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

AA–AK13A–TK

February 1984

This manual contains information you will need for using MicroPower/Pascal application development tools on either an RSX–11M/M–PLUS host system or a VAX/VMS host system in VAX–11 RSX compatibility mode. The manual includes directions for using MicroPower/Pascal utility programs, the MicroPower/Pascal compiler, and the MPBUILD automated build procedure.

This manual replaces the *MicroPower/Pascal–VMS VAX–11 Host User's Guide*, AA–V594A–TE, for VAX/VMS users.

Operating System: RSX–11M Version 4.1
RSX–11M–PLUS Version 2.1
VAX/VMS Version 3.4

Software: MicroPower/Pascal–RSX Version 1.5
MicroPower/Pascal–VMS Version 1.5

To order additional documents from within DIGITAL, contact the Software Distribution Center, Northboro, Massachusetts 01532.

To order additional documents from outside DIGITAL, refer to the instructions at the back of this document.

digital equipment corporation · maynard, massachusetts

First Printing, February 1984

Printed in U.S.A.

A postage-paid READER'S COMMENTS form is included on the last page of this document. Your comments will assist us in preparing future documentation.

CONTENTS

			Page
Preface			ix
CHAPTER	1	INTRODUCTION	1-1
	1.1	OVERVIEW OF MICROPOWER/PASCAL COMPONENTS	1-2
	1.1.1	Application Development Tools	1-2
	1.1.2	Runtime System Software	1-4
	1.2	OVERVIEW OF THE DEVELOPMENT PROCESS	1-6
	1.2.1	The Build Cycle	1-6
	1.2.2	Relationship of MPBUILD to the Build Cycle	1-8
	1.2.3	Loading and Debugging	1-8
	1.3	LOGICAL DEVICES FOR MICROPOWER/PASCAL FILES	1-9
	1.3.1	Logical Device MP: for MicroPower/Pascal-RSX	1-9
	1.3.2	Logical Device MICROPOWER$LIB for MicroPower/Pascal-VMS	1-10
	1.4	USER SET-UP PROCEDURE FOR VAX/VMS	1-10
CHAPTER	2	MICROPOWER/PASCAL COMPILER OPERATION	2-1
	2.1	FILE SPACE REQUIREMENTS	2-1
	2.2	COMPILER INVOCATION AND COMMAND LINE	2-2
	2.2.1	Compilation Command Line Syntax	2-2
	2.2.2	Command Line Usage Rules	2-4
	2.3	COMPILATION OPTIONS	2-5
	2.3.1	Syntax and Use of Source Code Options	2-6
	2.3.2	Option Semantics	2-7
	2.3.2.1	Runtime Checking Code (/CHeck:xxx)	2-7
	2.3.2.2	Debug Symbol Information (/DEbug)	2-7
	2.3.2.3	Extended Statistics (/EXtra)	2-8
	2.3.2.4	Filter Unused Declaration (/FIlter-decls)	2-8
	2.3.2.5	Instruction Set (/INstr:xxx)	2-9
	2.3.2.6	MACRO-11 Output Code (/MAcro)	2-10
	2.3.2.7	No Real-Time Predefinitions (/NOpred)	2-10
	2.3.2.8	Standard Pascal Only (/STandard)	2-10
	2.3.2.9	Selective Listing Control (NOLIST and LIST)	2-10
	2.4	COMPILATION LISTING	2-11
CHAPTER	3	INTRODUCTION TO APPLICATION BUILDING	3-1
	3.1	BUILD CYCLE OVERVIEW	3-4
	3.1.1	Phase 1: Build the Kernel	3-5
	3.1.2	Phase 2: Build the RX02 (DY) Device Handler Process	3-6
	3.2	BUILDING THE KERNEL	3-7
	3.2.1	Create and Assemble Configuration File	3-8
	3.2.2	Merge Configuration Object File	3-9

	3.2.3	Relocate Kernel Module	3-10
	3.2.4	Create Memory Image File	3-12
	3.2.4.1	Building a Memory Image for Debugging or Down-Line Loading	3-13
	3.2.4.2	Building a Memory Image for Booting	3-13
	3.2.4.3	Building a Memory Image for a ROM/RAM Environment	3-15
	3.3	BUILDING DIGITAL-SUPPLIED SYSTEM PROCESSES	3-15
	3.3.1	Edit and Assemble/Compile Prefix Module	3-16
	3.3.2	Merge System Process	3-17
	3.3.3	Relocate Handler	3-19
	3.3.4	Install System Process in Memory Image	3-20
	3.4	BUILDING USER-WRITTEN STATIC PROCESSES	3-21
	3.4.1	Compile/Assemble Static Process Source File	3-21
	3.4.2	Merge Static Process	3-22
	3.4.3	Relocate Static Process	3-24
	3.4.4	Install Static Process in Memory Image	3-25
	3.5	AUTOMATED APPLICATION BUILDING	3-26
	3.6	DEBUGGING AND REBUILDING THE APPLICATION	3-27
CHAPTER	4	EDITING THE CONFIGURATION FILES AND PREFIX MODULE FILES	4-1
	4.1	CONFIGURATION FILE	4-1
	4.1.1	Function of the Configuration File	4-1
	4.1.2	Prototype Configuration File CONFIG.MAC	4-3
	4.1.3	Assembling and Merging the Configuration File	4-6
	4.1.4	Modifying the Configuration File Macros	4-7
	4.1.4.1	CONFIGURATION	4-7
	4.1.4.2	DEVICES	4-8
	4.1.4.3	ENDCFG	4-8
	4.1.4.4	KXT11	4-9
	4.1.4.5	KXT11C	4-15
	4.1.4.6	MEMORY	4-18
	4.1.4.7	PRIMITIVES	4-19
	4.1.4.8	PROCESSOR	4-20
	4.1.4.9	RESOURCES	4-21
	4.1.4.10	SYSTEM	4-23
	4.1.4.11	TRAPS	4-24
	4.2	PREFIX MODULES	4-25
	4.2.1	Prefix Module Function	4-26
	4.2.2	How Prefix Modules Fit into the Build Procedure	4-28
	4.2.3	Editing Prefix Modules: Mapped and Unmapped PDP-11 Target Processors	4-29
	4.2.3.1	ADV11 or AXV11 A-to-D Converter (AA) Handler	4-29
	4.2.3.2	AAV11-C/AXV11-C D-to-A Output Prefix File, ABPFX.PAS	4-31
	4.2.3.3	Line-Frequency Clock Process (CLOCK)	4-32
	4.2.3.4	TU58 (DD), RL02 (DL), MSCP (DU), and RX02 (DY) Handlers	4-33
	4.2.3.5	KWV11-C Real-Time Clock (KW) Handler	4-35
	4.2.3.6	KXT11-C Q-BUS Arbiter Handler Prefix File (KX)	4-37
	4.2.3.7	DRV11-J (XA) Handler	4-38
	4.2.3.8	DLV11 (XL) Handler	4-39
	4.2.3.9	DPV11 Synchronous Serial Line (XP) Handler	4-43
	4.2.3.10	DRV11 (YA) Handler	4-45
	4.2.3.11	SBC-11/21 (FALCON and FALCON-PLUS) 8255 PIO Handler (YF)	4-46

	4.2.4	Editing Prefix Modules: KXT11-C Peripheral Processor	4-46
	4.2.4.1	Line-Frequency Clock Process (CLOCK)	4-47
	4.2.4.2	TU58 DECtape II (DD) Handler Prefix File	4-47
	4.2.4.3	KXT11-C Slave Q-BUS Handler Prefix File (KK)	4-47
	4.2.4.4	KXT11-C DTC (DD) Handler	4-48
	4.2.4.5	KXT11-C Asynchronous Serial Line (XL) Handler	4-49
	4.2.4.6	KXT11-C Synchronous Serial Line (XS) Handler	4-50
	4.2.4.7	KXT11-C Parallel Ports and Timer/Counter (YK) Handler	4-51
	4.2.5	Assembling or Compiling Prefix Modules	4-60
CHAPTER	5	MERGING OBJECT MODULES	5-1
	5.1	FUNCTIONS OF MERGE	5-2
	5.1.1	Resolving Intermodule Global References	5-3
	5.1.2	Updating Relocation Records	5-3
	5.1.3	Satisfying Library References	5-3
	5.1.3.1	Ordering of Multiple Object Libraries	5-4
	5.1.3.2	Ordering of All MERGE Inputs	5-5
	5.2	ROLE OF MERGE IN THE BUILD CYCLE	5-6
	5.2.1	Merging the System Configuration File	5-7
	5.2.2	Merging Each Static Process	5-7
	5.2.3	Optimizing the Kernel	5-7
	5.3	INVOCATION AND USE OF MERGE	5-9
	5.3.1	Command Line Format	5-9
	5.4	SECTION MAP	5-11
	5.5	MERGE OPTIONS	5-13
	5.5.1	Debug Symbols (/DE)	5-13
	5.5.2	Include Module from any Library (/IN)	5-15
	5.5.3	Module Name (/NM)	5-15
	5.5.4	Library File Identification (/LB)	5-16
	5.5.5	Extract Modules From Specific Library (/LB:module:...)	5-16
	5.5.6	Version Number (/VR)	5-17
CHAPTER	6	RELOCATING MERGED OBJECT MODULES	6-1
	6.1	FUNCTIONS OF RELOC	6-2
	6.2	ROLE OF RELOC IN THE BUILD CYCLE	6-3
	6.3	INVOCATION AND USE OF RELOC	6-3
	6.3.1	Command Line Format	6-4
	6.4	RELOCATION MAP	6-6
	6.5	RELOC OPTIONS	6-8
	6.5.1	Alphabetical Symbol Listing (/AB)	6-9
	6.5.2	Align First RW Section at 4K-Word Boundary (/AL)	6-9
	6.5.3	Debug Symbols (/DE)	6-10
	6.5.4	Disable Section Sort (/DS)	6-10
	6.5.5	Extend Section to Specified Size (/EX)	6-11
	6.5.6	Program/Process Name (/NM)	6-11
	6.5.7	Base Address for Specified Program Section (/QB)	6-12
	6.5.8	First RO Section at Specified Address (/RO)	6-12
	6.5.9	First RW Section at Specified Address (/RW)	6-13
	6.5.10	Short Map (/SH)	6-13
	6.5.11	Round Up Section Size (/UP)	6-13
	6.5.12	Program Version Number (/VR)	6-13

	6.5.13	Wide Map (/WI)	6-14
	6.5.14	Value of Undefined Locations (/ZR)	6-14
CHAPTER	7	BUILDING THE MEMORY IMAGE	7-1
	7.1	FUNCTIONS OF MIB	7-2
	7.1.1	Creating a Memory Image File	7-2
	7.1.2	PASDBG Load Format	7-3
	7.1.3	Bootstrap Load Format	7-3
	7.1.4	PROM Programmer Format	7-4
	7.1.5	Installing Static Processes	7-5
	7.1.6	Deleting Static Processes	7-5
	7.1.7	Installing a Bootstrap	7-6
	7.1.8	Creating a Map File	7-6
	7.1.9	Initializing the Debug Symbol File	7-6
	7.1.10	Installing Debug Symbols for a Process	7-6
	7.1.11	Deleting Debug Symbols for a Process	7-6
	7.1.12	Compacting a Mapped Memory Image	7-7
	7.2	ROLE OF MIB IN THE BUILD CYCLE	7-7
	7.3	INVOCATION AND USE OF MIB	7-8
	7.3.1	Command Line Format	7-9
	7.4	MIB OPTIONS	7-11
	7.4.1	Install Bootstrap (/BS)	7-11
	7.4.2	Compact Mapped Image (/CM)	7-12
	7.4.3	Delete Process and/or Symbols (/EX)	7-13
	7.4.4	Exception Group Code (/GC)	7-13
	7.4.5	Kernel Installation (/KI)	7-14
	7.4.6	Process Priority (/PR)	7-14
	7.4.7	Align Specified Program Section (/QB)	7-15
	7.4.8	Small Output Memory Image (/SM)	7-15
	7.5	COMMAND LINE EXAMPLES	7-16
	7.6	HOW RELOC AND MIB INTERACT	7-18
	7.7	MIB MEMORY MAP	7-24
CHAPTER	8	METHODS OF APPLICATION LOADING	8-1
	8.1	DOWN-LINE LOADING THE APPLICATION	8-1
	8.2	BOOTSTRAPPING THE APPLICATION FROM A STORAGE DEVICE	8-1
	8.3	PLACING YOUR APPLICATION IN PROM	8-2
CHAPTER	9	MAKING A VOLUME BOOTABLE ON THE TARGET	9-1
	9.1	FUNCTIONS OF COPYB	9-1
	9.2	INVOKING COPYB	9-2
CHAPTER	10	HOST/TARGET LINE SETUP FOR DEBUGGING	10-1
	10.1	HOST/TARGET TERMINAL LINE REQUIREMENTS FOR PASDBG-RSX	10-1
	10.1.1	Assigning and Allocating the Terminal Line	10-2
	10.1.2	Setting and Showing Line Speed	10-2
	10.1.3	Precautionary SLAVE Setting	10-2
	10.2	HOST/TARGET TERMINAL LINE REQUIREMENTS FOR PASDBG-VMS	10-3
	10.2.1	Assigning and Allocating the Terminal Line	10-3
	10.2.2	Line Protection Codes	10-4
	10.2.3	Setting and Showing Line Speed	10-4
	10.3	LINE-RELATED ERRORS	10-5

APPENDIX	A	THE MICROPOWER/PASCAL FILE SYSTEM	A-1
	A.1	THE DSP PROCESS	A-1
	A.2	THE DSPPFX PREFIX FILE	A-2
	A.3	PASCAL FILE SYSTEM SUPPORT LIBRARY	A-3
	A.4	MACRO-11 FILE SYSTEM SUPPORT LIBRARIES	A-3
APPENDIX	B	MPBUILD APPLICATION-BUILDING PROCEDURE	B-1
	B.1	CAPABILITIES AND LIMITATIONS OF MPBUILD	B-1
	B.2	THE MPBUILD DIALOG	B-3
	B.2.1	Dialog Structure	B-3
	B.2.2	Options, Usage Rules, and Defaults	B-4
	B.2.3	Dialog Description	B-6
	B.2.3.1	Kernel and Global-Information Section	B-6
	B.2.3.2	System-Process Section	B-9
	B.2.3.3	Beginning of User-Process Build Phase	B-11
	B.2.3.4	Pascal User-Process Section	B-12
	B.2.3.5	MACRO User-Process Section	B-13
	B.2.3.6	Bootstrap Section	B-15
	B.2.3.7	End of Dialog	B-15
	B.3	ADDING A PROCESS TO AN EXISTING APPLICATION IMAGE	B-15
	B.4	ERROR MESSAGES	B-16
INDEX			Index-1

FIGURES

Figure	1-1	Kernel Build Phase	1-6
	1-2	System Process Build Phase	1-7
	1-3	User Process Build Phase	1-7
	1-4	Application Image Loading	1-9
	2-1	Compilation Listing: Program with Errors, No /DE Switch	2-12
	2-2	Compilation Listing: Errors Removed, /DE Switch Used to Show Statement Numbers	2-13
	3-1	Input and Output Files for Build Process	3-4
	3-2	Build the Kernel	3-7
	3-3	Build DIGITAL-SUPPLIED System Processes	3-15
	3-4	Build User-Written Pascal Static Processes	3-21
	4-1	The Configuration File and the Build Procedure	4-2
	4-2	Prototype Configuration File	4-4
	4-3	Sample User Error-Handling Interface Routines	4-13
	4-4	ADV11-C/AXV11-C Handler Prefix Module (AAPFX.PAS)	4-30
	4-5	AAV11-C/AXV11-C Analog Output Prefix Module (AAPFX.PAS)	4-32
	4-6	Clock Process Prefix Module (CKPFX.MAC)	4-33
	4-7	TU58 Handler Prefix Module (DDPF.MAC)	4-34
	4-8	RL02 Handler Prefix Module (DLPFX.MAC)	4-34
	4-9	MSCP Handler Prefix Module (DUPFX.MAC)	4-35
	4-10	RX02 Handler Prefix Module (DYPFX.MAC)	4-35
	4-11	KWV11-C Clock Handler Prefix Module (KWPFK.PAS)	4-36
	4-12	KXT11-C Q-BUS Handler Prefix Module for Arbiter (KXPFX.MAC)	4-37
	4-13	DRV11-J Handler Prefix Module (XAPFX.MAC)	4-39

4-14	DLV11 Handler Prefix Module (XLPFX.MAC)	4-43
4-15	DPV11 Device handler Prefix Module (XPPFX.MAC)	4-44
4-16	DRV11 Handler Prefix File (YAPFX.PAS)	4-45
4-17	SBC-11/21 PIO Handler Prefix Module (YFPFX.MAC)	4-46
4-18	TU58 Handler Prefix Module for KXT11-C (DDPFXK.MAC)	4-47
4-19	KXT11-C Q-BUS Handler Prefix Module for Slave (KKPFXK.MAC)	4-48
4-20	KXT11-C DTC Device Handler Prefix File (QDPFXK.MAC)	4-49
4-21	LINDF$ Macros from KXT11-C XL Prefix File (XLPFXK.MAC)	4-50
4-22	KXT11-C Synchronous Serial Line Prefix Module (XSPFXK.PAS)	4-51
4-23	KXT11-C Parallel Port/Timer-Counter Prefix File (YKPFXK.MAC)	4-59
5-1	MERGE Utility Input and Output files	5-2
5-2	MERGE's Part in the Build Cycle	5-6
5-3	Sample MERGE Section Map	5-12
6-1	RELOC Utility Input and Output	6-1
6-2	RELOC's Part in the Build Cycle	6-3
6-3	Sample Relocation Map	6-7
7-1	MIB Utility Input and Output	7-1
7-2	PASDBG or Bootstrap Load Format .MIM File	7-4
7-3	PROM Programmer Format .MIM File	7-5
7-4	MIB's part in the Build Cycle	7-8
7-5	MIB Memory Map	7-25
A-1	The DSPPFX Prefix File	A-2

TABLES

Table	2-1	Compilation Options	2-4
	3-1	Build Utility Programs	3-3
	4-1	Processes and Prefix Modules for Mapped and Unmapped PDP-11 Targets	4-27
	4-2	Processes and Prefix Modules for KXT11-C Peripheral Processors	4-28
	5-1	MERGE Switches	5-13
	6-1	RELOC Switches	6-8
	7-1	MIB Switches	7-11

PREFACE

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. Because MicroPower/Pascal development can be performed in either an RSX-11M/M-PLUS or a VAX/VMS (with VAX-11 RSX) host operating system environment, this manual describes differences in operating procedures where required for the respective host systems.

Chapter Summary

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 the MicroPower/Pascal compiler.

Chapter 3 introduces the MERGE, RELOC, and MIB build utility programs and describes the normal application-build cycle in sufficient detail for building most MicroPower/Pascal applications. The chapter provides useful background information for using the MPBUILD automated build procedure.

Chapter 4 provides detailed information about creating or modifying system configuration files and system-process prefix files, which are required inputs to the build cycle. The chapter also describes the contents of the several system-process object module libraries.

Chapters 5 to 7 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 those options, not covered in Chapter 3, when building applications for unusual target configurations that cannot be handled by the MPBUILD automated build procedure.

Chapters 8 and 9 contain information about loading application images. Chapter 8 provides an overview of application loading methods, and Chapter 9 describes the use of the COPYB utility program, which is required for preparing a bootable storage volume for use on a target system boot device.

Chapter 10 explains how to set the required characteristics of host/target serial lines to be used for down-line loading/debugging via PASDBG.

Appendix A explains how to build components of the MicroPower/Pascal file system and the Directory Service Process into an application. Appendix B describes the MPBUILD command procedure, which automates much of the application-building process for most applications.

Document Conventions

The following conventions are used in this manual:

- In interactive examples, user input appears in **boldface** type to differentiate it from system output.

- All user inputs other than control characters are terminated with a carriage return. The symbol used in this manual where needed to explicitly 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, pressing the CTRL and Z keys simultaneously produces the CTRL/Z character, which signals "no more input" to RSX-11 utility programs. Such key combinations are represented as <CTRL/x> -- for example, <CTRL/Z> or <CTRL/C>.

- In descriptions of command syntax, uppercase letters represent fixed elements, such as command names and option switches, which you must type as shown. Lowercase letters represent a variable, such as a file specification, for which you must supply a value. Square brackets ([]) enclose optional elements of a command; you can include the item in brackets or omit it as appropriate to the situation. The ellipsis symbol (...) represents repetition; you can repeat the item that precedes the ellipsis.

- The capital letter O and the number zero are represented as follows:

  letter O: O

  number Ø: Ø

- 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 a PDP-11 host system running under RSX-11M or RSX-11M-PLUS. MicroPower/Pascal-VMS software provides the development tools on a VAX/VMS host system that supports VAX-11 RSX. 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 capabilites 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 develop the application code in a high-level language, as a set of cooperating concurrent processes that employ semaphore-based constructs for synchronization and intercommunication. The language supported is a superset of ISO Pascal, with extensions for both real-time programming and modular implementation techniques. (The MACRO-11 assembly language can optionally be used for coding part or all of an application.)

A MicroPower/Pascal application does not require a conventional operating system for its target runtime environment. Instead, the user-coded processes are combined, in the target memory image, with the MicroPower/Pascal kernel -- a tailorable set of executive modules -- and with selected system processes such as device handlers. 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 runtime executive (kernel)

- Modern semaphore-based architecture for concurrent processing, permitting efficient multitasking 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 functionality, compatible with RT-11

1.1 OVERVIEW OF MICROPOWER/PASCAL COMPONENTS

MicroPower/Pascal software components comprise both host development tools and runtime 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/debug your applications in the target system. Most of the development tools -- those used to build the memory image -- are RSX-11 programs common to both MicroPower/Pascal-RSX and MicroPower/Pascal-VMS. These programs execute in compatibility mode on a VAX-11 host system.

The runtime 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 runtime executive support and device handling needed by user-written processes. MicroPower/Pascal runtime support is included in the target memory image, as needed by a given application, during the application build cycle.

The runtime system software is by nature specific to the target system and is identical across all host versions of the product. The commonality of runtime 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-11, VMS or RT-11 host environment.

1.1.1 Application Development Tools

The major MicroPower/Pascal development tools are the following:

- The MPPASCAL Extended Pascal compiler, which is an RSX-11 program that executes in compatibility mode on a VAX-11 host. The source language supported by the compiler is the same as that supported by the MicroPower/Pascal-RT version, and both compilers generate identical object code.

  The extended Pascal source language is described in the MicroPower/Pascal Language Guide. The compiler command interface is described in Chapter 2 of the present 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 "knows about" processes, Pascal data types, user-defined data types, kernel structures, and target system states. It permits debugging in terms of 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, that is, 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 per host system are described in Chapter 10 of the present manual.

- 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 MPPASCAL 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). In each step, one component of the application memory image -- the kernel, a system process, or a user process -- is built. That is, the component is merged, relocated as required for the target environment, and installed in the memory image file. Each utility provides options that permit various combinations of the following target system characteristics:

  - Mapped or unmapped memory
  - 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 -- either for debugging or for independent execution -- can be bootstrapped from a target storage device, or can be blasted into PROM for permanent installation in the target. (Note that software support for PROM blasting is not a component of the MicroPower/Pascal product but is available separately.)

  The build utilities are RSX-11 programs that execute in compatibility mode on a VAX-11 host system. Their direct use is described in Chapters 3, 5, 6, and 7. (As a convenient alternative to direct use, the build utilities can be invoked indirectly and "automatically" via the high-level MPBUILD 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 dialog with the user and generates a customized build-command file based on the user's 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 discussed further in Sections 1.2.2 and 3.5 and is described in detail in Appendix B.

- The COPYB utility, which produces a bootable storage volume to be used for loading an application from a target system device. COPYB is an RSX-11 program that executes in compatibility mode on a VAX-11 host system. The use of COPYB is described in Chapter 9.

1.1.2 Runtime System Software

MicroPower/Pascal runtime software is supplied primarily in the form of 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 runtime system components are the following:

- The kernel, the processor-specific, executive portion of the MicroPower/Pascal runtime 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 -- that allow processes to cooperate in a real-time fashion. 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, that is, tailorable to the requirements of a specific application, and 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. A kernel is built by merging a user-prepared system configuration file -- containing a set of configuration macro calls -- with one of the kernel libraries. The result of the merge is a configured kernel object module, which is then relocated and installed 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.)

  Chapters 1, 2, and 3 of the MicroPower/Pascal Runtime Services Manual provide a functional description of the kernel. Chapters 3 and 4 of the present manual describe the steps involved in building a kernel; Chapter 4 in particular describes the system configuration macros and their use.

- MicroPower/Pascal device handlers, also called drivers, are system processes that provide hardware-level support for target I/O devices and device interfaces. The handler processes are supplied in object form, one handler per module, in three handler object libraries: DRVK.OLB for a KXT11-C peripheral-processor target system, DRVU.OLB for other unmapped target systems, and DRVM.OLB for mapped targets.

  You include in the target memory image only those handlers required for the application. A handler process is built by merging a DIGITAL-supplied prefix module file -- DLPFX.MAC for the DL handler, for instance -- with one of the handler libraries. The prefix module pulls the corresponding handler object module from the library and configures it with application-dependent hardware parameters: vector and CSR addresses, interrupt priority, and number of units per

controller, for example. The handler prefix modules contain user-modifiable default values for those parameters. After the merged handler module is relocated, the process image is installed in the memory image file.

Chapter 4 of the MicroPower/Pascal Runtime Services Manual provides a functional description of the device handlers. Chapters 3 and 4 of the present manual describe the steps involved in building a handler process, Chapter 4 in particular describes the prefix module files and their use.

- The Directory Service Process (DSP) is a part of the optional MicroPower/Pascal target file system. This system process provides RT-11-compatible file and directory services for block-structured storage devices. The DSP must be included in the application image if any user process does I/O on such a device (other than by direct requests to a device handler). The DSP is modular with respect to the file and directory maintenance functionality it can provide; for example, file deletion or renaming, and directory initialization or display are optional functions. The DSP is supplied as an object library, with one or several functions per library module. Because the DSP is implemented in Pascal, the DSP library is provided in four versions: DSPNHD.OLB, DSPEIS.OLB, DSPFIS.OLB, and DSPFPP.OLB, corresponding to the four possible instruction sets implemented by the target hardware.

  You build a DSP process by merging the prefix module file DSPPFX.MAC with one of the DSP libraries and with the corresponding version of the Pascal OTS library, LIBxxx.OLB. The prefix module pulls the appropriate combination of object modules from the library as determined by the functional options specified in the prefix module. By default, the DSPPFX module configures a full-function DSP. You can modify the DSPPFX.MAC file prior to assembling and merging it, to eliminate unneeded functionality and thus reduce the size of the process.

  Chapter 5 of the MicroPower/Pascal Runtime Services Manual describes the MACRO-11 user's interface to functions performed by the DSP. (The Pascal user's interface is implicit, consisting of the OPEN procedure and other related I/O procedures.) Chapter 3 and Appendix A of the present manual describe the steps involved in building the DSP; Appendix A in particular describes the DSP prefix module and its use.

- Per-process file system support consists of object-library modules that are included in individual user static processes. The compiler or the assembler generates references to the merge-time file system support routines as needed. These routines both supplement and interact with the DSP and are supplied in two object libraries: FSLIB.OLB for processes implemented in Pascal and MACFS.OLB for processes implemented in MACRO-11.

  In general, this support is required by any Pascal program that uses the OPEN procedure and related I/O operations. The support is also needed by any MACRO-11 program that uses the QIO$ or PARSE$ macros. When a user process requiring file system support is built, the appropriate library, FSLIB or MACFS, must be specified in the MERGE step in order to correctly satisfy references made to support routines in the user program module. See Chapter 3 and Appendix A for further information.

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 -- to reflect a target hardware configuration change, for example -- the application image file must be totally rebuilt, beginning with the kernel.)

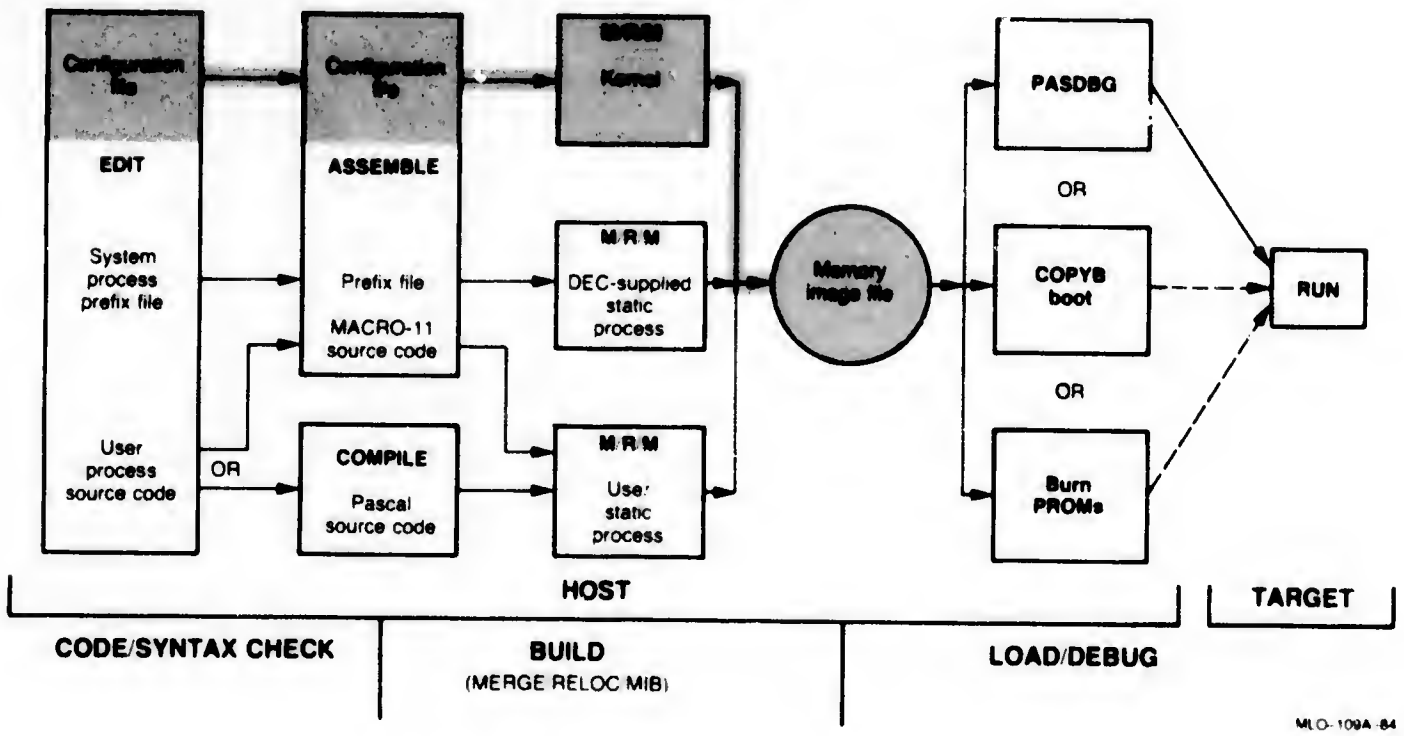

Figure 1-1 Kernel Build Phase

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 per system process in any total build cycle or any partial build cycle starting with this phase. For example, if the application requires three system processes -- two standard I/O device handlers and a line-clock process -- the system-process build phase consists of a 3-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.

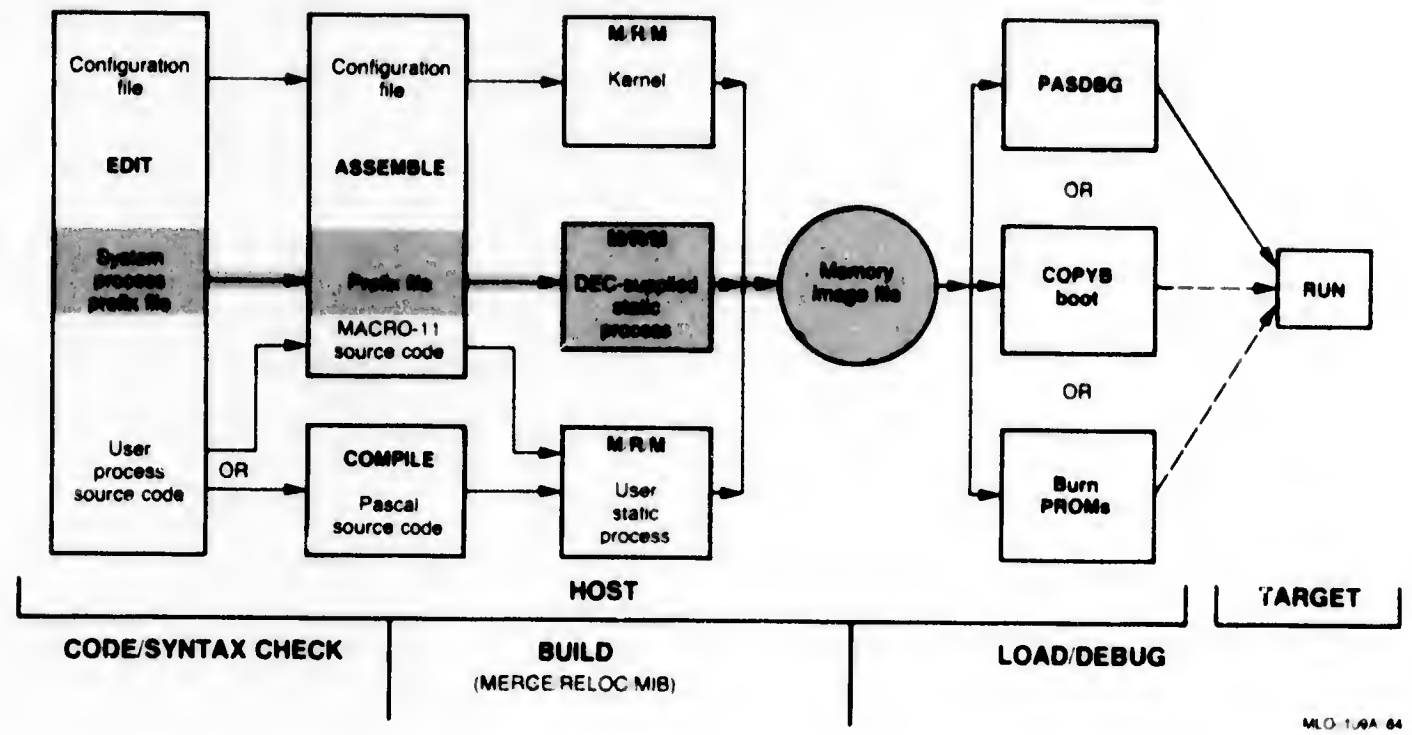

Figure 1-2 System Process Build Phase

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 per 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."

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

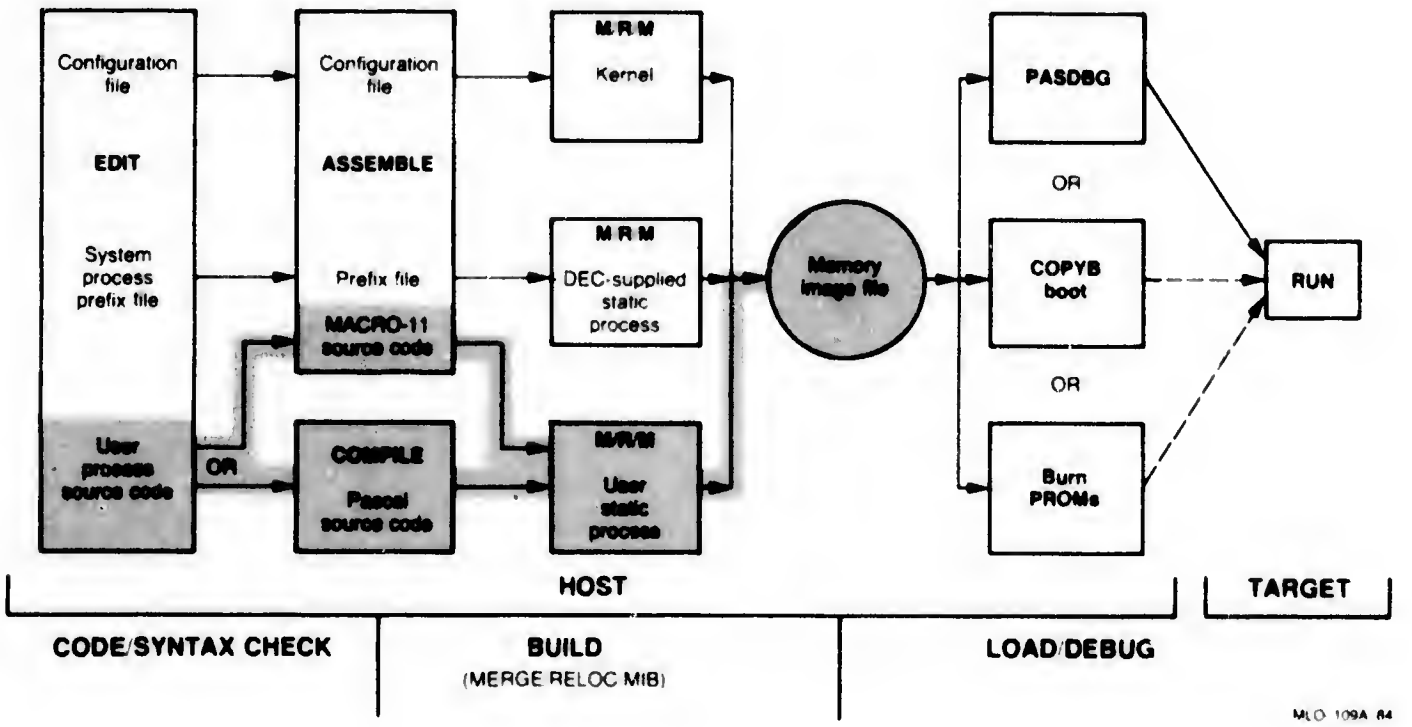

Figure 1-3 User Process Build Phase

1.2.2 Relationship of MPBUILD to the Build Cycle

As shown in Figures 1-1 through 1-3, each step through a 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 analogous 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, as described in Chapter 3, ordinarily you will not need to invoke them directly -- a highly repetitive and laborious procedure. Instead, you can use the MPBUILD command procedure to automate most of the application building process.

When invoked, the MPBUILD procedure conducts an interactive question/answer sequence, or dialog, with the user. As requested by the user during the dialog, MPBUILD indirectly "performs" a complete build cycle or a selected portion of the build cycle. More accurately, MPBUILD generates an intermediate command file that, when executed, invokes the many operations required for a full or a partial build -- repeated compilations or assemblies, merges, and so on. 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 "live examples" of application building.

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

1. A system configuration file, for the kernel build phase
2. One or more device-handler or file-system process prefix modules, for the system process build phase
3. 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; that is, the corresponding compilation or assembly step can be bypassed for any given input module.

The several DIGITAL-supplied libraries required for building any given component of the image are automatically "supplied" in the command lines generated by MPBUILD. That is, you do not specify any standard system inputs in the MPBUILD dialog. MPBUILD is versatile enough to be used for building most applications, with very few exceptions. See Appendix B for 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 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 is not a part of the MicroPower/Pascal product but is available separately.) Since only very limited possibilites 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.

