    )
@774446/177401     (              001          01    )
@774450/177432     (              032          1A    )
@774452/177706     (              306          C6    )
```

Here, the appropriate NCP SET NODE HARDWARE ADDRESS command would be:

```
NCP> SET NODE UPOWER HARDWARE ADDRESS AA-00-03-01-1A-C6
```

The NCP commands place the information about node UPOWER in the host system's volatile database. However, information in the volatile database must be reentered after each host system reload. To make that information part of the permanent database, use the DEFINE command rather than SET on VMS or use the CFE program rather than the NCP program on RSX.

The MicroPower/Pascal image (.MIM) file does not have an appropriate format for Ethernet down-line loading and must be converted to a format that the bootstrap loader will accept. You perform that conversion by running the MKBOOT (VMS) or MKB (RSX) program provided on the MicroPower/Pascal distribution kit. MKBOOT/MKB prompts for an input file name, and you respond with the .MIM file name. MKBOOT then prompts for an output file name, and you respond with the name of the loader format file, which has the .SYS file type. MKBOOT then converts the .MIM file into a formatted .SYS file.

After the image format conversion, you use the following command to specify the .SYS file as the image to be loaded into the target system's main memory:

```
NCP> SET NODE UPOWER LOAD FILE file-specification
```

After the down-line load is complete, the MicroPower/Pascal network services, if present in the target application, can take advantage of the information that the host system provided. In particular, the host system has informed the target system of the proper node number. If your application image is mapped, the NSP will use the host-supplied node number, so you need not specify the address information in the NSP prefix file, thus allowing the same MicroPower/Pascal application image to be run on multiple machines in a network. You enable those capabilities by specifying NETBOOT=YES to the SYSTEM macro in your kernel configuration file.

Once a MicroPower/Pascal application is initialized and is running, the Ethernet (QN) driver can respond to a host system request to reinitiate the Ethernet down-line load sequence. You enable that capability by specifying NETTRIGGER=YES to the SYSTEM macro in your kernel configuration file. You can then issue the following NCP command:

```
     NCP> TRIGGER NODE UPOWER                                    <- VMS
or
     NCP> TRIGGER NODE UPOWER PHYSICAL ADDRESS xx-xx-xx-xx-xx-xx  <- RSX
```

That command causes a "reboot" message to be sent to the target machine, which causes the MicroPower/Pascal QN driver to load the firmware from the DEQNA ROM and begin the loading sequence. You can use that command to reload the same image or, with an NCP SET NODE UPOWER LOAD FILE command (which must precede the TRIGGER), to load a new image.

Note

If multiple host systems contain enough information to service a down-line load request from the target system, the first to respond will perform the load, independent of which host executed the TRIGGER request. The other systems may report a load timeout error. If the system that performed the load was not the system that executed the TRIGGER, the wrong image is loaded into the target. You can avoid such problems by issuing the NCP command CLEAR NODE UPOWER HARDWARE ADDRESS on host systems that should not respond to the down-line load request.

13.2 DECNET/DDCMP Down-Line Loading

The DDCMP down-line loading method pertains only to the final production image of the application. Loading for debugging is done with PASDBG.

DECnet/DDCMP down-line loading uses DECnet network facilities on the host system (RSX or VMS) and a down-line bootstrap loader residing on the target machine. This loader resides in the firmware ROM on the target processor board.

Note

See your target processor documentation on how to initiate the DDCMP down-line load procedure.

Together those software components use DDCMP to copy a MicroPower/Pascal image file from the host system to the main memory of the target machine. Once the MicroPower/Pascal software is in the target memory, it gains control of the processor and begins executing. The MicroPower/Pascal image need not contain the DDCMP driver (CS) or the network service process (NSP) to be down-line loaded. You should, however, include those processes if the application requires network communication support at run time.

For the host operating system to provide the down-line load facility, it must be able to recognize a boot request message from the target system. Also, the MicroPower/Pascal target machine must be described in the host system's network node database.

The principal tool used to control the network for VMS or RSX is the Network Control Program (NCP). We recommend that you become familiar with the NCP utility. Please refer to the network manager's guide for your host system for information on NCP, including how to invoke it and the privileges required.

Note

The DDCMP circuit is the name of the host system's hardware device controller, which is connected to the same asynchronous serial line as the target machine. Depending on your configuration, the correct specification might be DLV–0. Check your configuration to determine the proper circuit specification.

To enable the host to recognize a boot request from DDCMP, the circuit must have service enabled:

```
NCP> SET CIRCUIT DLV-0 SERVICE ENABLED
```

To enter a MicroPower/Pascal target machine into the host system's network node database, you issue NCP commands to store the target machine's node name and address and host circuit for loading. For example, for the node "UPOWER::":

```
NCP> SET NODE UPOWER ADDRESS 5.410
NCP> SET NODE UPOWER SERVICE CIRCUIT DLV-0
NCP> SET NODE UPOWER LOAD FILE file-specification    <- described below

NCP> SET NODE UPOWER SECONDARY LOADER MICROPOWER$LIB:SECDLV.SYS  <- VMS
NCP> SET NODE UPOWER TERTIARY LOADER MICROPOWER$LIB:TERDLV.SYS
or
NCP> SET NODE UPOWER SECONDARY LOADER dev:[5,54]SECDLV.SYS       <- RSX
NCP> SET NODE UPOWER TERTIARY LOADER dev:[5,54]TERDLV.SYS
```

The node name and address may have already been specified when your network was installed. Make sure that each node in your network has a unique address and name.

The NCP commands place the information about node UPOWER in the host system's volatile database. However, information in the volatile database must be reentered after each host system reload. To make that information part of the permanent database, use the DEFINE command rather than SET on VMS or use the CFE program rather than the NCP program on RSX.

The MicroPower/Pascal image (.MIM) file does not have an appropriate format for DDCMP down-line loading and must be converted to a format that the bootstrap loader will accept. You perform that conversion by running the MKBOOT (VMS) or MKB (RSX) program provided on the MicroPower/Pascal distribution kit. MKBOOT/MKB prompts for an input file name, and you respond with the .MIM file name. MKBOOT then prompts for an output file name, and you respond with the name of the loader format file, which has the .SYS file type. MKBOOT then converts the .MIM file into a formatted .SYS file.

After the image format conversion, you use the following command to specify the .SYS file as the image to be loaded into the target system's main memory:

```
NCP> SET NODE UPOWER LOAD FILE file-specification
```

After the down-line load is complete, the MicroPower/Pascal network services, if present in the target application, can take advantage of the information that the host system provided. In particular, the host system has informed the target system of the proper node number. If your application image is mapped, the NSP will use the host-supplied node number, so you need not specify the address information in the NSP prefix file, thus allowing the same MicroPower/Pascal application image to be run on multiple machines in a network. You enable those capabilities by specifying NETBOOT=YES to the SYSTEM macro in your kernel configuration file.

Once a MicroPower/Pascal application is initialized and is running, the DDCMP driver (CS) can respond to a host system request to reinitiate the DDCMP down-line load sequence. You enable that capability by specifying NETTRIGGER=YES to the SYSTEM macro in your kernel configuration file. You can then issue the following NCP command:

```
NCP> TRIGGER NODE UPOWER                              <- VMS or RSX
```

That command causes a "reboot" message to be sent to the target machine, which causes the MicroPower/Pascal CS driver to load the firmware from the Processor ROM and begin the loading sequence. You can use that command to reload the same image or, with an NCP SET NODE UPOWER LOAD FILE command (which must precede the TRIGGER), to load a new image.

Note

When you are using the trigger command in this way, you should be aware that the CS driver initiates the boot sequence of the processor bootROMs. You must configure the processor bootROM to boot from the DL device. If the processor is configured to boot from another device, results are unpredictable.

Appendix A
Interaction of RELOC and MIB

This appendix shows many examples of RELOC and MIB command lines to perform various application builds.

A.1 Relocating Mapped Static Processes

You can use the /RO, /RW, or /QB options of RELOC to override the default virtual addressing if needed, as for a driver mapped process.

If a mapped process is to be used in a mixed ROM/RAM configuration, you must use the /AL option of RELOC, which starts the RW (RAM) segment on a 4K-word virtual address boundary. In addition, you can use the option in other situations to get particular effects, as described in Chapter 10.

Case 1: Use First Available Memory in Image

MAPPED—RAM-only

- RELOC command line:

```
RLC>pimfile=mobfile
```

  RELOC assigns virtual addresses to the read-only segment starting at 0 and to the read/write segment at the first available address following the read-only segment—that is, with no virtual-address separation between the two segments. RELOC produces the virtually relocated .PIM file.

- MIB command line:

```
MIB>outmim=pimfile,inputmim/SM
```

  MIB installs the read-only and read/write segments contiguously in the first available memory and sets up the process's memory-management register (PAR) values accordingly.

MAPPED—ROM/RAM

- RELOC command line:

```
RLC>pimfile=mobfile/AL
```

  RELOC assigns virtual addresses to the read-only segment starting at 0 and to the read/write segment starting at the next available 4K-word boundary, because of the /AL option. RELOC produces the virtually relocated .PIM file.

- MIB command line:

```
MIB>outmim=pimfile,inputmim/SM
```

  MIB starts the read-only segment and the read/write segment in the first available ROM and RAM areas, respectively, and sets up the process's memory-management register (PAR) values accordingly. Thus, the two segments are always separated and are mapped by different PARs.

Case 2: You Supply Start Addresses for RO and RW Segments

MAPPED—RAM-only

- RELOC command line:

```
RLC>pimfile=mobfile/RO:virtual-addr[/RW:virtual-addr]
```

  RELOC assigns virtual addresses to the read-only and read/write segments, using the starting address(es) you have specified. The addresses must reflect 4K-word address boundaries, corresponding to the virtual base of a given PAR. Expressed as an octal byte address, each value must be a multiple of 20000, where the multiplier can be 0 through 7. If you do not specify the /RW option, the read/write segment is allocated addresses contiguous with the read-only segment—no PAR separation between the two segments.

  You would use this form of RELOC command line for a device driver (driver mapped) process, since the read-only and read/write segments of such a process have to be mapped by, as well as fit within, PARs 2 and 3 respectively. For example, the following command relocates the DL driver:

```
RLC>DLHANDLR=DLHANDLR/RO:40000/RW:60000
```

  This use of /RO and /RW forces the special PAR 2 and PAR 3 mapping required for a driver's code and impure-data segments.

- MIB command line:

```
MIB>outmim=pimfile,inputmim/SM
```

  MIB places the process image in the first available physical memory, treating the read-only and read/write segments as separately allocatable units if the read/write segment is not virtually contiguous with the read-only segment. If the two segments are virtually separated—/RW was used—the two segments will probably still be contiguous in physical memory but not necessarily if the available memory in the image is fragmented or discontinuous.

  In any case, MIB sets up the process's PAR values to reflect the process's placement in the memory image.

MAPPED—ROM/RAM

- RELOC command line:

```
RLC>pimfile=mobfile[/RO:virtual-addr]/RW:virtual-addr
```

  RELOC assigns virtual addresses to the read-only and read/write segments, using the starting address(es) you have specified. The addresses must reflect 4K-word virtual address boundaries, as described above for a mapped RAM-only image. If you do not specify the /RO option, the read-only segment is origined at 0 by default. You must specify the /RW option (or /AL) for mapped ROM/RAM to ensure the necessary PAR separation between the two segments.

  As described for mapped RAM-only, you would use this form of RELOC command line for a device driver (driver mapped) process, specifying /RO:40000 and /RW:60000 to achieve the required PAR 2 and PAR 3 mapping.

- MIB command line:

```
MIB>outmim=pimfile,inputmim/SM
```

  MIB starts the read-only segment and the read/write segment in the first available ROM and RAM areas, respectively, and sets up the process's memory-management register (PAR) values accordingly. Thus, the two segments are always separated and are mapped by different PARs.

Case 3: You Supply Addresses for Specific, Named Program Sections

Positioning of process segments in memory by specification of individual p-sect names and addresses involves use of the RELOC /QB option, and, for a mapped image, possibly use of the MIB /QB option also. See Chapters 10 and 11 for a detailed description of the RELOC and MIB /QB options respectively.

Ordinary cases do not require the /QB option. You need to use /QB only if your process has special internal location or mapping dependencies that cannot be satisfied through use of the /RO and /RW or /AL options, which cover all commonly encountered relocation requirements by implicit reference to the first RO section and the first RW section of a program. Those options are, in effect, convenient specialized forms of the /QB option. Consider the following equivalences between the "implicit" /RO, /RW, and /AL options and the RELOC /QB option:

- /RO:mmmmm is equivalent to specifying /QB:1st-RO-section:mmmmm—that is, /QB:-.ALST.:mmmmm in any normal MicroPower/Pascal program. (The .ALST. p-sect is generated for the Pascal PROGRAM heading and by the DFSPC$ macro call in MACRO–11; see Chapter 10.)
- /RW:nnnnn is equivalent to specifying /QB:1st-RW-section:nnnnn—that is, /QB:.CDAT.:-nnnnn for a Pascal implementation or /QB:.MDAT.:nnnnn for a MACRO implementation in which the PURE$, PDAT$, and IMPUR$ macros are used exclusively for program sectioning.
- /AL is equivalent to specifying /QB:1st-RW-section:xxxxx, where "1st-RW-section" is again .CDAT. or .MDAT., and xxxxx is the next 4K-word boundary address following the RO section addresses. (/AL is intended specifically for mapped usage.)

Thus, you need use the /QB option only for specifying p-sects other than the first within the read-only or read/write segments.

MAPPED—RAM-only or ROM/RAM; locate named p-sects at specific virtual addresses

- RELOC command line:

```
RLC>pimfile=mobfile[/RO:mmm]/QB:psect-xxx:nnn:psect-yyy:ppp
```

or, equivalently:

```
RLC>pimfile=mobfile/QB[:.ALST.:mmm]:psect-xxx:nnn:psect-yyy:ppp
```

The /QB option can specify up to eight section names and addresses. Assuming that p-sect .ALST. precedes "psect-xxx" and "psect-xxx" precedes "psect-yyy" in the sorted input module, RELOC does the following:

- Starts the process image at virtual page address mmm if /RO or p-sect .ALST. is specified or at virtual address 0 if the first RO section is not specified
- Assigns contiguous addresses to any unspecified p-sects falling between .ALST. and psect-xxx
- Starts psect-xxx at virtual page address nnn, if possible, and assigns contiguous addresses to any unspecified p-sects falling between psect-xxx and psect-yyy
- Starts psect-yyy at virtual page address ppp, if possible, and assigns contiguous addresses to any unspecified p-sects following psect-yyy

The specified address values must be on 4K-word boundaries and, for other than the first RO section, must be high enough to allow for the allocation of all preceding program sections. If not, RELOC issues a warning message—"Specified section start too low, ignored—psectname"—ignores the invalid start address specification, and starts the corresponding specified section at the next available address above the last-allocated section.

If the memory image is ROM/RAM, one of the sections specified in the /QB option must be the first RW section, to start that section at a 4K-word virtual address boundary so that MIB can map the read/write segment separately in RAM.

- MIB command line:

```
MIB>outmim=pimfile,inputmim/SM
```

MIB places the process image in the first available physical memory on a first-fit basis, treating each discontinuous virtual segment as a separately allocatable unit. In a RAM-only image, all the segments will probably be contiguous in physical memory—ignoring the 32-word physical boundary breaks needed between noncontinuous pages—but not necessarily if the available memory in the image is fragmented or noncontinuous. In a ROM/RAM memory image, all read/write segments are physically separate from the read-only segments, of course, but the segments within each group are physically contiguous if available ROM and RAM areas permit.

In any case, MIB sets up the process's PAR values to reflect the process's placement in the memory image.

MAPPED—RAM-only or ROM/RAM; locate one or more named p-sects at specific physical addresses

- RELOC command line:

```
RLC>pimfile=mobfile[/RO:mmm]/QB:psect-xxx:nnn:psect-yyy:ppp
```

  or, equivalently:

```
RLC>pimfile=mobfile/QB[:.ALST.:mmm]:psect-xxx:nnn:psect-yyy:ppp
```

  Again, the /QB option can specify up to eight section names and addresses. RELOC performs virtual relocation of the process image exactly as explained in the preceding example. The difference in this example is in the MIB step.

- MIB command line:

```
MIB>outmim=pimfile,inputmim/QB:psect-name:phys-blk-num:psect-name.../SM
```

  In this command line, "psect-name" can be the name of any program section that was relocated to start on a virtual page (4K-word) boundary, and nnn is a "memory block" offset value as described in Section 11.4.5. As a more specific example, the following command line fixes the physical start address for the read/write program section .CDAT. at—for simplification—octal location 100000 (1000*100):