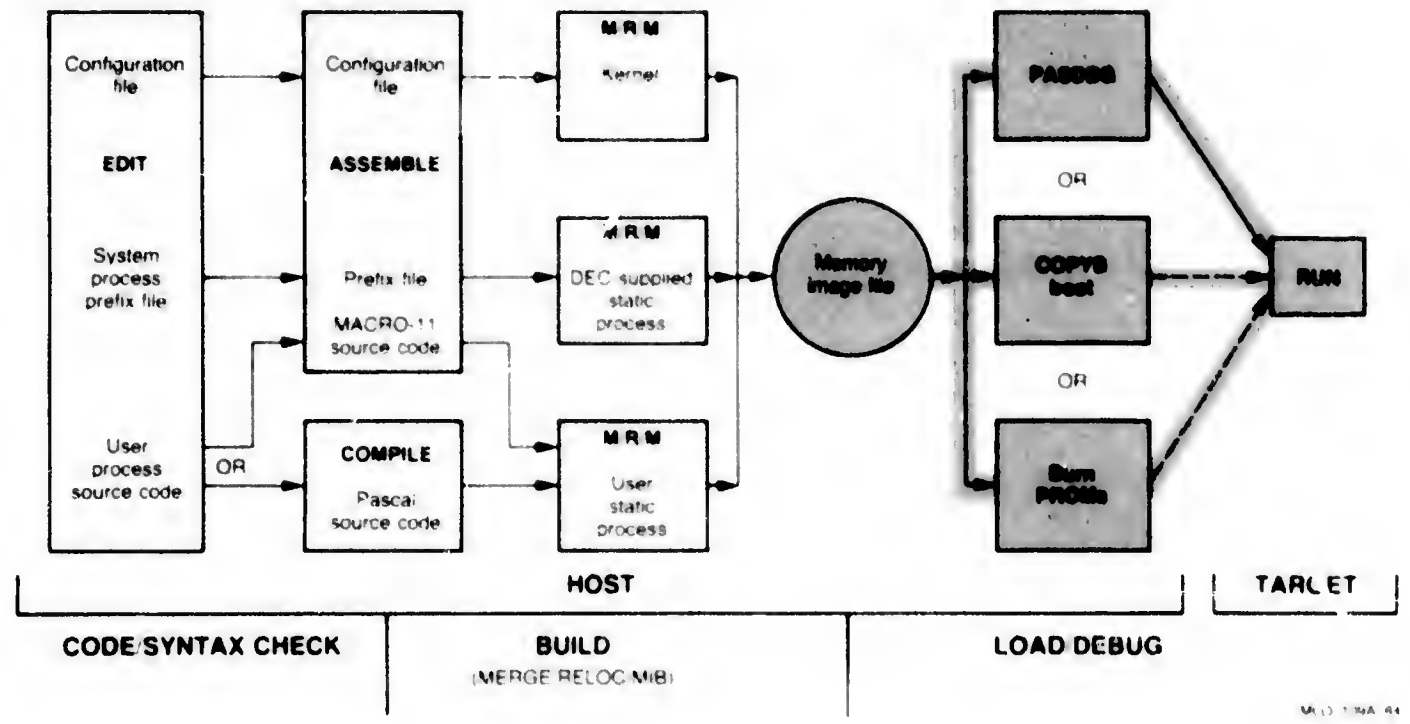

Figure 1-4 Application Image Loading

1.3 LOGICAL DEVICES FOR MICROPOWER/PASCAL FILES

1.3.1 Logical Device MP: for MicroPower/Pascal-RSX

On an 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 -- configuration and prefix source files, 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: will point to the same physical device.

1.3.2 Logical Device MICROPOWERSLIB for MicroPower/Pascal-VMS

On a VAX/VMS host system, the system logical name MICROPOWERSLIB identifies the logical device -- physical device and directory -- on which all MicroPower/Pascal-VMS files reside. By installation default, MICROPOWERSLIB is defined as a subdirectory emanating from SYSSLIBRARY.

1.4 USER SET-UP PROCEDURE FOR VAX/VMS

Before using any of the MicroPower/Pascal facilities under VAX/VMS, you should execute the command file MICROPOWERSLIB:MPSETUP.COM. The MPSETUP command procedure establishes logical definitions for the MPxxxx "command" symbols, such as MPPASCAL, MPBUILD, and MPMERGE, described in later chapters, and for PASDBG. These definitions allow you to the invoke, in a convenient, shorthand fashion, the MicroPower/Pascal-VMS executable components that reside in MICROPOWERSLIB. For example, you can invoke the MicroPower/Pascal compiler simply with the symbol MPPASCAL, which is defined as RUN MICROPOWERSLIB:MPP.

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:

```
$ @MICROPOWERSLIB:MPSETUP
```

CHAPTER 2

MICROPOWER/PASCAL COMPILER OPERATION

The MicroPower/Pascal compiler, MPPASCAL, 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 MPPASCAL 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 of 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. Each of these libraries corresponds to one of the instruction set options described in Table 2-1. All Pascal object modules must be merged with one of these libraries in addition to any special support library that might be required, such as FSLIB.OLB or RHSLIB.OLB.

2.1 FILE SPACE REQUIREMENTS

During a compilation, the compiler opens four temporary files for use as intermediate working space. 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. That approximation ordinarily includes the space needed for both an object and a listing file, but an unusually large listing might impose an additional requirement.

2.2 COMPILER INVOCATION AND COMMAND LINE

For a PDP-11 RSX-11M/M-PLUS Development System: Assuming that MPPASCAL has been installed according to installation-procedure defaults, you invoke it by the task name MPP, as follows:

```
>[MCR] MPP
MPP>
```

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

For a VAX-11 RSX (Compatibility Mode) Development System: If you have executed the MPSETUP.COM procedure, you can invoke the compiler by the logical symbol MPPASCAL, as follows:

```
$ MPPASCAL
MPP>
```

(Section 1.4 describes the MPSETUP command procedure.)

Most of the examples in this chapter show MPP as the invocation name; use MPP or MPPASCAL as appropriate to your host system. Following the compiler's command line prompt, MPP>, you can enter a command line specifying the input and output files and any needed options, or you can specify a command file that contains the required command line(s) -- see Section 2.2.2.

2.2.1 Compilation Command Line Syntax

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 at least one output file (object or listing) must be specified. Except as noted in Section 2.2.2, a standard RSX-11 or VMS file specification is accepted, as appropriate to your host system. Standard defaults are supplied for device, directory, and version number. The file type default is OBJ unless the /MA option is used.

listing-file
: A file specification for the listing output file. The listing file is optional, but at least one output file (object or listing) must be specified. Except as noted in Section 2.2.2, a standard RSX-11 or VMS file specification is accepted, as appropriate to your host system. Standard defaults are supplied for device, directory, and version number. If a device or a directory is specified for the object file, however, the same value is applied as a default for the corresponding field of the listing file specification. The file type default is LST.

source-file
A file specification for the input source file. Except as noted in Section 2.2.2, a standard RSX-11 or VMS file specification is accepted, as appropriate to your host system. Standard defaults are supplied for device, directory, and version number. The file type default is PAS.

option-list
One or more compilation options, or command switches, specified in the following form:

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

Table 2-1 summarizes the command-line switch names and values and lists the corresponding source code option, if any. Section 2.3 describes each option in detail. Only the first two characters of a switch name are significant in a command line. Switch values are always three characters. Several examples of switch 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
```

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.LIS = A**

   Compile source file A.PAS and produce listing file C.LIS 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 2-1.)

Table 2-1
Compilation Options

Switch Name / Value	Corresponding Source Option	Purpose
CHeck:		Generate runtime checks for:
IND	INDEXCHECK	Array index values
MAT	MATHCHECK	Division by Ø
POI	POINTERCHECK	Null pointer value
RAN	RANGECHECK	Variable value range
STA	STACKCHECK	Stack overflow
DEbug	(none)	Include symbol information for PASDBG debugger
EXtra-stats	(none)	List extended compilation statistics
FIlter-dcls	(none)	Filter out (discard) unused source declarations
INstructions:		Generate instructions for:
EIS	(none)	EIS hardware option
FIS		FIS hardware option
FPP		FPP hardware option
LS2		LSI-11/2 with ROM and EIS
NHD		No special hardware
MAcro	(none)	Generate MACRO-11 output code
NOpredfl	(none)	Disable real-time definition file PREDFL.PAS
STandard	STANDARD	Flag nonstandard features
(none)	NOLIST	Disable source listing
	LIST	Enable source listing

Note: The significant characters of a switch name are shown in uppercase. Any additional characters are optional and are ignored by the compiler.

2.2.2 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 VMS 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.

5. One source file is accepted per compilation; the file must contain one complete compilation unit, that is, one PROGRAM or MODULE.

6. If the /MA switch is specified, the "object" output file will contain MACRO-11 assembly code instead of binary object code, and the file type defaults to MAC.

7. If the /FI switch 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, as in the following VAX-11 RSX example using lengthy device/directory specifications:

```
$ MPPASCAL
MPP> DRA1:[JOHNJONES.MPPPROJECT]FOOBAR, FOOBAR -<RET>
MPP> = DRA1:[JOHNJONES.MPPPROJECT]FOOBAR/CHECK:IND-<RET>
MPP> :RAN/DEBUG/EXTRA/INSTR:FIS/STAND<RET>
```

9. A command line file can be specified (@file-spec) in response to the MPP> prompt in place of a direct command line. The default file type for a command file specified at MPP> level is .CMD, regardless of the host system. (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 per invocation.

11. A comment line is accepted in response to an MPP> prompt. A comment line is indicated by a leading semicolon (;). The compiler ignores the entire line and prompts for more input. Comment lines are useful for documenting a command file. (A null line -- RETURN only -- is also accepted; the compiler responds with a software version number and a prompt.)

2.3 COMPILATION OPTIONS

The compilation options are summarized in Table 2-1. As indicated in the table, some options can be requested only in the command line, some can be requested in either the command line or the source code, and one (LIST/NOLIST) can be requested only in the source code. The syntax for option switches specified in the command line is described in Section 2.2.1. The syntax and use of options specified in the source code are described below.

The semantics of both command-line and source-code options are given in Section 2.3.2.

2.3.1 Syntax and Use of Source Code Options

Unlike command line switches, source code 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 runtime checking code only for selected procedures. The source code option names are as follows:

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

Source code 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:

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

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

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

Invalid examples:

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

NOTE

> The source 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 code. For example, if MATHCHECK appears twice without an intervening NOMATHCHECK, NOMATHCHECK must appear twice in order to effectively disable the option.

2.3.2 Option Semantics

Options specified in the command line affect the entire compilation unit. Options specified in the source code affect only the portion of the compilation unit in which the option is enabled. All compilation options except /IN:NHD and LIST are initially disabled.

2.3.2.1 **Runtime Checking Code (/CHeck:xxx)** - The /CH:xxx switches (source code xxxxCHECK) cause the compiler to include code in the object module to check for various 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 indices 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. (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 runtime checks are useful for program debugging. Use of any checking option causes the generated code to be larger than it would otherwise be. The checking options are disabled by default.

2.3.2.2 **Debug Symbol Information (/DEbug)** - The /DE switch causes the compiler to include symbol-definition information in the object file for debugging purposes. The debug option allows symbolic debugging of the compiled program using 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.

2.3.2.3 **Extended Statistics (/EXtra)** - The /EX switch causes the compiler to provide 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.

2.3.2.4 **Filter Unused Declararation (/FIlter-decls)** - The /FI switch causes the compiler to filter out, or discard, 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 I 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 switch may permit it to compile sucessfully.

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

1. 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.

2. An identifier is considered used if it appears in the formal parameter list of a subprogram that is used.

3. A subprogram is considered used if there is a possible program path from the main level or from a used subprogram to that subprogram.

4. A type identifier is considered used if it is subordinate to a used constant, variable, or subprogram identifier.

5. A type identifier is considered used if it is a scalar type and if one or more of its enumerated elements is used.

6. A variable or a subprogram declared with the GLOBAL or INITIALIZE attribute is considered used.

7. All outer-level variable declarations of the compilation unit are considered used if the compilation unit has the OVERLAID attribute.

8. 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 switch, 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 will produce either of two kinds of listing, depending on the results of the compilation. If one or more syntax errors are encountered, the compiler will produce an abbreviated error listing identifying the error(s) detected. (Other errors may be present and, if so, will show up when the corrected version is recompiled.) If the program compiles correctly, however, the compiler produces a listing in which all filtered identifiers are indicated by the annotation "*** IDENTIFIER(S) NOT REFERENCED" in combination with up-arrow (^) characters. Note that any unused declarations supplied by PREDFL.PAS (Section 2.3.2.7) are effectively filtered but are not reported in the summary of unreferenced declarations, since declarations from PREDFL.PAS never appear in a listing.

2.3.2.5 **Instruction Set (/INstr:xxx)** - The /IN:xxx switch indicates the class of optional instructions, if any, that the compiler can use in the generated code. Indirectly, the switch 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 inherent in 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-C 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.

2.3.2.6 **MACRO-11 Output Code (/MAcro)** - The /MA switch causes 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. The MACRO-11 option is disabled by default.

2.3.2.7 **No Real-Time Predefinitions (/NOpred)** - The /NO switch 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. This option is disabled by default.

2.3.2.8 **Standard Pascal Only (/STandard)** - The /ST switch (source code STANDARD) causes the compiler to issue 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. This option is disabled by default.

2.3.2.9 **Selective Listing Control (NOLIST and LIST)** - The source code options NOLIST and LIST allow you to suppress the listing of selected parts of the source program. NOLIST causes the compiler to discontinue listing of the source file at the point at which NOLIST is encountered. A subsequent occurrence of LIST causes the compiler to resume listing. These options are often used bracketing a %INCLUDE directive, to suppress listing of the included file, as follows:

```
    ...

(*$NOLIST*)
%INCLUDE 'some-file-spec'
(*$LIST*)
    ...
```

Because the NOLIST and LIST options operate on a counter -- as opposed to a binary switch -- associated with listing output, they should be used in symmetric pairs in the order stated. Note in particular that if LIST is specified prior to NOLIST, NOLIST must be specified twice to effectively disable listing output, since listing is initially enabled.

If a listing file is not specified in the command line, the listing control options are ignored.

2.4 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 switch in the command line and specify both an object and a listing file, the listing will include 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 switch but omit the listing file specification, the compiler will display the lines-per-minute compilation statistic at your terminal.)

Also, use of the /FI (filter declarations) switch with a listing file specification will affect the form of listing file produced, as described in Section 2.3.2.4.

Figures 2-1 and 2-2 show two standard listings illustrating the common characteristics of an MPPASCAL listing file. Figure 2-1 shows the listing of a program containing syntax errors that was compiled without the /DE (debug) switch. Figure 2-2 shows a listing of the same program but without errors and compiled with the /DE switch. 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. Statement numbers, if any (DEBUG compilations only)
8. Error diagnostics, if any, and summaries
9. Elapsed time of compilation and average speed
10. List of options selected

```
                ERROR SUMMARY
(8)
 Line    26 - Undefined identifier
 Line    31 - Undefined identifier

(1) SUGEXAM
(2) 15:16:37  28-Nov-83 Monday (3) PASCAL V#1.05  Page 1-1
(4) File: SUGEX1

      Line          Source
      ------------------
   (5) 1 (1)        [ SYSTEM(MICROPOWER), DATASPACE(2000),
       2 (2)          STACKSIZE(200), PRIORITY (25) ] PROGRAM SUGEXAMP;
       3 (3)
       4 (4)        VAR
       5 (5)          Play, Esc : CHAR;
       6 (6)          I, J : INTEGER;
       7 (7)
   (6) 8 (8)        %INCLUDE 'change.pas'
   2   9 (1)        (* Include file with definition of Procedure Changecharacteristics *)
   2  10 (2)
   2  11 (3)        PROCEDURE Changecharacteristics;
   2  12 (4)        BEGIN
   2  13 (5)          WRITE (Esc, '<',                  (* Enter ANSI mode *)
   2  14 (6)                 Esc, CHR (91), '?81',      (* Turn off auto-repeat *)
   2  15 (7)                 Esc, CHR (91), '?21');     (* Enter VT52 mode *)
   2  16 (8)        END;
      17 (9)
      18 (10)       BEGIN
      19 (11)         Esc := CHR (155);
      20 (12)         Changecharacteristics;
      21 (13)         Play := 'y';
      22 (14)         WHILE (Play = 'Y') OR (Play = 'y') DO
      23 (15)           BEGIN
      24 (16)             WRITE (Esc, 'H', Esc, 'J');
      25 (17)             WRITE ('Enter a character, please: ');
      26 (18)             READLN (Let);
                                 84
(8)
*** 84: Undefined identifier
      27 (19)             WRITE (Esc, 'H');
      28 (20)             FOR J := 1 TO 19 DO
      29 (21)               BEGIN
      30 (22)                 FOR I := 1 TO 79 DO
      31 (23)                   WRITE (Let);
                                      84
(8)
*** 84: Undefined identifier
      32 (24)                 WRITELN;
      33 (25)               END;
      34 (26)             WRITE ('Go again ? Y/N ');
      35 (27)             READLN (Play);
      36 (28)           END;
      37 (29)       END.

(8) There were 2 lines with errors diagnosed.

(9) Compilation required 3 seconds.
    Average compilation speed was 744 lines/min.

    Options Selected

(10) No Special Instructions
     No Real-Time Pre-definitions
```

Figure 2-1 Compilation Listing: Program with Errors, No /DE Switch

```
(1) SUGEXAM
(2) 15:18:18  28-Nov-83 Monday  (3) PASCAL V01.05  Page 1-1
(4) File: SUGEX2

      Line  Stmt   Source
      ------------------
   (5) 1  (1)          [ SYSTEM(MICROPOWER), DATASPACE(2000),
       2  (2)            STACKSIZE(200), PRIORITY (25) ] PROGRAM SUGEXAMP;
       3  (3)
       4  (4)          VAR
       5  (5)            Play, Esc, Let : CHAR;
       6  (6)            I, J : INTEGER;
       7  (7)
   (6) 8  (8)          $INCLUDE 'change.pas'
   2   9  (1)          (* Include file with definition of Procedure Changecharacteristics *)
   2  10  (2)
   2  11  (3)          PROCEDURE Changecharacteristics;
   2  12  (4)          BEGIN
   2  13  (5)       (7)1   WRITE (Esc, '<',                  (* Enter ANSI mode *)
   2  14  (6)                     Esc, CHR (91), '?81',      (* Turn off auto-repeat *)
   2  15  (7)                     Esc, CHR (91), '?21');     (* Enter VT52 mode *)
   2  16  (8)         2 END;
      17  (9)
      18  (10)         BEGIN
      19  (11)      1    Esc := CHR (155);
      20  (12)      2    Changecharacteristics;
      21  (13)      3    Play := 'y';
      22  (14)      4    WHILE (Play = 'Y') OR (Play = 'y') DO
      23  (15)             BEGIN
      24  (16)      5        WRITE (Esc, 'H', Esc, 'J');
      25  (17)      6        WRITE ('Enter a character, please: ');
      26  (18)      7        READLN (Let);
      27  (19)      8        WRITE (Esc, 'H');
      28  (20)      9        FOR J := 1 TO 19 DO
      29  (21)                 BEGIN
      30  (22)     10            FOR I := 1 TO 79 DO
      31  (23)     11                WRITE (Let);
      32  (24)     12            WRITELN;
      33  (25)                 END;
      34  (26)     13        WRITE ('Go again ? Y/N ');
      35  (27)     14        READLN (Play);
      36  (28)             END;
      37  (29)     15 END.

(8) There were no lines with errors diagnosed.

(9) Compilation required 10 seconds.
    Average compilation speed was 222 lines/min.

    Options Selected

(10) Debugging
     No Special Instructions
     OBJ Code
     No Real-Time Pre-definitions
```

Figure 2-2 Compilation Listing: Errors Removed, /DE Switch Used to Show Statement Numbers

CHAPTER 3

INTRODUCTION TO APPLICATION BUILDING

A MicroPower/Pascal application memory image consists of three kinds of software:

- The DIGITAL-supplied kernel, which provides basic runtime support and services for all processes in the application. The kernel is organized so that it can be tailored to application requirements during the build procedure. In addition to modular code and data, the kernel includes a user-configured memory area from which it allocates system data structures for processes: PCBs, semaphores, ring buffers, message packets, and so on. (See Chapters 1 and 2 of the MicroPower/Pascal Runtime Services Manual.)

- DIGITAL-supplied static processes, called system processes, which provide device-handling services for user processes. The system processes include I/O device handlers (drivers), the clock service process, and the Directory Service Process (DSP), a part of the file system. A typical application might contain three or four system processes. (See Chapter 1 of the MicroPower/Pascal Runtime Services Manual.)

- The user-written static processes that constitute the application-specific portion of the target software.

All these components must be combined into a single memory image that can be loaded, executed, and debugged. You use the build utilities -- MERGE, RELOC, and MIB -- to accomplish that. Although you can run the individual utilities yourself, as described here and in Chapters 5 through 7, a supplied command procedure, MPBUILD, automates much of the process. For building most applications, DIGITAL suggests that you use MPBUILD. The present chapter and Chapter 4 provide the necessary background information. Section 3.5 contains a brief discussion of automated versus "manual" application building, and Appendix B provides details on running MPBUILD.

NOTE

The basic unit, or building block, of the build procedure is a static process, which may incorporate any number of dynamic subprocesses. (The kernel is also one build unit.)

INTRODUCTION TO APPLICATION BUILDING

In each phase of the build cycle, you merge, relocate, and install in the memory image one component of the application: first the kernel and then each system or user static process. The major functions of the build utilities are as follows:

1. MERGE -- Logically combines separately assembled or compiled object modules and any required library modules into one merged object module (.MOB file type). The merged object module represents either the kernel or a static process.

   MERGE resolves references between input modules and selects needed modules from object libraries. When merging a process, MERGE also satisfies references made within the process to kernel entry points, by means of a kernel symbol table (.STB) file created by RELOC during the kernel build phase.

2. RELOC -- Relocates the code and data segments contained in a merged object module to produce a process image (.PIM) file.

   RELOC physically combines and reorders the individual zero-origined program sections, grouping them according to their read-only (RO) or read/write (RW) attribute. RELOC groups all RO program sections into one RO segment and all RW sections into one RW segment. The RW segment follows the RO segment. (The program sections are ordered alphabetically within each segment.) RELOC then assigns appropriate physical (for unmapped case) or virtual (for mapped case) addresses to the program sections. RELOC provides many user-controlled relocation options, primarily for mapped and ROM/PAM target environments.

   Like the .MOB file, the .PIM file represents one piece of the application -- either the kernel or a static process -- but in a form the MIB utility can use. RELOC optionally produces a symbol table (.STB) file for each relocated component. MERGE uses this file to satisfy process-to-kernel references, and MIB uses it to produce debug-symbol information.

3. MIB (Memory Image Builder) -- In a kernel build phase, MIB initializes a memory image (.MIM) file and installs the kernel image in the .MIM file. In subsequent phases, MIB successively installs static process images (.PIM file contents) in the .MIM file to produce a complete, executable memory image. MIB also produces an optional debug-symbol (.DBG) file from information contained in the .STB files created by RELOC.

Table 3-1 summarizes the principal functions, inputs, and outputs of the build utilities. Figure 3-1 illustrates the operations the build utilities perform with respect to their primary input and output files. (The relocation shown in Figure 3-1 applies to an unmapped target system only.)

Table 3-1
Build Utility Programs

Utility Function	Input File Type	Input File Description	Output File Type	Output File Description
MERGE -- Creates a single object module from several input object modules; resolves intermodule references; logically concatenates program sections	.OBJ	Object modules from Pascal compilations, MACRO-11 assemblies	.MOB	Merged object module
			.MAP	Section map file
	.STB	Symbol table file	.AUX	Auxiliary output file
RELOC -- Combines and sorts program sections by read-only and read/write attribute and orders the program sections alphabetically in each category; relocates each program section; relocates symbol information and places it in a symbol file	.MOB	Merged object module	.PIM	Process image file
	.MIM	Memory image file (unmapped only)	.MAP	Memory map file
			.STB	Symbol table file
MIB -- Constructs a file containing an executable memory image consisting of a kernel and one or more static processes; constructs optional debug file	.PIM	Process image file	.MIM	Memory image file
	.MIM	Memory image file	.MAP	Memory image map file
	.STB	Symbol table file	.DBG	Debug file

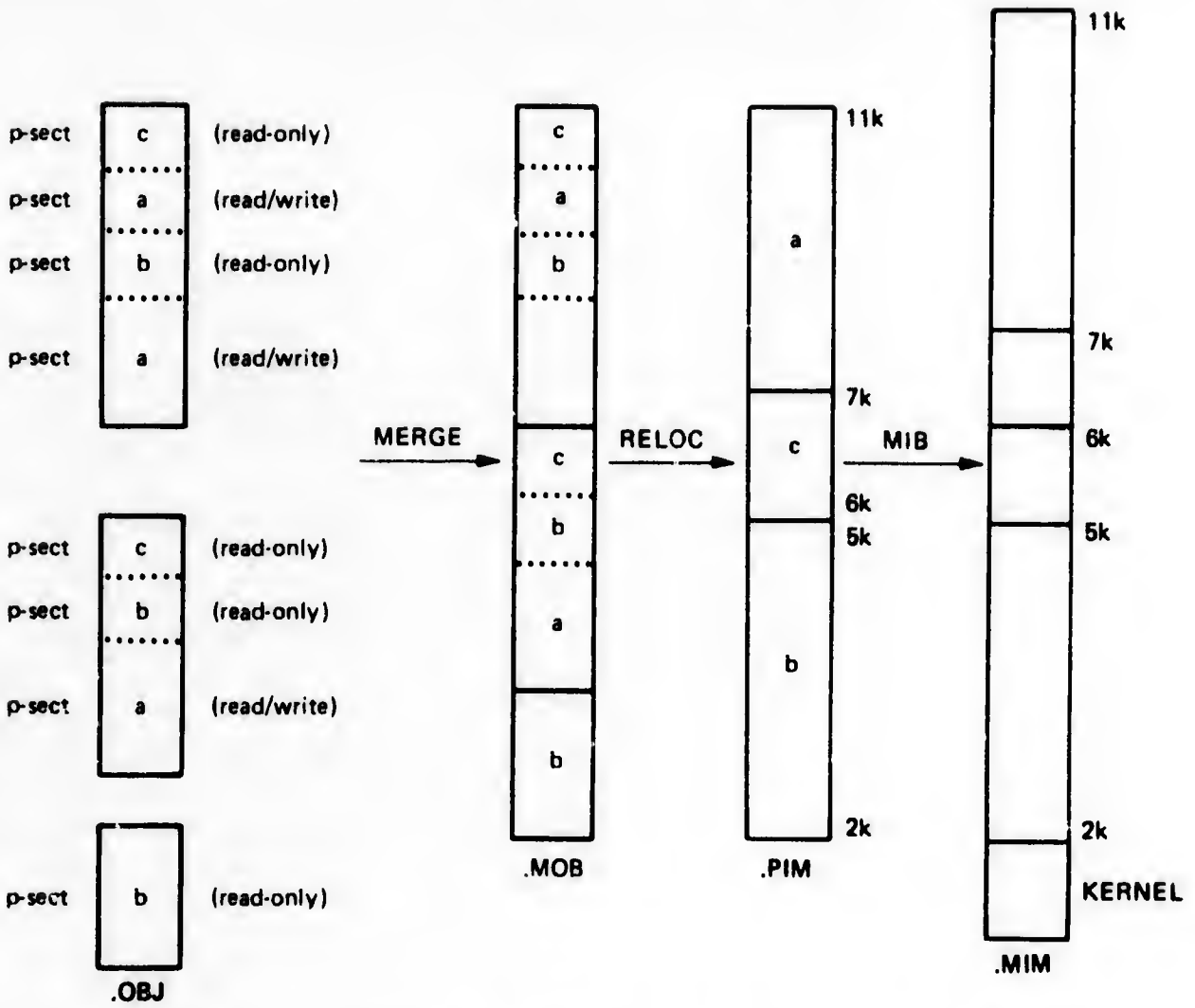

Figure 3-1 Input and Output Files for Build Process

3.1 BUILD CYCLE OVERVIEW

Application building begins when you have developed your application programs to the point of initial testing on the target system. Now an application image must be built for loading and execution under control of the symbolic debugger, PASDBG.

A typical development effort consists of many build/rebuild cycles. The following list summarizes the phases and steps involved in one build cycle:

1. Build the kernel

    a. Create or edit a system configuration file and assemble the file
    b. Merge the configuration file with the kernel library
    c. Relocate the kernel and create the kernel symbol table
    d. Construct a memory image file with kernel installed

2. Build DIGITAL-supplied system processes

   a. Edit prefix file for required handler or clock process and assemble or compile prefix file
   b. Merge prefix file with driver library and kernel symbol table from phase 1
   c. Relocate system process
   d. Insert system process in memory image file
   e. Repeat this phase for each additional system process required

3. Build user-written static processes

   a. Create static process source file(s) and compile or assemble source file(s)
   b. Merge process file(s) with OTS library, if any, and the kernel symbol table from phase 1
   c. Relocate static process
   d. Insert static process in the memory image file
   e. Repeat this phase for each additional user process

4. Debug, then return to phase 3, 2, or 1, as required, to rebuild

Sections 3.2 to 3.4 describe each build phase in detail.

An overview of just the first two phases of a build cycle is presented here, prior to the detailed descriptions, to give a feeling for the use of MERGE, RELOC, and MIB. The overview example consists of an initial application build for a mapped target system (an LSI-11/23), incorporating debugger support in the application. Assume that one of the system processes the application requires is the RX02 (DY) device handler. You build the kernel in phase 1; the DY handler process in phase 2.

3.1.1 Phase 1: Build the Kernel

The kernel build phase consists of four steps. First, you create and assemble a configuration file for your application. Second, you merge the configuration file with the kernel module library. Third, you relocate the kernel object module. Fourth, you use MIB to install the kernel in a new memory image file.

STEP 1: You edit and assemble a system configuration file for your specific application needs, as described in Chapter 4. This file, consisting of a set of macro calls, describes your target hardware and specifies a kernel software configuration for your application. The kernel configuration you specify for the initial build cycle is necessarily tentative, to be refined by trial and error in subsequent build cycles.

You normally use one of the configuration files supplied in the kit as a starting point for editing your initial configuration file. For this example the appropriate base file would be CFDMAP.MAC, "DMAP" indicating "debug support and mapped." After editing, you assemble your customized configuration file -- named KERNL1.MAC, for example -- with the COMM.MLB system macro library to produce a KERNL1.OBJ file.

STEP 2: You merge the KERNL1.OBJ file with PAXM.OLB, the mapped version of the kernel object module library. The resulting merged kernel object module, KERNL1.MOB, is now tailored for your application. You must specify the debug option (/DE) in the MERGE command line so that MERGE will include special debug-symbol information in the .MOB file.

STEP 3: You invoke RELOC with KERNL1.MOB as the input file and specify a kernel image file and a symbol table file as primary outputs, named KERNL1.PIM and KERNL1.STB. You again must specify the /DE option in the RELOC command line so that RELOC will pass along, via the .STB file, the debug-symbol information included by MERGE. Also, because you are building a mapped application, you must use a special relocation option to force correct mapping of the kernel's read/write (data) segment. (See Section 3.2.3 for details.) RELOC relocates the kernel, extracts and writes symbol-definition information to the .STB file, and writes the kernel image to the .PIM file. Information in the .STB file is used both by MIB in step 4 and by MERGE in subsequent phases.

STEP 4: You invoke MIB with KERNL1.PIM and KERNL1.STB as inputs and specify memory-image and debug-symbol files as outputs, named APPLC1.MIM and APPLIC.DBG, for example. You must specify the /KI (kernel installation) option in the MIB command line to indicate that the input PIM file contains a kernel image rather than a process. Doing so causes MIB to create and initialize an "empty" memory image file and to install the kernel image in it. Because a .DBG file was requested, MIB creates and initializes a debug-symbol (.DBG) file, updates the debug-symbol records from the KERNL1.STB file, and writes them to the .DBG file.

3.1.2 Phase 2: Build the RX02 (DY) Device Handler Process

The DY handler build phase consists of four steps. First, you edit and assemble the prefix file for the handler. Second, you merge the prefix module with the appropriate handler object library. Third, you relocate the DY handler object module. Fourth, you install the handler image in the memory image file. These steps are for a handler implemented in MACRO. If the handler were implemented in Pascal, you would use the Pascal compiler instead of the MACRO assembler and merge the handler prefix module with both the handler library and a Pascal OTS library (LIBxxx.OLB).

STEP 1: You edit the DY handler prefix file, DYPFX.MAC, as needed to match your target hardware configuration, as described in Chapter 4. Then you assemble DYPFX.MAC with COMM.MLB, the mapped version of the system macro library, to obtain a DYPFX.OBJ file.

STEP 2: You merge the DYPFX.OBJ file and the KERNL1.STB file, produced in phase 1, with DRVM.OLB, the version of the handler object library for mapped target systems. This merge produces an object file containing the DY handler process, which you can name DYHAND.MOB.

MERGE uses information in the kernel .STB file to satisfy kernel symbol references made by the process being merged. You do not use the /DE option in this build phase, since system-process symbol information is not normally useful for application debugging.

STEP 3: You invoke RELOC with DYHAND.MOB as the input file and get a process image file, DYHAND.PIM, as output. Because you are building a mapped application, you must use two special relocation options to force proper mapping of the handler's code and data areas according to the rules for driver-mapped processes. (See Section 3.3.3 for details.)

STEP 4: You invoke MIB with DYHAND.PIM and the existing memory image file, APPLC1.MIM, as input. You also specify an output MIM file, which can have the same name as the input MIM or a different one, such as APPLC2, for example. MIB allocates physical memory to the handler process based on next-available space in the memory image, sets up the process's mapping register (PAR) values, and installs the process image in a new, extended copy of the input MIM file. Thus, the resulting memory image consists of the kernel and the DY process. In subsequent phases, additional system and user processes will be built and added to the memory image, as required, in much the same way as described for the DY handler.

The following sections describe each build phase in considerably greater detail, providing a command format and several command examples for each step.

3.2 BUILDING THE KERNEL

The four steps involved in building the kernel image are described in detail below. 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.)

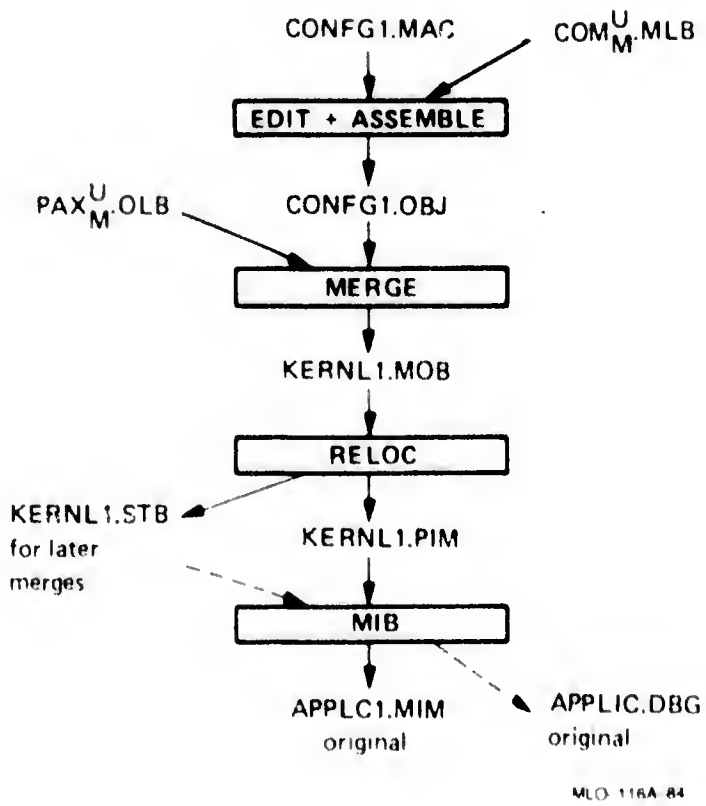

Figure 3-2 Build the Kernel

3.2.1 Create and Assemble 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 -- free-memory resources, primitive modules, trap processors -- and describe the target hardware configuration. The configuration macros are described in Chapter 4. The configuration file configures the kernel object module 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 "stand-alone" testing or use -- that is, without PASDBG.

As described in Chapter 4, you can create a configuration file for your application by modifying one of the sample configuration files included in the distribution kit. The subsequent command example assumes that your edited configuration file is named CONFG1.MAC and that it resides in your default directory.

The configuration source file must be assembled with one of the two MicroPower/Pascal system macro libraries: COMU.MLB for unmapped applications or COMM.MLB for mapped applications. You 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 this command line, x is U for an unmapped application or M for a mapped application. The MACRO-11 assembler will use the specified macro library, COMM.MLB or COMU.MLB, to satisfy the macro references in CONFG1.MAC.

In this example, the output of assembling the configuration file is an object file called CONFG1.OBJ.

NOTE

This and subsequent examples use the fictitious symbol "mpp-lib:" to stand for a device/directory specification for a MicroPower/Pascal library or other DIGITAL-supplied file. In your own command lines, you 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-11/RSX (compatibility mode) development system, the standard substitution for mpp-lib: would be "MICROPOWERSLIB:". (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 fictitious 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.

3.2.2 Merge Configuration Object File

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

General Command Format

```
>[MCR] MRG              (Native RSX-11)
or
$ MPMERGE               (VAX-11/RSX compatibility mode)

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-11 multiuser tasks with the standard task names determined by installation-procedure defaults. The VAX-11/RSX program invocation names, such as MPMERGE, also assume that you have executed the MPSETUP.COM procedure (Section 1.4). See Chapters 2, 5, 6, and 7 for further details about program invocation.

Default File Types

```
MRG> .MOB, .MAP, .AUX = .OBJ, .OBJ, ...
```

An object library file must be identified as such with the /LB (library) switch. The default type for a file specified with /LB is .OLB.

The "mobfile" in the command format represents MERGE's primary output, the merged object module (.MOB) file. See Section 5.3 for a full description of the MERGE command line.

The /DE switch must be specified in the command line if you will subsequently debug the application with the PASDBG symbolic debugger. The debugger needs special kernel symbol information that /DE causes MERGE to generate in the output object module.

The following command examples assume that the merged kernel object file is to be named KERNL1.MOB.

MAPPED

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

UNMAPPED

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

The CTRL/Z response to the second utility prompt shown in these examples causes an exit from the utility. (The second prompt will be omitted from subsequent utility command examples.)

NOTE

When you build a nondebug version of the application for down-line load and go, bootstrapping, or burning into PROM, your build commands will differ somewhat from those used in previous build operations. For instance, since debug symbols are not required, you omit the /DE switch from all command lines for the build cycle.

3.2.3 Relocate Kernel Module

Invoke the RELOC utility to relocate the kernel object module and produce a kernel image (.PIM) file and kernel symbol table (.STB) file. The symbol table file is required for all subsequent process MERGE steps and for producing the debug symbol (.DBG) file in the following MIB step.

General Command Format

```
     >[MCR] REL              (Native RSX-11)
or
     $ MPRELOC               (VAX-11/RSX compatibility mode)

     REL>[pimfile][,mapfile][,stbfile]=mobfile[,mimfile][/switches]
```

Default File Types

```
     REL> .PIM, .MAP, .STB = .MOB, .MIM
```

See Sections 6.3 and 6.5 for a full description of the RELOC command line and all applicable options.

The three RELOC command examples given below differ only in the type of command line switches that are required or appropriate for different target memory environments. (For simplicity, none of the examples specifies a relocation map file.)

MAPPED (RAM-only or ROM/RAM)

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

UNMAPPED RAM-only

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

UNMAPPED ROM/RAM

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

The first KERNL1 in each of the command lines names the output image file, KERNL1.PIM, and 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. The /DE switch directs RELOC to process the debug symbol information -- called internal symbol directory (ISD) records -- contained in the MOB file and to include that information in the STB file along with the kernel's global symbol definitions.

In a mapped application, the kernel's read/write memory (data) segment must be mapped by page address registers (PARs) 4, 5, and 6. (Memory segmentation and mapping conventions are described in Sections 2.1.6 and 2.1.7 of the MicroPower/Pascal Runtime Services Manual.) In the mapped command line example, 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. (As illustrated in Figure 3-1, RELOC physically combines and reorders program sections so that all read-only sections precede all read/write sections prior to their relocation.)

An unmapped RAM-only application has no special relocation requirements or constraints. Therefore, you do not need to use a relocation option in the RELOC command line for that case.

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, 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.

Note that the /DE switch is not appropriate for the ROM/RAM case, since PASDBG can be used only with an application image built for a RAM-only target.

3.2.4 Create Memory Image File

You invoke the MIB utility program 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 debug symbol (.DBG) file as output.

The /SM (short memory image) switch shown in this and subsequent MIB examples is discretionary but is generally recommended. Its effect is to limit the size of the .MIM file created in this step to the minimum needed for installing the kernel image. (In subsequent MIB steps, the "short" .MIM file is used as input, and MIB creates a new, larger copy as output.) 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. In that case, process images can be installed directly into the original full-size .MIM file in subsequent MIB steps; you specify the existing .MIM file as an output. No separate input .MIM file need be specified. Updating the full-size .MIM file "in place" is not a particularly safe mode of operation, however, since the .MIM file could be corrupted at some point due to an error and no back-up versions would exist.

The /SM option allows you to conserve file space while retaining interim versions of the memory image in a compact form, either for back-up purposes or for use in a later partial-rebuild cycle. (Also, there is no point in ending up with a "final" memory image that is larger than the memory space allocated to the installed components, that is, the loadable portion of memory.)

General Command Format

```
     >[MCR] MIB              (Native RSX-11)
or
     $ MPMIB                 (VAX-11/RSX compatibility mode)

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

Default File Types

```
     MIB> .MIM, .MAP, .DBG = .PIM, .MIM, .STB
```

See Sections 7.3 and 7.6 for a full description of the MIB command line and all applicable options.

MIB can construct a memory image for any of the following purposes:

- For loading and debugging under control of the PASDBG symbolic debugger -- RAM-only target environment, no bootstrap in the image, DSM included in the kernel.
- For down-line loading only via the PASDBG LOAD/EXIT command -- RAM-only target environment, no bootstrap in the image, no DSM in the kernel. (Once the memory image is loaded via LOAD/EXIT, no further interaction takes place between the target system and the debugger.)
- For bootstrapping from a disk or disklike device on the target system -- RAM-only target environment, appropriate bootstrap installed in the image, no DSM in the kernel.

- For "blasting" into PROM with the PROM Programmer -- ROM/RAM target environment, no bootstrap in the image, no DSM in the kernel.

The type of memory image file MIB constructs depends partly on the type of memory specified in the configuration file and partly on the options you use in the build steps.

3.2.4.1 **Building a Memory Image for Debugging or Down-Line Loading** - The following example shows the basic form of 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; that is controlled by the configuration file (Section 3.2.1). (The example is applicable to either a mapped or an unmapped application.)