```
MIB>outmim=pimfile,inputmim/QB:.CDAT.:1000/SM
```

  This command line assumes that the .CDAT. program section has been relocated at some virtual page boundary—whether by the RELOC /RW, /AL, or /QB option makes no difference.

  MIB places the virtual program segment(s) that precede the segment beginning with section .CDAT. in first available physical memory. MIB then starts the segment beginning with section .CDAT. at physical location 100000. Any additional noncontiguous segments are allocated memory following the segment located by the /QB option, on a next-available basis. (In this example, if MIB finds location 100000 already used when it attempts to allocate the .CDAT. section, it issues the fatal error message "Specified memory for /QB section unavailable or nonexistent.")

  MIB sets up the process's PAR values to reflect the placement of the process segments in the memory image.

A.2 Relocating Unmapped Static Processes

For a RAM-only target, the RO and RW segments can be placed contiguously in memory. For a ROM/RAM target, however, they must be separated, and starting addresses must be provided. Normally, RELOC determines those addresses for you by looking them up in the input .MIM file; alternatively, you can use the /RO, /RW, or /QB options.

RELOC can obtain the needed address information by inspecting the existing .MIM file in order to find the next available memory location(s) in the current memory image. You must always include the name of the existing memory image (.MIM) file as an input in the RELOC command line when you build an unmapped process.

Case 1: Use First Available Memory in Image

UNMAPPED—RAM-only or ROM/RAM

- RELOC command line:

```
RLC>pimfile=mobfile,mimfile
```

  RELOC inspects the existing .MIM file to determine the next available physical starting addresses for the process's read-only and read/write segments and relocates the process accordingly. In the RAM-only case, the process segments are relocated contiguously, assuming that sufficient continuous memory is available in the image. In the ROM/RAM case, the RO and RW segments are relocated disjointly in the first available ROM and RAM, respectively.

- Then, invoke MIB with the same input .MIM file that you specified for RELOC:

```
MIB>outmim=pimfile,inputmim/SM
```

  MIB installs the static process in the memory image at the physical starting address(es) fixed by RELOC.

Case 2: You Supply Start Addresses for RO and RW Segments

UNMAPPED—RAM-only or ROM/RAM

- RELOC command line:

```
RLC>pimfile=mobfile/RO:phys-addr[/RW:phys-addr],mimfile
```

  RELOC assigns physical addresses to the read-only and read/write segments, using the starting address(es) you have specified. (Presumably, you have inspected a RELOC map of the process being built to determine segment sizes and a MIB map of the existing image to determine available start locations.) If you do not specify the /RW option for a RAM-only image, RELOC allocates the read/write segment contiguously with the read-only segment. You must specify the /RW option for a ROM/RAM image to origin the read/write segment at some location in RAM. RELOC produces the physically relocated .PIM output file. Although RELOC does not use the input .MIM file to determine starting addresses in this case, RELOC does check the .MIM file to be sure that it is unmapped.

- MIB command line:

```
MIB>outmim=pimfile,inputmim/SM
```

  MIB installs the static process in the memory image at the physical starting address(es) fixed by RELOC.

Case 3: You Supply Addresses for Specific, Named Program Sections

Positioning of process segments in memory by specification of individual p-sect names and addresses involves use of the RELOC /QB option for an unmapped image. See Chapters 10 and 11 for a detailed description of the RELOC /QB option.

The /QB option is not ordinarily required. You need to use /QB only if your process has special "internal" location or mapping dependencies that cannot be satisfied through use of the /RO and /RW options, which cover all commonly encountered relocation requirements by implicit reference to the first RO section and the first RW section of a program. These options are

convenient, specialized forms of the /QB option. Consider the following equivalences between the "implicit" /RO and /RW options and the RELOC /QB option:

- /RO:mmmmm is equivalent to specifying /QB:1st-RO-section:mmmmm—that is, /QB:-.ALST.:mmmmm in any normal MicroPower/Pascal program. (The .ALST. p-sect is generated for the Pascal PROGRAM heading and by the DFSPC$ macro call in MACRO–11; see Chapter 10.
- /RW:nnnnn is equivalent to specifying /QB:1st-RW-section:nnnnn—that is, /QB:.CDAT.:-nnnnn for a Pascal implementation or /QB:.MDAT.:nnnnn for a MACRO implementation in which the PURE$, I-DAT$, and IMPUR$ macros are used exclusively for program sectioning.

Thus, you need use the /QB option only for specifying p-sects other than the first within the read-only or read/write segments.

UNMAPPED—RAM-only or ROM/RAM

- RELOC command line:

```
RLC>pimfile=mobfile/RO:mmm/QB:psect-xxx:nnn:psect-yyy:ppp,mimfile
```

  or, equivalently:

```
RLC>pimfile=mobfile/QB:.ALST.:mmm:psect-xxx:nnn:psect-yyy:ppp,mimfile
```

  (The /QB option can specify up to eight section names and addresses.) Assuming that p-sect .ALST. precedes "psect-xxx" and "psect-xxx" precedes "psect-yyy" in the sorted input module, RELOC does the following:

  - Starts p-sect .ALST. at physical address mmm and assigns contiguous addresses to any unspecified p-sects falling between .ALST. and psect-xxx
  - Starts psect-xxx at physical address nnn, if possible, and assigns contiguous addresses to any unspecified p-sects falling between psect-xxx and psect-yyy
  - Starts psect-yyy at physical address ppp, if possible, and assigns contiguous addresses to any unspecified p-sects following psect-yyy

  The specified address values nnn and ppp must be high enough to allow for the allocation of all preceding program sections. If not, RELOC issues a warning message, ignores the invalid start address specification, and starts the corresponding specified section at the next available address above the last-allocated section.

  If the memory image is ROM/RAM, one of the sections specified in the /QB option must be the first RW section in order to start that section at some point in RAM.

- MIB command line:

```
MIB>outmim=pimfile,inputmim/SM
```

  MIB installs the several discontinuous segments of the static process in the memory image at the physical addresses fixed by RELOC.

Appendix B
Extended Disk (XD) and DRV11 (YA) Drivers

This appendix gives the build instructions for the XD and YA drivers. Those drivers differ from other MicroPower/Pascal drivers in build requirements and in other respects. (The XD and YA drivers are also the only two MicroPower/Pascal drivers to be coded in Pascal.)

B.1 Extended Disk (XD) Driver

The XD driver permits MicroPower/Pascal applications to perform Pascal file-structured I/O to disk devices with greater than 65536 blocks by partitioning the physical disk unit into multiple partitions. Each physical disk unit is treated as a single controller with one or more logical units. The XD driver conceptually resides "between" the ACP and the physical disk driver. In response to I/O requests issued by the ACP, the XD driver performs the intermediate processing necessary to complete the operation.

Note

Using the XD driver entails the participation of file system OTS routines, the ACP, and a physical disk driver (normally the DU driver). Alternatively, you can perform non-file-structured I/O to disk devices with greater than 65,536 blocks by issuing send/receive requests to a physical disk driver.

Unlike other MicroPower/Pascal drivers, the extended disk (XD) driver is not included in the driver object libraries. Instead, it is distributed as a source file, XDDRV.PAS, on the kit. To use the XD driver, you must edit XDDRV.PAS as appropriate for your application, compile it with its include files (EXC.PAS, IOPKTS.PAS, and GSINC.PAS), and build it into your application as a static user process. (In response to the MPBUILD question "Is this process a device driver?", you should respond "NO".)

During the application build, you must merge the XD driver with the module GETSET.PAS.

Your application image must also include the ACP and physical disk driver system processes.

No prefix file exists for the XD driver. User changes are made to the driver source. See Chapter 5 for a more detailed description of the building of user processes. For more information on disk drivers, see Chapter 4 of the *MicroPower/Pascal I/O Services Manual.*

B.2 DRV11 (YA) Driver

Unlike other MicroPower/Pascal drivers, the DRV11 (YA) driver is not included in the driver object libraries. Instead, it is distributed as two source modules—the driver proper (YADRV.PAS) and the driver prefix file (YAPFX.PAS). The DRV11 (YA) driver is available for applications that require it, or as a base for editing, or as an example of a device driver coded in Pascal.

You must compile both YAPFX.PAS and YADRV.PAS and merge the output object modules to build the DRV11 driver process. In the following examples of command lines, the unmapped case is arbitrarily chosen for illustration.

To compile both the prefix module and the driver module, you can use the appropriate form of Pascal command line given below:

```
> [MCR] MPP
MPP> YAPFX=YAPFX/IN:NHD     (RSX)
or
$MPPASCAL YAPFX/IN=(NHD)    (VMS)

> [MCR] MPP
MPP> YADRV=YADRV/IN:NHD     (RSX)
or
$MPPASCAL YADRV/IN=(NHD)    (VMS)
```

You must use the NHD option when compiling the Pascal modules for an unmapped system, regardless of your target hardware instruction set. You must merge the unmapped driver with DRVU.OLB, FILSYS.OLB, and LIBNHD.OLB. For a mapped system, compile the prefix file, substituting the EIS option for the NHD option, regardless of your target hardware instruction set. Then merge the mapped driver, substituting DRVM for DRVU and LIBEIS for LIBNHD.

The following example merges the DRV11 driver process for an unmapped application:

```
>[MCR] MRG (for RSX) or $MPMERGE (for VMS)
MRG>YAPFX=YAPFX,YADRV,KERNL1.STB,mpp-lib:DRVU/LB,FILSYS/LB,LIBNHD/LB
```

The OTS object library files, FILSYS and LIBNHD, satisfy references to OTS routines found in the unmapped YA driver object module. The driver object library file, DRVU, satisfies references to common driver routines. To merge a mapped YA driver, substitute DRVM for DRVU and substitute the LIBEIS library for the LIBNHD library.

The RELOC and MIB steps are the same as for a driver written in MACRO.

See Chapter 4 for a more detailed description of the building of DIGITAL-supplied system processes. For more information on parallel line drivers, see Chapter 6 of the *MicroPower/Pascal I/O Services Manual.*

Appendix C
.MIM File Format

The following diagram shows the format of the .MIM file for MicroPower/Pascal V2. If the file does not contain a bootstrap, the file starts at block 0 for RT, block 1 for RSX and VMS. The block numbers are different because RT numbers its virtual blocks for files starting from 0, and RSX and VMS number theirs starting from 1. If the file contains a bootstrap, the .MIM file header starts at the block following the bootstrap. MicroPower bootstraps are two blocks long for unmapped applications and three blocks long for mapped applications. For RT, therefore, the file header starts at block 2 for unmapped applications and at block 3 for mapped applications. For RSX and VMS, it is block 3 for unmapped applications and block 4 for mapped applications.

If you dump a .MIM file, keep in mind that the RT and RSX dumpers display from left to right on each line (offset 0 on the left) but that the VMS dumpers display from left to right for octal dumps (offset 0 is on the right).

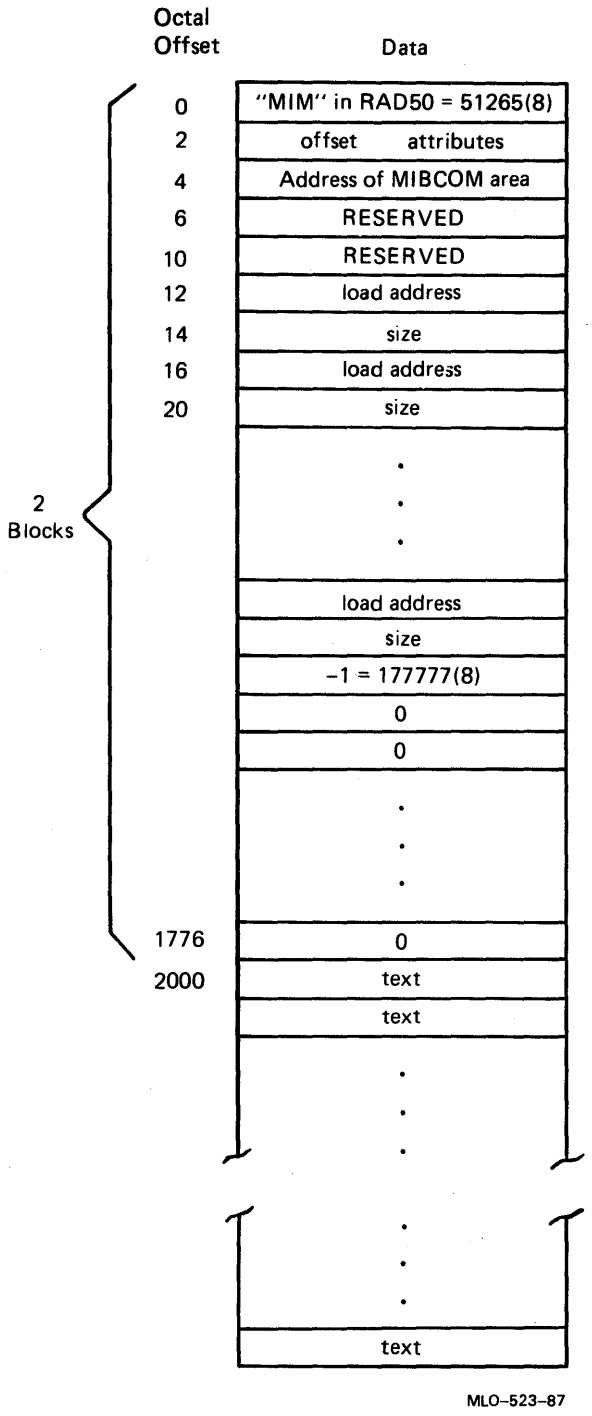
The load addresses in the diagram are memory addresses that are used when the .MIM file is loaded (not addresses of the text in the .MIM file). For unmapped applications, all addresses and sizes are actual 16-bit octal values (number of bytes). For mapped applicatons, all addresses and sizes are 16-bit values representing units of 100(octal)-byte memory blocks. The offset field value is an actual 16-bit value (number of bytes). To obtain full byte addresses and sizes for mapped applications, multiply the value by 100(octal) (equivalent to adding two zeros to the octal value). For example, the value 6734(octal) represents the actual value 673400(octal).

The offset field, used only for mapped applications, is always 0 for unmapped applications. It gives the offset from the memory block address in the MIBCOM address field to the start of the MIBCOM area. For example, if the value in the MIBCOM address field is 235(octal) and the offset is 12(octal), the actual memory address of the MIBCOM area is 23500(octal)+12(octal)=23512(octal).

Attributes byte bit definitions are as follows:

```
bit 0 - 1, mapped application; 0, unmapped
bit 1 - 1, ROM/RAM application; 0, RAM-only
bit 2 - 1, J11 mapping (support for separation  of  I-space  and
        D-space and for supervisor-mode shared libraries); 0, no
        J11 mapping
bit 3 - 1, CMR21 processor; 0, a processor other than CMR21
```

Figure C–1: .MIM File Format



To convert octal offsets within the file to block number and offset within a block, split the offset into two parts. The last three octal digits represent the octal offset within the block, and the rest of the octal digits represent the octal block number on RT systems (one less than the octal block number on RSX and VMS systems). For example, offset 56342(octal) is block 56(octal)=46(decimal) on RT systems and offset 342(octal) within the block. It is block 56(octal)+1=57(octal)=47(decimal) on VMS and RSX systems and the same offset within the block (342 octal).

The text in the .MIM file for each load segment immediately follows the text for the previous segment. To locate the start of the text in the .MIM file for a segment, therefore, add the sizes of all previous segments to find the offset from the start of text in the file. Then add the octal offset of the start of text (varies depending on the operating system and the type, if any, of bootstrap: mapped or unmapped). Finally, convert the octal offset within the file to a block number and offset within the file.

Index

A

Active Page Register (APR), 6-4
Address space, saving, 6-4
Application
  building, 1-3, 1-7, 2-6, 2-7
  development
    debug phase, 5-8
    rebuild phase, 5-8
  MIM file, 2-7
Assembling user processes, 5-3

B

Bootstrap
  file
    installing, 11-6
    MPBUILD dialog, 2-11
    removing, 11-6
  load format
    MIM file, 11-3
Build cycle, 1-3, 2-1
  automated, 2-1
  MERGE utility, 9-6
  MIB utility, 11-7
  partial, 2-6
  transition point, 2-10
  user input, 1-7
  XD driver, B-1
  YA driver, B-2
Build utilities
  overview, 1-3

C

Command files
  generated
    MPBUILD, 1-7, 2-7, 2-11
  intermediate, 2-7
Command format
  MERGE, 3-4, 4-5, 5-4
  MIB, 3-8, 4-7, 5-7
  Pascal, 5-3
  RELOC, 3-5, 4-6, 5-6
Compiler
  options, 8-3
    command line, 8-9
    MPBUILD, 2-8
    source program, 8-8
Compiling
  user processes, 5-3
Configuration file, 1-4, 2-6
  assembling, 3-3
  creating, 3-2
COPBOT.PAS
  See also COPYB utility
  copy bootstrap program, 12-4
COPYB utility, 1-3
  See also COPBOT.PAS
  command line examples, 12-2
  functions, 12-1
CS driver
  down-line loading, 13-6
  run-time support, 13-4

D

DBG files
  initializing, 11-6
  MPBUILD, 2-7
DDCMP method
  down-line loading, 13-4
Debugger Service Module (DSM), 2-7, 3-3, 7-1
Debugging
  processes with shared libraries, 6-14

Debugging support
  build cycle, 2-7
/DEBUG option
  Pascal, 8-9
DEBUG parameter
  SYSTEM configuration macro, 7-1
Debug symbol information
  MPBUILD, 2-7
  Pascal programs, 8-9
DECnet
  down-line loading, 13-2
DEQNA address
  down-line loading, 13-3
Device driver, 4-2
  object libraries, 1-4
  overview, 1-4
  prefix modules, 1-4, 2-9
  user supplied, 2-11
Down-line loading
  DDCMP method, 13-4
  DECnet, 13-2
  Ethernet, 13-2
  hardware address, 13-3
  PASDBG, 7-1
DRVM.OLB, DRVU.OLB, 1-4

E

Ethernet
  down-line loading, 13-2

F

File options
  /LIB, /LIST, /OBJ, /MAP, /MAC, /PAS, 2-4
FILSYS.OLB, 8-1

G

Global references
  MERGE utility, 9-3

H

Host system, 1-1, 1-2

I

I&D-space
  restrictions, 6-2
  separation, 6-2
    build cycle, 6-3
    kernel configuration, 6-1
/IDS option
  MPBUILD dialog, 2-4
INCLUDE files, Pascal
  PREDFL.PAS, 8-12
Installing
  bootstrap file, 11-6
  debug symbols
    shared library, 11-7
    static process, 11-7
  shared libraries, 11-6
  static processes, 5-5, 11-5
  system processes, 4-5
    memory image, 4-7
Interrupt Service Routines (ISR)
  shared libraries, 6-5

J

J11 processor, 6-1

K

Kernel
  build cycle, 2-6, 2-7
    MIB utility, 3-7
  functions, 1-4
  mapped
    relocating, 3-5
  merging
    debugging, 3-5
    mapped target, 3-4
    unmapped target, 3-5
  MIM file, 2-6, 3-7
  object libraries, 1-4