```
>[MCR] MIB
MIB>APPLC1,,APPLIC=KERNL1,,KERNL1/KI/SM
```

In this example, APPLC1 names the output memory image file, APPLC1.MIM, and APPLIC names the output debug-symbol file, APPLIC.DBG. The first KERNL1 in the command line specifies the input kernel image file, KERNL1.PIM, and the second 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) switch; it indicates to MIB that the .PIM file contains a kernel image rather than a process image. That causes MIB to 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.

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 a 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 modified MIB command line looks as follows:

```
>[MCR] MIB
MIB>APPLC1=KERNL1/KI/SM
```

3.2.4.2 **Building a Memory Image for Booting** - The following example shows the basic form of MIB command line used to create a memory image file with a bootstrap installed. This type of .MIM file can be used for booting the application from a target TU58, RL01/RL02, RX50, RD51, or RX02 device, after processing the completed file with the COPYB utility. (See Chapter 8 for information on various methods of loading the application.)

```
>[MCR] MIB
MIB>APPLC1=KERNL1/KI/SM/BS:"bootstrap-filespec"
```

Note the quote signs enclosing the bootstrap file specification in the /BS (bootstrap) switch. They 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 include the following:

DDBOTU	Unmapped TU58 bootstrap
DLBOTU	Unmapped RL01/RL02 bootstrap
DYBOTU	Unmapped RX02 bootstrap
DUBOTU	Unmapped MSCP-class disk bootstrap (RX50, RD51)
DDBOTM	Mapped TU58 bootstrap
DLBOTM	Mapped RL01/RL02 bootstrap
DYBOTM	Mapped RX02 bootstrap
DUBOTM	Mapped MSCP-class disk bootstrap (RX50, RD51)

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

In this example, APPLC1 names the output memory image file, APPLC1.MIM, and KERNL1 specifies the input kernel image file, KERNL1.PIM. The /KI switch must be used in this step, as explained in the preceding section. The /BS option causes MIB to install 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 floppy diskette, DLBOTx is the RL01/RL02 bootstrap, and DUBOTx is the MSCP-class disk bootstrap (RX50, RD51).

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).

Note that a bootstrap can be added to a memory image at any point in the development process; it does not have to be done in the kernel installation step. For example, a bootstrap can be added in a separate MIB operation following a complete build cycle, using a MIB command line of the form:

```
outmim=,inmim/SM/BS:"bootfile"
```

In this command format, outmim is the new .MIM file in which a bootstrap is to be installed, and inmim is an existing .MIM file that lacks a bootstrap. (Once installed, a bootstrap can be neither removed nor replaced.) Deferring bootstrap installation until the end of a build cycle would allow you, for example, to use different bootstraps with the same memory image -- assuming that you have several possible target boot devices.

The /SM option in this example is again discretionary; its effect and implications are described in Section 3.2.4.

3.2.4.3 **Building 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>APPLC1=KERNL1/KI/SM
```

In this example, APPLC1 names the output memory image file, APPLC1.MIM, and KERNL1 specifies the input kernel image file, KERNL1.PIM. MIB creates the .MIM file and installs the kernel image in it. The /KI switch must be used in this step, as explained in Section 3.2.4.1.

This example assumes that the configuration file used in the kernel MERGE 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).

As in the preceding MIB examples, the /SM switch is discretionary; see Section 3.2.4.

3.3 BUILDING DIGITAL-SUPPLIED SYSTEM PROCESSES

After building the kernel, you are ready to build the DIGITAL-supplied system processes your application needs. The four steps involved for each system process are described below. Figure 3-3 illustrates the system process build phase.

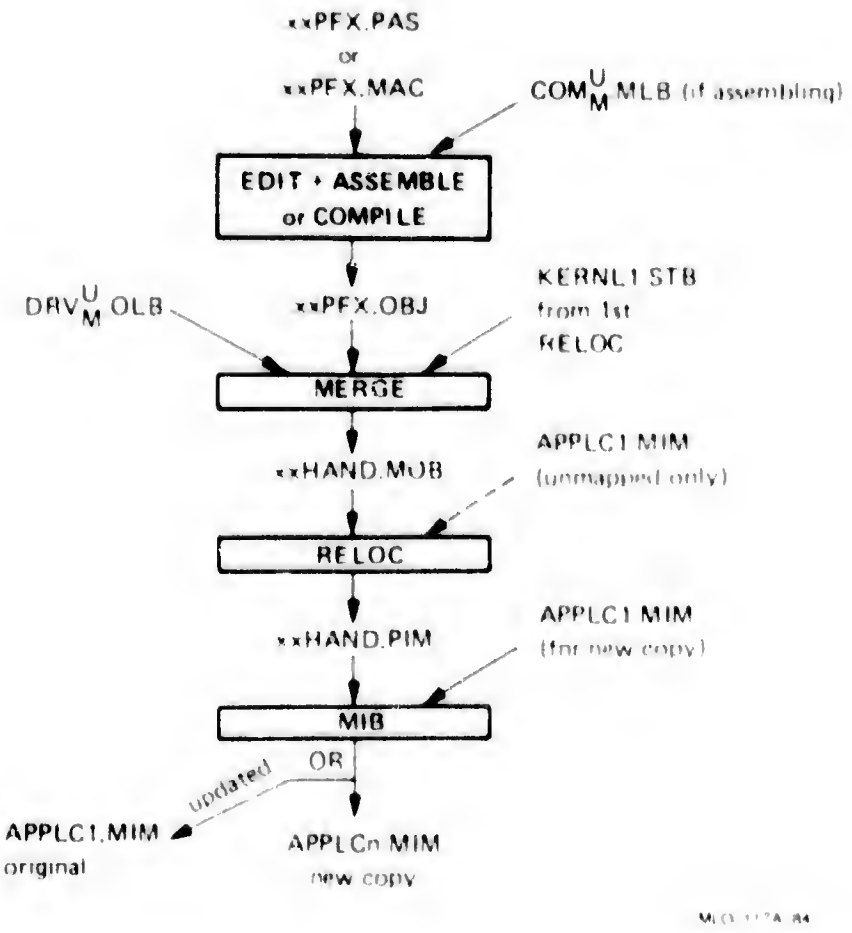

Figure 3-3 Build DIGITAL-Supplied System Processes

3.3.1 Edit and Assemble/Compile Prefix Module

The DIGITAL-supplied system processes -- I/O device handler and clock processes -- are listed in Tables 4-1 and 4-2. These processes are supplied in object form in three object module libraries: DRVM.OLB for mapped targets, DRVU.OLB for unmapped LSI-11 or FALCON targets, and DRVK.OLB for unmapped KXT11-C targets. As shown in Table 4-1, each system process has a corresponding prefix module file in either MACRO-11 (.MAC) or Pascal (.PAS) source form. The prefix module has two functions in a system process MERGE step: to select the required object module from the DRVx.OLB handler library and to supply that module with device-specific parameters, such as CSR/vector addresses.

Therefore, for each system process to be included in your application, you must inspect and, possibly, modify the matching prefix module and then assemble or compile it as appropriate. For example, to build the RX02 (DY) device handler into your application, you edit the DY prefix module DYPFX.MAC as needed to reflect your target hardware and assemble the module. Section 4.2 describes each prefix module in detail and explains the default parameters that you may need to modify.

NOTE

The Directory Service Process (DSP) -- part of the MicroPower/Pascal file system -- is not included in the DRVx libraries. The DSP is supplied in the object libraries DSPNHD.OLB, DSPFIS.OLB, DSPEIS.OLB, and DSPFPP.OLB. The corresponding prefix module file is DSPPFX.MAC. Appendix A describes DSPPFX.MAC and the other software components required for building an application that includes the file system.

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 this command, x is U for an unmapped application or M for a mapped application. (As explained in a previous note, 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 the user's default directory.

To compile a prefix module written in Pascal -- for example, the KWPFX.PAS (KWV11 clock handler) prefix file -- you can use the following minimum form of Pascal command line:

```
>[MCR] MPP
MPP>KWPFX=KWPFX/IN:xxx
```

In this command, the instruction set option xxx is either NHD or EIS. You must use the NHD option when compiling a Pascal prefix module for an unmapped system, regardless of your target hardware instruction set, and you must merge the resulting object module with DRVU.OLB or

DRVK.OLB and LIBNHD.OLB. For a mapped system, you must use the EIS option, again regardless of your target hardware instruction set, and merge the resulting module with DRVM.OLB and LIBEIS.OLB.

Assembling the Directory Service Process prefix module, DSPPFX.MAC, is a special case; it must be assembled with the file system macro library, FSMAC.MLB, instead of the standard COMU.MLB or COMM.MLB system macro library. Assuming that the DSP prefix file is in your default directory, suitably modified as required, you assemble the DSP prefix file with the following form of MACRO-11 command:

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

The FSMAC.MLB library defines the dspf$ macro used in the DSPPFX.MAC file.

3.3.2 Merge System Process

Merge the prefix object module for a handler with the kernel symbol table and the appropriate device-handler object library: DRVU.OLB for an unmapped LSI or FALCON target system, DRVK.OLB for an unmapped KXT11-C target system, or DRVM.OLB for any mapped target system. The kernel symbol table file is needed to satisfy references to primitive services in the kernel. The DRVx library supplies the required handler object module.

General Command Format

```
>[MCR] MRG              (Native RSX-11)
or
$ MPMERGE               (VAX-11/RSX compatibility mode)

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

Default File Types

```
MRG> .MOB, .MAP, .AUX = .OBJ, .OBJ, ...
```

The default type for a file specified with the /LB (library) switch is .OLB.

The example below merges the DY (RX02) handler prefix object module, DYPFX.OBJ, with the DRVU or DRVM library and the kernel symbol table, KERNL1.STB, created in the phase 1 RELOC step. (The DRVK library does not contain a DY handler.) The output is the merged device-handler object module DYHAND.MOB.

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

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

Note that the /DE option is not used in this phase, since system-process symbols are not ordinarily used in user-level debugging.

Merging a device handler implemented in Pascal, such as the YA, AA, or KW handler, involves an additional library file. The handler's prefix object module must be merged with the LIBNHD.OLB Pascal OTS library for an unmapped system or with LIBEIS.OLB for a mapped system in addition to the DRVx object library and the kernel STB file.

The following example merges the KW programmable clock handler process for a mapped application:

```
>[MCR] MRG
MRG>KWHAND=KWPFX,KERNL1.STB,mpp-lib:DRVM/LB,LIBEIS/LB
*<CRTL/Z>
```

The OTS library file LIBEIS.OLB satisfies references to OTS routines found in the mapped KW handler object module that is pulled from DRVM.OLB. (The DIGITAL-supplied handlers that are implemented in Pascal have been compiled for unmapped systems with the /IN:NHD compilation option, and thus they must be merged with LIBNHD, regardless of the instruction set supported by the target system. For mapped systems, the same handlers have been compiled with the /IN:EIS compilation option and must be merged with LIBEIS, again regardless of the actual target instruction set.)

NOTE

The Directory Service Process (DSP) -- part of the MicroPower/Pascal file system -- is a special case among system processes with respect to the MERGE step required for it. (The DSP is not a device handler and does not reside in the DRVx libraries.) The DSP is implemented in Pascal -- although its prefix module, DSPPFX.MAC, is written in MACRO-11 -- and its object modules are supplied in the four special object library files DSPNHD.OLB, DSPEIS.OLB, DSPFIS.OLB, and DSPFPP.OLB. (Choose the version that matches your target system instruction set.) The corresponding Pascal OTS library must also be included in the merge.

The following example merges the DSP for a target system that supports the FPP instruction set:

```
>[MCR] MRG
MRG>DSP=DSPPFX,KERNL1.STB,mpp-lib:DSPFPP/LB,LIBFPP/LB
MRG><CRTL/Z>
```

DSP names the output process image file, DSP.MOB. DSPPFX specifies the prefix module DSPPFX.OBJ. See Section 3.3.1 for the corresponding assembly step. DSPFPP specifies the FPP version of the DSP object module library, DSPFPP.OLB. LIBFPP specifies the corresponding version of the Pascal OTS library. To merge the DSP for other than an FPP target system, substitute the appropriate version of the DSP and OTS libraries in the command line shown above.

3.3.3 Relocate Handler

Invoke RELOC to relocate the object module DYHAND.MOB. This operation produces the device-handler process image file DYHAND.PIM, which will be installed in the application memory image file.

General Command Format

```
>[MCR] REL              (Native RSX-11)
or
$ MPRELOC               (VAX-11/RSX compatibility mode)

REL>[pimfile][,mapfile][,stbfile]=mobfile[,mimfile][/switches]
```

Default File Types

```
REL> .PIM, .MAP, .STB = .MOB, .MIM
```

MAPPED

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

UNMAPPED

```
>[MCR] REL
REL>DYHAND=DYHAND,APPLC1
```

In both examples, the first DYHAND names the output process image file DYHAND.PIM, and the second specifies the input merged object file DYHAND.MOB. In the mapped example, the special relocation switches /RO:40000/RW:60000 are needed to ensure correct mapping of the handler's code and data segments according to the requirements for driver-mapped processes. (See Section 2.1.7 of the MicroPower/Pascal Runtime Services Manual.) These two switches force the handler's code (read-only) segment to be mapped by PAR 2 and its data (read/write) segment to be mapped by PAR 3. (The size of a device handler's code segment is limited to 4K words, the address range of a single PAR. The same is true for the data segment.) The /RO:40000 switch forces the handler'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 switch forces the handler's first read/write program section to start at virtual address 60000, corresponding to PAR 3.

If you are relocating a user-written device handler, check the RELOC map to ensure that the process's RW segment size does not exceed 8K (20000 octal) bytes; the entire segment must be mapped by PAR3 only.

In the unmapped example, APPLC1 specifies the memory image file created in the previous phase, APPLC1.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 building an unmapped process; it works for both RAM-only and ROM/RAM targets. As an alternative to specifying the .MIM file, however, you can specify physical relocation addresses in the command

line, using the /RO, /RW, or /QB switches. (You are unlikely to need to use the alternative method.)

NOTE

The Directory Service Process (DSP) is an exception among system processes with respect to relocation for a mapped environment. The DSP is not a device handler and thus does not have driver process mapping. The DSP has general-process mapping and therefore is relocated like most user-written processes in the mapped case (Section 3.4.3). The special relocation switches /RO and /RW are not used, but the /AL switch must be employed if building for a mapped ROM/RAM environment. (For an unmapped target system, there is no distinction between system and user processes in regard to relocation options.)

The following sample command lines relocate the DSP merged object module, DSP.MOB, for different target environments:

```
REL>DSP=DSP              Mapped RAM-only

REL>DSP=DSP/AL           Mapped ROM/RAM

REL>DSP=DSP,APPLC1       Unmapped
```

See Section 3.4.3 for an explanation of the /AL (align R/W segment) option.

3.3.4 Install System Process in Memory Image

Invoke MIB to add the current process to the memory image.

General Command Format

```
>[MCR] MIB              (Native RSX-11)
or
$ MPMIB                 (VAX-11/RSX compatibility mode)

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

Default File Types

```
MIB> .MIM, .MAP, .DBG = .PIM, .MIM, .STB
```

In the following example, the DY handler 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>APPLC1=DYHAND,APPLC1/SM
```

The first APPLC1 in the command line names the output .MIM file, and the second specifies the existing APPLC1.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. The /SM switch again limits the size of the new .MIM file to the minimum needed to accommodate the existing memory image plus the newly installed process.

Since the output .MIM file was given the same name as the input .MIM file, MIB creates a higher-numbered version. If you wanted to keep the input .MIM file as possible input to a subsequent partial-rebuild cycle, starting at the system process phase, you would probably give the output file a different name, such as APPLC2.

3.4 BUILDING USER-WRITTEN STATIC PROCESSES

After building the DIGITAL-supplied static processes, you are ready to build your own static processes. The four steps involved for each user process are described below. Figure 3-4 illustrates the user process build phase.

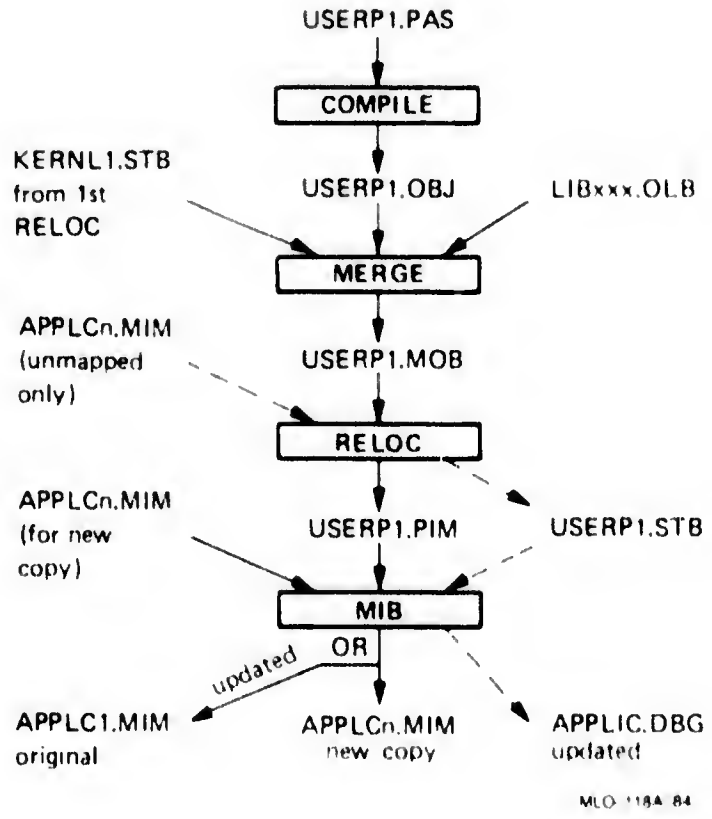

Figure 3-4 Build User-Written Pascal Static Processes

3.4.1 Compile/Assemble Static Process Source File

Invoke the MicroPower/Pascal compiler to compile a user process written in Pascal. Chapter 2 provides a complete description of compiler operation.

General Command Format

```
>[MCR] MPP              (Native RSX-11)
or
$ MPPASCAL              (VAX-11/RSX compatibility mode)

MPP>[objfile][,listfile]=sourcefile[/switches]
```

Default File Types

```
MPP> .OBJ, .LST = .PAS
```

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
```

The /DE switch causes the compiler to include debug symbol information in the object file. (The compiler produces debug records for both local and global source symbols.) The /IN:EIS switch specifies the target system instruction set; other values for /IN: include NHD (default), FIS, and FPP. (Chapter 2 describes all compilation options.) No list file is requested in the example.

Invoke 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 /E:DBG option recognized by MACRO-11 produces a different type of debug symbol records; the option should not be used.) Valid debug symbol records for global symbols can, however, be generated by MERGE.

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 this command, COMx is COMM for a mapped application or COMU for an unmapped application. If your MACRO-11 source program contains any file system macro calls, it must be assembled with the FSMAC.MLB library as well as with COMx.MLB, as described in Appendix A. In that case, the assembly command line would look as follows:

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

3.4.2 Merge Static Process

Each static process 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 four MicroPower/Pascal OTS libraries support FP-11 floating-point hardware (LIBFPP.OLB), FIS floating-point hardware (LIBFIS.OLB), EIS hardware (LIBEIS.OLB), and no special hardware (LIBNHD.OLB). Use the OTS library that matches your target hardware instruction set.

General Command Format

```
      >[MCR] MRG              (Native RSX-11)
or
      $ MPMERGE               (VAX-11/RSX compatibility mode)

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

Default File Types

```
      MRG> .MOB, .MAP, .AUX = .OBJ, .OBJ, ...
```

The default type for a file specified with the /LB (library) switch is .OLB.

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

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

The KERNL1.STB file satisfies references to kernel primitive-service entry points. 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.) Note that the /DE switch is used only on the USERP1.OBJ module. Ordinarily, you would not want the globals of the OTS routines contained in LIBxxx.OLB, and you should not use the /DE switch on an STB file in any case.

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

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

The only difference between this example and the preceding, Pascal-oriented one is that no object-time library is required for user processes written in assembly language unless the process uses the file system. See Appendix A concerning the MACFS.OLB file system support library.

You can also use MERGE to combine, or "premerge," a set of individually compiled or assembled user modules into one object module, as in the following example:

```
      >[MCR] MRG
      MRG>USERP3.OBJ/DE=PROGP3,P3MOD1,P3MOD2
```

The /DE option specified on the output side causes debug records to be copied from all the input object files to the output object file. (If MERGE finds no valid debug records in an input module, as in the MACRO-11 case, it generates them for the global symbols of that module.) The effect is the same as if /DE were appended to each of the input file names. The .OBJ file type is specified for the output file so that MERGE will not apply the default .OB type to the output. The merged object file thus obtained -- presumably containing all the user-written code for a static process -- can in turn be used as input to a further MERGE operation in which it would be merged with an STB file and an OTS library, for example. (The purpose of the example is simply to show that MERGE can be used incrementally, that is, in an iterative fashion within one "multiple merge" step.)

Note that you can also merge your own user module 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:LIBxxx/LB
```

NOTE

Refer to Appendix A for information about merging a process that uses MicroPower/Pascal file system services. Almost all Pascal I/O procedures and functions imply the use of the file system, as do the MACRO-11 QIO$ and related macro calls.

3.4.3 Relocate Static Process

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

General Command Format

```
>[MCR] REL              (Native RSX-11)
or
$ MPRELOC               (VAX-11/RSX compatibility mode)

REL>[pimfile][,mapfile][,stbfile]=mobfile[,mimfile][/switches]
```

Default File Types

```
REL> .PIM, .MAP, .STB = .MOB, .MIM
```

If the application is unmapped, RELOC needs physical start 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 that for an unmapped process, you must include the name of the existing memory image file -- APPLC1, 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 switches 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 switch, 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 3.3.3 shows how to relocate a driver-mapped process.)

UNMAPPED RAM-only

```
>[MCR] REL
REL>USERP1,,USERP1=USERP1,APPLC1/DE
```

UNMAPPED ROM/RAM

```
>[MCR] REL
REL>USERP1=USERP1,APPLC1
```

MAPPED RAM-only

```
>[MCR] REL
REL>USERP1,,USERP1=USERP1/DE
```

MAPPED ROM/RAM

```
>[MCR] REL
REL>USERP1=USERP1/AL
```

The command line in both of the 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 switch 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.) That 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 will force 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.)

3.4.4 Install Static Process in Memory Image

Invoke MIB to add 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.

General Command Format

```
>[MCR] MIB              (Native RSX-11)
or
$ MPMIB                 (VAX-11/RSX compatibility mode)

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

Default File Types

```
MIB> .MIM, .MAP, .DBG = .PIM, .MIM, .STB
```

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,,APPLIC=USERP1,APPLC1,USERP1/SM
```

On the left-hand, or output, side of the equal sign, APPLC1 names the output .MIM file, and APPLIC specifies the existing APPLIC.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 higher-numbered version. You might, of course, assign a different name -- APPLC2 for example -- to the output .MIM file. Also, 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 APPLIC.DBG file.

3.5 AUTOMATED APPLICATION BUILDING

The preceding sections have described the steps involved in a complete build cycle for a normal range of applications. You do not have to execute each of those individual steps yourself, however, or construct a command file to do so. The DIGITAL-supplied MPBUILD command procedure can generate a customized build-command file for you, based on your responses to an interactive dialog in which you supply information about your target and identify the components to be built. When invoked, the generated command file will "drive" the entire build cycle. Appendix B describes the MPBUILD dialog in detail.

Before using MPBUILD for the first time, you should have already accomplished the following:

- Obtained "clean" compilations or assemblies of all user-process source programs, to preclude source code errors during the build. (MPBUILD accepts build inputs in source or object form, however.)
- Created an initial system configuration file for your application, as described in Chapter 4.
- Performed any required editing of the handler process prefix files to be used in the build cycle, as described in Chapter 4, to match your target I/O configuration.

(You can also edit the configuration and prefix module files during the MPBUILD dialog; this feature is convenient for making minor modifications for a rebuild.)

MPBUILD effectively hides the details of utility command syntax and relieves you of an otherwise considerable and repetitive clerical burden. MPBUILD allows you to perform a total build cycle, starting with the kernel and handler processes, or to perform a partial build cycle, starting with a previously built kernel/handler image, for economical rebuilds.

3.6 DEBUGGING AND REBUILDING THE APPLICATION

Having completed the initial build cycle, you are ready to start debugging your application and rebuilding it as required. This section discusses some of the considerations involved.

The sample command sequence below causes the debugger to load the application's debug symbol file, APPLIC.DBG, and then to load the APPLC1.MIM image into the target system over a serial communication line.

```
>[MCR] PDB              (RSX-11 version of debugger)
or
$ PASDBG                (VAX/VMS version of debugger)

PASDBG> LOAD/SYMBOL APPLIC
PASDBG> LOAD/TARGET APPLC1
```

(If the MIM and DBG files have the same name, a single "LOAD file-name" command is sufficient.) Operation of PASDBG from this point is described in the MicroPower/Pascal Debugger User's Guide.

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/handler .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/handler image. The MPBUILD procedure facilitates that by subdividing the entire build cycle into two partial build cycles, one for building a kernel/handler 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. That 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 switch to allow the compiler to perform full optimization. (That can reduce the size of some processes significantly, as well as increasing 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 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 will be down-line loaded using the PASDBG LOAD/EXIT command, for initial stand-alone testing.

Chapter 7 describes some special capabilities of the MIB utility that might allow you to avoid a rebuild entirely in some cases, or to make a partial rebuild operation more efficient. For example, you can change the start-up priority or exception group code of a currently installed process, or you can delete a process from the memory image -- and compact the image if it is mapped -- and then add an updated version of only that process.

CHAPTER 4

EDITING THE CONFIGURATION FILES AND PREFIX MODULE FILES

The first step in building a new application for a target system is creating a configuration file for your specific application. You usually do this by modifying a copy of one of the configuration files included in the distributed MicroPower/Pascal software. The configuration file supplies the build utility programs with basic information about the target system hardware configuration and the kernel software needed for your application. During application construction, you must also modify, assemble, and merge a prefix module for each DIGITAL-supplied system process, such as device handlers, clock process, or the directory service process, you use. The automated build procedure MPBUILD allows you to edit files during the automated build process. Refer to Appendix B for details.

This chapter provides the following information about configuration and prefix module files:

- The functions of the configuration file
- The role of the configuration file in the build procedure
- Configuration file macros
- A prototype configuration file
- The functions of the prefix modules
- The role of the prefix modules in the build procedure
- A description of each prefix module
- Directions for assembling or compiling the prefix modules

4.1 CONFIGURATION FILE

This section tells you what a configuration file does and how to use the configuration macros to support your hardware and software.

4.1.1 Function of the Configuration File

The configuration file supplies the build utility programs with the information required for configuring the kernel and the application memory image. That information includes the following:

- The type of target processor and its hardware options
- The target memory configuration
- The interrupt vector locations used by all devices installed in the target system
- The trap-processing modules that the application will require
- The amount of RAM memory to be used by the kernel for its stack, for message packets, and for dynamic data structures
- The primitive service modules to be included in the kernel

The configuration file lets you supply that information in the form of configuration macros written in MACRO-11 assembly language. The expansions for the macros are contained in a system macro library (COMM.MLB or COMU.MLB) that you must specify to MACRO-11 when you assemble the configuration file.

When you assemble the configuration file and merge it with your kernel module library (PAXU.OLB or PAXM.OLB), you build a kernel that supports your hardware and software configuration. Thus, you indirectly tell the build utilities how to build the application. Figure 4-1 illustrates the role of the configuration file in the kernel build procedure.

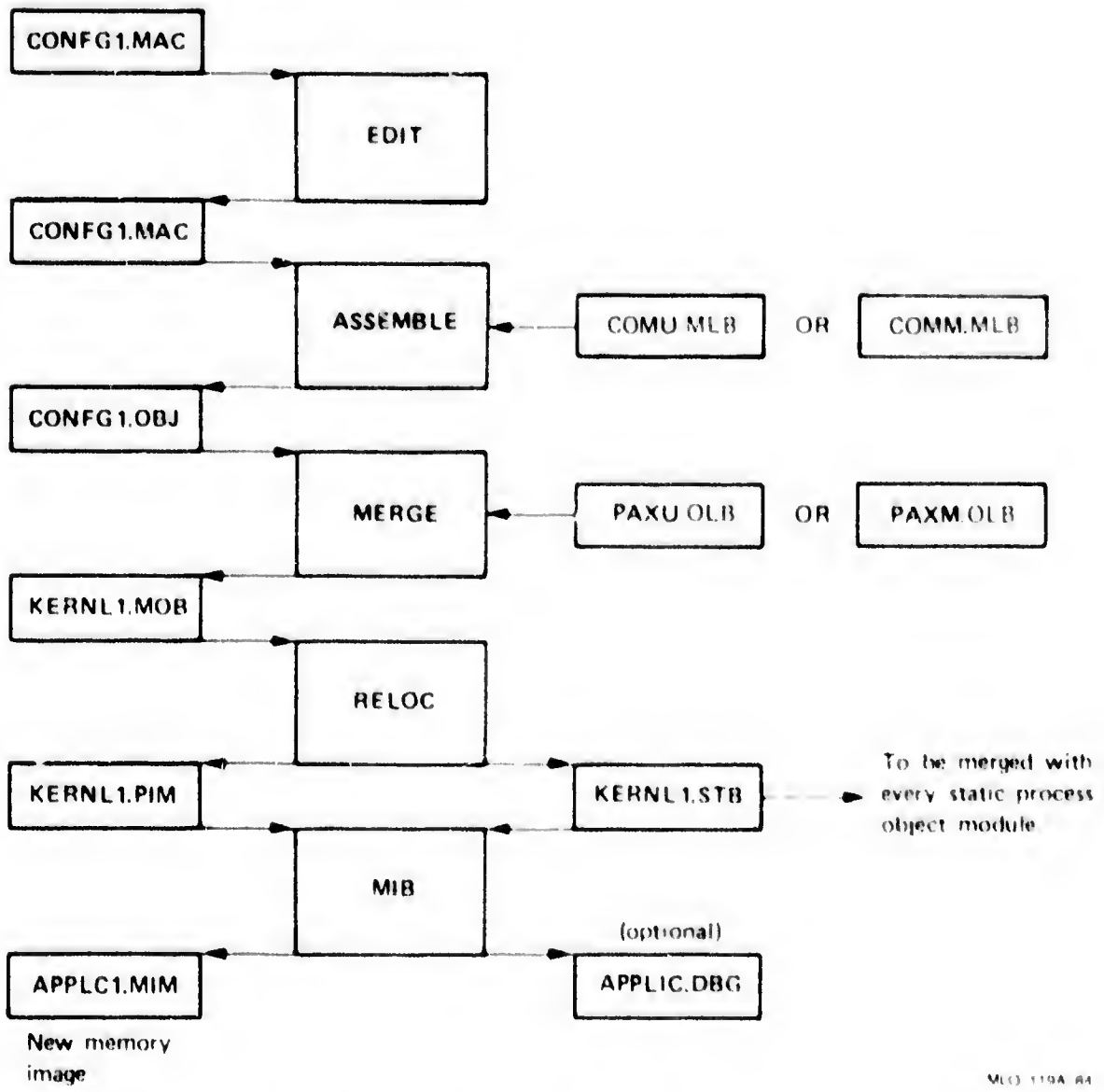

Figure 4-1 The Configuration File and the Build Procedure

4.1.2 Prototype Configuration File CONFIG.MAC

Figure 4-2 shows the prototype configuration file, CONFIG.MAC, included in the MicroPower/Pascal distribution kit to illustrate configuration file features. The prototype configuration file relies almost completely on configuration default values. That file supports a system with the following characteristics:

Processor	LSI-11/23 without memory mapping or floating-point options
Memory	One contiguous 28K-word (57,344-byte) RAM segment, base address 0, no parity checking
Device vectors	Console terminal serial line (vector addresses 60, 64); line clock (vector address 100)
Kernel	No debugger service module; default kernel stack size (104 bytes unmapped); 20 message packets; 3,000 bytes for system data structures; all primitive service modules included
Traps processed	TR4, T10, BPT, EMT, and TRP (LSI-11 defaults).

```
(1)         .enabl  LC

;  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
;  ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
(2) ;
;  COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.

(3)         .enabl  GBL

(4)         .mcall  CONFIGURATION
;+
;
; Module name: CONFIG.MAC
;
; System: MicroPower/Pascal
;
; Functional Description:
;
; This module describes the hardware and system software configuration on
; which the application system is to run. It must be edited by the user
; and assembled. The resulting object module is used to build the kernel.
;
; The following set of macros may be used in this file. The CONFIGURATION
; macro must be the first macro evoked. The ENDCFG macro must be the last.
; A configuration  file must contain at the minimum the CONFIGURATION,
; SYSTEM, PROCESSOR, MEMORY, DEVICES, and ENDCFG macros.
;
;       CONFIGURATION   [name]
;       SYSTEM  optimize[=NO],debug[=NO]
;       PROCESSOR mmu=[NO], fpu[=FP11;=FIS], type[=L1123], vector[=1000]
;       MEMORY base[=0], size[=0], type[=RAM], parity[=NO], csr[=0]
;       DEVICES v1,v2,...,v6
;       RESOURCES stack[=..KIS],packets[=20.],structures[=3000.],ramtbl[=20.]
;       PRIMITIVES p1[=ALL],p2,p3,p4,p5,p6
(5)     KXT11 nxm[=HALT], break[=TRAP], syshalt[=ODTROM], level7[=TRAP],
;             slu1[=9600], slu2[=9600]
;       TRAPS   t1[=ALL;t1],t2,t3,t4,t5,t6,t7,t8
;               ALL - TR4, T10, BPT, EMT, and TRP (std. LSI-11 set)
;               TR4 - Trap to 4 (bus timeout)
;               T10 - Trap to 10 (reserved instruction)
;               BPT - Breakpoint instruction trap
;               EMT - EMT instruction trap
;               TRP - TRAP instruction trap
;               MPT - Memory parity error
;               FIS - FIS exception trap
;               FPP - FPP exception trap
;               MMU - Memory management fault
;               BRK - FALCON (SBC-11/21) BREAK level-7 trap
;       ENDCFG
;
; If the SYSTEM macro optimize argument is given a value of "YES", the
; RESOURCES, TRAPS, and PRIMITIVES macros must be used to specify the
; actual services and options desired. If the default optimize option
; "NO" is chosen, the RESOURCES, TRAPS and PRIMITIVES macros are
; defaulted within the SYSTEM macro and should not be explicitly evoked
; in the configuration file. If PROCESSOR type=T11, then the KXT11 macro
; must be evoked, and the PROCESSOR macro must precede the SYSTEM macro.
;
;-

(6)         .sbttl  System Configuration File

(7) CONFIGURATION

(8) SYSTEM          debug=NO, optimize=NO

(9) PROCESSOR       mmu=NO

(10) MEMORY         base=0, size=<28.*32.>, type=RAM

(11) DEVICES        60,64,100

(12) ENDCFG

(13) .end
```

Figure 4-2 Prototype Configuration File

The memory size parameter (size=nnn) in the MEMORY macro is specified in units of 64 bytes; thus, the effective size specification in CONFIG.MAC is 896*64, equaling 57,344 bytes of memory. The expression <28.*32.> is used instead of 896 to indicate, by convention, a 28K-word memory segment. Similarly, 32,768 bytes of memory could be specified by the expression <16.*32.> in place of 512 to indicate a 16K-word memory segment.

NOTE

In MACRO-11 assembly language notation, a number in the source code representing a decimal integer value must be terminated by a decimal point. MACRO-11 interprets numbers not ending with a decimal point as octal values. Also, angle brackets are used instead of parentheses to delimit an expression.

To create a configuration file for your own application, you can copy and edit either the prototype CONFIG.MAC or one of the working configuration files also included in the distribution kit. The additional files that can be used as a base for editing are:

CFDFAL.MAC	FALCON target, with debug support
CFDFPL.MAC	FALCON-PLUS target, with debug support
CFDKTC.MAC	KXT11-C target, with debug support
CFDMAP.MAC	Mapped LSI-11 target, with debug support
CFDUNM.MAC	Unmapped LSI-11 target, with debug support

These configuration files are used for hardware-installation verification purposes. CFDUNM.MAC is used, for example, by the automatic verification procedure for an LSI-11 target, and CFDFAL.MAC is used for a FALCON target.

Section 4.1.4 describes the configuration macros in detail, providing the information you need to change or to add configuration macros in a copy of the file you choose to modify.

Figure 4-2, a listing of CONFIG.MAC, has circled numbers identifying its components. Components numbered 1, 3, 4, 6, and 13 identify MACRO-11 general assembler directives that you must leave in the file. Component number 2 is the copyright statement; component number 5 is a cursory description of the configuration macros that you can delete from the file if you wish. Lines beginning with a semicolon (;) are comment lines, which MACRO-11 does not assemble.

You can modify lines 7 through 12, as described in Section 4.1.4, to suit your configuration. You can change the arguments to these macros and add other macros. Six configuration macros must be included in the configuration file; other macros are included as needed. The first macro in the file must be CONFIGURATION; the second and third, PROCESSOR and SYSTEM (in either order); and the last, ENDCFG. You must also include the MEMORY and DEVICES macros. In summary, the configuration file must contain at least the following macros:

```
CONFIGURATION
PROCESSOR
SYSTEM
MEMORY
DEVICES
ENDCFG
```

Also, if the processor type is T11 (FALCON) or T11+ (FALCON-PLUS), you must include the KXT11 macro in the file. If the processor type is KXT11-C, you must include the KXT11C macro in the file.

The configuration file macros are summarized below.

Macro	Function
CONFIGURATION	Identifies file as system configuration file; must be first macro in file
DEVICES	Defines the set of vectors used by all devices installed in the system (six vectors per call); may be invoked more than once
ENDCFG	Ends the system configuration file; must be last macro in file
KXT11	Describes trap modes and serial line baud rates for the SBC-11/21 (FALCON or FALCON-PLUS)
KXT11C	Configures the kernel for the KXT11-C peripheral processor and adds a KXT11-C power-up module
MEMORY	Describes a single, contiguous segment of memory in the target hardware; mandatory; may be invoked more than once
PRIMITIVES	Defines the set of primitive modules to be included in the kernel (six primitives per call); may be invoked more than once
PROCESSOR	Describes the type of target processor and its hardware options; must be invoked as the second or third macro in file
RESOURCES	Specifies the amount of RAM memory to be included in the kernel for its stack, for message packets, and for dynamic data structures
SYSTEM	Determines whether the kernel should be optimized and whether it should include the debugger service module; must be invoked as the second or third macro in file
TRAPS	Defines the set of trap processors required for application (six traps per call); may be invoked more than once

4.1.3 Assembling and Merging the Configuration File

When your configuration file contains the macros appropriate for your system, you assemble the file with the required system macro library, COMU.MLB or COMM.MLB, to produce an object module, as described in Chapter 3. You then merge the object module with the kernel module library (PAXU.OLB or PAXM.OLB) to produce a kernel object module, as also described in Chapter 3.

4.1.4 Modifying the Configuration File Macros

If none of the supplied configuration files meets your needs, you must modify one of the files. You should copy the selected file to a file with a different name and edit the newly copied file. Doing so will preserve the base file for future build procedures.

NOTE

In the following text, all numeric values except addresses are expressed in decimal unless otherwise indicated. Addresses are expressed in octal. In the macro syntax definitions and examples, however, MACRO-11 notation rules apply: decimal numbers end with a decimal point; octal numbers do not.

CONFIGURATION

4.1.4.1 **CONFIGURATION** - The CONFIGURATION macro identifies a file as a system configuration file. This macro takes one argument, name, which generates a .IDENT statement identifying the particular configuration file. The .IDENT statement is a MACRO-11 assembler directive that labels the object module with the supplied name.

The CONFIGURATION macro must be the first macro in a configuration file.

Syntax

```
CONFIGURATION name
```

name
: An identifier of up to six characters, each of which must be a valid RAD50 character. If you do not include the argument, no .IDENT statement is generated, and MERGE defaults the name to the first .IDENT statement it finds in the files being merged.

Example

```
CONFIGURATION C1
```

DEVICES

4.1.4.2 **DEVICES** - Use the DEVICES macro to define the set of interrupt vectors used by all devices installed on the target system. You can specify up to six vectors per statement.

Interrupt vectors are locations in low memory that contain the address of a device's interrupt dispatch block (IDB) and the new processor status word. Each device must be installed so that it interrupts at one of the specified vector addresses.

The DEVICES macro allocates a unique IDB for each vector specified in the macro. Since a vector may be in ROM, it must permanently point to its IDB, through which the kernel dispatches interrupts to interrupt service routines (ISRs). Therefore, in order to permit runtime connection of ISRs to interrupts, IDBs must be allocated in RAM, and such allocation must be done during the build procedure. See Chapter 8 of the MicroPower/Pascal Runtime Services Manual for a full description of IDBs and interrupt dispatching.

NOTE

When building an application with debugging support, do not specify the console-terminal vector addresses, 60 and 64, which are used by the debugger's host-to-target serial line.

Syntax

```
DEVICES v1,v2,...,v6
```

vn

The address of the first word of each vector used by the application. You can specify up to six addresses per DEVICES macro, and you can use more than one DEVICES macro in a configuration file.

Example

```
DEVICES 300,310,320
```

ENDCFG

4.1.4.3 **ENDCFG** - Use the ENDCFG macro to end the system configuration file. ENDCFG must be the last macro in a configuration file.

Syntax

```
ENDCFG
```

KXT11

4.1.4.4 **KXT11** - Configuration files used for SBC-11/21 (FALCON or FALCON-PLUS) applications require the KXT11 macro; configuration files used for LSI-11 or other PDP-11 microcomputer applications do not. If T11 or T11+ is the processor type specified in the PROCESSOR macro, you must also include the KXT11 macro to provide information on traps and serial line baud rates for the SBC-11/21.

Syntax

```
KXT11  nxm[=HALT],break[=TRAP],syshalt[=ODTROM],level7[=TRAP],
       slu1[=9600],slu2[=9600]
```

nxm

·lects the hardware action taken on a nonexistent memory (bus timeout) access. Parameter values are HALT or TRAP. You should select the HALT trap in normal circumstances, since the factory hardware configuration is nxm = HALT. TRAP is a level-7 interrupt through vector 140. HALT is a trap to the restart address.

break

Selects the hardware action taken whenever the console BREAK key is pressed or the LSI-11 Bus BHALT L signal line is asserted. Parameter values are HALT or TRAP. You should select the TRAP value in normal circumstances, since the factory hardware configuration is break = TRAP. TRAP is a level-7 interrupt through vector 140. HALT is a trap to the processor restart address.

NOTE

You cannot specify the same parameter value for both nxm and break.

syshalt

Specifies the action to be taken when a HALT instruction is executed or BHALT L is asserted. Refer to the section "Choosing Values for SYSHALT and LEVEL7," below, for help in choosing the syshalt parameter. The syshalt values ODT, ODTROM, HANG, or USER specify the following actions:

ODT	Jump to ODT module in the application kernel.
ODTROM	Jump to Macro-ODT firmware at location 173000.
HANG	Go into an infinite jump-to-self loop (BR .).
USER	Execute user-written routine; refer to "User-Written Error-Handling Routines $BHUSR, $NHUSR, and $NUHNG," below.

level7

Specifies the action taken when an interrupt through vector 140 occurs. Refer to the section "Choosing Values for SYSHALT and LEVEL7," below, for help in choosing the level7 parameter. This parameter is ignored unless break=TRAP, nxm=HALT, and syshalt=HANG are also specified. Level7 values ODT, USER, TRAP, and IGNORE specify the following actions:

ODT — Jump to ODT module in application kernel.

USER — Execute user-written routine (global symbol $NUHNG). The level7 routine must provide response for break interrupts only.

TRAP — Enter kernel exception-handling mechanism with exception type equal to Execution Error (EX$EXC), subcode Break (ESSBRK).

IGNORE — Dismiss interrupt; that is, execute an RTI instruction.

slu1

Specifies the initialization baud rate for serial port 1, which is normally the console port. Slu1 values (bits per second) are 300, 600, 1200, 2400, 4800, 9600, 19,200, and 38,400. The slu1 value must agree with the transmit and receive baud rates set on the console terminal or configured on the host system for down-line loading and debugging. Do not use a decimal point following the specified speed value; decimal radix is implicit for this parameter.

NOTE

When initially entered, software ODT sets the baud rate for serial port 1 to the baud rate of the attached device (console terminal or host system) upon receiving a leading carriage return on the console line. However, the kernel subsequently overrides that setting with the specified slu1 value when the application starts up.

slu2

Specifies the initialization baud rate for serial port 2. Parameter values are specified in the same manner as for serial port 1.

NOTE

Software ODT automatically sets the baud rate for serial port 2 to 38,400 bps whenever a TU58 bootstrap command is issued -- for example, DD <RET>.

Choosing Values for SYSHALT and LEVEL7

The parameters for syshalt and level7 are interdependent, and only certain combinations are valid. The interaction of syshalt and level7 depends on the SBC11/21 (FALCON or FALCON-PLUS) board jumper selections for BREAK and TRAP. Valid combinations and their effects when a BREAK, HALT, nonexistent memory reference (NXM), or power-up occur are listed below. Note that the power-up address is hardware configured as well, and the power-up parameter that you specify must agree with the hardware address.

SBC11/21 Board Jumpers BREAK=TRAP, NXM=HALT

SYSHALT=ODT, LEVEL7 any value

```
BREAK   --> Transfer to software ODT
HALT    --> Transfer to software ODT
NXM     --> Kernel trap to 4 exception
Powerup --> Start execution at location 0
```

SYSHALT=ODTROM, LEVEL7 any value

```
BREAK   --> Transfer to ROM ODT
HALT    --> Transfer to ROM ODT
NXM     --> Kernel trap to 4 exception
Powerup --> Transfer to ROM ODT
```

SYSHALT=HANG, LEVEL7=ODT

```
BREAK   --> Transfer to software ODT
HALT    --> Hang
NXM     --> Kernel trap to 4 exception
Powerup --> Start execution at location 0
```

SYSHALT=HANG, LEVEL7=USER

```
BREAK   --> Transfer to user routine $NUHNG (see section
            (3) of Figure 4-3)
HALT    --> Hang
NXM     --> Kernel trap to 4 exception
Powerup --> Start execution at location 0
```

SYSHALT=HANG, LEVEL7=TRAP

```
BREAK   --> Kernel break exception
HALT    --> Hang
NXM     --> Kernel trap to 4 exception
Powerup --> Start execution at location 0
```

SYSHALT=HANG, LEVEL7=IGNORE

```
BREAK   --> Ignored
HALT    --> Hang
NXM     --> Kernel trap to 4 exception
Powerup --> Start execution at location 0
```

SYSHALT=USER, LEVEL7 any value

```
BREAK   --> Transfer to user routine $NHUSR (see section
            (1) of Figure 4-3)
HALT    --> Transfer to either software ODT or  ROM  ODT
            (see section (1) of Figure 4-3)
NXM     --> Kernel trap to 4 exception
Powerup --> Start execution at location 0 if no ROM  ODT
            on SBC11/21;  otherwise, transfer to ROM ODT
```

SBC11/21 Board Jumpers BREAK=HALT, NXM=TRAP

SYSHALT=ODT, LEVEL7 any value

```
BREAK   --> Transfer to software ODT
HALT    --> Transfer to software ODT
NXM     --> Kernel trap to 4 exception
Powerup --> Start execution at location 0
```

SYSHALT=ODTROM, LEVEL7 any value

```
BREAK   --> Transfer to ROM ODT
HALT    --> Transfer to ROM ODT
NXM     --> Kernel trap to 4 exception
Powerup --> Transfer to ROM ODT
```

SYSHALT=HANG, LEVEL7 any value

```
BREAK   --> Hang
Halt    --> Hang
NXM     --> Kernel trap to 4 exception
Powerup --> Start execution at location 0
```

SYSHALT=USER, LEVEL7 any value

```
BREAK   --> ODT or Hang (user choice;  see  section  (2)
            of Figure 4-3)
HALT    --> ODT or Hang (user choice;  see  section  (2)
            of Figure 4-3)
NXM     --> Transfer to user routine $BHUSR (see section
            (2) of Figure 4-3)
Powerup --> Start execution at location 0
```

User-Written Error-Handling Routines $BHUSR, $NHUSR, and $NUHNG

If you specify SYSHALT=USER or LEVEL7=USER, you can transfer control to your own error-handling routine when a BREAK or nonexistent memory reference occurs. In addition, you can choose whether to transfer to software ODT or ROM ODT for other situations. Figure 4-3 shows examples of the three required types of user interface routines. Choose the one required by your application, as determined by your choice of break, nxm, syshalt, level7, and board jumpers. Edit a copy of FALUSR.MAC supplied on the MicroPower/Pascal kit, extract the section applicable to your application, insert the global address of your error-handling routine in place of "xxx" at point (4), and assemble. Write your error-handling routine and assemble. Merge $BHUSR, $NHUSR, or $NUHNG from FALUSR.MAC, together with your error-handling routine, with the rest of the kernel.

```
        .title  FALUSR - Sample FALCON interface modules
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1984 BY DIGITAL EQUIPMENT CORPORATION.
;               ALL RIGHTS RESERVED.
;
        .mcall  MACDF$,IODF$
        .mcall  FALDF$

        macdf$
        iodf$
        faldf$

        krn0$   orig=YES

; Functional Description:
;
;       This module is a sample interface module showing how to write
;       a kernel module to handle break, halt, NXM and trap to 4 on
;       a FALCON or FALCON-PLUS. A user written routine is required for
;       the following KXT11 macro parameters:
;
;               nxm=HALT, break=TRAP, syshalt=USER
;               User written module must resolve global symbol: $NHUSR
;       or
;               nxm=HALT, break=TRAP, syshalt=HANG, level7=USER
;               User written module must resolve global symbol: $NUHNG
;       or
;               nxm=TRAP, break=HALT, syshalt=USER
;               User written module must resolve global symbol: $BHUSR
;
;-------------------------------------------------------------------------
(4)     .sbttl  $NHUSR
;+
;
;       nxm=HALT, break=TRAP, syshalt=USER
;       User written module must resolve global symbol: $NHUSR
;
;       This module must define the Break vector at 140.
;
; One of the following globals should also be included depending on the
; whether or not the ODT ROM is present.
; Note that these modules also redefine vector
; 140, so the /I switch to merge should be used to include these modules
; and the user written routine should be included in a second, incremental
; merge operation.
;
;       $NHODR          ; Should be included if ODT rom is present
;       $NHODT          ; Should be included if Macro ODT is present
;
; The commands to MERGE to cause this are:
;
;       Run MERGE
;       CONFIG=CONFIG,PAXU/I
;       Global Symbol? $NHxxx
;       CONFIG=CONFIG.MOB,FALUSR
;       ^C
;
;       where $NHxxx is either $NHODR, $NHODT.
;
;-
        abk$

; Specify vector data for T-11
; Define Break vector at location 140

$A:
.       = $A + V.FBRK           ; Set Break vector

$NHUSR::
```

(Continued on next page)

Figure 4-3 Sample User Error-Handling Interface Routines

```
(4)             .word   xxx             ; Address of user break vector
                .word   PR.7            ; routine

;-------------------------------------------------------------------

(2)             .sbttl  $NUHNG

;+
;       nxm=HALT, break=TRAP, syshalt=HANG, level7=USER
;       User written module must resolve global symbol: $NUHNG
;
;       This routine defines the dispatch needed for a user break
;       handling routine through vector 140.
;
;       BREAK/BHALT must be jumpered to trap to 140 (break=TRAP
;       option).
;-
        abk$

$A:

.       = $A + V.FBRK           ; Set Break vector

$NUHNG::
(4)             .word   xxx             ; Address of user break vector
                .word   PR.7            ; routine

;-------------------------------------------------------------------

(3)             .sbttl  $BHUSR
;+
;
;       nxm=TRAP, break=HALT, syshalt=USER
;       User written module must resolve global symbol: $BHUSR
;
;       This module defines the NXM vector at location 140.
;
;       NXM must be jumpered to trap to 140 (nxm=TRAP option).
;
; One of the following globals should also be included depending on the
; desired BREAK/HALT option. Note that these modules also redefine vector
; 140, so the /I switch to merge should be used to include these modules
; and the user written routine should be included in a second, incremental
; merge operation.
;
;       $BHODR          ; Should be included if ODT rom is present
;       $BHODT          ; Should be included if Macro ODT is present
;       $BHHNG          ; Should be included if HANG is desired on HALT/BREAK
;
; The commands to MERGE to cause this are:
;
;       Run MERGE
;       CONFIG=CONFIG,PAXU/I
;       Global Symbol? $BHxxx
;       CONFIG=CONFIG.MOB,FALUSR
;       ^C
;
;       where $BHxxx is either $BHODR, $BHODT, $BHHNG.
;
;-
        abk$

$A:

.       = $A + V.FBRK           ;Setup NXM vector

$BHUSR::
(4)             .word   xxx             ;Address of user-written NXM
                .word   PR.7            ;routine

.end
```

Figure 4-3 Sample User Error-Handling Interface Routines (Cont)

KXT11C

4.1.4.5 **KXT11C** - If KXT11C is the processor type specified in the PROCESSOR macro, you must also include the KXT11C macro to provide information to configure the kernel for the KXT11-C peripheral processor and include the correct power-up module.

Syntax

```
KXT11C bhalt[=NO],reset[=IGNORE],clock[=OFF],power[=BOOT],map[=0]
```

BHALT
Indicates if Q-BUS debug traps (B-halts) should call the routine at $KXTDB. You can put a PASDBG breakpoint there.

NO Provides no debug traps on B-halt (default)

YES Provides debug traps on B-halt

RESET
Indicates the action to take on a Q-BUS reset.

IGNORE KXT11-C ignores all Q-BUS resets (default)

BOOT Q-BUS resets cause a KXT11-C powerup

RSTBOT Q-BUS resets cause a KXT11-C reset then powerup

INTRPT Simulates a hardware interrupt through vector 220. You can connect to the interrupt and do any processing needed by the application. When finished, you can restart the system by exiting the interrupt service routine with an RTS PC instruction.

CLOCK
Indicates whether to enable the clock interrupts on the KXT11-C.

OFF Clock is not enabled (default)

ON Clock is enabled

POWER
Indicates the action to take on a power fail and subsequent power restart.

BOOT No power fail or power restart processing is done. Every trap through the restart vector is treated like a power-up, unless the bhalt or reset flags are set, in which case action is taken for them instead (default).

NONVOL Uses the power fail and power restart subroutines. When debugging with PASDBG, the action taken on INIT/RESTARTs depends on the contents of location 175002. Prior to restarting, set bit 9 for a powerup, or set bit 10 to test power recovery. The kernel simulates a trap through the interrupt vector

at 214 when power returns if your application has connected to this interrupt. You can connect a routine to the vector to do any restart processing your application needs. When the interrupt completes, the system reinitializes to reconfigure the peripheral chips, which were not protected during power fail. Your routines can use the kernel's Recovery Indicator (KXT$RI) to tell the difference between a cold start and a restart. KXT$RI will be nonzero if a restart is in progress.

MAP

Sets the KXT11-C memory configuration. The map parameter must be set to an integer between 0 and 7 (default is 0) that corresponds to the memory map jumper settings on the KXT11-C. The parameter selects the address of the power fail flags that native firmware will also use. The last two words (highest address) of native RAM are used for that purpose, and 60 bytes before that are privately used by native firmware. Thus, the last 64 bytes of native RAM must not be configured by the MicroPower/Pascal MEMORY macro.

The macro also adds a trap handler (module KSLU2) so that the standard vector for the multiprotocol chip is split into four separate vectors for each channel. The vectors that are available to your application are the following:

- 140 -- Channel A receive character interrupt
- 144 -- Channel A transmit character interrupt
- 150 -- Channel A receive error interrupt
- 154 -- Channel A Modem Control Interrupt
- 160 -- Channel B receive character interrupt
- 164 -- Channel B transmit character interrupt
- 170 -- Channel B receive error interrupt
- 174 -- Channel B Modem Control Interrupt

Power-Up and Power-Fail Routine

The functions provided by the power-up routine are listed below. The power fail and restart functions are provided by separate subroutines and are not built into your application unless you select the NONVOL option for the POWER parameter in the configuration macro. If not selected, power fail and power restart are replaced by the standard power-up routine.

- **Power fail:** At power fail, the power-fail flag KXT$RI is set, any I/O is stopped, and a "wait" is issued.
- **Power restart:** Subsequent powerup simulates an interrupt through vector 214. You may connect to that vector during powerup if an interrupt service routine is to be called during power restart. You then have the opportunity to modify or set flags in the battery backed-up portion of the RAM before reinitializing the kernel.

- **Power-up:** At initialization time, the kernel's RAM is zeroed; then a standard powerup is performed. The entry point to the power-up routine is $KXTPU and is declared as a global.

- **B-halt:** Since a b-halt signal from the Q-BUS simulates a trap through the power-up vector at 24, the power-up routine must also handle that condition. If B-halt traps are enabled and one is received, the power-up module passes through a debug point ($KXTDB) and then returns from the interrupt.

- **Reset:** Q-BUS resets also trap through the power-up vector at 24 and must be handled by the power-up routine. If Q-BUS resets are not being ignored, the kernel takes one of three actions as selected by the configuration macro. The power-up module can be entered, an I/O reset can be issued and then the power-up module entered, or a "reset interrupt" can be simulated through the vector at 220, giving you the ability to take any action desired.

Power-Fail/Power-Restart Rules

- You must ensure that all code -- both kernel and user -- has been located in ROM and that all stack areas, kernel RAM, and user nonvolatile read/write memory has been located in battery backed-up RAM.

- In the restart interrupt handling routine, you must decide what action the application needs when the power restart occurs. If no action is needed, you do not need to connect to the interrupt. Your initialization routines may test the kernel's 1-word restart indicator (KXT$RI); if nonzero, a power restart rather than a cold start is in progress.

During debugging, the following rules apply.

- When you use the debugger (PASDBG) along with the NONVOL feature, the LOAD and INIT/RESTART commands from the debugger enter the power-up module but bypass the native firmware, where the type of power-up is defined -- cold start or restart. Therefore, you must manually set flags to cause the desired action before issuing the restart command, as explained below.

- To down-line load an application that is using the NONVOL feature, use the PASDBG ODT mode to set bit 9 at octal address 175002. The load command can now be used. To restart a loaded application, use the ODT mode to set the same bit; then issue the INIT/RESTART command.

- To test the power-fail and restart sequence, use ODT mode to set bit 10 of octal location 175002 and issue an INIT/RESTART command.

- If you load or restart an application using the NONVOL feature without first setting one of the flags, the target will crash. If that happens, enter ODT mode and follow the instructions above to repeat the down-line load.

- You must decide in the interrupt-handling restart routine what action the application needs when the power restart occurs. If no action is needed, the interrupt does not have to be connected to. Your initialization routines may test the kernel's 1-word recovery indicator (KXT$RI); if the value is

nonzero, a power restart rather than a cold start is in progress.

MEMORY

4.1.4.6 **MEMORY** - Use the MEMORY macro to describe a single, contiguous segment of memory in the target hardware. You must specify the MEMORY macro once for each contiguous ROM or RAM segment. You can use the MEMORY macro a maximum of 168 times in a program.

To describe your target application memory, you need to know the following specifics about it: the type of memory -- read-only memory (ROM) or read/write memory (RAM) -- the base address of the memory segment in 64-byte increments, the size of the segment in 64-byte increments, and the presence or absence of memory parity checking.

If you specify memory parity checking in the MEMORY macro, the MPT trap processor is automatically included in the kernel.

All RAM memory used in a MicroPower/Pascal application is assumed to be volatile, regardless of its physical characteristics. No special power-fail restart support is provided for nonvolatile RAM.

NOTE

The last two words (highest address) of native RAM are used for the power fail flags, and 60 bytes before that are privately used by native firmware. Thus, the last 64 bytes of native RAM must not be configured by the MicroPower/Pascal MEMORY macro.

Syntax

```
MEMORY base[=0],size[=NNNNN],type[=RAM],parity[=NO],csr[=0]
```

base
: The base address of the memory segment divided by 100 if specified in octal or by 64 if specified in decimal. The default base is 0.

size
: The size of the memory segment, expressed in 64-byte units. You must supply that value.

type
: The type of memory, ROM or RAM. The default memory type is RAM.

parity
: The memory parity option, which is present or not depending on the kind of memory used. The default is no parity checking.

csr

The address of the memory parity controller's control status register (CSR) for that segment of memory. The default csr is 0. (This parameter is meaningful only if parity=YES.)

Example

```
MEMORY base=1000, size=256., type=RAM, parity=NO
```

That statement describes a memory segment consisting of 16,384 bytes of RAM, originating at location 100000 -- 32768(decimal). The segment is not parity-checked. Note that in a DIGITAL-supplied configuration file, the size value 256(decimal) in that example would appear as <8.*32.>, an equivalent expression intended to indicate an 8K-word segment size, by notational convention.

PRIMITIVES

4.1.4.7 **PRIMITIVES** - Use the PRIMITIVES macro to select the primitive-service modules to be included in the kernel for your application. Primitives can be selected individually only when optimize=YES has been specified in the SYSTEM macro. You can specify individual primitives by name, or you can specify all primitives with one argument (ALL). If you specify optimize=YES in the SYSTEM macro and if you also include the PRIMITIVES macro, you must include the PRIMITIVES macro between the SYSTEM macro and the ENDCFG macro in the configuration file.

NOTE

If you use the special "repetitive merge" method of optimizing primitive modules, which involves the use of MERGE's auxiliary file, do not include the PRIMITIVE macro in the configuration file for the kernel build steps. Specify optimize=YES in the SYSTEM macro, however. See Section 5.2.3 for information on optimizing the kernel with respect to primitive modules by means of the "repetitive merge with auxiliary file" method.

If you specify optimize=NO in the SYSTEM macro, all primitives are included by default. Do not include the PRIMITIVES macro in the configuration file in that case.

Six primitives can be specified each time the PRIMITIVES macro is used in the configuration file. Chapter 3 of the MicroPower/Pascal Runtime Services Manual describes the primitives and gives the 4-character mnemonics you use to specify them. (Omit the trailing dollar sign in the primitive name; for instance, the WAIT$ primitive is specified as WAIT in the PRIMITIVES macro.)

Syntax

```
PRIMITIVES p1,p2,p3,p4,p5,p6
```

p1..p6
: Represent either the names of up to six primitives that can be specified in each use of the PRIMITIVES macro or the specification ALL to include all primitives.

Examples

```
PRIMITIVES  ALL                 (include all primitives)

PRIMITIVES  CRST,DAPK,DEXC      (include specific primitives)
```

PROCESSOR

4.1.4.8 **PROCESSOR** - Use the PROCESSOR macro to describe the target processor and the hardware options on it. The processor-type symbol may be L1123 (LSI-11/23), L112 (LSI-11 or LSI-11/2), KXT11C (KXT11-C), T11 (SBC-11/21 FALCON), or T11+ (SBC-11/21 FALCON-PLUS). Your target configuration may also include the KT-11 memory-management unit and either of two floating-point options (FP-11 or FIS). The PROCESSOR macro must be included in the configuration file, immediately preceding or following the SYSTEM macro.

If the PROCESSOR macro specifies that the target includes the FP-11 or the FIS floating-point instruction set, the FP-11 or FIS trap processor is included in the kernel. Note that the KEF11 and FPF11 hardware options are equivalent for configuration purposes; both support the FP-11 instruction set.

When you use the PROCESSOR macro, you can also indicate where the effective vector area ends by specifying the next free address above it. This permits kernel code to begin at a location below the end of the standard vector area -- below 1000 for an LSI processor or below 400 for a T11 processor. If the vector parameter is not specified, the standard vector area for the target processor is assumed.

Syntax

```
PROCESSOR mmu=[NO],fpu,type[=L1123],vector[=1000]
```

mmu
: Specifies the presence (YES) or absence (NO) of the KT-11 memory-management unit.

fpu
: Either the FP-11 floating-point option (specify fpu=FP11) or the FIS floating-point option (specify fpu=FIS). If the target processor does not have floating-point hardware, omit the fpu argument.

type
: The type of processor: L1123, L112, KXT11C, T11, or T11+. The default is L1123.

vector
The address of the next free location above the highest interrupt or trap vector the application uses. The default is 1000 for an L1123 or L112 processor and 400 for a T11 processor.

Examples

```
PROCESSOR mmu=YES, fpu=FP11, type=L1123

PROCESSOR fpu=FIS, type=L112, vector=400

PROCESSOR type=T11
```

NOTE

The PROCESSOR macro must precede the SYSTEM macro in the configuration file if you specify type=T11 or type=T11+ in the PROCESSOR macro and if you specify optimize=NO in the SYSTEM macro.

RESOURCES

4.1.4.9 **RESOURCES** - Use the RESOURCES macro to modify the default amounts of read/write memory (RAM) to be included in the kernel's data space for the following purposes:

- Message packets -- the free-packet pool
- System data structures other than packets -- the free-memory pool
- The kernel-interrupt stack
- The free-RAM table, used by MIB during memory image building

You include the RESOURCES macro in the configuration file only if you specify optimize=YES in the SYSTEM macro.

The free-packet pool and the free-memory pool form the kernel's system-common memory area. See Chapter 2 of the MicroPower/Pascal Runtime Services Manual for information on system-common memory organization before using the packets or structures arguments of the RESOURCES macro. The default free-packet pool permits runtime allocation of 20 packets; thus, 20 packets are available for use by processes at the same time. That number has been found adequate for most applications and may be excessive for some. Since packets are reusable, the optimum number of packets is generally a space/performance tradeoff, but too few packets relative to application needs can cause obscure real-time problems. If space is not an overriding consideration, specify the maximum number of packets that could be required simultaneously by all processes in your application.

The default free-memory pool provides 3,000 bytes for allocation of all other system data structures allocated at runtime: PCBs,

semaphores, and ring buffers. (See Chapter 2 of the MicroPower/Pascal Runtime Services Manual for structure sizes.) The default pool size is sufficient for complex unmapped applications with many processes and structures and should be adequate for many mapped applications. Mapped applications require a larger free-memory pool than do unmapped applications -- assuming the same number of processes and structures -- due in part to the much larger mapped PCB size. In either case, if the pool is too small, a runtime failure will occur because of a process's inability to create another process or a needed data structure or the kernel's inability to initialize all static processes. If space is not a constraint, start with 4K bytes for the free-memory pool for a large mapped application; the pool size can be reduced in a later build cycle if experience shows it to be excessive. (While debugging, you can use the SHOW FREE STRUCTURES command to assess the amount of unused pool space at various points of system operation.)

The default kernel-interrupt stack size -- currently 104 bytes unmapped or 144 bytes mapped -- is a "safe" value and is sufficient for virtually all applications. Additional kernel stack might be needed if the application software includes -- in addition to three standard device handlers of different hardware priorities -- a user-written interrupt service routine that pushes more than five words (unmapped) or seven words (mapped) onto the stack. In any case, you should allocate additional stack space only if you encounter kernel-stack overflow problems at runtime. You can determine whether a kernel-stack overflow has occurred by examining the guard word at kernel location $KSTKL. If overflow has occurred, the guard word will contain a value other than the preset octal value 42557 -- defined by the kernel symbol $GRD.

The free-RAM table is used by MIB to keep track of unallocated RAM memory areas during image building. That table is retained in the memory image as part of kernel data. The default table allows five entries. Overflow of the table at build time is indicated by the MIB warning message "?MIB-W-Kernel free RAM table too fragmented." Some space may be wasted in the memory image if the table overflows.

Memory in addition to that specified by RESOURCES is included in the kernel for its fixed-size private data area and for interrupt dispatch blocks (IDBs). The amount of space allocated for IDBs depends on the number of vectors specified in DEVICES; see Chapter 8 of the MicroPower/Pascal Runtime Services Manual for details.

Syntax

```
RESOURCES   stack[=..KIS],packets[=20.],structures[=3000.],
            ramtbl[=20.]
```

stack
: The size, in bytes, of the kernel interrupt stack. The default size, supplied by the kernel symbol ..KIS, is 104 bytes unmapped or 144 bytes mapped. The recommended method of increasing the stack size, where necessary, is to specify the symbol ..KIS plus an increment n, in an expression of the form <..KIS+n>. (The default size could change in a future version; use of the symbol guards against a potential version skew.)

packets
: The number of message packets that can be allocated concurrently from the free-packet pool. The default is 20 packets.

structures
: The size, in bytes, of the free-memory pool, from which system data structures are allocated. The default is 3,000 bytes.

ramtbl
: The size, in bytes, of the kernel free-RAM table. You can use this argument to increase the number of 4-byte entries that can be accommodated. The default size is 20 bytes (five entries). If specified, the value must be a multiple of 4.

Example

```
RESOURCES packets=30., structures=4096., ramtbl=32.
```

This macro statement allocates space for 30 message packets, 4K bytes of memory for other dynamic structures, and a 32-byte free-RAM table accommodating eight entries. The macro also allocates a standard size kernel stack by default.

SYSTEM

4.1.4.10 **SYSTEM** - The SYSTEM macro controls a set of default kernel-configuration values and determines whether debugging support is included in the kernel. The SYSTEM macro must appear in the configuration file and must be the second or third macro in the file, either preceding or following the PROCESSOR macro.

The two SYSTEM macro arguments, optimize and debug, take YES or NO values. If you specify NO for the optimize argument, the SYSTEM macro supplies all default values for the RESOURCES, TRAPS, and PRIMITIVES macros, and those three macros may not appear in the configuration file. Effectively, optimize=NO produces a standard kernel-software configuration, with default system-common memory resources, all standard trap processors, and all primitive service modules.

If you specify YES for the optimize argument, the default values for the RESOURCES, TRAPS, and PRIMITIVES macros are inhibited, and you can use those macros to tailor the kernel to your application requirements. The RESOURCES macro lets you specify the amount of RAM to be included in the kernel's data space for various purposes. (RESOURCES must appear in the configuration file in that case.) The TRAPS and PRIMITIVES macros let you specify the set of trap processors and primitive service modules to be included in the kernel. (If you use the special "repetitive MERGE" method of optimizing the kernel with respect to primitive modules, as described in Section 5.2, you must omit the PRIMITIVES macro from the configuration file for the final kernel merge step.)

If you specify YES for the debug argument, the debug service module (DSM) is included in the kernel. Include the DSM only if you intend to use the PASDBG symbolic debugger to load and debug the application. If you do not plan to use PASDBG, or will use PASDBG for loading only (LOAD/EXIT command), the DSM must not be included in the kernel (debug=NO). If the DSM is present, it automatically gains control at start-up time; that would cause a system crash in a nondebug target configuration or if PASDBG LOAD/EXIT were used.

NOTE

If you specify optimize=YES but do not include the RESOURCES macro in the configuration file, an assembly error is generated. The PROCESSOR macro must precede the SYSTEM macro if optimize=NO and the PROCESSOR type value is T11 or T11+.

Syntax

SYSTEM optimize[=NO],debug[=NO]

optimize
: Specifies whether you want to optimize the kernel by using additional configuration macros (YES) or want the default kernel configuration (NO). The default is NO.

debug
: Specifies whether the debug service module should be included in the kernel. The default is NO.

Example

```
SYSTEM optimize=YES,debug=YES
```

TRAPS

4.1.4.11 **TRAPS** - Use the TRAPS macro to define the set of trap-processing modules required for your application. (You use this macro only if the SYSTEM macro specifies optimize=YES.) You need to include in the kernel the processors that handle the kinds of traps that your application may generate. If a given trap processor is not included and if the corresponding trap occurs, the trap is effectively ignored.

A trap to 10 (T10) occurs if an application issues an illegal instruction. A trap to 4 (TR4) occurs if an application refers to an odd address or to a nonexistent location or if a bus timeout occurs. (Note that the T-11 processor does not generate a TR4 but does generate the BRK trap that the other processors do not. BRK denotes the T-11 processor's break-character trap.) Since assembly language EMT instructions generate emulator traps (EMTs), you need the trap processor if you explicitly use EMT instructions in the application.

A memory-management trap (MMU) occurs when an application refers to a virtual address that is not mapped to a physical address. A memory parity trap (MPT) occurs if the memory parity hardware detects a memory parity error. Floating-point traps (FPP or FIS) occur when the floating-point hardware detects a floating-point exception or error. Each of these four traps is generated only if the specific hardware option is present on the target processor.

If the PROCESSOR macro specifies that the target processor's hardware options include the memory-management unit, the MMU trap module is included automatically. Likewise, if PROCESSOR specifies that the

target includes the FP-11 or FIS floating-point processor option, the FPP or FIS trap module is included. Memory parity hardware specified in the MEMORY macro implies the MPT trap module.

If you specify ALL in the TRAPS macro, the TR4, T1Ø, BPT, EMT, and TRP trap processors are included; BRK is also included if the processor type is T11 or T11+. If the SYSTEM macro specifies optimize=NO, TRAPS ALL is assumed by default.

Syntax

```
TRAPS t,t,t,t,t,t
```

t

Either the name of one of the trap processors in the following list or, if you use the ALL argument, the TR4, T1Ø, BPT, EMT, and TRP processors. You can specify up to six arguments per macro call.

The trap processors are listed below:

BPT	B[illegible]point instruction trap
BRK	T-11 processor break character trap
EMT	EMT instruction trap
FIS	FIS exception trap
FPP	FP11 exception trap
MMU	Memory-management fault
MPT	Memory parity error
T1Ø	Trap to 1Ø (reserved instruction)
TR4	Trap to 4 (bus timeout)
TRP	TRAP instruction trap

Example

```
TRAPS ALL

TRAPS TR4,T1Ø,EMT,TRP
```

4.2 PREFIX MODULES

To build a DIGITAL-supplied handler process, you must assemble or compile the corresponding prefix module and merge it with one of the three device-handler object libraries. This section describes how the prefix modules function in the build procedure, how to edit them, and how to assemble or compile them.

The supplied system processes fall into two major categories: those for use on a KXT11-C peripheral processor and those for use on any other PDP-11 family microprocessor.

KXT11-C Handler Library DRVK.OLB

The library of handlers for use with the KXT11-C processor is called DRVK.OLB. That library contains the following device handlers in object module format:

```
CLOCK.OBJ    (Line clock process)
DDDRVK.OBJ   (TU58 DECtape II)
KKDRVK.OBJ   (KXT11-C arbiter/slave protocol)
```

```
QDDRVK.OBJ   (KXT11-C DMA transfer controller)
XLDRVK.OBJ   (KXT11-C asynchronous serial line)
XSDRVK.OBJ   (KXT11-C synchronous serial line)
YKDRVK.OBJ   (KXT11-C parallel port and counter/timer)
```

Mapped and Unmapped Handler Libraries DRVM.OLB and DRVU.OLB

The handlers for other supported processors are further divided into two libraries. The library of unmapped device handlers for other processors is called DRVU.OLB; the library of mapped handlers for other processors is DRVM.OLB. Each of those libraries contains the following device handlers in object module format:

```
AADRV.OBJ   (ADV11/AXV11 A-to-D converter)
CLOCK.OBJ   (line clock process)
DDDRV.OBJ   (TU58 DECtape II)
DLDRV.OBJ   (RL02 disk)
DUDRV.OBJ   (MSCP disk)
DYDRV.OBJ   (RX02 diskette)
KWDRV.OBJ   (KWV11 real-time clock)
KXDRV.OBJ   (KXT11-C arbiter/slave protocol)
XADRV.OBJ   (DRV11-J parallel line)
XLDRV.OBJ   (DLV11 serial line)
XPDRV.OBJ   (DPV11 synchronous line)
YADRV.OBJ   (DRV11 parallel line)
YFDRV.OBJ   (SBC-11/21 parallel line)
```

Although the different libraries may have modules of the same name, such as DDDRV.OBJ, the modules are NOT interchangeable. You must use the library DRVK.OLB when building an application for a KXT11-C target system, and either DRVM.OLB or DRVU.OLB (mapped or unmapped) when building an application for any other target system.

4.2.1 Prefix Module Function

The prefix module has two purposes. First, it statically allocates the impure area required for the device handler. Second, it defines certain device-specific parameters. The prefix module defines, for example, the number of controllers that the handler supports, the address of the CSR for each controller, the address of the interrupt vector for each CSR, and the units supported on each controller.

The prefix module provides the handler with a table of controller vectors and CSR addresses; it also allocates an impure area for each controller. The handler process uses that storage.

The software distribution kit contains the handler prefix modules in either MACRO-11 or Pascal source format. The supplied prefix modules specify default parameters. You can edit a copy of a prefix module to change the parameters before you assemble or compile it and merge it with the handler library.

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. Table 4-2 shows the DIGITAL-supplied system processes for KXT11-C peripheral processors.

Table 4-1
Processes and Prefix Modules for Mapped and Unmapped PDP-11 Targets

Process	Device Name	Prefix Module
ADV11 or AXV11 A-to-D converter	AA	AAPFX.PAS
Line-frequency clock	CK	CKPFX.MAC
TU58 DECtape II	DD	DDPFX.MAC
RLØ2 disk	DL	DLPFX.MAC
MSCP disk	DU	DUPFX.MAC
RXØ2 diskette	DY	DYPFX.MAC
KWV11 real-time clock	KW	KWPFX.PAS
KXT11-C arbiter/slave protocol	KX	KXPFX.MAC
DRV11-J (four ports) parallel line interface	XA	XAPFX.MAC
DLV11 asynchronous serial line interface	XL	XLPFX.MAC XLPFXD.MAC XLPFXF.MAC
DPV11 synchronous serial line	XP	XPPFX.MAC
DRV11 parallel line interface	YA	YAPFX.PAS
SBC-11/21 parallel line interface	YF	YFPFX.MAC

Table 4-2
Processes and Prefix Modules for KXT11-C Peripheral Processors

Process	Device Name	Prefix Module
Line-frequency clock	CK	CKPFX.PAS
TU58 DECtape II	DD	DDPFXK.MAC
KXT11-C arbiter/slave protocol	KK	KKPFXK.MAC
KXT11-C DMA transfer controller	QD	QDPFXK.MAC
KXT11-C asynchronous serial line interface	XL	XLPFXK.MAC
KXT11-C synchronous serial line interface	XS	XSPFXK.PAS
KXT11-C parallel port timer/counter	YK	YKPFXK.MAC

Prefix module ABPFX.PAS, not listed in Table 4-1, is a special case in that no system process corresponds to it. That prefix module must be merged with any user-written Pascal application process that uses the DIGITAL-supplied procedure WRITE_ANALOG_WAIT. That procedure supports digital-to-analog output operations on an AAV11-C or AXV11-C device. See Appendix G of the MicroPower/Pascal Runtime Services Manual for details.

NOTE

The prefix module DSPPFX.MAC corresponds to the Directory Service Process (DSP), which is part of the MicroPower/Pascal file system. See Appendix A for a description of the DSPPFX.MAC file and information on building elements of the file system.

4.2.2 How Prefix Modules Fit into the Build Procedure

Building an application consists of four major phases, as follows:

1. Building the kernel
2. Building DIGITAL-supplied static processes
3. Building user-written static processes
4. Debugging and rebuilding the application

You use the prefix modules when you build DIGITAL-supplied static processes (in phase 2, above).

Refer to Chapter 3 for an overview of the build procedure.

4.2.3 Editing Prefix Modules: Mapped and Unmapped PDP-11 Target Processors

This section illustrates the pertinent portions of each prefix module and describes the statements and fields you may need to modify in order to reflect your target I/O configuration. Certain minor or tutorially nonessential items in the prefix module source files have been omitted from the representations in this section. The omitted items include listing-control directives, copyright text, and extensive comment sequences.

4.2.3.1 ADV11 or AXV11 A-to-D Converter (AA) Handler - The AAPFX.PAS prefix module for the ADV11-C/AXV11-C Analog Input Converter (AA) handler is reproduced in Figure 4-4. The AA handler supports either an ADV11-C Analog Input board or the input side of an AXV11-C Analog Input/Output board. The following paragraphs describe the user-modifiable constant declarations in the AAPFX.PAS prefix module.

The aa_process_priority constant sets the process software priority to the standard value for the AA handler. The aa_fork_priority constant sets the handler's interrupt service routine fork priority. You need not modify either of those values unless you attempt to fine-tune the priority relationship between the AA handler and other processes in the application.

The aav_hardware_priority constant sets the default ADV11-C/AXV11-C device interrupt priority.

The aav_csr_addr constant specifies the factory configured CSR address for the device, and the aav_dbr_addr constant specifies the corresponding data buffer register (DBR) address. (The DBR address always follows the CSR address by two bytes.) Modify those values if your device is not configured for the standard CSR address.

The aav_vec_addr constant specifies the factory-configured normal interrupt vector address for the device. The aav_erv_addr constant specifies the corresponding error interrupt vector address. (The error vector address always follows the normal vector address by four bytes.) Modify those values if your device is not configured for the standard vector address.