  optimization, 3-11
  phase
    MERGE utility, 3-4
  relocating
    debugging, 3-7
    unmapped RAM-only target, 3-6
    unmapped ROM/RAM target, 3-6
  symbol table
    creating, 3-5
  unmapped
    relocating, 3-6

L

LIBNHD.OLB, LIBEIS.OLB, LIBFIS.OLB, LIBFPP.OLB, 8-1

/LIB option
  MPBUILD dialog, 2-4
Libraries
  DRVM.OLB, DRVU.OLB, 1-4
  LIBNHD.OLB, LIBEIS.OLB, LIBFIS.OLB, LIBFPP.OLB, 8-1
  macro, 3-3
    restriction, 2-2
  PAXM.OLB, PAXU.OLB, 1-4
/LIST option
  MPBUILD dialog, 2-4
Load format
  bootstrap, 11-3
  PASDBG, 11-3
Loading
  down-line with DDCMP, 13-4
  down-line with DECnet, 13-2
  down-line with Ethernet, 13-2
  down-line with PASDBG, 7-1

M

/MAC option
  MPBUILD dialog, 2-4
Macro libraries, 3-3
  MPBUILD restriction, 2-2
Map file
  creating, 11-6
/MAP option
  MPBUILD dialog, 2-4
Mapped
  applications, 4-5, 6-3
  memory image, 5-7
  static processes
    relocating, A-1
MERGE utility, 1-3, 2-2
  build cycle, 9-6
  command line examples, 9-7, 9-8
  command line format, 3-4, 4-5, 5-4
  functions, 9-3
  global references, 9-3
  input file order, 9-5
  kernel phase, 3-4
    debugging, 3-5
    mapped target, 3-4
    unmapped target, 3-5
  object library order, 9-5
  options, 9-12
    debug symbols (/DE), 9-13
MERGE utility
  options (cont'd.)
    extract module (/LB:module:...), 9-15
    include module (/IN), 9-14
    library file identification (/LB), 9-15
    module name (/NM), 9-15
    supervisor-mode shared library (/SL), 9-16
    user-mode shared library (/UL), 9-16
    version number (/VR), 9-16
  reference resolution, 9-4
  section maps, 9-10
  shared libraries, 9-7
  static process, 9-7
Merging
  static processes, 5-4
  system configuration file, 9-6
MIB utility, 1-3, 2-2, 4-5
  build cycle, 11-7
  command format, 3-8
  command line examples, 11-8, 11-9
  command line format, 4-7, 5-7
  functions, 11-2
  kernel building, 3-7
  memory map, 11-14
  options, 11-11
    align p-sect (/QB), 11-13
    exception group code (/GC), 11-12
    install bootstrap (/BS), 11-11
    kernel installation (/KI), 11-12
    process priority (/PR), 11-13
    remove bootstrap (/RB), 11-14
    small memory image (/SM), 11-14
  static process, 5-5
  system processes, 4-7
MIM files
  conversion
    MKBOOT program, 13-3, 13-5
    MKB program, 13-3, 13-5

MIM files (cont'd.)
creating, 11-2
booting, 3-9
debugging, 3-8
down-line loading, 3-9
ROM/RAM environment, 3-10
down-line loading, 7-1
installing system processes, 4-7
kernel, 3-7
names, 4-7
MKBOOT program
MIM file conversion, 13-3, 13-5
MKB program
MIM file conversion, 13-3, 13-5
MPBUILD procedure, 1-3, 2-1
dialog, 2-5
errors, 2-11, 2-12
limitations, 2-2
macro library restriction, 2-2
system processes, 2-4, 2-9
user processes, 2-4, 2-10
MPBUILD utility, 5-8
MPPASCAL compiler
command syntax, 8-2
compilation options, 8-3, 8-9
OTS libraries, 8-1
overview, 8-1
PREDFL.PAS file, 8-12
source-program options, 8-8
MPSETUP.COM file, 1-9
MPxxxx symbol definitions, 1-9

N

NETBOOT option
SYSTEM macro, 13-3, 13-5
NETTRIGGER option
SYSTEM macro, 13-3, 13-5
Network Control Program (NCP), 13-2
network node database, 13-2, 13-4
Network node database
Network Control Program (NCP), 13-2, 13-4
Network Service Process (NSP)
run-time support, 13-2, 13-4

O

Object libraries
DRVM and DRVU, 1-4
Object libraries (cont'd.)
LIBNHD, LIBEIS, LIBFIS, LIBFPP, 8-1
PAXM and PAXU, 1-4
SUPEIS, SUPFPP, 8-1
/OBJ option
MPBUILD dialog, 2-4
Optimizing the kernel, 3-11
Options
MIB utility, 11-11
RELOC utility, 10-9
OTS
libraries, Pascal, 8-1
object libraries, 6-6
routines, 6-1, 6-3

P

Pascal
command format, 5-3
compiler, 1-2
INCLUDE files
PREDFL.PAS, 8-12
OTS libraries, 8-1
PASDBG, 1-2
build cycle, 2-7
LOAD/EXIT command, 7-1
load format, 11-3
/PAS option
MPBUILD dialog, 2-4
PAXM.OLB, PAXU.OLB, 1-4
Position Independent Code (PIC)
shared library, 6-5
PREDFL.PAS, 8-12
Prefix modules
assembling, 4-4
device driver, 1-4, 2-9
editing, 4-2
merging
device driver object library, 4-5
kernel symbol table, 4-5
PROCESSOR macro, 6-1
Programmer format
PROM, 11-4
PROM
programmer format, 11-4

Q

QN driver