The aa_device constant specifies the device type symbol to be returned by the Get Characteristics function; the default definition of aa_device reflects an AXV11-C device. If your device is an ADV11-C, comment out, or delete, the default definition and remove the leading comment bracket preceding the alternative definition.

```
{
 COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.
}

MODULE AAPFX;

(*
   ADV11-C (AXV11-C) A/D Handler Prefix Module

   Defines CSR and VECTOR addresses, driver interrupt priority
   (aav_hardware_priority (aa$hpr) ), and driver process priority
   ( aa_process_priority (aa$ppr) ).
-)

[$NOLIST]
%INCLUDE 'DEF:IOPKTS'
[EXTERNAL($AADRV)] PROCEDURE AADRV;
  EXTERNAL;                                 { Pulls the driver from library }
[$LIST]

        { For differing hardware configurations, change the values
          of the symbols in the CONST section below. }

CONST
  aa_process_priority = 16#;                { Standard process priority, }
  aa_fork_priority = 16#;                   { Standard fork priority }
  aav_hardware_priority = 4;                { and interrupt priority, }
                                            { for ADV11-C device}
  aav_csr_addr = %O'17#4##';                { Address of CSR }
  aav_dbr_addr = %O'17#4#2';                { Address of DBR }
  aav_vec_addr = %O'4##';                   { Address of vector }
  aav_erv_addr = %O'4#4';                   { Address of error vector }

                { Type of A/D converter }

{ aa_device    = IT$ADV11C;                 { ADV11-C 16 channel A/D with no D/A }
  aa_device    = IT$AXV11C;                 { AXV11-C 16 channel A/D with 2 D/A
                                              channels }

VAR
  proc_prio     : [GLOBAL (aa$ppr)] INTEGER;
                { Driver process priority }

  fork_prio     : [AT(aa_fork_priority), GLOBAL (STSFPR)] UNSIGNED;
                { Global constant }

  dev_prio      : [GLOBAL (aa$hpr)] INTEGER;
                { Hardware interrupt priority}

  dev_type      : [GLOBAL (aa$htp)] IOSTYPE_REAL_TIME;
                { Type of hardware device: ADV11-C or AXV11-C, }

  csr           : [AT(aav_csr_addr), GLOBAL ($aacsr)] UNSIGNED;
                { CSR address }

  dbr           : [AT(aav_dbr_addr), GLOBAL ($aadbr)] INTEGER;
                { Data Buffer Register address }

  vector        : [AT(aav_vec_addr), GLOBAL ($aavec)] UNSIGNED;
                { interrupt vector address }

  err_vector    : [AT(aav_erv_addr), GLOBAL ($aaerv)] UNSIGNED;
                { error interrupt vector address }

 { Declared in AADRV module }

  version       : [EXTERNAL ($aaver)] PACKED ARRAY[1..n] OF CHAR;

[ GLOBAL ($AAPFX) ] PROCEDURE aapfx;

BEGIN
   proc_prio := aa_process_priority;        { Final process priority }
   dev_prio :=
        aav_hardware_priority * #;          { In PSW placement }
   dev_type := aa_device;                   { Type of A/D converter }
   version := 'X#1.##';                     { Note: Should match the
                                              version in AADRV module }
END;   {PROCEDURE aapfx}
```

Figure 4-4 ADV11-C/AXV11-C Handler Prefix Module (AAPFX.PAS)

4.2.3.2 **AAV11-C/AXV11-C D-to-A Output Prefix File, ABPFX.PAS** - The DIGITAL-supplied Pascal procedure WRITE_ANALOG_WAIT supports digital-to-analog output operations through either an AAV11-C Analog Output board or the output side of an AXV11-C Analog Input/Output board. The WRITE_ANALOG_WAIT procedure is described in Appendix G of the MicroPower/Pascal Runtime Services Manual. The ABPFX.PAS prefix module supplies hardware configuration information for the WRITE_ANALOG_WAIT procedure declaration. The ABPFX.PAS prefix module must be merged with any user process that incorporates the WRITE_ANALOG_WAIT procedure, and with RHSLIB.OLB.

The ABPFX.PAS prefix module is reproduced in Figure 4-5. The following paragraphs describe the constant and variable declarations in that prefix module.

The ab_dac_addr constant specifies the address of the first output buffer register on the device to be supported. The default definition of ab_dac_addr assumes the standard configuration for an AAV11-C device. The AAV11-C has four buffer registers, DAC 0 through DAC 3, and the standard address assignment of DAC 0 is 170440.

The alternative, commented-out definition of ab_dac_addr assumes the standard configuration for an AXV11-C device. The AXV11-C has two output buffer registers, DAC 0 and DAC 1, and the standard address assignment of DAC 0 is 170404, the standard CSR address plus 4. If your device is an AXV11-C or is not configured for the standard base register address, modify the constant declaration section accordingly.

The ab_dac_nchan constant specifies the number of output channels on the device to be supported. The effective declaration of ab_dac_nchan is suitable if the supported device is an AAV11-C. The alternative, commented-out declaration of ab_dac_nchan is suitable if the supported device is an AXV11-C. If your device is an AXV11-C, comment out, or delete, the effective declaration and remove the leading comment bracket preceding the alternative declaration.

```
{
 COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.
}

MODULE ABPFX;

{+
   AAV11-C/AXV11-C Digital to Analog Converter Prefix Module

   Defines buffer addresses
-}

        { For differing hardware configurations, change the values
          of the symbols in the CONST section below. }

CONST
  ab_dac_addr    = %O'170440';          { Standard buffer address for D/A
                                          output buffers for AAV11-C }

{ ab_dac_addr    = %O'170404';          { Standard buffer address for D/A
                                          output buffers for AXV11-C }

  ab_dac_nchan   = 4;                   { Number of output channels for
                                          AAV11-C }

{ ab_dac_nchan   = 2;                   { Number of output channels for
                                          AXV11-C }

VAR
  dac            : [AT(ab_dac_addr), GLOBAL ($abdac)]
                   ARRAY[1..ab_dac_nchan] OF INTEGER;

  dachan         : [AT(ab_dac_nchan),GLOBAL ($abnch)]
                   UNSIGNED;
```

Figure 4-5 AAV11-C/AXV11-C Analog Output Prefix Module (ABPFX.PAS)

4.2.3.3 **Line-Frequency Clock Process (CLOCK)** - The clock process prefix module is reproduced in Figure 4-6. The following paragraphs describe the symbol definitions in the prefix module.

CK$CSR defines the clock's CSR address, if any. If the clock's CSR is not at that address on the target, you must modify the statement. If your target clock has no CSR, do not modify the statement.

CK$VEC defines the clock's interrupt vector address. If the clock does not interrupt at that address on the target, you must modify the statement.

CK$FPR defines the handler's interrupt service routine fork priority.

CK$HPR defines the clock's hardware interrupt priority.

CK$PPR defines the clock process software priority. That software priority value should be higher than any other device handler's process software priority.

CK$TPS defines the number of ticks per second, or frequency, of the clock. If the target's clock frequency is 50 Hz, change that value to 50.

CK$IPR defines the clock process initialization priority. That priority is higher than that of I/O device handlers.

```
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1983, 1984 BY DIGITAL EQUIPMENT CORPORATION. ALL
;  RIGHTS RESERVED.
        .GLOBL  $CLOCK          ; Haul in the clock driver
CK$CSR  ==      177546          ; -> Command/status register address
        ; CHANGE THE ABOVE ADDRESS TO 177520 FOR THE KXT11-CA:
        ;CK$CSR == 177520       ; -> CSR addr for KXT11-CA
;
CK$VEC  ==      100             ; -> Clock interrupt vector
CK$FPR  ==      57344.          ; Clock ISR fork priority (7*65536/8)
CK$HPR  ==      6               ; Clock hardware interrupt priority
CK$PPR  ==      224.            ; Clock process software priority
CK$TPS  ==      60.             ; Number of ticks per second
CK$IPR  ==      251.            ; Process initialization priority
        .end
```

Figure 4-6 Clock Process Prefix Module (CKPFX.MAC)

4.2.3.4 **TU58 (DD), RL02 (DL), MSCP (DU), and RX02 (DY) Handlers** - The TU58 DECtape II device handler prefix module DDPFX.MAC is reproduced in Figure 4-7. Another version, DDPFXK.MAC, reproduced i Figure 4-18, should be used when you are building a DD handler for use with the KXT11-C peripheral processor. The RL02 device handler prefix module is shown in Figure 4-8, the MSCP-class disk prefix module in Figure 4-9, and the RX02 device handler prefix module in Figure 4-10. The four prefix modules are almost identical. The following paragraphs describe the fields of the prefix modules that correspond to the circled numbers on the figures.

1. The **drvcf$** macro contains a field (nctrl) for the number of controllers on the target to be supported by the handler. The dname field specifies the first two characters of the corresponding request-queue semaphore name.

   If you edit the nctrl field to change the number of controllers to 2, you must delete the semicolon in front of the second ctrcf$. Doing so includes the second ctrcf$ macro for the second controller. Note that the name of the first controller is A and that of the second, B. Those names supply the third character of the corresponding request-queue semaphore names -- for example, DYA and DYB for the DY handler.

2. The **ctrcf$** macro gives the name, number of units, CSR and vector addresses, and unit numbers for the controller. You can edit those fields if your controller does not conform to the defaults.

   The field **units=** specifies the unit numbers of the devices attached to the controller. The designation 0:1 refers to Unit 0 and Unit 1. For an RX02, 0 and 1 are the only possible unit numbers, but you can edit that field if you have only one unit <0>. Note that <0,1> and <0:1> are equivalent.

   Note that the interrupt vectors specified in those macros must also be specified in the system configuration file, using the DEVICES macro.

3. The DY$, DD$, and DL$ definitions specify software priorities for the driver processes and the hardware interrupt priority for the controllers. All controllers have the same priorities.

```
.title  DDPFX   - TU58 Device driver prefix module

; COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.

        .mcall  drvcf$
        .mcall  ctrcf$

(3) DD$PPR  ==      175.            ; Process priority
    DD$FPR  ==      175.*256.       ; Fork process priority
    DD$HPR  ==      4               ; TU58 hardware priority
    DD$IPR  ==      250.            ; Process initialization priority

    (1) drvcf$  dname=DD,nctrl=1
(2) ->  ctrcf$  cname=A,nunits=2.,csrvec=<176520,320>,units=<0:1>

        .end
```

Figure 4-7 TU58 Handler Prefix Module (DDPFX.MAC)

```
.title  DLPFX   - RL02 Device driver prefix module

; COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.

        .mcall  drvcf$
        .mcall  ctrcf$

(3) DL$PPR  ==      184.            ; Process priority
    DL$FPR  ==      184.*256.       ; Fork process priority
    DL$HPR  ==      4               ; RL02 hardware priority
    DL$IPR  ==      250.            ; Process initialization priority

    (1) drvcf$  dname=DL,nctrl=1
(2) ->  ctrcf$  cname=A,nunits=1.,csrvec=<174400,160>,unit=<0:0>
;       ctrcf$  cname=B,nunits=2.,csrvec=<174410,164>,units=<0,1>

        .end
```

Figure 4-8 RL02 Handler Prefix Module (DLPFX.MAC)

```
        ; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
        ; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
        ;
        ; COPYRIGHT (c) 1983, 1984 BY DIGITAL EQUIPMENT CORPORATION. ALL
        RIGHTS
        ;    RESERVED.
                .mcall  drvcf$
                .mcall  ctrcf$
(3)     DU$PPR  ==      184.            ; Process priority
        DU$FPR  ==      184.*256.       ; Fork process priority
        DU$HPR  ==      4               ; MSCP hardware priority
        DU$IPR  ==      250.            ; Process initialization priority
             (1)drvcf$  dname=DU,nctrl=1
           (2)->ctrcf$  cname=A,nunits=3.,csrvec=<172150,154>,units=<0:2>
        ;       ctrcf$  cname=A,nunits=3.,csrvec=<174344,154>,units=<0:2>
                .end
```

Figure 4-9 MSCP Handler Prefix Module (DUPFX.MAC)

```
.title  DYPFX   - RX02 Device driver prefix module

; COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.

        .mcall  drvcf$
        .mcall  ctrcf$

(3) DY$PPR  ==      184.            ; Process priority
    DY$FPR  ==      184.*256.       ; Fork process priority
    DY$HPR  ==      5               ; RX02 hardware priority
    DY$IPR  ==      250.            ; Process initialization priority

       (1)drvcf$  dname=DY,nctrl=1
     (2)->ctrcf$  cname=A,nunits=2.,csrvec=<177170,264>,units=<0:1>
;       ctrcf$  cname=B,nunits=2.,csrvec=<177200,270>,units=<0,1>

        .end
```

Figure 4-10 RX02 Handler Prefix Module (DYPFX.MAC)

4.2.3.5 **KWV11-C Real-Time Clock (KW) Handler** - The prefix module for the KWV11-C Real-Time Clock (KW) handler is reproduced in Figure 4-11. The following paragraphs describe the user-modifiable constant declarations in that prefix module.

The kw_process_priority constant sets the process software priority to the standard value for the KW handler. The kw_fork_priority constant sets the handler's interrupt service routine fork priority. You need not modify either of those values unless you attempt to fine-tune the priority relationship between the KW handler and other processes in the application.

The kw_hardware_priority constant sets the default KWV11-C device interrupt priority.

The kw_csr_addr constant specifies the standard CSR address for the device, and the kw_bpr_addr constant specifies the corresponding buffer/preset register (BPR) address. The BPR address always follows the CSR address by two bytes. Modify those values if your device is not configured for the standard CSR address.

The kw_vec_addr constant specifies the standard interrupt vector address for the device. Modify that value if your device is not configured for the standard vector address.

```
{
 COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.
}

MODULE KWPFX;

{+ KWV11-C Driver Prefix Module

   Defines CSR and VECTOR addresses, driver interrupt priority
   (kw_hardware_priority (kw$hpr) ), and driver process priority
   ( kw_process_priority (kw$ppr) ).
-}

        { For differing hardware configurations, change the values
          of the symbols in the CONST section below. }

CONST
   kw_process_priority = 160;           { Standard process priority }
   kw_fork_priority = 16000;            { Standard fork priority }
   kw_hardware_priority = 4;            { and interrupt priority }
                                        { for KWV11-C device}
   kw_csr_addr = %O'170420';            { Address of KWV11-C CSR }
   kw_bpr_addr = %O'170422';            { Address of KWV11-C BPR }
   kw_vec_addr = %O'440';               { Address of KWV11-C vector }

VAR
   proc_prio :  [GLOBAL (kw$ppr)] INTEGER;
                { Driver process priority }

   fork_prio :  [AT(kw_fork_priority), GLOBAL (STSFPR)] UNSIGNED;
                { Global constant }

   dev_prio :   [GLOBAL (kw$hpr)] INTEGER;
                { Hardware interrupt priority}

   kw_csr :     [AT(kw_csr_addr), GLOBAL ($kwcsr)] UNSIGNED;
                { CSR address }

   kw_bpr :     [AT(kw_bpr_addr), GLOBAL ($kwbpr)] UNSIGNED;
                { Buffer/Preset Register address }

   kw_vec :     [AT(kw_vec_addr), GLOBAL ($kwvec)] UNSIGNED;
                { interrupt vector address }

 { Declared in KWDRV module }

   version :    [EXTERNAL ($kwver)] PACKED ARRAY[1..6] OF CHAR;

[ EXTERNAL($KWDRV)] PROCEDURE KWDRV;    { Pull driver OBJ from library }
  EXTERNAL;

[ GLOBAL ($KWPFX) ] PROCEDURE kwpfx;

BEGIN
   proc_prio := kw_process_priority;    { Final process priority }
   dev_prio :=
        kw_hardware_priority * 32;      { in PSW placement }
   version := 'X01.00';                 { Note: Should match the
                                          version in KWDRV module }
END;   {PROCEDUEE kwpfx}
```

Figure 4-11 KWV11-C Clock Handler Prefix Module (KWPFX.PAS)

4.2.3.6 **KXT11-C Q-BUS Arbiter Handler Prefix File (KX)** - The KX handler, running on a Q-BUS arbiter processor, establishes and controls a protocol with one or more KK handlers running on KXT11-C slave processors for passing commands and data between the arbiter and the slave(s). The prefix module for the arbiter's KX handler is reproduced in Figure 4-12.

The KX prefix module uses the drvcf$ and ctrcf$ macros and specifies hardware, fork, process, and initialization priorities by assigning values to four parameters.

The **drvcf$** macro contains a field (nctrl) for the number of controllers on the target to be supported by the handler. The dname field specifies the first two characters of the corresponding request-queue semaphore name.

The **ctrcf$** macro gives the name, number of units, CSR and vector addresses, and unit numbers for the controller. You can edit those fields if your controller does not conform to the defaults. Only a single ctrcf$ macro can be used in the KX prefix file.

The field **units=** specifies the unit numbers of the devices attached to the controller. Each KXT11-C board plugged into the Q-BUS defines one unit number. The designation <0:2> specifies units 0, 1, and 2. Unit numbers must be consecutive, starting at 0.

Note that the interrupt vectors specified in those macros must also be specified in the system configuration file, using the DEVICES macro.

The KX$xxx definitions specify software priorities for the driver processes and the hardware interrupt priority for the KXT11-C boards.

```
.titleKXPFX- KXT11-CA Two port memory device driver
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1983, 1984 BY DIGITAL EQUIPMENT CORPORATION.
;   ALL RIGHTS RESERVED.

.mcallDRVCF$
.mcallCTRCF$

KX$PPR==175.; Process priority
KX$FPR==175.*256.; Fork process priority
KX$HPR==4; Hardware priority
KX$IPR==250.; Process initialization priority
;
drvcf$dname=KX,nctrl=1
ctrcf$cname=A,nunits=2.,csrvec=<162110,500,162120,504>,units=<0>
;ctrcf$cname=B,nunits=2.,csrvec=<162150,510,162160,514>,units=<0>
;ctrcf$cname=C,nunits=2.,csrvec=<162210,520,162220,524>,units=<0>
;ctrcf$cname=D,nunits=2.,csrvec=<162250,530,162260,534>,units=<0>
;ctrcf$cname=E,nunits=2.,csrvec=<162310,540,162320,544>,units=<0>
;ctrcf$cname=F,nunits=2.,csrvec=<162350,550,162360,554>,units=<0>
;ctrcf$cname=G,nunits=2.,csrvec=<175410,560,175420,564>,units=<0>
;ctrcf$cname=H,nunits=2.,csrvec=<175450,570,175460,574>,units=<0>
;ctrcf$cname=I,nunits=2.,csrvec=<175510,600,175520,604>,units=<0>
;ctrcf$cname=J,nunits=2.,csrvec=<175550,610,175560,614>,units=<0>
;ctrcf$cname=K,nunits=2.,csrvec=<175610,620,175620,624>,units=<0>
;ctrcf$cname=L,nunits=2.,csrvec=<175650,630,175660,634>,units=<0>
;ctrcf$cname=M,nunits=2.,csrvec=<175710,640,175720,644>,units=<0>
;ctrcf$cname=N,nunits=2.,csrvec=<175750,650,175760,654>,units=<0>
.end
```

Figure 4-12 KXT11-C Q-BUS Handler Prefix Module for Arbiter (KXPFX.MAC)

4.2.3.7 **DRV11-J (XA) Handler** - The DRV11-J device handler prefix module is reproduced in Figure 4-13. See Chapter 4 of the MicroPower/Pascal Runtime Services Manual for prerequisite information about XA-handler unit numbers. The following paragraphs describe the fields of the prefix module that correspond to the circled numbers on the figure.

1. The **drvcf$** macro specifies the handler name for the request-queue semaphore and the number of DRV11-J devices (controllers) on the target to be supported by the handler. You can edit that field to change the number of controllers; if you do, however, you must add a ctrcf$ macro for each controller.

2. The ctrcf$ macro specifies the controller name, number of units -- ports and bit-interrupt lines -- the controller supports, CSR and vector addresses, and the unit numbers of supported units. Change cname = A to cname = B for the second controller, cname = C for the third, and so on. The cname field supplies the third character for the corresponding request-queue semaphore name.

   When specifying nunits, be sure to include the decimal point. If the DRV11-J device on your target supports fewer than 16 units -- the maximum is 16 -- edit that field..

   The csrvec field specifies both the initial CSR address (CSRA) and the first of 16 consecutive vector addresses associated with the device. You can edit that field if your device is not configured for those starting addresses.

   Note that you must use the DEVICES macro to specify all 16 of those interrupt vectors in the system configuration file. If the first vector is 400, for example, you must specify 400, 404, 410, 414, 420, 424, 430, 434, 440, 444, 450, 454, 460, 464, 470, and 474.

   The units field specifies the unit numbers of the ports and sense lines supported by the controller. The designation 0:15 specifies units 0 through 15. Each unit number has a fixed significance, as defined by the XA handler. Unit numbers 0 through 3 refer to parallel ports A, B, C, and D of the DRV11-J, respectively. Unit numbers 4 through 15 refer to the individual bit-interrupt (sense) lines 0 through 11 of port A, respectively. The handler uses the values specified in the units field to validate the unit numbers given at run time in XA-handler function requests. You can edit that field to restrict the range of valid unit numbers for your device configuration. For example, the units specification units=<1,2,4:9> indicates that only ports B and C (units 1 and 2) can be referred to in parallel-I/O read or write requests and that only the sense lines 0 through 5 of port A (units 4 through 9) can be referred to in bit-interrupt enable and disable requests.

   You can specify unit numbers in one of two ways. You can enumerate all the unit numbers explicitly, separating them with commas:

   ```
   nunits=8.,units=<1,2,3,4,5,6,7,8>
   ```

   You can also use a colon (:) to show a range of units:

   ```
   nunits = 8.,units = <0,2,7:12>
   ```

NOTE

Decimal numbers are assumed in the units field. Do not include a decimal point.

3. Various priorities associated with the driver are given in that field, including the device interrupt priority.

4. The J$RPRI definition sets rotating priorities on or off; 1 = rotating priorities, 0 = fixed priorities. J$HIGH sets interrupt polarity to high or low; 0 = low-level polarity, 1 = high-level polarity. See Chapter 4 of the MicroPower/Pascal Runtime Services Manual for a description of the assumptions made by the handler, based on those settings.

```
.title  XAPFX   - DRV11-J Device driver prefix module

; COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.

        .mcall  drvcf$
        .mcall  ctrcf$

(3) XA$PPR  ==      175.            ; Process priority
    XA$FPR  ==      175.*256.       ; Fork process priority
    XA$HPR  ==      4               ; DRV-11J hardware priority
    XA$IPR  ==      250.            ; Process initialization priority

(4) J$RPRI  ==      1               ; Set for rotating priorities
    J$HIGH  ==      0               ; Clear for low level interrupt polarity

      (1) drvcf$  dname=XA,nctrl=1
    (2) → ctrcf$  cname=A,nunits=16.,csrvec=<164160,400>,units=<0:15>

        .end
```

Figure 4-13 DRV11-J Handler Prefix Module (XAPFX.MAC)

4.2.3.8 **DLV11 (XL) Handler** - The DLV11 device handler prefix module XLPFX.MAC is reproduced in Figure 4-14. There are three other XL prefix files: XLPFXD.MAC, for building an LSI application with debug support, XLPFXF.MAC, for building a FALCON or FALCON-PLUS application with debug support, and XLPFXK.MAC, for building a KXT11-C application with debug support. Section 4.2.4.5 discusses XLPFXK.MAC. The other three prefix files differ primarily in the default CSR and vector addresses they provide. The default CSR and vector assignments are as follows:

- In XLPFX.MAC, CSR=177560 and VEC=60 (LSI console terminal port)
- In XLPFXD.MAC, CSR=176500 and VEC=300 (Nonconsole terminal port)
- In XLPFXF.MAC, CSR=176540 and VEC=120 (FALCON or FALCON-PLUS SLU2 port)

The fields of the XLPFX prefix module, described below, are illustrated in Figure 4-14.

$XLPRM (item 1 in Figure 4-14) labels a word specifying the number of serial line units to be supported by the XL handler. You must modify that value to reflect the number of lines described in the prefix module, that is, the number of line-definition (LINDF$) macros used in the file.

NOTE

When building an appplication with debug support, you must not specify a LINDF$ macro for the host-to-target serial line used by PASDBG. That line, which must be connected to the target's console-terminal port, is handled directly by the debug service module. (The default line definition in the XLPFX.MAC file assumes an LSI application without debug support; it describes a line connected to the LSI console terminal port -- CSR 177560.)

LINDF$ (item 2 in Figure 4-14) is a macro call that supplies information about each supported line. One LINDF$ macro must be used per serial line and must supply at least the vector and CSR addresses (receive side only) for the line. A line type must also be specified for other than the basic DLV11 type of line interface unit; the symbol TT$DLE must be specified, for example, for a DLV11-E interface unit. An initial line speed must be specified if a line is configured for programmable baud rate. The other LINDF$ arguments pertain to optional start-up time creation and connection of input and output ring buffers for a given line.

Unit number assignments correspond to the order in which the LINDF$ macros occur in the prefix file. That is, the first or only LINDF$ macro implicitly defines unit 0, the second defines unit 1, the third defines unit 2, and so on.

The complete LINDF$ macro keyword syntax is as follows:

```
LINDF$ [name],vec,csr,typ[=TT$DL],rnam,rsiz[=12.],ratt,rmod,xnam,
       xsiz[=80.],xatt,xmod,spd
```

vec

The receive-side interrupt vector address for a given line; the transmit vector for that line is assumed to follow the receive vector by four bytes. For example, vec=300 specifies a receive vector at location 300 and implies a corresponding transmit vector at location 304.

csr

The receive-side CSR (RCSR) address for a given line; the transmit CSR (XCSR) for that line is assumed to follow the RCSR by four bytes. For example, csr=176500 specifies an RCSR at location 176500 and implies a corresponding XCSR at location 176504.

typ

The set of functional capabilities to be supported by the line, in terms of an interface unit type:

TT$DL for the minimum common functions provided by a DLV11 or DLV11-J interface

TT$DLE for a DLV11-E interface

TT$DLF for a DLV11-F interface

TT$DLT for a DLART-type interface jumpered for programmable baud rate.

Additional information on type TT$DLT is given below. The default is typ=TT$DL. Note that type TT$DL can be specified for any kind of interface unit, provided that only common DLV11 line functions are utilized.

rnam

The name of a receive ring buffer to be created and automatically connected to the line by the XL handler at start-up time. The name must contain six characters, including trailing blanks, if needed -- for example, rnam=<INBUF >. The rnam parameter is optional.

rsiz

The size, in an even number of bytes, of the receive buffer. The default size is 12 bytes. The rsiz parameter is meaningful only if you specify rnam.

ratt

The access attribute of the receive buffer. Specify ratt=SA$RIR for record-oriented input access (default) or ratt=SA$RIS for stream-oriented input access. The ratt parameter is meaningful only if you specify rnam.

rmod

Receive-side mode bits. Specify rmod=F$XCHK to enable automatic XON/XOFF checking.

xnam

The name of a transmit ring buffer to be created and automatically connected to the line by the XL handler at start-up time. The name must contain six characters, including trailing blanks if needed -- for example, xnam=<OUTBUF>. The xnam parameter is optional.

xsiz

The size, in an even number of bytes, of the transmit buffer. The default size is 80 bytes. The xsiz parameter is meaningful only if you specify xnam.

xatt

The access attribute of the transmit buffer. Specify ratt=SA$ROR for record-oriented output access (default) or ratt=SA$ROS for stream-oriented output access.

xmod

Transmit-side mode bits; currently unused.

spd

The initial, or start-up, speed setting for a line configured for programmable baud rate. For example, spd=1200. specifies an initial line speed of 1200 baud.

The spd argument sets the receive-side baud rate in the XL handler for the serial line at start-up time, assuming that the line speed is programmable. The spd argument is valid only if the type value is TT$DLE, TTSDLF, or TT$DLT. If the line type is TTSDLE or TTSDLF, the valid speed values are 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, or 19,200. (The value 134 indicates 134.5 baud.) If the line type is TT$DLT, the valid speed values are 300, 600, 1200, 2400, 4800, 9600, 19,200, or 38,400.

The optional ring buffer connection capability is provided primarily to support terminal I/O in Pascal via the standard Pascal files INPUT and OUTPUT. The Pascal OTS assumes that the line to be associated with the predeclared file variables INPUT and OUTPUT has been automatically connected to ring buffers named XLI0 and XLO0 -- input and output sides, respectively -- by the XL handler at start-up time.

The type symbol TT$DLT indicates a serial line of the type implemented on the MXV11-B multifunction board and on the FALCON or FALCON-PLUS SBC-11/21 microcomputer. That type of line supports programmable baud rate in addition to common DLV11 capabilities. For an MXV11-B line configured for programmable baud rate, specify typ=TT$DLT and give the spd argument. For an MXV11-B line with hardwired baud rate, indicate the line type as TT$DL. For an SBC-11/21 serial line, specify typ=TT$DLT and give the spd argument. To use a baud rate that has already been set -- by the kernel, as specified in the KXT11 configuration macro, for example -- omit the spd argument.

The XL$... definitions (item 3 in Figure 4-14) specify software priorities associated with the handler process and the hardware priority for all serial line interrupts.

Note that the interrupt vectors specified and implied in the XLPFX.MAC prefix file must also be specified in the system configuration file, using the DEVICES macro.

```
; COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.

        .mcall  MACDF$,IODF$,QUEDF$,DRVDF$,XLISZ$,LINDF$

        macdf$
        quedf$
        drvdf$
        xlisz$  GLOBAL

        dfalt$  F$XCHK,4000

;+
;
;                       [MANY COMMENT LINES DELETED HERE]
;
; LINDF$ macro examples:
;
; $XLPRM:: .word 2        ; Define two lines
;
;       LINDF$  csr=177560,vec=60,typ=TT$DLF,rnam=<XLI0 >,rsiz=10.,
;               ratt=SA$RIS,rmod=F$XCHK,xnam=<XLO0 >,xsiz=80.,xatt=SA$ROS
;
;       LINDF$  csr=176500,vec=300
;
;       .end
;
; Defines a line on unit 0 with predefined stream-attribute buffers for
; both receive and transmit sides and defines another line on unit 1.
; Unit 0 above has X-OFF/X-ON checking enabled. The controller for unit 0
; is a DL11-F. Note that you must pass a 6 character blank padded string
; for the ring buffer structure names as shown above.
;
;-

        .GLOBL  $XL     ;Haul in the XL driver from the library

(3) XL$PPR  ==      175.            ; Process priority
    XL$FPR  ==      175.*256.       ; Fork process priority
    XL$HPR  ==      4               ; hardware priority
    XL$IPR  ==      250.            ; Process initialization priority

        pdat$
(1) $XLPRM::.word   1       ; Define only one line

(2) LINDF$  csr=177560,vec=60,rmod=F$XCHK,rnam=<XLI0  >,xnam=<XLO0  >

        .end
```

Figure 4-14 DLV11 Handler Prefix Module (XLPFX.MAC)

4.2.3.9 **DPV11 Synchronous Serial Line (XP) Handler** - The prefix module XPPFX.MAC for the DPV11 (XP) handler is reproduced in Figure 4-15. The following paragraphs describe the fields of the prefix module that correspond to the circled numbers on the figure.

1. SXPPRM labels a word specifying the number of synchronous serial lines to be supported by the XP handler. The handler can support a maximum of 16 lines. Modify the value, if necessary, to reflect the number of synchronous serial lines used in your application.

2. The DPVDF$ macro specifies the receive-side CSR and vector addresses for each supported line. The transmit-side CSR and vector addresses are assumed to follow the specified receive-side addresses by four bytes each. You must specify one DPVDF$ macro call for each DPV11 line interface unit on

the target system. The unit numbers assigned to the lines correspond to the order in which the DPVDF$ macro calls occur in the file. That is, the first or only DPVDF$ macro implicitly defines unit 0, the second defines unit 1, and so on.

3. The XP$... definitions specify software priorities associated with the handler process and the hardware priority for all synchronous line interrupts.

Note that the interrupt vectors specified and implied in the prefix module must also be specified in the system configuration file, using the DEVICES macro.

```
; COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.

        .mcall  MACDF$,IODF$,QUEDF$,DRVDF$,XPISZ$,DPVDF$

        macdf$
        quedf$
        drvdf$
        xpisz$  GLOBAL

;+
;                       [COMMENT BLOCK DELETED HERE]
;
; DPVDF$ example:
;
; $XPPRM:: 2                        ; Define two DPV-11 devices
;
;       DPVDF$  csr=160010,vec=300
;       DPVDF$  csr=160020,vec=310
;
;       .end
;
; The first DPVDF$ macro defines a line on unit 0 and the second defines
; another line on unit 1.
;
;-

        .GLOBL  $XP                     ; Pull in the DPV driver from the library

(3) XP$PPR  ==      175.                ; Process priority
    XP$FPR  ==      175.*256.           ; Fork process priority
    XP$HPR  ==      5                   ; hardware priority
    XP$IPR  ==      250.                ; Process initialization priority

        pdat$
(1) $XPPRM::.word   1                   ; Define only one DPV-11 device by de'aul

(2) DPVDF$  csr=160010,vec=300

        .end
```

Figure 4-15 DPV11 Device Handler Prefix Module (XPPFX.MAC)

4.2.3.10 **DRV11 (YA) Handler** - The DRV11 device handler prefix module is reproduced in Figure 4-16. The following paragraphs describe the user-modifiable parts of the file that correspond to the circled numbers on the figure.

1. The STD_PROC_PRIO constant sets the process software priority to the standard value for the YA handler. The STD_INT_PRIO constant sets the default DRV11 device interrupt priority.

2. The CSR address is assigned the default value 167770. That is the factory-configured CSR address for the device.

3. The VECTOR_REQA and VECTOR_REQB assignments set the output (REQ A) interrupt vector address to 340 and the input (REQ B) interrupt vector address to 344, respectively. Those are the factory-configured vector addresses for the device.

4. The MAPPED assignment indicates that the target system environment is unmapped. Change the assigned value to TRUE if your target system is mapped.

```
{
 COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION.  ALL RIGHTS RESERVED.
}
MODULE YAPFX;
{+ DR Driver Prefix Module

   Defines CSR and VECTOR addresses, driver interrupt priority
   (DRPRIO), and driver process priority (STDPRIO).
-}
CONST
(1)   STD_PROC_PRIO = 160;                  { Standard process priority }
      STD_INT_PRIO = 4;                     { and interrupt priority }
                                            { for DRV11 device}

VAR
      STDPRIO:[GLOBAL] integer;             {Driver process priority}
      DRPRIO:[GLOBAL] integer;              {Interrupt priority}
      CSR:[GLOBAL] unsigned;                {CSR address--used in
                                             driver as DRV11_CSR record}
      VECTOR_REQA:[GLOBAL(REQA)] unsigned;  {vector address, A request}
      VECTOR_REQB:[GLOBAL(REQB)] unsigned;  {vector address, B request}
      MAINTENANCE:[GLOBAL(maint)] boolean;
      MAPPED:[GLOBAL(mapped)] boolean;

 {Declared in YADRV module}
      VERSION:[EXTERNAL($YAVER)]packed array[1..6] of char;
[GLOBAL] Procedure InitCSR;

Begin
      STDPRIO := STD_PROC_PRIO;             {Final process priority}
      DRPRIO := STD_INT_PRIO * 32;          {In PSW placement}
(2)   CSR := %O'167770';                    {CSR INBUF at 167774,
                                             CSR OUTBUF at 167772}
(3)   VECTOR_REQA := %O'340';
      VECTOR_REQB := %O'344';
      VERSION := 'V01.01';                  {Note: should match the
                                            version in DRDRV module}
      MAINTENANCE := FALSE;                 {Not maintenance mode}
(4)   MAPPED := FALSE;                      {Not mapped system}

End;    {Procedure InitCSR}
```

Figure 4-16 DRV11 Handler Prefix File (YAPFX.PAS)

4.2.3.11 **SBC-11/21 (FALCON and FALCON-PLUS) 8255 PIO Handler (YF)** - The SBC-11/21 PIO prefix module is reproduced in Figure 4-17. The following paragraphs describe the symbol definitions in the prefix module.

YF$PPR defines the process software priority for the YF handler. The field can be modified to fine-tune the relationship between the handler and other processes in the application.

YF$FPR defines the handler's interrupt service routine fork priority. By default, the value of YF$FPR is set to 256 times the software priority.

YF$IPR defines the process initialization priority. That priority should be equal to that of all other I/O device handlers.

YF$HPR defines the hardware interrupt priority. For the SBC-11/21 parallel port, the level is fixed at 5.

YF$AIO defines the data direction for the 8-bit data port A; 0 = output, 1 = input. If the default direction of that port is reversed, jumpers M59 through M66 on the FALCON or FALCON-PLUS board must be changed from their factory configuration to reflect the prefix module settings.

YF$BIO defines the data direction for the 8-bit data port B; 0 = output, 1 = input. If the default direction of that port is reversed, jumpers M59 through M66 on the FALCON or FALCON-PLUS board must be changed from their factory configuration to reflect the prefix module settings.

```
.title  YFPFX   - FALCON (SBC-11/21) 8255 PIO Device driver prefix module
; COPYRIGHT (c) 1982 BY DIGITAL EQUIPMENT CORPORATION. ALL RIGHTS RESERVED.

        .GLOBL  $YF             ; Haul in the driver from the library

YF$PPR  ==      176.            ; Process software priority
YF$FPR  ==      176.*256.       ; ISR Fork priority
YF$IPR  ==      250.            ; Process initialization priority
YF$HPR  ==      5               ; Hardware interrupt priority

YF$AIO  ==      1               ; Port A is input
                                ; Set to 0 for Port A output
YF$BIO  ==      0               ; Port B is output
                                ; Set to 1 for Port B input

        .end
```

Figure 4-17 SBC-11/21 PIO Handler Prefix Module (YFPFX.MAC)

4.2.4 Editing Prefix Modules: KXT11-C Peripheral Processor

The prefix modules applicable to the KXT11-C peripheral processor are described below. In general, you edit the prefix modules for the KXT11-C exactly as you would edit prefix modules for any other target processor. Because the KXT11-C is a fixed configuration, you will probably need to modify few paramaters in the prefix files.

4.2.4.1 **Line-Frequency Clock Process (CLOCK)** - Except for a CSR value of 177520, the clock process prefix module, CKPFX.MAC, is the same for both the KXT11-C and other target processors. Refer to the description of CKPFX.MAC in Section 4.2.3.3.

4.2.4.2 **TU58 DECtape II (DD) Handler Prefix File** - The DD prefix file for the KXT11-C is essentially the same as that for other target processors. However, the CTRCF$ macro has an extra argument to specify the baud rate, as shown in Figure 4-18.

The TU58 uses one or more of the KXT11-C serial lines. If you use a TU58 on a KXT11-C, be sure to modify the XL prefix file so that you do not try to use the serial line with the XL handler as well. If you use the XS handler, which always uses multiprotocol channel A, be sure to configure the TU58 to use some other line.

```
        .title  DDPFXK.MAC - TU58 Device handler prefix file for KXT11-C
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1984 BY DIGITAL EQUIPMENT CORPORATION.
;               ALL RIGHTS RESERVED.

; This file contains an example of the prefix file that is required
; to configure the TU58 device handler on a KXT11-C
;
 .SBTTL BEGINNING OF TU58 CONFIGURATION
;
; bring in macro definitions from library
        .mcall  drvdf$,drvcf$
        .mcall  ctrcf$
;
;
DD$PPR  ==      175.            ; Process priority
DD$FPR  ==      175.*256.       ; Fork process priority
DD$HPR  ==      4               ; TU58 hardware priority
DD$IPR  ==      250.            ; Process initialization priority
;
;  use the handler definition macros
        drvdf$
        drvcf$ dname=DD,nctrl=1
;
;  configure multiprotocol channel B on the KXT11-C, to drive TU58
 ctrcf$ cname=A,nunits=2.,csrvec=<175710,160>,units=<0:1>,typrm=<TT$DM,9600>
;
        .end
```

Figure 4-18 TU58 Handler Prefix Module for KXT11-C (DDPFXK.MAC)

4.2.4.3 **KXT11-C Slave Q-BUS Handler Prefix File (KK)** - There are no parameters to modify in the KK prefix module. The module references the global $KK, and this extracts the KK handler from the device driver library when the application is being built.

```
        .TITLE  KKPFXK.MAC - KXT11CA Connect to QBUS Handler

; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1984 BY DIGITAL EQUIPMENT CORPORATION.
;               ALL RIGHTS RESERVED.

; This module is used only to pull the KXT11CA communications with
; the QBUS handler (KK) out of the device driver library.  It also
; defines the priorities of the KK driver and hardware.
;
;
; Cause KK driver to be fetched from library

        .globl  $KK


; Definitions of Priorities in this driver
KK$HPR  == 5            ; Hardware priority
KK$IPR  == 250.         ; Initialization priority
KK$PPR  == 170.         ; Process priority
KK$FPR  == 170.*256.    ; Fork priority
;
        .END
```

Figure 4-19 KXT11-C Q-BUS Handler Prefix Module for Slave (KKPFXK.MAC)

4.2.4.4 **KXT11-C DTC (QD) Handler** - The prefix file for the QD handler is shown in Figure 4-20. The QD handler uses the DMA transfer controller (DTC) of the KXT11-C board to perform direct memory access transfers between local memory on the KXT11-C board and other memory or I/O devices. Those can be:

- Q-BUS memory of the arbiter CPU
- A Q-BUS I/O device on the arbiter CPU's Q-BUS
- Other locations in local memory on the KXT11-C
- A local I/O device connected directly to the KXT11-C

The priorities, CSR, vector, and number of units are standard for the KXT11-C board. The QD prefix file, as supplied, should be appropriate for all applications and not need modification.

```
        .title  QDPFXK  - KXT11-C DTC device driver

; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1984 BY DIGITAL EQUIPMENT CORPORATION.
;               ALL RIGHTS RESERVED.

        .MCALL  DRVCF$
        .mcall  ctrcf$

QD$PPR  ==      175.            ; Process priority
QD$FPR  ==      175.*256.       ; Fork process priority
QD$HPR  ==      4               ; DTC hardware priority
QD$IPR  ==      250.            ; Process initialization priority

        drvcf$  dname=QD,nctrl=1
        ctrcf$  cname=A,nunits=2.,csrvec=<174400,110>,units=<0:1>

        .end
```

Figure 4-20 KXT11-C DTC Device Handler Prefix File (QDPFXK.MAC)

4.2.4.5 **KXT11-C Asynchronous Serial Line (XL) Handler** - The XL prefix module to configure the XL handler for a KXT11-C, XLPFXK.MAC, is very similar to the XL prefix files XLPFX.MAC, XLPFXD.MAC, and XLPFXF.MAC. However, because the configuration of the board is fixed, fewer modifications to the file are required. There are always three serial lines: the console line at vector 60, multiprotocol channel A, and multiprotocol channel B. You should normally use three LINDFS macros to configure the XL handler, as illustrated in Figure 4-21 and described below. However, if you connect a TU58 to the KXT11-C, the TU58 handler will use one of the serial lines. If this is the case, be sure that XLPFXK.MAC does not also define the same line for the XL handler. If you use the XS handler, the XS handler will always use multiprotocol channel A; in that case, be sure to omit channel A from the XL handler configuration.

typ

The following typ values are permitted in the LINDFS macro for the XL handler with the KXT11-C:

- TT$DLT f the console channel at vector 60
- TT$DM for KXT11-C multiprotocol channel data only
- TT$DMP for KXT11-C multiprotocol channel with partial modem control. Multiprotocol channel A is the only channel that can be of this type. In this case, another handler may use channel B.
- TT$DMM for KXT11-C multiprotocol channel with full modem control. Multiprotocol channel A is the only channel that can be of this type, and then only when the XL handler (or no handler) is in control of multiprotocol channel B.

vec
The vectors for the asynchronous lines on the KXT11-C are:

Line	Vector
Console	60
Channel A	140
Channel B	160

csr The control status registers for the three lines are:

Line	CSR
Console	177560
Channel A	175700
Channel B	175710

The other LINDFS arguments correspond to the LINDF$ arguments in XLPFX.MAC, described above.

```
{
 THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
 ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.

 COPYRIGHT (c) 1982, 1984 BY DIGITAL EQUIPMENT CORPORATION.
               ALL RIGHTS RESERVED.
}

;   Console
LINDFS  typ=TT$DL,csr=177650,vec=60,rmod=F$XCHK,rnam=<XLI0>,rsiz=80.,
        xnam=<XLO0>,xsiz=80.
;
;Multiprotocol channel B with ring buffers, but without modem control
LINDFS  typ=TT$DM,csr=175710,vec=160,rmod=F$XCHK,rnam=<XLI1  >,rsiz=134.,
        xnam=<XLO1  >,xsiz=80.,spd=9600
;
;   Multiprotocol channel A with ring buffers and full modem control
LINDFS  typ=TT$DMM,csr=175700,vec=140,rmod=F$XCHK,rnam=<XLI2  >,rsiz=134.,
        xnam=<XLO2  >,xsiz=80.,spd=9600
```

Figure 4-21 LINDFS Macros from KXT11-C XL Prefix File (XLPFXX.MAC)

4.2.4.6 **KXT11-C Synchronous Serial Line (XS) Handler** - The XS prefix file, shown in Figure 4-22, configures the serial line unit of the KXT11-C multiprotocol chip. The CSR, vector, and priorities are standard for the serial line port and the prefix file, as supplied, should be appropriate for all applications and not need modification. The XS handler always uses multiprotocol channel A.

```
{
 THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
 ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.

 COPYRIGHT (c) 1982, 1984 BY DIGITAL EQUIPMENT CORPORATION.
               ALL RIGHTS RESERVED.
}

[system(MICROPOWER)]module XS_prefix;

const

  csr                   = %o'175700';    { The CSR address of the device }
  vector                = %o'140';       { The vector address of the device }
  priority_driver       = 160;           { The driver execution priority }
  priority_interrupt    = 4;             { The driver hardware int priority }
  priority_fork         = 160;           { The driver fork priority }

var

  driver_priority       : [external($xsdpr)] PRIORITY_RANGE;
  interrupt_priority    : [external($xsipr)] BYTE_RANGE;
  fork_priority         : [external($xsfpr)] PRIORITY_RANGE;
  device_address        : [external($xsdad)] UNIVERSAL;
  device_vector         : [external($xsdve)] VECTOR_RANGE;

[global($xsini)]procedure init_line_vars;

begin
  driver_priority := priority_driver;
  interrupt_priority := priority_interrupt * 32;
  fork_priority := priority_fork;
  device_address := csr;
  device_vector := vector;
end;
```

Figure 4-22 KXT11-C Synchronous Serial Line Prefix Module (XSPFXK.PAS)

4.2.4.7 **KXT11-C Parallel Ports and Timer/Counter (YK) Handler** - Figure 4-23 shows the YKPFXK.MAC parallel port and Timer/Counter prefix file.

The KXT11-C parallel port and timer/counter prefix file uses four configuration macros. An initialization macro, YKCIS, defines symbols and other macros. The second, YKCPS, configures a port. The third, YKCTS, configures a timer. The fourth, YKCES, marks the end of the configuration list. The second and third macros update internal symbols as they are used. The symbols' values indicate what features have been selected. Those values are used to prevent the user from attempting to select an invalid configuration. Error messages are output to the listing file or terminal if a conflicting option is requested.

The configuration macros generate a data table that is used by the handler at system initialization time to configure the peripheral processor hardware registers.

Configuration Initialization Macro-YKCI$

Syntax

```
YKCI$
```

The configuration initialization macro YKCI$ must be the first macro used in the YK handler prefix file. YKCI$ has no parameters associated with it; it initializes symbols and defines the remainder of the macros and submacros to be used.

Port Configuration Macro-YKCP$

Syntax

```
YKCP$ chan=[A],ptype=[YK$BIT],hsh=[YK$INL],dskw=[0],out=[0],
      inv=[0],inl=[0],oco=[FALSE],plnk=[FALSE],dma=[FALSE],
      pat=[FALSE]
```

This macro is used to configure a particular port. The macro can be used by listing parameters in the order that they are defined or by using the KEYWORD=VALUE format, where the value can be a sum of bit definitions in the case of the direction, polarity, or one's catcher specifications. If all parameters for a port cannot fit on a single line, the macro can be reused for the same channel and the remainder of the parameters specified on the second usage. Any parameter omitted will take on the default value as specified above in square brackets. Each parameter is described below.

chan
: Specifies the channel number to use. Permissible values are A, B, or C.

ptype
: Specifies the port type: bit, input, output, or bidirectional. Permissible values are:

  YK$BIT -- bit port
  YK$INP -- input port
  YK$OUT -- output port
  YK$BID -- bidirectional port

hsh
: Specifies the type of handshake mode: interlocked, strobed, 3-wire, or pulsed.

  YK$INL -- interlock
  YK$STR -- strobed
  YK$3WI -- 3-wire (IEEE 488)
  YK$PUL -- pulsed

NOTE

Timer 3 must be configured and a run-time request sent in order to use pulsed handshake. The timer set command must be sent prior to sending the first command to the port.

dskw
: Specifies the deskew time, in cycles. Permissible values are 0, 2, 4, 6, 8, 10, 12, 14, or 16.

out
: Specifies the I/O direction for each bit. If set, the bit is output. The following symbols may be OR'd together to define output bits.

  Mnemonic Value (octal)

YK$B0	1
YK$B1	2
YK$B2	4
YK$B3	10
YK$B4	20
YK$B5	40
YK$B6	100
YK$B7	200

inv

Specifies polarity of the bit. If a bit is set, the corresponding input or output bit of the interface register is inverted. The symbols YK$B0 through YK$B7 listed above may be OR'd together to define inverted bits.

inl

Specifies which input bits should have the one's catcher attribute. If a bit is set, the corresponding input bit of the interface register will have the one's catcher enabled. The symbols YK$B0 through YK$B7 listed above may be OR'd together to define one's catcher bits.

oco

Specifies which output bits have open drain. If not specified, outputs will be active pull up.

plnk

If specified as TRUE, ports A and B will be linked together to form one 16-bit port.

dma

If specified, the port will use DMA.

pat

If specified, the port will use pattern recognition.

Timer Configuration Macro-YKCT$

This macro is used to configure a particular timer. The macro can be used by listing the parameters in the order that they are defined or by using the KEYWORD = VALUE format. If all parameters to be specified for a timer cannot fit on a single line, the macro can be reused for the same timer number and the remainder of the parameters specified on the second usage. Any parameter omitted takes on the default value specified as follows:

Syntax

```
YKCT$  tnum=[1],texto=[NO],textc=[NO],textt=[NO],textg=[NO],
       tretre=[NO],tout=[YK$TPL],tlnk=[YK$TIN]
```

tnum

Defines the timer number: 1, 2, or 3

texto

NO -- Disables timer output
YES -- Enables timer output

```
textc

     NO  -- Disables timer external count
     YES -- Enables timer external count

textt

     NO  -- Disables timer external trigger
     YES -- Enables timer external trigger

textg

     NO  -- Disables timer external gate
     YES -- Enables timer external gate

tretre

     NO  -- Disables timer retrigger
     YES -- Enables timer retrigger

tout
     Defines the type of timer output

     YK$TPL -- Pulse output
     YK$T1S -- One shot
     YK$TSQ -- Square wave

tlnk
     Defines the interaction of timers 1 and 2

     YK$TIN -- Timers are independent
     YK$1G2 -- Timer 1 output gates timer 2
     YK$1T2 -- Timer 1 output triggers timer 2
     YK$1I2 -- Timer 1 output is timer 2 input
```

End Configuration Macro-YKCE$

This macro must be used after all port or timer macros have been used. The end macro builds the configuration table from the local symbols that were defined while the other two macros were being used.

Parameter Interaction

Many inputs, outputs, and internals in the parallel port and timer/counter chip are multiplexed among the various functions. Thus, several features are mutually exclusive of other features, or are not available for a particular port or combination of ports.

Error messages can occur during assembly of the YK prefix file if a chosen combination of parameters is not a viable configuration. Refer to the MicroPower/Pascal Messages Manual for a list of the possible error messages.

The tables below show which combinations of parameters are invalid for the YKCP$ macro when configuring ports A, B, and C. If marked with an X, the parameter combination is invalid for the port. For example, when configuring port A, you cannot specify an inl value if you specify ptype=YK$INP, YK$OUT, or YK$BID. The tables do not consider invalid combinations among ports; refer to the configuration notes for those.

General Port and Timer Configuration Notes

Handshake Signals

- If timer 3 has external output, bit 0 of port C cannot be a handshake signal.
- Only a single port can specify pulsed handshake.
- Only a single port can specify 3-wire handshake.
- If one port uses 3-wire handshake, other port must be a bit port.

Port Outputs

- Output lines of ports A and B must all be open collector or all be active pullup.
- If timer 1 has external output, bit 4 of port B must be an output.
- If timer 2 has external output, bit 0 of port B must be an output.
- If timer 3 has external output, bit 0 of port C must be an output.

Timer External Outputs

- Port B must be a bit port to use timer 1 or timer 2 external output.
- If timer 1 has external output, bit 4 of port B must be an output.
- If timer 2 has external output, bit 0 of port B must be an output.
- Port C must be configured in order to use timer 3 external output.
- To use external output for timer 3, port C must be configured.
- If timer 3 has external output, bit 0 of port C must be an output.
- If timer 3 has external output, bit 0 of port C cannot be a handshake signal.

Invalid Combinations of Parameters in YKCP$ Macro When Configuring Port A

port A	YK$ BIT	YK$ INP	YK$ OUT	YK$ BID	YK$ INL	YK$ STR	YK$ PUL	YK$ 3WI	dskw	out	inv	inl	oco	plnk	dma	pat
YK$BIT		X	X	X	X	X	X	X	X					X		
YK$INP	X		X	X					X	X		X				
YK$OUT	X	X		X						X		X				
YK$BID	X	X	X				X	X		X		X				
YK$INL	X					X	X	X								
YK$STR	X				X		X	X								
YK$PUL	X			X	X	X		X								
YK$3WI	X			X	X	X	X									
dskw	X	X														
out		X	X	X												
inv																
inl		X	X	X												
oco																
plnk	X															
dma																
pat																

Port A Configuration Notes

- If port A is a bit port (ptype=YK$BIT), bits 0-3 and bits 4-7 must all be inputs or all be outputs. Thus, if port A is a bit port, the parameter **out** can have only the values 0, 17, 360, or 377(octal).
- If ports A and B are linked, port A cannot be a bit port.
- To use handshake signals on port A, port C must be configured.
- The value of oco for ports A and B must be the same.
- If port A is bidirectional, port B must be a bit port.

Invalid Combinations of Parameters in YKCP$ Macro When Configuring Port B

port B	YK$ BIT	YK$ INP	YK$ OUT	YK$ BID	YK$ INL	YK$ STR	YK$ PUL	YK$ 3WI	dskw	out	inv	inl	oco	plnk	dma	pat
YK$BIT		X	X	X	X	X	X	X	X							
YK$INP	X		X	X					X	X		X		X		
YK$OUT	X	X		X						X		X		X		
YK$BID	X	X	X				X	X		X		X		X		
YK$INL	X					X	X	X								
YK$STR	X				X		X	X								
YK$PUL	X			X	X	X		X								
YK$3WI	X			X	X	X	X									
dskw	X	X														
out		X	X	X												
inv																
inl		X	X	X												
oco																
plnk		X	X	X												X
dma																
pat														X		

Port B Configuration Notes

- If ports A and B are linked, port B must be a bit port.
- To use handshake signals on port B, port C must be configured.
- The value of oco for ports A and B must be the same. That is, the output lines of ports A and B must all be open collector or all be active pullup.
- If port B is bidirectional, port A must be a bit port.

Invalid Combinations of Parameters in YKCP$ Macro When Configuring Port C

port C	YK$ BIT	YK$ INP	YK$ OUT	YK$ BID	YK$ INL	YK$ STR	YK$ PUL	YK$ 3WI	dskw	out	inv	inl	oco	plnk	dma	pat
YK$BIT		X	X	X	X	X	X	X	X					X	X	X
YK$INP	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
YK$OUT	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
YK$BID	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
YK$INL	X	X	X	X		X	X	X	X					X	X	X
YK$STR	X	X	X	X	X		X	X	X					X	X	X
YK$PUL	X	X	X	X	X	X		X	X					X	X	X
YK$3WI	X	X	X	X	X	X	X		X					X	X	X
dskw	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
out		X	X	X					X					X	X	X
inv		X	X	X					X					X	X	X
inl		X	X	X					X					X	X	X
oco		X	X	X					X					X	X	X
plnk	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
dma	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
pat	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Port C Configuration Notes

- Port C is 4 bits wide; therefore, values for inp, inl, and out can have values only in the range 0 to 17(octal).
- To use handshake signals on ports A and B, port C must be configured.
- Port C handshake inputs cannot be defined as outputs.
- Port C handshake outputs cannot be defined as inputs.

```
        .TITLE  YKPFXK.MAC - PIO and COUNTER/TIMER PREFIX

; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1984 BY DIGITAL EQUIPMENT CORPORATION.
;               ALL RIGHTS RESERVED.

;       This module is an example of using the special configuration
; macros for the Parallel I/O and Counter-Timers on the KXT 11C.
; The header at the beginning of the module lists the features that are
; being configured.
;
;       This configuration provides:
;
;       1.  4 switch inputs
;       2.  4 LED driver outputs
;       3.  8 line parallel output port, with pattern match or DMA
;       4.  2 pulsed handshakes (1 input, 1 output) for output port
;       5.  1 status input line from parallel device
;       6.  Timer 3 provides delay time in pulsed handshake
;       7.  Timer 1 and 2 are linked (timer 1 = least sig word)
;
;DEFINE PRIORITIES FOR YK HANDLER
YK$HPR  == 5            ; Hardware priority
YK$IPR  == 250.         ; Initialization priority
YK$PPR  == 180.         ; Process priority
YK$FPR  == 180.*256.    ; Fork priority

; CALL:  INITIALIZE MACRO
        .MCALL  YKCI$
        YKCI$

;
; PORT A
;  bit port' for reading switches and driving LEDs
;  bits 0,1,2,3 are inputs.... bits 4,5,6,7 are inverted outputs
        YKCP$   CHAN=A,PTYPE=YK$BIT,OUT=<YK$B4+YK$B5+YK$B6+YK$B7>
        YKCP$   CHAN=A,INV=<YK$B4+YK$B5+YK$B6+YK$B7>
;
; PORT B
; parallel output port, with pulsed handshake
        YKCP$   CHAN=B,PTYPE=YK$OUT,HSH=YK$PUL,PAT=YES,DMA=YES
;
; PORT C
;  handshake signals for port B
;  bit 0 = acknowledge (input)
;  bit 1 = data available (inverted open collector output)
;
;  status input from external device
;  bit 2 = non-inverted input
;
        YKCP$   CHAN=C,OUT=<YK$B1>,INV=YK$B1,OCO=YES


; TIMER 1
        YKCT$   TNUM=1

; TIMER 2
; timer 1 output is timer 2 input
        YKCT$   TNUM=2,TLNK=YK$1I2

; TIMER 3
        YKCT$   TNUM=3

; END CONFIGURATION
        YKCE$

.END
```

Figure 4-23 KXT11-C Parallel Port/Timer-Counter Prefix File (YKPFXK.MAC)

4.2.5 Assembling or Compiling Prefix Modules

You must assemble or compile each prefix module before you can merge it with the appropriate device handler library to create the device handler process. Prefix modules written in MACRO-11 must be assembled with either the COMU macro library for an unmapped target system (including a KXT11-C) or with the COMM macro library for a mapped target system. Sample commands for assembling prefix modules are shown in chapter 3.

Prefix modules written in Pascal must be compiled with the MicroPower/Pascal compiler, using either the IN:NHD or the IN:EIS compilation option, depending on whether the intended target system is unmapped or mapped. If the target is an unmapped system (including the KXT11-C), compile for the NHD instruction set, regardless of the instruction set actually supported on your target system. If the target is a mapped system, compile the prefix module for the EIS instruction set. Sample compilation commands for Pascal prefix modules are shown in Chapter 3.

All Pascal-implemented handlers supplied in the DRVU and DRVK object libraries were compiled using the /IN:NHD option; all Pascal-implemented handlers supplied in the DRVM object library were compiled using the /IN:EIS option.

CHAPTER 5

MERGING OBJECT MODULES

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 run MERGE yourself, as described below, or you can, for most applications, use the automated build procedure MPBUILD.COM, described in Appendix B. (The build-command file generated by MPBUILD invokes MERGE implicitly as required.)

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. The GSD records hold information needed to resolve those intermodule references.

The ISD records contain information on all symbols, including global and local (Pascal only), used in the object module. PASDBG uses this information for 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 option in effect. MERGE can generate ISD records (for global symbols only) for input modules that do not already contain ISDs, such as modules created by the MACRO-11 assembler and library modules.

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 describes:

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

Figure 5-1 shows the MERGE utility's input and output files.

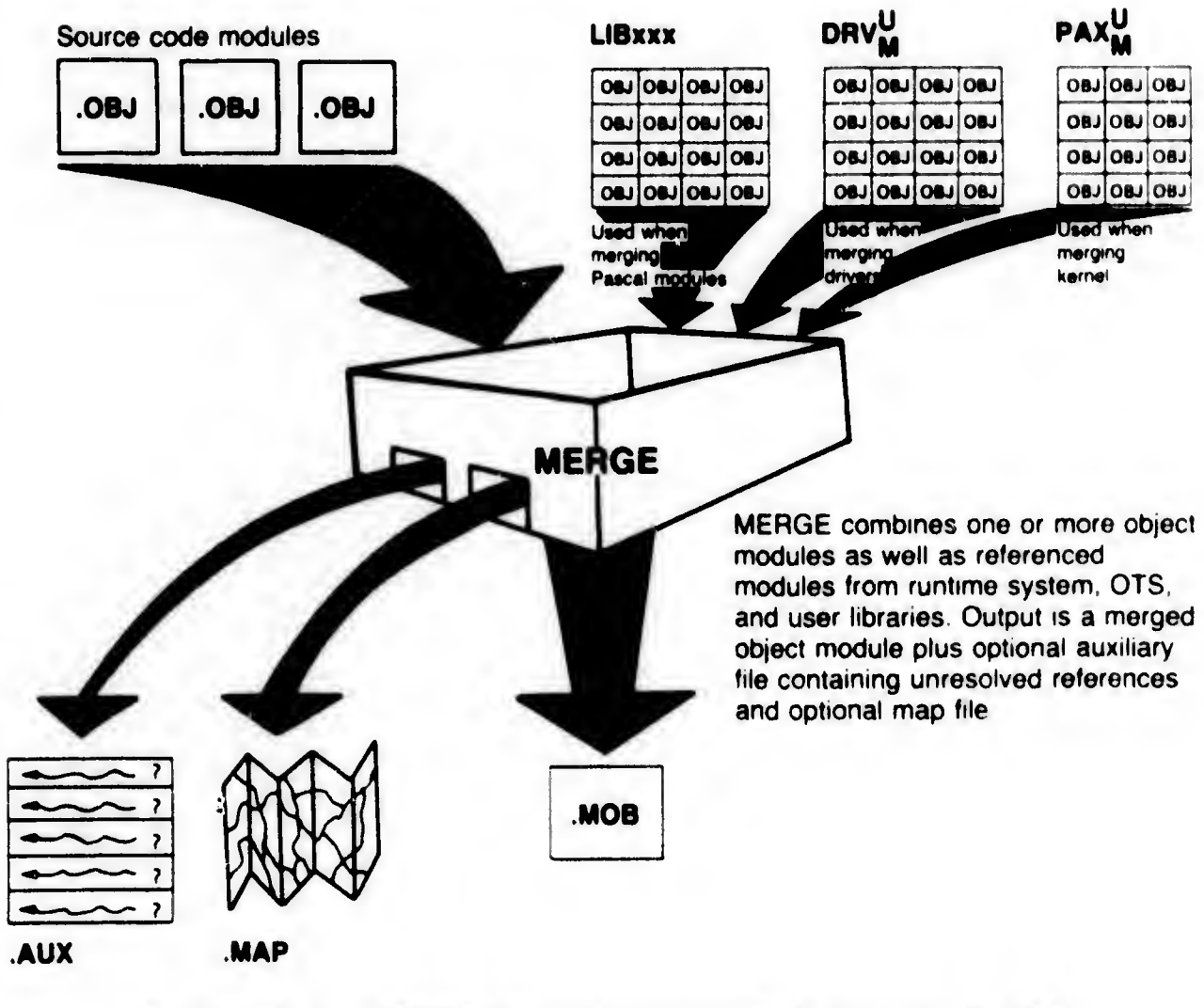

Figure 5-1 MERGE Utility Input and Output Files

5.1 FUNCTIONS OF MERGE

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

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

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; any unresolved global references; and the files included in the merge.

MERGE can also be used to optimize the kernel in respect to 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.

5.1.1 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 and == operator in MACRO-11. MERGE looks through the input object modules to find and keep track of, in its symbol table, all global symbol definitions and external references. MERGE tries to find a global definition to satisfy each external reference. When it matches a global reference with a global definition, MERGE deletes the reference but retains the definition.

MERGE consolidates global symbol definitions for program sections having the same name. A program section is a unit of instructions, data, or both that can be relocated independently. The MicroPower/Pascal compiler divides a module into program sections. When MERGE finds several program sections having the same name but that are in different object modules, it creates a single valid program section name entry in the merged object module. MERGE places all global symbol definitions for a program section after that 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.

5.1.2 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 start location is changed. When MERGE combines program sections, it updates the RLD records to reflect the amount of text included in each section from other modules.

5.1.3 Satisfying Library References

One of MERGE's most important functions is to resolve references to object library modules. 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 library files specified in the MERGE command line. During the search phase, when MERGE finds a global definition in a library 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 "satisfies" references to library routines and data structures.

The MicroPower/Pascal software package includes four versions of the Pascal Object Time System (OTS) library: LIBNHD.OLB, LIBEIS.OLB, LIBFIS.OLB, and LIBFPP.OLB. (Each version supports one of the possible PDP-11 instruction set options.) When merging object modules created by the MicroPower/Pascal compiler, you include the appropriate OTS library for your target system -- conventionally referred to as LIBxxx.OLB -- in the MERGE command line. If MERGE cannot satisfy all symbol references within the input object modules it is merging, it searches LIBxxx.OLB and/or any other object library you might specify to find matching global definitions. If any unresolved references remain after MERGE scans the libraries, it issues the warning message "Undefined globals:" followed by a list of the unresolved symbols.

Note that MERGE searches libraries in the order in which they appear in the MERGE command and satisfies references on a "first found" basis. Also, some library modules contain references to other libraries, as discussed below. Therefore, the order in which multiple libraries are specified is often important. In general, the common Pascal OTS library should always be specified last when used in a merge operation.

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

1. PAXM.OLB and PAXU.OLB -- the mapped and unmapped versions of the kernel module library, used when merging a system configuration file to build a kernel.

2. DRVM.OLB, DRVU.OLB, and DRVK.OLB -- the mapped, unmapped, and KXT11-C versions of the device handler (driver) library, used when merging a handler prefix module to build a device handler process.

3. FSLIB.OLB -- the Pascal file-system support library, used when merging a Pascal-implemented user process that requires file system support.

4. MACFS.OLB -- the MACRO-11 file-system support library, used when merging a MACRO-implemented user process that requires file system support.

5. DSPNHD.OLB, DSPEIS.OLB, DSPFIS.OLB, and DSPFPP.OLB -- four versions of the Directory Service Process (DSP) module library, differentiated by target instruction-set support. (The DSP is implemented in Pascal.) One of these libraries, DSPxxx.OLB, is used when merging the DSPPFX.OBJ module to build a DSP for an application requiring file system support.

6. RHSLIB.OLB -- the MicroPower/Pascal Analog Subsystem interface library, used when merging a Pascal-implemented process that employs the Pascal A/D and D/A procedure interfaces to the AA and KW handlers and the AB "pseudo handler." (RHSLIB.OLB is the object-time complement of the RHSDSC.PAS INCLUDE file, which provides external declarations of the analog interface procedures.)

Chapters 3 and 4 and Appendix A of this manual discuss and give examples of using the PAXx, DRVx, DSPxxx, FSLIB, and MACFS libraries. Appendix G of the MicroPower/Pascal Runtime Services Manual describes the RHSLIB procedures.

5.1.2.1 **Ordering of Multiple Object Libraries** - MERGE attempts to resolve any unsatisfied references found within a library object module that it extracts: one module may refer to another in the same library or in a different library. MERGE does so by further, iterative searching of the current library and, if necessary, by searching any additional libraries that you specify. MERGE will not "look back," however, to a previously searched library in the attempt to resolve a reference. Thus, the symbol-resolution process can be affected by the order in which you specify input files, especially libraries. When building any Pascal-implemented device handler, for example, which requires merging a handler prefix module with the handler library (DRVx.OLB) and the Pascal OTS library (LIBxxx.OLB), you must be careful about the order of the libraries in the MERGE command. A handler prefix module always contains at least one

reference to the required handler object module in the handler library, which causes that module to be "pulled in" from library and combined with the prefix module. If the handler module was implemented in Pascal -- as is the case for the AA, KW, and YA handlers -- that module in turn contains references to common Pascal OTS modules in LIBxxx. Therefore, the handler library specification must precede the OTS library specification in the MERGE command line. (As a general rule, the prefix module itself does not contain references to symbols in LIBxxx.)

As a specific example, the following command sequence compiles the AAPFX.PAS prefix file and correctly merges the AA handler process for a mapped application:

```
>[MCR] MPP    or    $ MPPASCAL
MPP>AAPFX = dev:[ufd]AAPFX/IN:EIS

>[MCR] MRG    or    $ MPMERGE
MRG>AAHAND=AAPFX,KERNEL.STB,dev:[ufd]DRVM/LB,dev:[ufd]LIBEIS/LB
```

The AAPFX.OBJ module contains references to global symbols defined in the AADRV module in DRVM.OLB, which causes that module to be extracted and combined with AAPFX. The AADRV module in turn contains references to globals defined in modules residing in LIBEIS.OLB. If the LIBEIS specification were to precede the DRVM specification in the command line, LIBEIS would be searched prior to DRVM, resolving no references, and the external references subsequently found in the AADRV module would remain unresolved. (The kernel STB file, created in a previous kernel relocation step, resolves the address values of kernel-routine entry points required by calls for primitive services made in the handler process. The STB file specification can appear anywhere in the command line relative to library files, since it is treated as an ordinary object module. The primary difference is that an STB file contains only GSD records.)

The possible interlibrary reference problem described for DRVx.OLB also holds true for DSPxxx.OLB, FSLIB.OLB, and RMSLIB.OLB. That is, most of the modules in those libraries also contain references to modules in the common Pascal OTS library. Thus, the general requirement is that LIBxxx.OLB, when it is used, must follow any other library also specified in the MERGE command. (LIBxxx.OLB modules do not themselves contain references to any other library, although many of them do contain external references that must be resolved by the kernel STB file.) A similar precaution must be observed for any user libraries that contain interlibrary references.

5.1.3.2 **Ordering of All MERGE Inputs** - In general, the recommended safe ordering of input files in the MERGE command line is as follows:

1. Specify all object modules (.OBJs) first
2. Specify the .STB file, if any is used, next
3. Specify any mutually unrelated object libraries next
4. Specify LIBxxx.OLB, if used, last

More specifically, if the OBJ input to a user-process merge operation consists of more than one object module, 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. That is mandatory for debugging purposes. 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.

5.2 ROLE OF MERGE IN THE BUILD CYCLE

You use MERGE at several stages in the application build cycle (see Figure 5-2). You merge the assembled system configuration file with the kernel library (PAXU or PAXM) to create the kernel object module. You then use MERGE to create a complete object module for each static process to be included in the application image. You merge the DIGITAL-supplied system processes required by your target hardware devices -- primarily device handlers -- as well as the user-written processes that constitute the application-specific part of the application memory image. Optionally, you can use MERGE to optimize the kernel with respect to primitive modules after a complete application has been developed and tested.

Refer to Chapter 3 for an overview of the application build cycle.

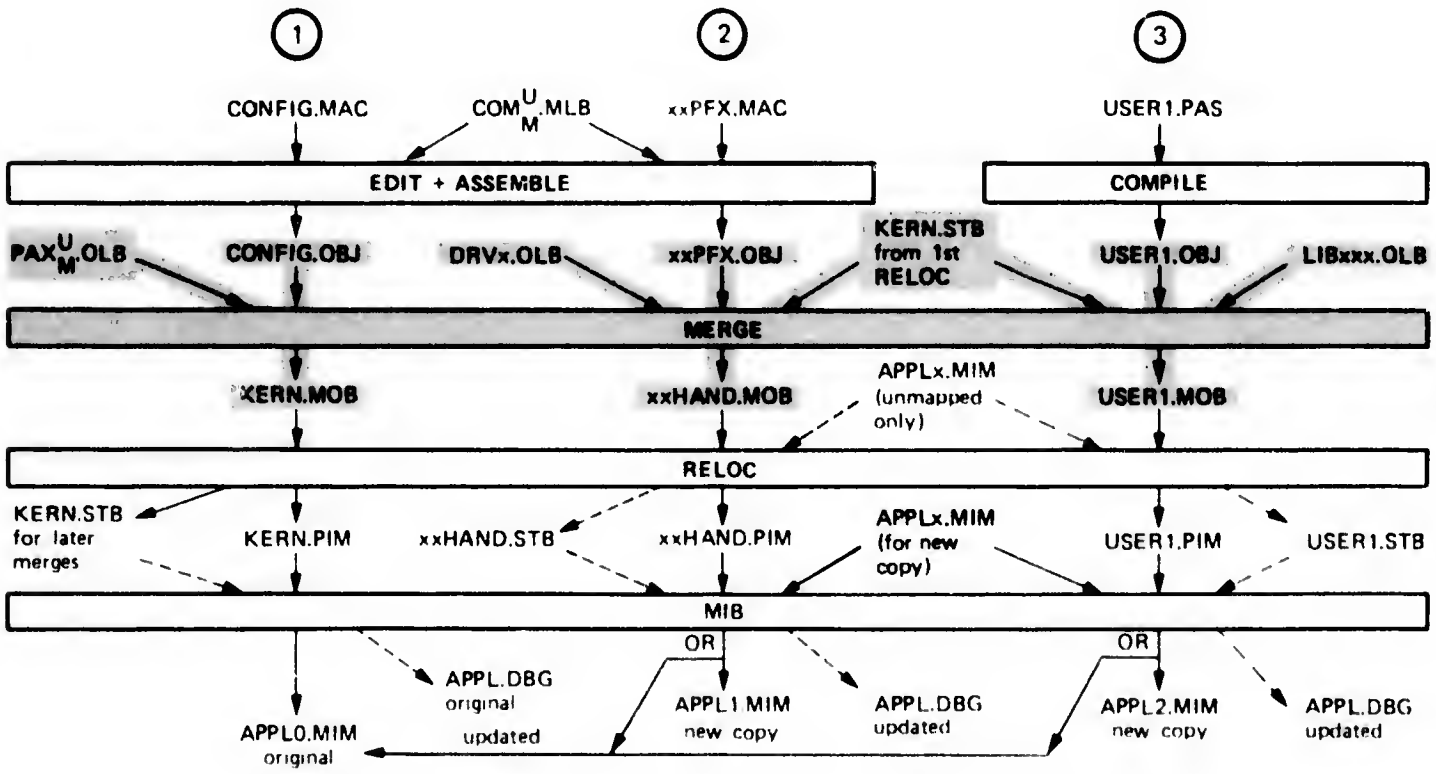

Figure 5-2 MERGE's Part in the Build Cycle

5.2.1 Merging the System Configuration File

By merging the configuration object module with the kernel library, you extract and configure the kernel library modules needed for your application. (Chapter 4 describes the system configuration macros and how to create or modify a configuration source file for assembly.) The output of the merge is a customized kernel object module.

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.

5.2.2 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 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 runtime references within a static process to kernel primitive service routines.

5.2.3 Optimizing the Kernel

You can use MERGE's optional auxiliary output file to build a kernel that includes only the primitive service routines that your application uses. That minimizes the size of the kernel. (The MERGE auxiliary file method of optimizing the kernel primitives is an alternative to using the PRIMITIVES macro in the system configuration file.)

Developing an application generally involves a number of build/debug/rebuild cycles, in which you have to debug and rebuild various static processes several times, 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. That involves a special use of MERGE to produce object files that contain, essentially, 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. 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 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. 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. 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>,,xxHANDLR=xxHANDLR,DRVx/LB
MRG>,,APASX=APASX,LIBxxx/LB
MRG>,,BPASX=BPASX,LIBxxx/LB
MRG>,,CPASX=CPASX,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. Specifying OPTIMIZE=YES causes kernel optimization via 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 for more information on editing 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>OPTKERNL=OPTCONFG,xxHANDLR.AUX,APASX.AUX,BPASX.AUX,-
MRG>CPASX.AUX, ... ,PAXx/LB
```

   In the kernel merge, the primitive references contained in the .AUX files will cause only those primitive modules actually 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 3.2.3), using OPTKERNL.MOB, and use the optimized kernel .STB file produced in that step when merging each of the static processes.

5.3 INVOCATION AND USE OF MERGE

For a PDP-11 RSX-11M/M-PLUS 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-11 forms of direct invocation may be used with respect to command line placement:

1. **>[MCR] MRG command-line**

2. **>[MCR] MRG @command-file**

   where the specified .CMD file contains one or more MERGE command lines

3. **>[MCR] MRG**
   MRG>**command-line** or **@command-file**
   MRG>

The format of the MERGE command line is described below. Note that only forms 1 and 2 of MERGE invocation can be used within a command file. Form 1 limits the entire line to 80 characters and precludes the use of continuation lines. Forms 2 and 3 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-11 RSX (Compatibility Mode) 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 .CMD; the file may contain one or more MERGE command lines. Type CTRL/Z in response to the MRG> prompt to exit.

5.3.1 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 a PDP-11 RSX-11M/M-PLUS host system. The standard location for DIGITAL-supplied MicroPower/Pascal files, such as libraries and prefix modules, is MP:[2,10] for a native RSX-11 system.

If you are using a VAX/VMS host in VAX-11 RSX compatibility mode, 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. At least one input file must be specified. (If no output file is specified, MERGE performs an error check only.)

[mobfile],[mapfile][,auxfile]=infile1[,infile2 ,...][/switches]

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

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. (An auxiliary output file, therefore, is a limited form of object module file, containing only GSDs for undefined symbols.) The default file type is .AUX. If an auxiliary file is specified, MERGE assumes that unsatisfied global references are expected, and no warning message is issued if any occur. (The references are listed in the section map, however, if any.) If this field is omitted, no auxiliary file is produced, and a warning message is issued for any unsatisfied references.

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 switch, /LB, must be appended to a library file specification. The default file type is .OBJ unless the /LB switch is specified; the type default for a library file is .OLB. The debug switch, /DE, can be appended to any input file specification. The /DE switch is normally used on object module files and may be specified for a library. In general, you should not specify the /DE switch for a .STB file.

/switches
: Any of the position-independent option switches summarized in Table 5-1. Multiple switches can be specified in a list of the following form:

    /switch1/switch2/...

  The position-independent switches are /IN:xxx, /NM:xxx, and /VR:n. The /DE, /LB, and /LB:xxx switches are position-dependent; they are intrinsically associated with a given file specification.

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 255 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. (The CTRL/Z response to the second MRG> prompt causes an exit from MERGE.) No auxiliary output file is produced, and no debug symbol information is propagated from or generated for the input modules. 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>PROCESS1=PROCESS1/DE,KERNDBG.STB,LIBFIS/LB
```

This command merges the object module PROCESS1, compiled from Pascal source code using the /DE and /IN:FIS compilation switches, with the kernel symbol table, KERNDBG.STB, and the OTS library, LIBFIS.OLB. KERNDBG.STB resolves references to kernel primitive entry points. LIBFIS.OLB resolves references to OTS modules, and MERGE includes the required OTS modules from the library in the merged object module PROCESS1.MOB. The /DE switch causes MERGE to propagate all debug symbol information (ISD records) it finds in PROCESS1.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 PROCESS1.PAS were compiled without /DE -- MERGE would generate such records for all global symbol definitions in the module.)

5.4 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 -- section name, section length, and section attributes
4. Unresolved global references, if any
5. List of files included in the merge

Figure 5-3 shows a sample MERGE section map.

```
(1) (2) MICROPOWER MERGE V01.00          Load Map          Tue 02-Feb-82 10:47:49

                        Title:  EXAMPL   Ident:  032004

(3) Section              Size     Attributes

    . ABS.              000000 (RW,I,GBL,ABS,OVR)
    .SDAT.              001750 (RW,D,GBL,REL,CON)
    .CODE.              003774 (RO,I,LCL,REL,CON)
    .ODAT.              000016 (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.              001532 (RO,D,LCL,REL,CON)
    .ALST.              000076 (RO,D,GBL,REL,CON)
    ..INI.              000002 (RO,D,GBL,REL,CON)
    .OTS.               006316 (RO,I,GBL,REL,CON)
    .ECOD.              000006 (RO,I,LCL,ABS,OVR)
    .SNDF.              177766 (RO,I,LCL,ABS,OVR)
    .HDDF.              177776 (RO,I,LCL,ABS,OVR)
    .STDF.              000010 (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)
    .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)
    .SYIM.              000076 (RW,D,GBL,REL,CON)
    ..BEG.              000000 (RO,D,GBL,REL,CON)
    ..ZND.              000000 (RO,D,GBL,REL,CON)

(4) Undefined Globals:

    $REXC
    $PELM
    $QELM
    $STPC
    $CRPC
    $DLST
    $SGNL
    $WAIT
    $DLPC
    $GTST
    $CHGP
    $CRST

(5) Files Included:

    DK :EXAMPL.OBJ
    DK :LIBRHD.OBJ
```

Figure 5-3 Sample MERGE Section Map

5.5 MERGE OPTIONS

Sections 5.5.1 through 5.5.6 describe the MERGE option switches that are summarized in Table 5-1.

Table 5-1
MERGE Switches

Switch	Meaning
/DE	Includes debug symbol information (ISD records) in the output object file. Kernel and user-process symbol information must be "passed along" 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.
/NM:name	Specifies the name to be assigned to the object module created during the merge operation; primarily useful for debugging purposes. This option overides the effect of a MACRO-11 .TITLE statement, if any, or a Pascal PROGRAM or MODULE name.
/LB	Identifies a file as an object library that is to take part 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 switch 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 switch (no arguments).
/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.