run-time support, 13-2

R

Relocating
  kernel for debugging, 3-7
  static processes, 5-5
    mapped, A-1
    unmapped, A-5
  system processes, 4-5
    mapped, 4-6
    unmapped, 4-6
Relocation map, 10-7
Relocation Symbol Directory (RLD)
  record updating, 9-4
RELOC utility, 1-3, 2-2, 4-5
  build cycle, 10-3
  command line examples, 10-4
  command line format, 3-5, 4-6, 5-6
  functions, 10-2
  kernel phase, 3-5
  options, 10-9
    align first RW section (/AL), 10-12
    alphabetical symbol listing (/AB), 10-12
    debug symbols (/DE), 10-14
    disable section sort (/DS), 10-14
    extend section size (/EX), 10-17
    first RO p-sect (/RO), 10-19
    first RW p-sect (/RW), 10-19
    I&D-space separation (/ID), 10-17
    p-sect base address (/QB), 10-18
    RO D-space starting address (/DR:n), 10-14
    round up section size (/UP), 10-20
    RW D-space starting address (/DW:n), 10-17
    short map (/SH), 10-19
    static process name (/NM), 10-18
RELOC utility
  options (cont'd.)
    supervisor-mode shared library (/SL), 10-20
    user library base address (/LS:name:addr), 10-18
    user-mode shared library (/UL[:addr]), 10-20
    value of undefined locations (/ZR), 10-21
    version number (/VR:xxx), 10-20
    wide map (/WI), 10-20
Removing
  bootstrap file, 11-6
Run-time software
  device drivers, 1-4
  kernel, 1-4
  overview, 1-2, 1-4

S

Shared library, 6-1, 6-3
  debugging processes, 6-14
  installing, 11-6
    debug symbols, 11-7
  MERGE utility, 9-7
  restrictions, 6-5
  supervisor-mode, 6-4
    building, 6-6
    kernel configuration, 6-1
    referencing, 6-13
  user-mode, 6-4
    absolute, 6-5, 6-9
    building multiple, 6-12
    mapped, 6-8
    referencing, 6-14
    relocatable, 6-5, 6-8
    unmapped, 6-7
Static process
  installing, 5-5, 11-5
    debug symbols, 11-7
  merging, 5-4
  relocating
    mapped, A-1
    unmapped, A-5
  RELOC utility, 5-5
SUPEIS.OLB, SUPFPP.OLB, 8-1
Supervisor-mode
  shared library, 6-4
    kernel configuration, 6-1

Supervisor-mode
  shared library (cont'd.)
    referencing, 6-13
Symbol definition file,
    MPSETUP.COM, 1-9
SYS file
  MKBOOT program, 13-3, 13-5
System configuration file, 1-4, 2-6
SYSTEM macro
  configuration file, 13-3
  NETBOOT option, 13-3, 13-5
  NETTRIGGER option, 13-3,
      13-5
System processes
  MPBUILD dialog, 2-4, 2-9
  overview, 1-4
  relocating
    mapped, 4-6
    unmapped, 4-6

T

Target system, 1-1, 1-2

U

Unmapped
  applications, 4-6, 6-3
  memory image, 5-7
  static processes
    relocating, A-5
User-mode
  shared library, 6-4
    absolute, 6-5, 6-9
    referencing, 6-14
    relocatable, 6-5, 6-8
User processes
  assembling, 5-3
  compiling, 5-3
  MPBUILD dialog, 2-4, 2-10
Utility programs
  MERGE, 2-2
  MIB, 2-2
  overview, 1-3
  RELOC, 2-2

X

XD driver
  build cycle, B-1

Y

YA driver
  build cycle, B-2

HOW TO ORDER
ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008
Rest of U.S.A. and Puerto Rico*	800–258–1710	Nashua, NH 03061
* Prepaid orders from Puerto Rico must be placed with DIGITAL's local subsidiary (809–754–7575)		
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order desk
Internal orders (for software documentation)	—	Software Distribution Center (SDC) Digital Equipment Corporation Westminster, MA 01473
Internal orders (for hardware documentation)	617–234–4323	Publishing & Circulation Serv. (P&CS) NR03–1/W3 Digital Equipment Corporation Northboro, MA 01532

MicroPower/Pascal–RSX/VMS
System User's Guide
AA–AK13C–TK

READER'S COMMENTS

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

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

_____

_____

_____

_____

_____

_____

_____

_____

_____

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

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

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

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

Do Not Tear — Fold Here and Tape

digital

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
MLO5–5/E45
146 MAIN STREET
MAYNARD, MA 01754–2571

Do Not Tear — Fold Here

Cut Along Dotted Line