5.5.1 Debug Symbols (/DE)

The purpose of the /DE switch is to propagate, or "pass 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 compiling under the /DE option.

MERGE can also create ISDs when necessary but only for global program symbols. Ordinarily, you will use MERGE to create rather than propagate ISDs only for input modules generated by the MACRO-11 assembler. (The standard MACRO-11 assembler produces ISD records when requested by the /E:DBG option, but those ISDs are not in MicroPower/Pascal format and therefore are considered as foreign if encountered by MERGE in an input module.)

When appended to an input file specification, the /DE switch has either of two effects, depending on whether the input file already contains any valid ISD records. If MERGE finds any valid ISDs in an input module or an extracted library module, it 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 causes MERGE to pass debug symbol information through to the output object module in either case, regardless of whether that information was generated initially by the Pascal compiler or is generated in the merge step.

If you specify /DE on the output .MOB file, MERGE propagates or creates ISD records as described above but for all the input files; the effect is exactly the same as if /DE were appended to each input file specification. Normally, you would not want to do that.

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. (The configuration module contributes no symbols useful for debugging; do not use /DE on it.) When building a user process for debugging, specify /DE on the library file(s) only if you specifically want global symbols from library modules for debugging purposes. (Note that DIGITAL-supplied libraries do not contain ISD records. MERGE must create ISDs if /DE is specified for any supplied library file.)

Generally, you should not specify the /DE switch for an STB file. Doing so in the standard case -- 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 will -- you will get the error "Bad ISD in file." Similarly, do not use /DE on the output side of a command line that specifies an STB file as an input. 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 "premerging" several user object modules into one OBJ file.

In example 1, MERGE includes ISD records in the output module for each of the input modules -- PRGMXYZ, MODX1, MODX2, and MODX3. In example 2, ISD records are included only for PROCXYZ.OBJ.

Example 1

```
MRG>PROCXYZ.OBJ/DE=PRGMXYZ,MODX1,MODX2,MODX3
```

Example 2

```
MRG>PROCXYZ=PROCXYZ/DE,KERNDBG.STB,LIBNHD/LB
```

Note that when you build an application for debugging, the module containing the PROGRAM compilation unit in Pascal or the .TITLE statement defining the static process name in MACRO-11 must be the first input file to MERGE. If it is not, and you do not use the /NM switch, the SET PROGRAM command in PASDBG will fail. In example 1, for instance, PRGMXYZ.OBJ must be the first input module specified in the command line, assuming that PRGMXYZ represents a Pascal PROGRAM unit or a MACRO-11 module containing the DFSPC$ (Define Static Process) macro and a corresponding .TITLE statement. (See the /NM switch description for further information.)

5.5.2 Include Module from any Library (/IN)

The format of the Include Module option switch 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 a library reference). The switch causes MERGE to extract and include in the output module any library modules it 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 it is not needed to satisfy a direct reference in one of the input object modules, or before it is needed to satisfy a reference in a subsequent library module. (This switch may be useful in solving a certain form of interlibrary reference conflict that cannot be solved solely by the order in which the several libraries are specified.)

5.5.3 Module Name (/NM)

The /NM:name switch allows you to specify the name of the object module to be created during the merge. The name argument can consist of from one to six RAD50 characters. The name specified by the switch supersedes the "normal" output module name that MERGE otherwise arrives at during its processing of input modules. For debugging purposes, the output module name, however established, must match the runtime name defined for the static process represented by a merged object module.

Ordinarily, you would use the name 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. For a static process implemented in Pascal, the module and static-process name correspondence necesary 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.

Normally, MERGE uses the module name of the first object module that it encounters in the input stream as the output module name. Note that 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. 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 runtime 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.

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 switch affecting its output STB file, which allows you in turn to override the module name determined in the MERGE step.)

5.5.4 Library File Identification (/LB)

The /LB switch, 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. (Note that /LB has a distinctly different meaning if specified with arguments, as described below.)

5.5.5 Extract Modules from Specific Library (/LB:module:...)

The /LB switch 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 switch 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 switch (no arguments). For example, consider the following command:

```
MRG>OUTMOD=INPUT1,MYLIB/LB:MODULX,MYLIB/LB,GENLIB/LB
```

The first specification of MYLIB causes the library module MODULX to be extracted from MYLIB.OLB and merged with INPUT1.OBJ. The second specification of MYLIB causes MYLIB also to be searched, 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.

5.5.6 Version Number (/VR)

The /VR:xxx switch allows you to specify a program version number or other identifier value for the output object module. The switch argument can consist of from one to six RAD50 characters. (The identifier appears in the maps produced by MERGE, RELOC, and MIB.) 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.

If you do not specify the /VR switch, 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 6

RELOCATING MERGED OBJECT MODULES

The RELOC utility program allocates memory for each program section, assigns base virtual addresses to the sections, sorts program sections by read-only or read/write attribute alphabetically within each category, relocates the program sections, and produces as output a process image (.PIM) file. RELOC also produces symbol table (.STB) files. See Figure 6-1.

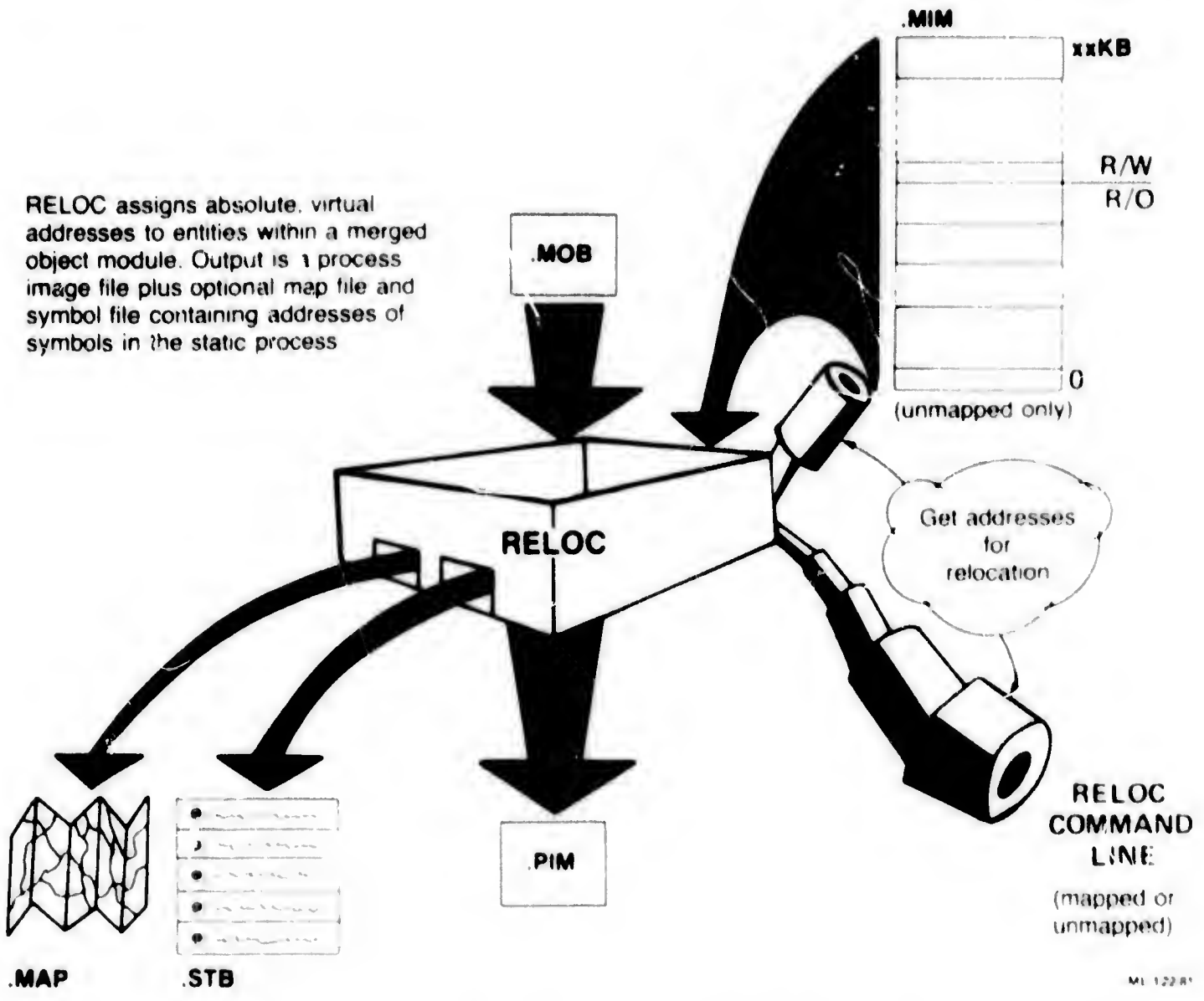

Figure 6-1 RELOC Utility Input and Output

RELOCATING MERGED OBJECT MODULES

This chapter describes:

- 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 below, or you can, for most applications, use the automated procedure MPBUILD.COM, described in Appendix B.

6.1 FUNCTIONS OF RELOC

After MERGE combines program sections, RELOC allocates memory for them by an appropriate relocation of addresses in all sections. Prior to relocation, the addresses of all symbols in each program section are offsets from a base address of 0, as recorded in the relocation directory of each p-sect.

RELOC assigns physical memory addresses for unmapped applications or virtual addresses for mapped applications to replace all the zero base addresses of the p-sects, allocating all sections contiguously by default. RELOC then adjusts all text records, using relocation directory information to assign new addresses.

RELOC normally sorts all p-sects by the read-only, read/write (RO/RW) attributes before it relocates them and puts them in the output module. The sorting of program sections 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 program sections alphabetically by section name within the read-only and read/write segments. RELOC concatenates text blocks of program sections having the same name. (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. ROM or RAM segments can be started at specific addresses, or RAM can be started on the next-available 4K-word virtual address boundary. You can specify the start location of given program sections in memory. (You supply either physical or virtual addresses as required by the application image.) You can extend a program section to a specific size. You can begin a program section 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, you can cause RELOC to include ISD records in the .STB file. The ISDs are necessary if you want to use the PASDBG debugger with the application. You can direct RELOC to provide a relocation map that contains program section names, sizes, starting addresses, and global symbols. You can alphabetize the map symbols and create a shortened map. Table 6-1, in Section 6.5, summarizes the RELOC options.

6.2 ROLE OF RELOC IN THE BUILD CYCLE

You use RELOC at several stages in the build cycle. 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, producing for each a separate .PIM file and optionally a separate .STB file for debugging. Then you use the MIB utility to insert each process in the memory image file. See Section 7.6 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. Refer to Chapter 3 for an overview of the application build cycle.

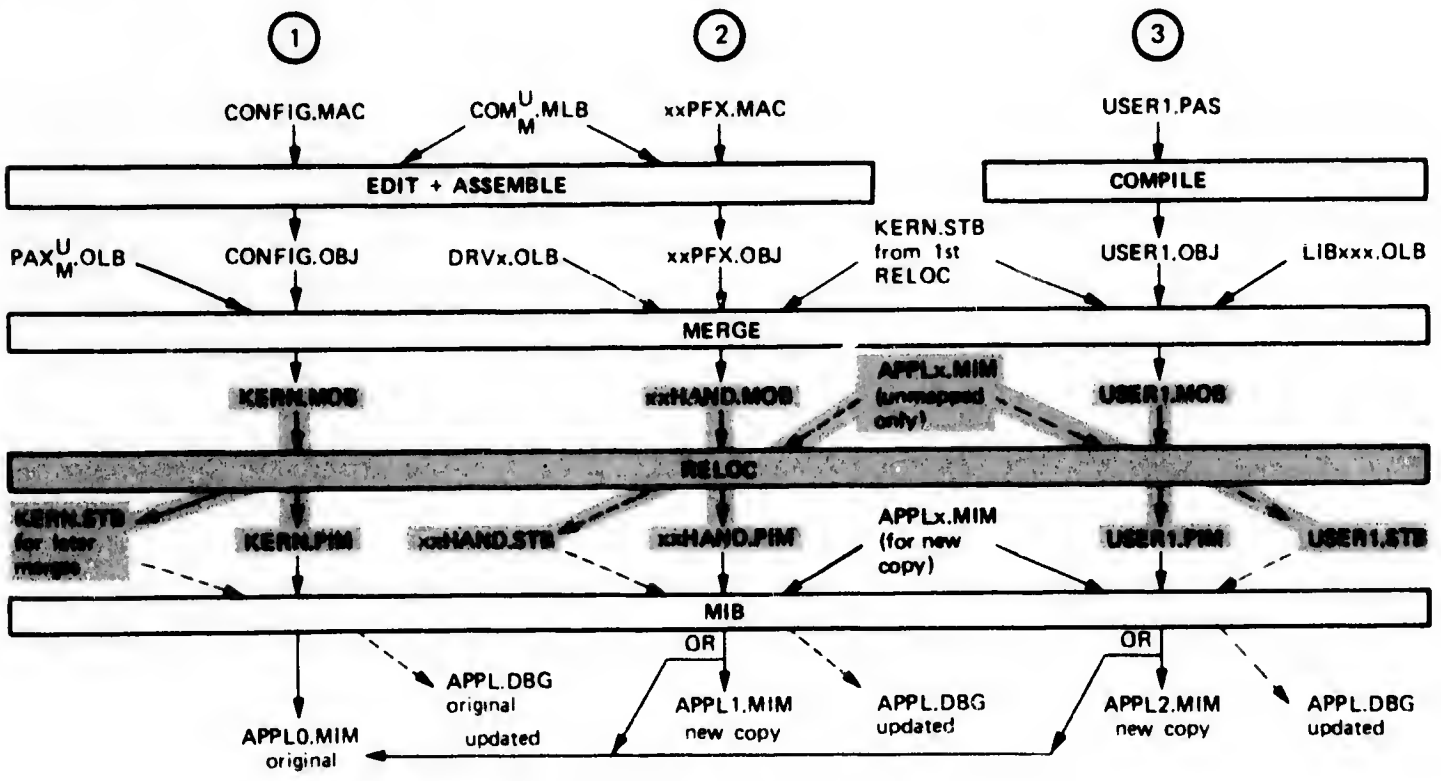

Figure 6-2 RELOC's Part in the Build Cycle

6.3 INVOCATION AND USE OF RELOC

For a PDP-11 RSX-11M/M-PLUS Development System: Assuming that RELOC has been installed according to installation-procedure defaults, you invoke it by the task name REL, as follows:

```
>[MCR] REL
```

(Precede "REL" with "MCR" only if your CLI mode is DCL.) The following three standard RSX-11 forms of direct invocation may be used with respect to command line placement:

1. **>[MCR] REL command-line**

2. **>[MCR] REL @command-file**

   where the specified .CMD file contains one or more RELOC command lines

3. **>[MCR] REL**
   REL>command-line or **@command-file**
   REL>

The format of the RELOC command line is described below. Note that only forms 1 and 2 of RELOC invocation can be used within a command file. Form 1 limits the entire line to 80 characters and precludes the use of continuation lines. Forms 2 and 3 can be used to issue several RELOC command lines within one invocation or to avoid the 80-character limit. Type CTRL/Z in response to the REL> prompt to exit.

For a VAX-11 RSX (Compatibility Mode) 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
REL>command-line or @command-file
REL>
```

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

6.3.1 Command Line Format

RELOC 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 a PDP-11 RSX-11M/M-PLUS host system. If you are using a VAX/VMS host in VAX-11 RSX compatibility mode, 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. At least one input file must be specified. (If no output file is specified, RELOC performs an error check only.)

```
[pimfile],[mapfile][,stbfile]=mobfile[,mimfile][/switches]
```

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 this field is omitted, 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 will consist 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, that is, relocated, values for each symbol, reflecting either physical or virtual addresses in the application memory image. (These entries are needed in the case of a kernel STB file for subsequent merging with processes.) In addition, if the /DE switch is specified, the STB file will also contain 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. The ISD records are needed when building either a kernel or a user process for debugging, since the STB file serves as 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 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. Specify an input mimfile if you are relocating a process for an unmapped memory image and you are not specifying any addresses 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. You may not specify a mimfile if you use any of the following option switches: /RO, /RW, /QB, /EX, /UP, or /AL.

/switches
Any of the option switches summarized in Table 6-1. All RELOC switches are position-independent. Multiple switches can be specified in a list of the following form:

/switch1/switch2/...

The following examples illustrate basic RELOC command line syntax.

Example 1

```
>[MCR] REL   or   $ MPRELOC
REL>PROCXYZ.PIM,PROCXYZ.MAP,PROCXYZ.STB=PROCXYZ.MOB/DE
REL><CTRL/Z>
```

This command relocates the merged object module PROCXYZ.MOB for a mapped application and produces the process image file, PROCXYZ.PIM, the map file PROCXYZ.MAP, and the symbol table file named PROCXYZ.STB. The /DE switch causes any debug symbol information (ISDs) contained in the .MOB file to be included in the .STB file. (The CTRL/Z response to the second REL> prompt causes an exit from RELOC.) Since only the standard file types are used, you could omit the file types and let RELOC assume the defaults, as follows:

```
REL>PROCXYZ,PROCXYZ,PROCXYZ=PROCXYZ/DE
```

Example 2

```
REL>PROCABC,PROCABC=PROCABC,APPLICUM
```

This command relocates the merged object file PROCABC.MOB based on physical starting addresses obtained by inspection of the unmapped APPLICUM.MIM memory image file. The outputs specified on the left of the equal sign are the process image file PROCABC.PIM and the map file PROCABC.MAP. The command does not request a symbol table file; therefore, no /DE switch is applicable. This form of RELOC command, specifying an input memory image file, is valid only for unmapped applications.

Example 3

```
REL>,PROCABC=PROCABC,APPLICUM
```

This command relocates the merged object file as described in example 2 but produces only a relocation map file.

6.4 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. PIM file specification and input module name (Title) and version (Ident)
4. Information about each program section processed by RELOC during the relocation -- section name, section attributes, section base address, section 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

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. However, you can use the /WI option to produce a 6-column map listing. Figure 6-3 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.)

```
(1) MICROPOWER RELOC V01.05 MU      Load Map      (2) Thu 08-Dec-83 16:35:38

(3) EXAMPLDB.PIM    Title:  EXAMPL  Ident:  342162

  Section   Address  Size          Global  Value   Global  Value   Global  Value

  (4). ABS.    000000 000000 (RW,I,GBL,ABS,OVR)

                                     $TERM  000000   $MAP   000000   SA$RIR 000000
                                     SA$ROR 000000   PR.0   000000   SA$OFF 000000
                                     CX$STD 000000   SA$IFF 000000   CM$DPV 000000
                                     CS$X25 000000   DC$DSK 000000   DK$DY2 000000
                                     IF$RDP 000000   RS$PRT 000000   RT$DRV 000000
                                     IS$NOR 000000   TT$DL  000000   DP.LNK 000000
                                     BS.FPT 000000   CD.LNK 000000   CS.FPT 000000
                                     PC.FLK 000000   EA$DIS 000000   E.ADDR 000000
                                     ES$NSC 000000   EH.MEM 000000   FB.ADR 000000
                                     FR.STA 000000   ID.PCB 000000   KC.R4  000000

                                        .
                                        .
                                        .

    .SQUE.     000000 000000 (RO,I,LCL,ABS,OVR)

    .STDF.     000000 000000 (RO,I,LCL,ABS,OVR)

    ..BEG.     043612 000000 (RO,D,GBL,REL,CON)

                                     $BINIT 043612

    ..INI.     043612 000002 (RO,D,GBL,REL,CON)

    ..ZND.     043614 000000 (RO,D,GBL,REL,CON)

                                     $EINIT 043614

    .CDAT.     043614 000074 (RW,D,GBL,REL,CON)

    .ODAT.     043710 000022 (RW,D,GBL,REL,OVR)

    .SDAT.     043732 001750 (RW,D,GBL,REL,CON)

                                     $BDAT  043732   $EDAT  045702

    .SYIM.     045702 000114 (RW,D,GBL,REL,CON)

                                     $FREEL 045702   $HPLCK 045704   $SPI   045720
                                     $APIM  045722   $CSLOC 045722   $IN    045724
                                     $OUT   045726   $AFLHD 045730   $DSPSD 045732
                                     $IFV   045746   $OFV   045772

(5) Transfer address =  000001

(6) Total ROM size =  014764

(7) Total RAM size =  002202
```

Figure 6-3 Sample Relocation Map

6.5 RELOC OPTIONS

This section describes the RELOC option switches summarized in Table 6-1.

NOTE

Any numeric value specified in a RELOC option switch 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 6-1
RELOC Switches

Switch	Meaning
/AB	Lists symbol names alphabetically within p-sects in the load map.
/AL	Aligns the first read/write program section 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.
/DS	Causes RELOC to leave program sections in the order of their occurrence in the input module, disabling the normal sorting of sections first by read-only and read/write attribute and then by alphabetical order of section names within the read-only and read/write "segments." (This option is used only in a few special cases.)
/EX:name:size	Extends a program section to a specified size, in bytes.
/NM:name	Specifies, for the output STB file, the name of the module to be relocated; overrides actual input module name (PASDBG "program name") for debugging purposes.
/QB:name:addr[:...]	Sets the base address for the named program section; multiple pairs of section names and base addresses can be specified.

(Continued on next page)

Table 6-1 (Cont)
RELOC Switches

Switch	Meaning
/RO:addr	Causes the first read-only program section to start at the specified physical or virtual address. (If used with /DS, the base of the program is unconditionally set to addr.)
/RW:addr	Causes the first read/write program section to start at the specified physical or virtual address.
/SH	Causes a shortened load map to be produced; the kernel symbols defined in the . ABS. p-sect are omitted.
/UP:name:value	Rounds up the length of the named program section so that the next free address is a whole-number multiple of the the specified value, which must be a power of 2.
/VR:xxx	Specifies a program version number or other "ident" that is to appear in the load map for the module to be relocated.
/WI	Causes a wide map to be produced -- six columns rather than the usual three; useful for creating a shorter map for line-printer listings.
/ZR:n	Sets the value of unaccessed locations in the image to n; the default value is 0.

6.5.1 Alphabetical Symbol Listing (/AB)

The /AB switch causes RELOC to list symbol names in the load map alphabetically within a given program section. 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.

6.5.2 Align First RW section at 4K-Word Boundary (/AL)

The /AL switch causes RELOC to start the first read-write (RW) program section found in the input module -- subsequent to the RELOC section 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 noncontiguous first RW program section. (MIB automatically positions the RW segment in RAM for a ROM/RAM target system.)

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, all program sections are treated as if they were read/write sections for purposes of positioning in physical memory. Special relocation in a mapped RAM-only environment is ordinarily needed only for driver-mapped device handler 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 -- you cannot also specify a memory image (.MIM) file as input. In addition, the /AL switch and the /RW or /DS switch are mutually exclusive.

6.5.3 Debug Symbols (/DE)

The /DE switch causes all internal symbol directory (ISD) records found in the input module to be suitably relocated and included in the output symbol table file. The /DE switch has no effect on the output PIM file. (The ISDs contain symbol information that is subsequently installed in the DBG file by MIB and used by PASDBG.) You must specify /DE when relocating a kernel for an application to be built with debug support and when relocating any process that you will want to debug via PASDBG.

6.5.4 Disable Section Sort (/DS)

The /DS switch prevents RELOC from performing its standard program section sort and causes all sections to be relocated in the process image according to the order of their occurrence in the input module file. Otherwise, RELOC first segregates the program sections into two groups, or segments, by RO/RW attribute, with the RO segment preceding the RW segment, and then sorts the sections alphabetically by p-sect name within each segment.

Many intrinsic MicroPower/Pascal mechanisms, both build-time and run-time, depend on RELOC's grouping and alphabetic sorting of sections for their proper functioning. In addition, the segregation of RO and RW sections is crucial for the physical memory segmention required by an actual or simulated ROM/RAM target environment. (See also the specific requirement concerning .ALST. described below.) Therefore, the DS option is not intended for general use and, if used, places full responsibility for correct p-sect ordering on the implementor. The switch is provided only for the unusual situation of interspersed ROM/RAM/ROM memory, as caused by use of low-addressed, on-board RAM in a ROM/RAM FALCON-PLUS target, in which case the MACRO programmer needs to control the ordering of program sections directly and unconditionally. Ordinarily, the source macros PURE$, PDAT$, and IMPUR$ (Chapter 3 of the MicroPower/Pascal Runtime Services Manual) suffice for control of program sectioning, when used in conjunction with RELOC sorting. (The /DS switch normally cannot be used for a process implemented in Pascal, since the compiler transparently provides all required program sectioning, and the Pascal programmer has no control over section order without resorting to a "dummy" MACRO-11 module.)

You cannot use the /RW or /AL switches if you use the /DS switch. Also, if you use /DS, the load map produced by RELOC does not show the ROM (RO segment) and RAM (RW segment) total sizes.

Note that one general requirement must be met by a static process for any kind of target system environment, as follows. The first relocatable program section 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 section 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 section is placed properly at the beginning of the process. (Zero-length p-sects such as . ABS. do not enter into consideration, of course, for section ordering purposes.) In a MACRO-11 source program, the Define Static Process (DFSPC$) macro call establishes the .ALST. p-sect, containing the static process list element generated by DFSPC$. Therefore, the MACRO programmer must not define any p-sect that will precede that section. It is recommended that only the standard MicroPower macros PURE$, PDAT$, and IMPUR$ be used for program sectioning in a MACRO-11 process implementation. (The MicroPower/Pascal compiler automatically generates the .ALST. p-sect for a PROGRAM compilation unit.)

6.5.5 Extend Section to Specified Size (/EX)

The /EX:name:size switch lets you extend a named program section to a specified size. (You can extend only one program section in a given static process.) The name argument in the switch is a p-sect name, and size is the total number of bytes you want allocated to the program section, expressed in octal. The size value must be an even number greater than the actual section size; if the value is less, RELOC ignores the switch and uses the actual section size. In any case, RELOC starts the following section at the next contiguous location.

You can use this option, for example, to extend a program section in order to increase the size of a stack or allocate patch space at the end of the section.

If you are building an unmapped application and use /EX, you cannot also specify a memory image (.MIM) file as input. That is, you must use the /RO switch, and possibly /RW also, in addition to /EX in the unmapped case.

6.5.6 Program/Process Name (/NM)

The /NM:name switch lets you specify a "program" (static-process) name for debugging purposes, overriding the module name GSD/ISD entries contained in the input merged object module. This switch is provided as an alternative to the MERGE /NM switch. (See the description of /NM for the MERGE utility.) The name argument may consist of from one to six RAD50 characters. The switch affects only the output STB file and, consequently, the DBG file updated in the subsequent MIB step.

The RELOC /NM switch has the same eventual effect on the program node name in the DBG file as does the MERGE /NM switch, and allows you to correct the same PASDBG-related problem caused by misuse of the .TITLE directive for a process implemented in MACRO-11. Further, the RELOC /NM switch permits you to override the effect of merging the input modules for any static process in the wrong order. If used, the name specified must match the runtime process-id defined for the static process, or the PASDBG SET PROGRAM command will fail.

6.5.7 Base Address for Specified Program Section (/QB)

The /QB switch lets you specify a base address for any named program section in the input module. In the absence of any special relocation switches, RELOC starts each section at the next available physical or virtual memory address. The /QB switch allows you to override the normal "next contiguous address" relocation of any given section if necessary to meet special application requirements or to satisfy special target-memory constraints. The format of the /QB switch 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 section. (You can specify up to eight section names and addresses.) For unmapped applications, /QB specifies a physical address; for mapped applications, /QB specifies a virtual address.

NOTE

In a mapped target system, any noncontiguous program section must start on a 4K-word virtual address boundary in order to satisfy mapping hardware requirements.

6.5.8 First RO Section at Specified Address (/RO)

The /RO:addr switch starts the first read-only (RO) program section found in the input module -- subsequent to the RELOC section sort -- at the specified address. (If you use /RO together with the /DS switch, the first section of the program is set to addr regardless of its access attribute. Note that /DS is not recommended for general use.) For unmapped applications, /RO specifies a physical address; for mapped applications, /RO specifies a virtual address.

If you are building an unmapped application and use /RO, you cannot also specify a memory image (.MIM) file as input. If your target system is ROM/RAM, you must use the /RW or /QB switch in addition to /RO.

NOTE

In a mapped target system, any noncontiguous program section must start on a 4K-word virtual address boundary in order to satisfy mapping hardware requirements.

Note that in any static process, the alphabetically first read-only p-sect must be named .ALST. to position the static process list element correctly in the memory image. In a MACRO-11 source program, the Define Static Process (DFSPC$) macro call establishes the .ALST. p-sect, containing the static process list element generated by DFSPC$. (The MicroPower/Pascal compiler automatically generates the .ALST. p-sect for a PROGRAM compilation unit.)

6.5.9 First RW Section at Specified Address (/RW)

The /RW:addr switch starts the first read/write (RW) program section found in the input module -- subsequent to the RELOC section 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, you cannot also specify a memory image (.MIM) file as input. In addition, the /RW switch and the /AL or /DS switch are mutually exclusive.

NOTE

In a mapped target system, any noncontiguous program section must start on a 4K-word virtual address boundary in order to satisfy mapping hardware requirements.

6.5.10 Short Map (/SH)

The /SH switch produces a "short" load map by excluding the kernel symbol definitions contained in the . ABS. p-sect from the map listing. The numerous kernel symbols (roughly 700) are ordinarily of no direct use for debugging. By default, the map file includes the kernel symbol definitions. If you omit the map file specification, the /SH switch is ignored.

6.5.11 Round Up Section Size (/UP)

The /UP:name:value switch rounds up the length of the named program section so that the next free address in the process image is a whole-number multiple of "value." The subsequent program section will normally start at that address boundary. The specified boundary value must be a power of 2. You can round only one section per process. The /UP option may be useful if you want to align a following data section, for example.

If you are building an unmapped application and use /UP, you cannot specify a memory image (.MIM) file as input.

6.5.12 Program Version Number (/VR)

The /VR:ident switch lets you override the program version number or other program identification value for the module being relocated. The ident argument may consist of from one 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 -- program identification GSD record -- found in the input module.

6.5.13 Wide Map (/WI)

The /WI switch 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 switch is ignored.

6.5.14 Value of Undefined Locations (/ZR)

The /ZR:nnn switch lets you set the value of undefined locations in the process image. The nnn 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 switch 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, and locations implied by use of the /EX or /UP switch, are affected by /ZR.

CHAPTER 7

BUILDING THE MEMORY IMAGE

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. The debug symbol file is needed when you use the PASDBG symbolic debugger. See Figure 7-1.

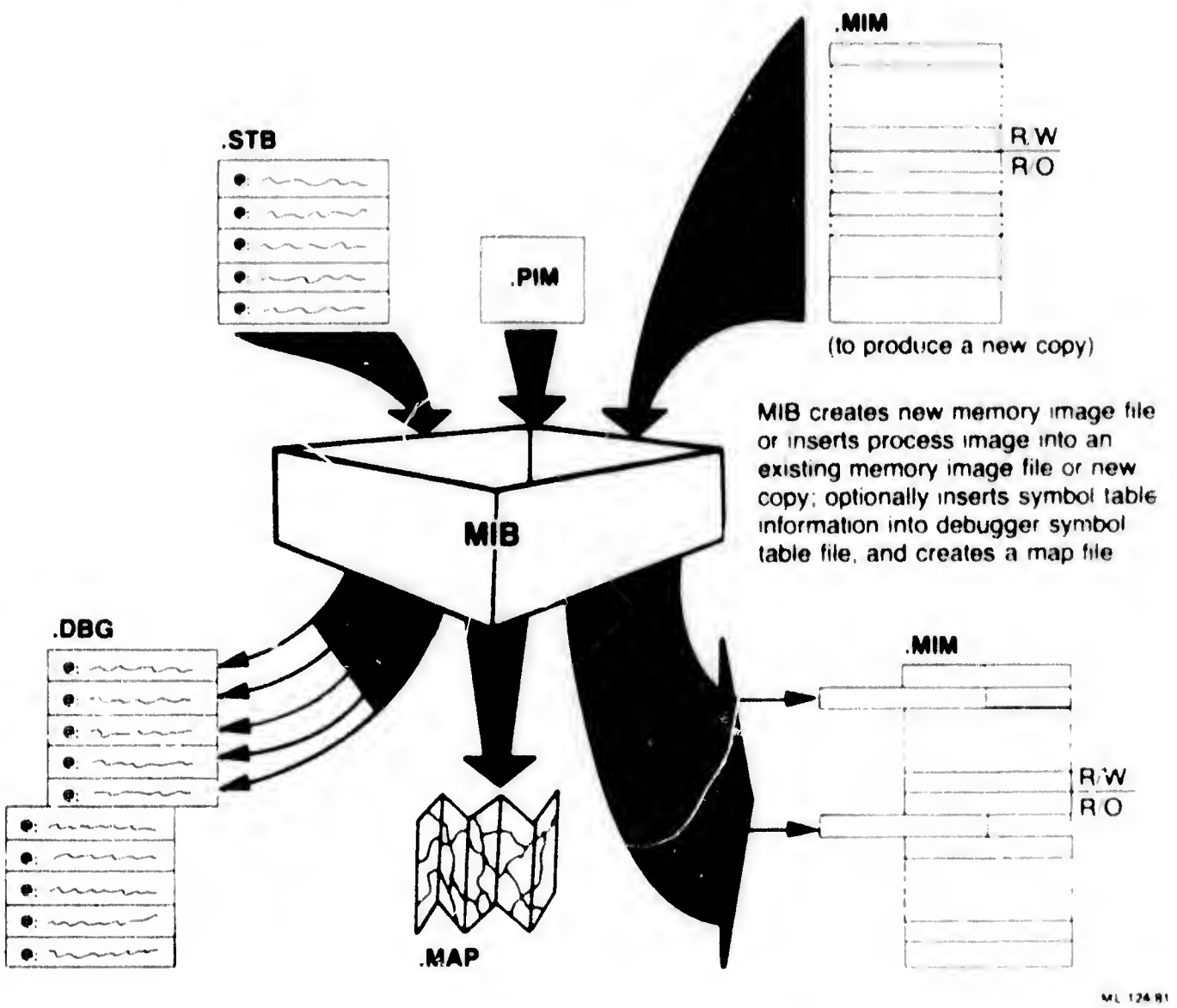

Figure 7-1 MIB Utility Input and Output

BUILDING THE MEMORY IMAGE

This chapter describes:

- MIB's functions
- MIB's role in the build cycle
- The invocation and use MIB
- The interaction between RELOC and MIB
- The listing file MIB optionally produces

You can run MIB yourself, as described below, or you may be able to use the automated build procedure MPBUILD, described in Appendix B.

7.1 FUNCTIONS OF MIB

The MIB utility program creates the memory image file, installs and removes static processes, prefixes bootstraps, creates the debug symbol file and installs and removes 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 replace one process, to change a process's start-up priority, or to change a process's exception group code.

7.1.1 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:

1. 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)

2. Bootstrap load format: RAM-only memory image, appropriate bootstrap in the MIM file, no DSM in the kernel

3. 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 will use PASDBG to either load and debug your application or to load it for independent execution. In either case, 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 will subsequently boot and load the application image from a target system disk or TU58 device.

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

7.1.2 PASDBG Load Format

A .MIM file in PASDBG load format contains a memory allocation table and a compressed memory image containing both read-only and read/write segments for all installed components. 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 flag word in the memory allocation table header indicates whether the memory image is for a mapped or an unmapped target. In addition, the flag word indicates whether the image is RAM-only or ROM/RAM. 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 7-2 illustrates both the PASDBG and bootstrap load formats.

7.1.3 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 7.2.) DIGITAL supplies bootstraps for all disk or disklike devices supported on a target system. You specify the appropriate bootstrap file for your system with the MIB /BS switch (Section 7.4.1), 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 FLX utility program, and then use the MicroPower/Pascal COPYB utility to make the volume bootable from a device on your target system. See Chapter 9 for further details.

You can install a bootstrap when you initially create the .MIM file in the kernel-build step, or you can install it at the end of a build cycle. The latter strategy is convenient if you might want the same memory image to be bootable from several different devices. (Once a bootstrap is installed in a memory image file, it cannot be removed.) 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.

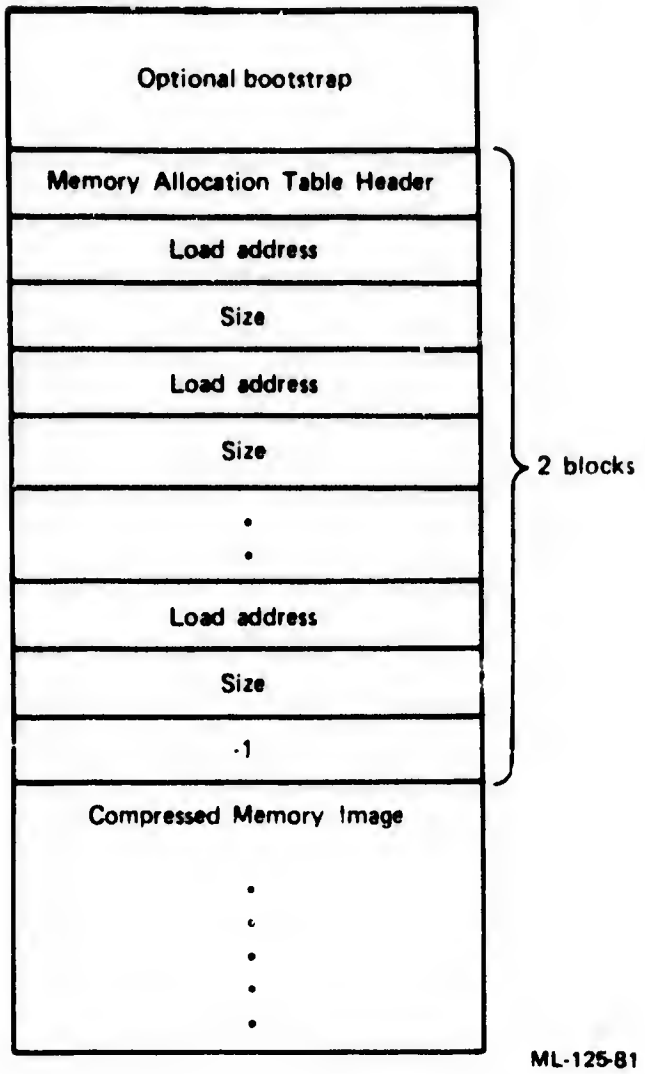

Figure 7-2 PASDBG or Bootstrap Load Format .MIM File

7.1.4 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-11 DECprom on a VAX/VMS host system, for example.)

Figure 7-3 shows a PROM programmer format memory image file.

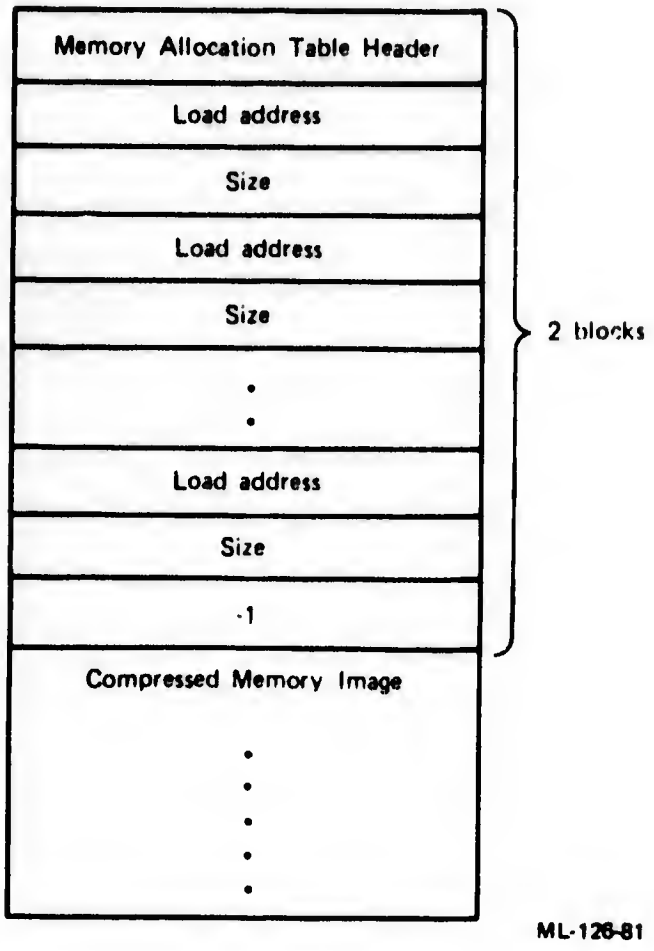

Figure 7-3 PROM Programmer Format .MIM File

7.1.5 Installing Static Processes

After you have used 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 specify both an input and an output MIM file in the successive MIB steps -- see description of the /SM (short-memory-image) switch in Sections 7.2 and 7.4.8. 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. MIB can also install debug symbol information for a given process in the optional debug symbol file.

7.1.6 Deleting Static Processes

You can use MIB to delete a static process from the memory image file. MIB removes the deleted process from the kernel's static process list, modifies the memory allocation table, and adds the freed memory to the kernel's free-memory list.

MIB can also remove the corresponding debug symbols from the debug symbol file in the same operation.

7.1.7 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 switch. 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 switches.

7.1.8 Creating a Map File

MIB can create a map file that shows the location and extent of the kernel, the currently 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 some other operation. Section 7.5 describes the MIB map.

7.1.9 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 correctly 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 STB file for the kernel from which MIB produces the initial portion of the DBG file. (The /DE switch must be used in both the MERGE and RELOC steps for the kernel.)

7.1.10 Installing Debug Symbols for a 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, the /DE switch 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.)

7.1.11 Deleting Debug Symbols for a Process

If you delete a static process from a .MIM file, you will want to delete the symbols for that process from the .DBG file as well. Include the .DBG file in the MIB command line when you specify the /EX switch. If you forget to delete the symbols from the .DBG file when you delete a static process from the .MIM file, you can delete them from the .DBG file as a separate step, again using /EX.

7.1.12 Compacting a Mapped Memory Image

Deletion of processes from a memory image file can result in possibly unusable "holes" in the image or, in the worst case, an overly fragmented, totally unusable image. (MIB detects and reports excessive image fragmentation as an unrecoverable error condition.) If the memory image is mapped, you can consolidate the free space in the image by using the /CM switch in a separate operation following a process deletion. In the compaction operation, MIB effectively rearranges the remaining processes in the MIM file to eliminate any gaps caused by deletions. (An unmapped image cannot be compacted.)

7.2 ROLE OF MIB IN THE BUILD CYCLE

You use MIB repeatedly to install each component needed in your target memory image (see Figure 7-4). 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 invoke 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 switch was used in the prior MIB step. MIB installs the static process in the output MIM file in accordance with the "rules" for the type of image being constructed -- mapped or unmapped, and RAM-only or ROM/RAM.

You can begin the MIB sequence by using the short-memory-image (/SM) switch together with the /KI switch 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 that case the size of the initial MIM file reflects the full extent of target memory specified in the kernel's configuration tables. That will allow 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. (A new generation of the MIM file is never created if only an output MIM file is specified for updating in place.)

If you choose to create a full-size MIM file, omitting the /SM switch, you 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 currently 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. Effectively, MIB copies the existing .MIM file contents to the new output file and extends the output file to accomodate the process being installed.

In general, to avoid confusion you should be consistent in using /SM; either use /SM 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 when and 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 might want to make a copy of the DBG file 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 currently installed debug symbols.

Chapter 3 provides an overview of the entire build cycle.

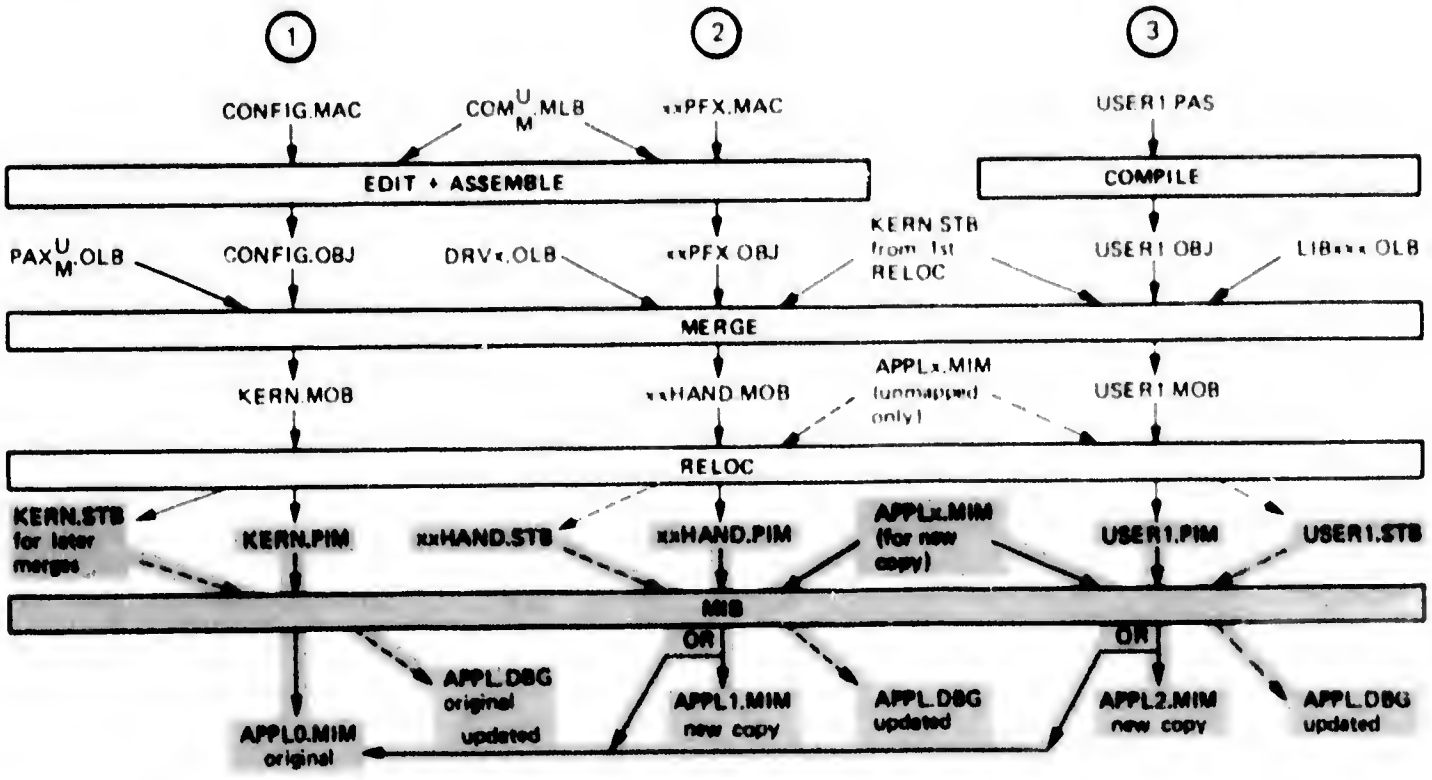

Figure 7-4 MIB's Part in the Build Cycle

7.3 INVOCATION AND USE OF MIB

For a PDP-11 RSX-11M/M-PLUS 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-11 forms of direct invocation may be used with respect to command line placement:

1. `>[MCR] MIB command-line`

2. `>[MCR] MIB @command-file`

   where the specified .CMD file contains one or more MIB command lines

3. **>[MCR] MIB**
   MIB>command-line or **@command-file**
   MIB>

The format of the MIB command line is described below. Note that only forms 1 and 2 of MIB invocation can be used within a command file. Form 1 limits the entire line to 80 characters and precludes the use of continuation lines. Forms 2 and 3 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-11 RSX (Compatibility Mode) 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.

7.3.1 Command Line Format

MIB accepts a command line in the form shown below. In that command line, all file specifications are in standard RSX format with respect to device and directory (UFD) information if you are using a PDP-11 RSX-11M/M-PLUS host system. If you are using a VAX/VMS host in VAX-11 RSX compatibility mode, 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. At least one output file must be specified for any given MIB operation.

```
[outmim],[mapfile][,dbgfile] =
                    [pimfile],[inputmim][,stbfile][/switches]
```

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 switch is specified, MIB attempts to open an existing file identified as "outmim" for updating. If the /SM switch is specified and the /KI switch is not, both input and output MIM files are required. (The presence or absence of an outmim specification, along with option switches, 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. A map file can be specified whenever an output or an input MIM file is specified.

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) switch; 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. (The presence or absence of a pimfile specification, along with option switches, determines in part the type of MIB operation to be performed.) If you specify a PIM file, indicating a kernel/process installation, you must also specify at least an output MIM file. This field is invalid if the /EX switch is also specified.

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 switch in the previous MIB step -- or if the /SM switch is specified 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 switch in the corresponding MERGE and RELOC steps. The default file type is .STB. You specify this field only if you are installing symbols in an output DBG file; the dbgfile field must also be specified.

/switches
: Any of the option switches summarized in Table 7-1. All MIB switches are position-independent. Multiple switches can be specified in a list of the following form:

```
/switch1/switch2/...
```

7.4 MIB OPTIONS

This section describes the MIB option switches that are summarized in Table 7-1.

Table 7-1
MIB Switches

Switch	Meaning
/BS:"file-spec"	Installs a bootstrap at the beginning of the memory image file; the bootstrap is copied from the specified file.
/CM	Compacts the physical memory layout in a mapped MIM file, consolidating the free space caused by process deletion. Invalid for an unmapped memory image.
/EX:name:...	Deletes one or more static processes from the memory image.
/GC:name:code	Changes the exception group code of a static process currently 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.
/PR:name:value	Changes the start-up priority of a static process currently installed in the memory image.
/QB:name:block[:...]	Aligns named program sections at specified physical locations in the memory image during a process installation step. (Valid for mapped applications only.)
/SM	Limits the output memory image file to the exact size of the kernel and the currently installed static processes.

7.4.1 Install Bootstrap (/BS)

The /BS:"file-spec" switch installs a bootstrap at the beginning of the memory image file. The file-spec argument identifies the particular 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
DYBOTU.BOT	RX01/RX02 diskette, unmapped target

(MSCP-class disks include the RX50 and RD51 devices.) A MIM file containing a bootstrap must be processed by the COPYB utility prior to its use, as described in Chapter 9. See Sections 3.2.4 and 7.1.3 for a 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 initially 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 top of memory, this requirement should never be a problem either. However, you should be aware of these bootstrap requirements.

7.4.2 Compact Mapped Image (/CM)

The /CM switch compacts the physical memory layout in a mapped memory image; /CM is illegal for an unmapped memory image. Use this option to consolidate available space in the image after you have deleted one or more static processes. When you delete a static process, the space occupied by the process remains as one or more "holes" in the memory image (at least two holes in the ROM/RAM case). The space can be reused only if a static process installed later is small enough to fit. After several deletions, however, free space may become excessively fragmented, causing subsequent MIB operation errors. When you use /CM, MIB consolidates the free space by moving process images so that the free space is contiguous. However, MIB will move only those process images that have not been specially placed in specific physical locations with the /QB switch.

You cannot use the /CM switch to compact physical memory during an operation that installs or deletes a process. Use /CM as a separate step after deleting a process from a mapped image. You must specify both an input .MIM file and an output .MIM file for memory compaction.

7.4.3 Delete Process and/or Symbols (/EX)

The /EX:name[:...] switch deletes one or more static processes from the memory image. The name argument specifies the runtime static-process name, as defined by the first six characters of the Pascal program name or by the DFSPC$ macro in a MACRO-11 implementation. You can specify up to eight process names in the /EX switch.

When you delete a static process, MIB also deletes that process's symbols from the debug symbol file if you specify one in the same command. The general form of MIB command for process deletion only is:

```
outmim[,mapfile] = [,inputmim/SM]/EX:process[:...]
```

The general form of MIB command for both process and symbol deletion is:

```
outmim,[mapfile],dbgfile = [,inputmim/SM]/EX:process[:...]
```

You can also use the /EX switch to delete symbols from the debug file without deleting a static process from the memory image at the same time. To do so, specify only the output DBG file and the /EX switch in a command line of the following form:

```
,,dbgfile = /EX:process[:...]
```

You might want to use this form of command if you had previously deleted a process but forgot to delete its symbols from the debug file.

You cannot use /EX in an operation that installs a process and/or symbols. That is, deletion can be performed only as a separate step from installation. An input MIM file is always optional in a process deletion step; the output MIM file can be modified in place in this case regardless of whether it is "small" or full size. See the /CM switch concerning image compaction in the case of a mapped image. (If an unmapped memory image becomes overly fragmented due to multiple process deletions, the only recourse is to rebuild the image from scratch.)

7.4.4 Exception Group Code (/GC)

The /GC:name:nnn switch changes the exception group code of a specified static process currently 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 name argument identifies the static process to be modified, which can be the process being installed by the current operation or another, previously installed process. The nnn argument 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.

Note that the exception group of a dynamic process cannot be modified by this option.

7.4.5 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 optional DBG, file is created regardless of whether or not a file by the same name already exists. In general, only the /SM and /BS switches are valid or meaningful in combination with /KI.

7.4.6 Process Priority (/PR)

The /PR:name:nnn switch changes the start-up priority of a static process currently installed in the memory image. This option allows you to change process start-up priority without rebuilding the application. Note that the /PR switch 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 name argument identifies the static process to be modified, which can be the process being installed by the current operation or another, previously installed process. The nnn argument specifies the value to which the priority should be changed. This value must be within the range 1 to 255 (1 to 377 octal). (Priorities 248 through 255 are reserved for startup 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.

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 effectively starts up at its specified running priority, which is not affected by the /PR option.)

7.4.7 Align Specified Program Section (/QB)

The /QB switch 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 switch for an unmapped image (the RELOC /QB switch is its equivalent in the unmapped case). The format of the /QB switch 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.) Note that each section specified in the MIB /QB switch 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 switch, effectively a "memory block" number, specifies this multiple. The value is assumed to be expressed in octal unless terminated by a decimal point. For example, to indicate the octal starting address 63000, you specify 630 as the block value (63000/100).

To load an entire static process starting at a desired physical address, specify a start location for the program section .ALST.; this section contains the static process list element and is always 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.)

7.4.8 Small Output Memory Image (/SM)

The /SM switch limits the output memory image file to the exact size of the kernel and the currently installed static processes. If /SM is not specified, 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 causes MIB to create 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 /SM is used 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.

7.5 COMMAND LINE EXAMPLES

The following examples show MIB commands for straightforward image-building operations. More complex examples are given in Section 7.6, which describes RELOC and MIB interaction. Device and directory specifications are omitted from the sample command lines; include those fields in your file specifications as required.

Example 1

```
>[MCR] MIB   or   $ MPMIB
MIB>APPLIC1.MIM,,APPLIC.DBG=KERNEL.PIM,,KERNEL.STB/KI/SM
```

This example creates and initializes the memory image file APPLIC1.MIM and installs the kernel image (KERNEL.PIM) in it. The /KI switch signals MIB that the input .PIM file contains a kernel image rather than a process.

The command also initializes the debug symbol file because an output .DBG file is specified together with the /KI switch. The kernel symbol table, KERNEL.STB, is required input when you create the debug file.

The /SM switch causes a small memory image file to be created, just large enough to contain the kernel. The file will be used as an input file in a subsequent MIB step.

Since all the files in the example above have standard file types, you could omit the file types in the command line and let MIB use the defaults, as follows:

```
MIB>APPLIC1,,APPLIC=KERNEL,,KERNEL/KI
```

Example 2

```
MIB>APPLIC2.MIM=PROCESSA.PIM,APPLIC1.MIM
```

or

```
MIB>APPLIC2=PROCESSA,APPLIC1
```

This command creates a new output .MIM file (APPLIC2.MIM) from the input .MIM file (APPLIC1.MIM) and installs the process image (PROCESSA.PIM) in it. Since APPLIC1.MIM was created in Example 1 as a small image file, it cannot be specified as an existing output file and updated in place as shown in Example 3. (Regardless of whether the existing MIM file is small or full size, however, you can always create a new output MIM file by specifying the existing file as an input. Note that this mode of MIB usage preserves the original MIM file, APPLIC1, as a backup or for possible reuse in a rebuild cycle.) The output file name can be the same as the input file name, of course, placing reliance on version number differences to distinguish between several "generations" of the MIM file.

Since the command line does not include the /SM switch -- mainly for the purposes of Example 3 -- the file APPLIC2 that MIB creates is full size, as large as required by the kernel's memory configuration table. Information in that table comes from the MEMORY macro(s) that you specified in the system configuration file at the beginning of the kernel build phase (see Chapters 3 and 4). If /SM were included, APPLIC2.MIM would again be a small image file, containing only the installed components from APPLIC1.MIM -- just the kernel in this case -- plus the new process from PROCESSA.PIM.

Example 3

```
MIB>APPLIC2.MIM,,APPLIC.DBG=PROCESSB.PIM,,PROCESSB.STB
```
or
```
MIB>APPLIC2,,APPLIC=PROCESSB,,PROCESSB
```

This command installs the process image (PROCESSB.PIM) in the full-size memory image file APPLIC2.MIM. Because the command does not specify an input memory image file, MIB does not create a new one. Instead, MIB opens the existing memory image file for updating and installs the process image in it, modifying the current file in place.

The command also installs the new process's debug symbols in the debug symbol file initialized in Example 1. The existing debug symbol file must be specified on the output side of the command line, and the symbol table (STB) file created by RELOC for the process in question must be specified as an input.

You will get an error if you have been using the /SM switch and then try to update a small .MIM file in place.

Example 4

```
MIB>APPLIC2.MIM,APPLIC2.MAP,APPLIC.DBG=PROCESSB.PIM,,PROCESSB.STB
```
or
```
MIB>APPLIC2,APPLIC2,APPLIC=PROCESSB,,PROCESSB
```

This command line is the same as the one in Example 3 except that it also produces the memory map file APPLIC2.MAP.

Example 5

You can perform the debug symbol installation operation separately from process image installation if desired. You might want to do so if, for example, you previously encountered an error when installing debug symbols in a combined operation that successfully installed a process.

```
MIB>,,APPLIC.DBG=,,PROCESSX.STB
```
or
```
MIB>,,APPLIC=,,PROCESSX
```

This command installs the debug symbol information contained in file PROCESSX.STB in the debug symbol file initialized in Example 1 and updated in Example 3. The debug symbols need not be installed in the DBG file in the same order that the corresponding processes are installed in the MIM file.

The following command line requests the converse operation, debug symbol deletion:

```
MIB>,,APPLIC.DBG=/EX:PROCSX
```

In this command, the /EX switch specifies the name of the static process whose symbols are to be deleted -- which is not necessarily the same as the name of the STB file from which they were installed. (In this case, presumably the debug symbols in question were installed from file PROCESSX.STB.)

7.6 HOW RELOC AND MIB INTERACT

The way in which MIB installs a process in the memory image is determined partly by the type of memory image being constructed and partly by the way the process was relocated in the corresponding RELOC step.

The following cases illustrate relocation and installation of a static process under different circumstances. To simplify the examples, no map, STB, or debug files are included in the command formats.

Case 1: Use First Available Memory in Image

UNMAPPED -- RAM-only or ROM/RAM

1. RELOC command line:

   REL>pimfile=mobfile,mimfile

   RELOC inspects the existing MIM file in order 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 for first-available ROM and RAM respectively.

2. You invoke MIB with the same input MIM file as was specified to RELOC:

   MIB>outmim=pimfile,inputmim/SM

   MIB installs the static process in the memory image at the physical starting address(es) fixed by RELOC.

MAPPED -- RAM-only

1. RELOC command line:

   REL>pimfile=mobfile

   RELOC assigns virtual addresses to the read-only segment starting at Ø 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.

2. MIB command line:

   MIB>outmim=pimfile,inputmim/SM

   MIB installs the read-only and read/write segments contiguously in first available memory and sets up the process's memory management register (PAR) values accordingly.

MAPPED -- ROM/RAM

1. RELOC command line:

   REL>pimfile=mobfile/AL

RELOC assigns virtual addresses to the read-only segment starting at Ø and to the read/write segment starting at the next available 4K-word boundary, because of the /AL switch. RELOC produces the virtually relocated .PIM file.

2. 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

UNMAPPED -- RAM-only or ROM/RAM

1. RELOC command line:

REL>pimfile=mobfile/RO:phys-addr[/RW:phys-addr]

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 switch for a RAM-only image, the read/write segment is allocated contiguously with the read-only segment. You must specify the /RW switch for a ROM/RAM image, of course, to origin the read/write segment at some location in RAM. RELOC produces the physically relocated .PIM output file.

2. 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.

MAPPED -- RAM-only

1. RELOC command line:

REL>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 specified values must reflect 4K-word address boundaries, of course, 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 Ø through 7.) If you do not specify the /RW switch, 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 handler (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:

REL>DLHANDLR=DLHANDLR/RO:40000/RW:60000

This use of /RO and /RW forces the special PAR 2 and 3 mapping required for a handler's code and impure data segments.

2. MIB command line:

MIB>outmim=pimfile,inputmim/SM

MIB places the process image in first available physical memory, treating the read-only and read/write segments as separately allocatable units if the read/write segment is not virtually continuous 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 for some reason.

In any case, MIB sets up the process's PAR values to reflect the process's placement in the memory image.

MAPPED -- ROM/RAM

1. RELOC command line:

REL>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 specified values must reflect 4K-word virtual address boundaries, as described above for a mapped RAM-only image.) If you do not specify the /RO switch, the read-only segment is origined at 0 by default. You must specify the /RW switch (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 handler (driver mapped) process, specifying /RO:40000 and /RW:60000 to achieve the required PAR 2 and PAR 3 mapping.

2. 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 switch and, for a mapped image, possibly use of the MIB /QB switch also. See Sections 6.5.7 and 7.4.7 for a detailed description of the RELOC and MIB /QB switches, respectively.

Use of the /QB switch is not ordinarily required. You would need to use /QB only if your process has special "internal" location and/or mapping dependencies that cannot be satisfied through use of the /RO and /RW or /AL switches, which cover all commonly encountered relocation requirements by implicit reference to the first RO section and the first RW section of a program. Effectively, those switches are convenient, specialized forms of the /QB switch. Consider the following equivalences between the "implicit" /RO, /RW, and /AL switches and the RELOC /QB switch:

- /RO:mmmmm is equivalent to specifying /QB:lst-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 Section 6.5.4.)

- /RW:nnnnn is equivalent to specifying /QB:lst-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:lst-RW-section:xxxxx, where "lst-RW-section" is again .CDAT. or .MDAT., and xxxxx is the first unused 4K-word boundary address following the RO section addresses. (/AL is intended specifically for mapped usage.)

Thus, you need use the /QB switch only for specifying p-sects other than the first within the read-only or read/write segments.

UNMAPPED -- RAM-only or ROM/RAM

1. RELOC command line:

   REL>pimfile=mobfile/RO:mmm/QB:psect-xxx:nnn:psect-yyy:ppp

   or, equivalently:

   REL>pimfile=mobfile/QB:.ALST.:mmm:psect-xxx:nnn:psect-yyy:ppp

   (The /QB switch 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 switch must be the first RW section, to start that section at some point in RAM.

2. 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.

MAPPED -- RAM-only or ROM/RAM; locate named p-sects at specific virtual addresses

1. RELOC command line:

REL>pimfile=mobfile[/RO:mmm]/QB:psect-xxx:nnn:psect-yyy:ppp

or, equivalently:

REL>pimfile=mobfile/QB[:.ALST.:mmm]:psect-xxx:nnn -<RET>
REL>:psect-yyy:ppp

(The /QB switch 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, 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 switch 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.

2. MIB command line:

```
MIB>outmim=pimfile,inputmim/SM
```

MIB places the process image in 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 of 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 for some reason. In a ROM/RAM memory image, all read/write segments will be physically separate from the read-only segments, of course, but the segments within each group will be 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

1. RELOC command line:

```
REL>pimfile=mobfile[/RO:mmm]/QB:psect-xxx:nnn:psect-yyy:ppp
```

or, equivalently:

```
REL>pimfile=mobfile/QB[:.ALST.:mmm]:psect-xxx:nnn -<RET>
REL>:psect-yyy:ppp
```

(Aga..., the /QB switch 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.

2. MIB command line:

```
MIB>outmim=pimfile,inputmim/QB:psect-name:phys-blk-num -<RET>
MIB>: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 7.4.7. As a more specific example, the following command line fixes the physical start address for the read/write program section .CDAT. at -- for simplicity's sake -- 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 switch makes no difference.

MIB places the virtual program segment(s) that precede the segment beginning with section .CDAT. in first available physical memory. In this example, that available memory area must be below location 100000. 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 switch, on a next-available basis. (In this example, if MIB finds location 100000 already used when it attempts to allocate the .CDAT. section, it ignores the /QB switch, allocates next-available space, and issues warning messages to that effect.)

MIB sets up the process's PAR values to reflect the placement of the process segments in the memory image.

7.7 MIB MEMORY MAP

If you specify mapfile in the command line, MIB creates a memory image map showing the following:

1. MIB version number
2. Date and time of operation
3. Target memory type
4. Boot device, if any
5. Memory layout -- physical memory present (base, size, and type) and available ROM and/or RAM in image (base and size)
6. Kernel information -- physical starting address and kernel segments (base and size)
7. Static process information -- process name and priority, process type and group, physical starting and termination addresses, stack address, and segment starting addresses and sizes

For unmapped images, starting addresses and sizes are direct byte values in octal. For mapped images, starting addresses and sizes are given indirectly as octal values representing units of 100(octal)-byte memory blocks. To obtain full byte addresses and sizes, multiply the values given in the map by 100.

Figure 7-5 shows a sample MIB map.

Unmapped:

```
(1) (2) MICROPOWER MIB V01.00   Application Map          Thu 20-Jan-82 01:46:42
(3) (4) MIM filename -- DK :EXAMPU.MIM Unmapped Ram-Only System Boot device: DY

(5) Physical Memory Segments

        Start   Size    Type
        000000  150000  Ram

    Available Ram

        Start   Size
        045246  112532


(6) Kernel Segments

    Segment Start: 000000 Size: 020724
    Kernel Start Address: 011506

(7) Static Process Information

    Name  Priority      Type     Cntxsw    Group      Term Adj. Stack  Start

    $XLADR  250     Drv-Access              1          022632   026450 021366
            Segment Start: 020724 Size: 035526

    EXAMPL  255     General    NCX          1          034730   043506 033704
            Context Switch Location: 045170 Initial Value: 043510
            Segment Start: 026452 Size: 016574
```

(Addresses and sizes are byte values.)

Mapped:

```
(1) MICROPOWER MIB V01.00   Application Map
(3) (4) MIM filename -- DK :EXAMP4.MIM Mapped Ram-Only System   Boot device: DY

(5) Physical Memory Segments

        Start   Size    Type
        000000  002000  Ram

    Available Ram

        Start   Size
        000475  001303


(6) Kernel Segments

    Segment Start: 000000 Size: 000166
    Segment Start: 000166 Size: 000044
    Kernel Start Address: 012450

(7) Static Process Information

    Name  Priority      Type     Cntxsw    Group      Term Adj. Stack  Start

    $XLADR  250     Drv-Access              1          041702   060530 040442
            Segment Start: 000232 Size: 000047
            Segment Start: 000301 Size: 000006
            Segments are movable

    EXAMPL  255     General    NCX          1          006256   015034 005232
            Context Switch Location: 016516 Initial Value: 015036
            Segment Start: 000307 Size: 000166
            Segments are movable
```

(Multiply all addresses and sizes by 100(octal) to get byte values.)

Figure 7-5 MIB Memory Map

CHAPTER 8

METHODS OF APPLICATION LOADING

To execute the 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, RLØ1/RLØ2 or RD51 disk, or RXØ2 or RX5Ø 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.

8.1 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, 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 Debug 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 via 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. Also, the host-side line speed and other characteristics must be set, and the logical-device TD: assigned, as described for debugging (see Chapter 1Ø).

8.2 BOOTSTRAPPING THE APPLICATION FROM A STORAGE DEVICE

If both the host- and target-system hardware configurations include a mass-storage device of the same type -- for example, DECtape II or RXØ2 diskette -- you can prepare a bootable volume on the host system and bootstrap the application directly from that volume on the target.

The following steps are necessary to load the application in that 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, via MPBUILD, or as a separate MIB operation, using the /BS option of the MIB utility program. (See Section 3.2.4.2 or Chapter 7.)

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 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 9 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.

8.3 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-11 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. Those 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 attempting 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.

CHAPTER 9

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 describes:

- COPYB's functions
- The invocation and use of COPYB

9.1 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. Likewise, 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 RSX-11 FLX 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 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 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.

9.2 INVOKING COPYB

For a PDP-11 RSX-11M/M-PLUS 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 VAX-11 RSX (Compatibility Mode) 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 either 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 brotable. That implies a prior use of the FLX 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 COPYB command. The first example includes a sample FLX operation.

Example 1

```
>[MCR] FLX    or    $ MCR FLX
FLX>DY0:/RT/IM=APPL5.MIM
FLX><CTRL/Z>

>[MCR] CPB    or    $ MPCOPYB
CPB>DY0:APPL5.MIM
```

The FLX command line transfers the Files-11 APPL5.MIM file in your default device/directory to RX02 diskette mounted in DY drive unit 0. The /RT switch requests version of output to RT-11 file format. (The command assumes that the output volume is already in RT-11 format.) The /IM switch requests an image-mode copy operation; that is, the file content is to be copied with no internal format modification.

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.)

Note that 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 will truncate 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.

Example 2

```
>[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 switch 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.

CHAPTER 10

HOST/TARGET LINE SETUP FOR DEBUGGING

Use of the PASDBG symbolic debugger is described in the MicroPower/Pascal Debugger User's Guide. (Each host system version of PASDBG -- RSX, VMS, and RT -- implements the same set of debugging commands.) Host and target hardware configuration requirements for debugging are described in the MicroPower/Pascal installation guide for your host system.

This chapter describes the software setup requirements for the host-system terminal line to be used by PASDBG as a host/target communication line. The line setup requirements are determined by the host operating system and differ somewhat for PASDBG-RSX and PASDBG-VMS. In both cases, however, you must assign the logical device name TD: to the line connected to your target system before invoking PASDBG, and the line must be set to a speed matching that of the target's console terminal port.

10.1 HOST/TARGET TERMINAL LINE REQUIREMENTS FOR PASDBG-RSX

PASDBG-RSX uses the standard RSX-11 terminal driver to handle the serial line through which it down-line loads and communicates with the target system. Therefore, any host system terminal line (TTnn:) can be connected to and serve as the communications link with your target system, provided that line has certain software-settable characteristics. The requirements are as follows:

- You must assign the logical name TD: to the line in question before invoking PASDBG.
- The send/receive speed of the line must match that of the target's console terminal port.
- As a precautionary measure, the line should be set to SLAVE mode before it is physically connected to the target system. The SLAVE setting prevents the host system from reacting to possible, but unlikely, line transients when power is applied to the target.

You may also want to allocate the line to yourself, to prevent another user from inadvertently accessing your target system between your debugging sessions.

Note that although you can freely allocate and assign a logical name to any terminal line, you must be a privileged user in order to set the speed or any other characteristic of a line other than the one you are logged in on.

10.1.1 Assigning and Allocating the Terminal Line

When invoked, PASDBG uses the logical name TD: to identify your host/target communication line. Before invoking PASDBG, therefore, you must associate that logical name with the line connected to the target you wish to load. If your host/target line has the physical device name TT12:, for example, you assign the logical name as follows:

```
>[MCR] ASN TT12:=TD:
```
or
```
>[DCL] ASSIGN TT12: TD:
```

No other user will be able to access that line while you are running PASDBG. When you exit PASDBG, however, another user would be able to access the line and, consequently, the target attached to it. If you do not wish to share the use of a given line and/or target system, you can allocate the line to yourself, as follows:

```
>ALL TT12:
```

The allocation protects the line while you are logged in. Use the DEALLOCATE command to release the line if desired.

10.1.2 Setting and Showing Line Speed

The speed, or baud rate, of a host/target terminal line must match the speed setting configured for the target's console terminal port, to which the line is attached. You can inspect the current speed setting and other characteristics of a line -- TT7:, for example -- as follows:

```
>[MCR] DEV TT7:
```
or
```
>[DCL] SHOW TERMINAL TT7:
```

To set line speed, you use the following privileged SET command or have it done for you by a privileged user:

```
>[MCR] SET /SPEED=TT7:nnnn:nnnn
```
or
```
>[DCL] SET TERMINAL TT7:/SPEED=(nnnn,nnnn)
```

where nnnn is the desired baud rate.

10.1.3 Precautionary SLAVE Setting

To safeguard the operating system from possibly "confusing" line transients, the host/target line can be set to SLAVE mode before the line is connected to the target, or at least before the target is turned on. This prevents the host system from responding to any spurious signals that may occur on the line during the target power-up sequence, before PASDBG is invoked. (PASDBG sets the line to SLAVE mode if necessary when you initiate a debugging session.)

The MCR or DCL command for setting SLAVE mode is as follows:

```
>[MCR] SET /SLAVE=TT7:
```

or

```
>[DCL] SET TERMINAL TT7:/SLAVE
```

You must be a privileged user to set characteristics for any line other than your "login" line, TI:. Since the terminal lines to be used as host/target lines will likely be dedicated to that purpose, the necessary commands for those lines can probably be placed in the system start-up file.

10.2 HOST/TARGET TERMINAL LINE REQUIREMENTS FOR PASDBG-VMS

PASDBG-VMS uses the standard VMS terminal driver to handle the serial line through which it down-line loads and communicates with the target system. Therefore, any host system terminal line (TTxn:) can be connected to and serve as the communications link with your target system, provided that line has certain software-settable characteristics. The requirements are as follows:

- You must assign the logical name TD: to the line in question before invoking PASDBG.
- The line's protection code must be set so that you have read/write access to the line. (Default device protection may not permit you such access, and the setting of device protections requires OPER privilege.)
- The send/receive speed of the line must match that of the target's console terminal port.

You may also want to allocate the line to yourself, to prevent another authorized user from inadvertently accessing your target system between debugging sessions. Note that although you can freely assign a logical name to any terminal line, you must have read/write access to a given line before you can SHOW or SET its speed, allocate it, or use it for debugging.

10.2.1 Assigning and Allocating the Terminal Line

When invoked, PASDBG uses the logical name TD: to identify your host/target communication line. Before calling PASDBG, therefore, you must associate that logical name with the line connected to the target you wish to load. If your host/target line has the physical device name is TTC5:, for example, you assign the logical name as follows:

```
$ ASSIGN TTC5: TD:
```

No other user will be able to access that line while you are running PASDBG. When you exit PASDBG, however, another authorized user would be able to access the line and, consequently, the target attached to it. If you do not wish to share the use of a given line and/or target system, you can allocate the line to yourself, as follows:

```
$ ALLOCATE TTC5:
```

Or, you can allocate and assign the logical name in a single operation:

```
$ ALLOCATE TTC5: TD:
```

Of course, the allocation protects the line only while you are logged in. (An attempt to allocate a line to which you lack access results in a "no privilege" error return.) Use the DEALLOCATE command to release the line.

10.2.2 Line Protection Codes

To use PASDBG or to SET or SHOW line speed, you must have read/write access to the terminal line you are going to use. Under VMS V3.0 and later, you normally have no effective access to any terminal line other than the one through which you log in. Therefore, you -- if you have sufficient privilege -- an operator, or the system manager must change the default protection code of the line(s) you are interested in to something useful, as shown in the following examples:

```
$ SET PROT/DEV=(W:RWED) TTxn:
```

This protection will let anyone use the line.

```
$ SET PROT/DEV/OWNER=[xxx,yyy]/PROT=(G:RWED) TTxn:
```

This protection will let anyone in the owner's group xxx use the line.

```
$ SET PROT/DEV/OWNER=[xxx,yyy]/PROT=(O:RWED) TTxn:
```

This protection will let only the owner [xxx,yyy] use the line.

To set line protection, you must have OPER privilege. Since the terminal lines to be used as host/target lines will likely be dedicated to that purpose, the protection definitions probably should be placed in the system start-up file, SYS$MANAGER:SYSTARTUP.COM.

10.2.3 Setting and Showing Line Speed

The speed, or baud rate, of a host/target terminal line must match the speed setting of the target's console terminal port. You can inspect the current speed setting of a line to which you have read/write access -- TTC5:, for example -- as follows:

```
$ SHOW TERMINAL TTC5:
```

To set line speed, you must first allocate the line and then use the SET command, as follows:

```
$ ALLOCATE TTC5:
  TTC5: allocated
$ SET TERMINAL/SPEED=nnnn TTC5:
```

Note that if you do not allocate the line before attempting to set its speed, the speed setting will fail, but you will not receive any error message to that effect.

10.3 LINE-RELATED ERRORS

PASDBG produces the following error message concerning problems with the host/target communication line:

?PASDBG-E-NOTD, Unable to access logical device TD:

This message indicates one of three possible conditions:

1. You have not assigned the logical device name TD:.

2. The line assigned as TD: is allocated to and/or in use by another user.

3. Under VAX/VMS, you do not have read/write access to the line assigned as TD:.

APPENDIX A

THE MICROPOWER/PASCAL FILE SYSTEM

If your Pascal source code contains I/O statements, such as the OPEN procedure, that manipulate files on mass-storage devices, or if your MACRO-11 application manipulates files via QIO$ macros or the equivalent, you must build both runtime and object-time elements of the file system into your application. Those elements consist of the Directory Service Process (DSP) and a library of file system support routines (FSLIB.OLB or MACFS.OLB). (See Chapter 9 of the MicroPower/Pascal Language Guide and Chapter 5 of the MicroPower/Pascal Runtime Services Manual.)

A.1 THE DSP PROCESS

The DSP is a DIGITAL-supplied system process that performs file directory operations for files on mass-storage devices. The directory format supported by the DSP is compatible with that of the RT-11 file system. The DSP process must be included in your application image if any block-structured storage device, such as a disk or TU58, is utilized.

The four DSP object libraries are DSPNHD, DSPFPP, DSPEIS, and DSPFIS, corresponding to the four possible target-hardware instruction sets. You modify the prefix file DSPPFX.MAC, assemble it, and then merge DSPPFX with the appropriate DSP and OTS libraries to produce the specific DSP process your application requires. (Note that although the DSP prefix file is written in MACRO, the DSP object modules contained in the DSPxxx libraries were implemented in Pascal.) Section A.2 describes the prefix file DSPPFX.MAC. The form of MERGE command line needed to create a DSP static process is as follows; xxx should be replaced by NHD, FPP, FIS, or EIS:

```
>[MCR] MRG
```

or

```
$ MPMERGE

MRG>YORDSP=DSPPFX.OBJ,KERNEL.STB,mpp-lib:DSPxxx/LB,LIBxxx/LB
```

The resulting YORDSP.MOB file contains the merged DSP process for your application. DSPPFX.OBJ is the DSP prefix file object module, DSPxxx is the appropriate DSP library for your hardware configuration, LIBxxx is the appropriate OTS library for your hardware configuration, and KERNEL.STB is the symbol table file that you created when you built your application's kernel.

You can build and install the DSP "automatically" as part of a build cycle generated by the the MPBUILD procedure, or you can use MERGE, RELOC, and MIB in a "manual" build cycle as described in Chapter 3. The DSP will work in both mapped and unmapped applications.

A.2 THE DSPPFX PREFIX FILE

You use this prefix file to tailor the DSP process for the needs of your application; that is, you can omit unneeded DSP functionality. Figure A-1 shows the text of this file as supplied. Numbered lines in the figure correspond to the following optional functions:

1. OPEN an existing file for reading or update
2. OPEN a new file
3. RENAME a file
4. DELETE a file
5. CLOSE a file (and make it permanent, if new)
6. PURGE an open file (delete file from the directory, if new)
7. OPEN a directory to be read
8. INITIALIZE the directory for a device
9. PROTECT a file in a directory

```
.nlist
        .enabl  LC
.list
.title   DSPPFX - RT-11 DSP (RTDSP) prefix module
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED OR
; COPIED ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (c) 1982, 1983 BY DIGITAL EQUIPMENT CORPORATION. ALL
; RIGHTS RESERVED.
;
        .mcall  dspfn$

;+
; DSPPFX
;
; DSP functions are selected from the following list of eight.
; To include a function, specify include=YES. To omit the
; function from the installed DSP, specify include=NO.
;
;-

    (1) dspfn$  option=LOOKUP, include=YES
(2) → dspfn$  option=ENTER,  include=YES
    (3) dspfn$  option=RENAME, include=YES
(4) → dspfn$  option=DELETE, include=YES
    (5) dspfn$  option=CLOSE,  include=YES
(6) → dspfn$  option=PURGE,  include=YES
    (7) dspfn$  option=READDIR,include=YES
(8) → dspfn$  option=INITDIR,include=YES
    (9) dspfn$  option=PROTECT,include=YES

        .end
```

Figure A-1 The DSPPFX Prefix File

After any desired editing, assemble the prefix file together with the file system macro library FSMAC.MLB as follow:

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

Make appropriate substitutions for "mpp-lib:" and "user-dir:" as required for your host system and the location of DSPPFX.MAC.

A.3 PASCAL FILE SYSTEM SUPPORT LIBRARY

Pascal-implemented user processes that perform file system operations must be merged with the file system support library FSLIB.OLB in addition to the standard OTS library LIBxxx.OLB. The form of MERGE command line required to build a user static process in this case is as follows, where xxx is replaced by NHD, FPP, FIS, or EIS:

```
MRG>YORPROC=YORPROC.OBJ,KERNEL.STB,mpp-lib:FSLIB/LB,LIBxxx/LB
```

This form of MERGE command line would be used in a user-process build step of a "manually" performed build cycle, as described in Chapter 3. Pascal file system support can be automatically and conveniently included, however, where required in a build cycle generated by the MPBUILD procedure.

A.4 MACRO-11 FILE SYSTEM SUPPORT LIBRARIES

A user process that is implemented in MACRO-11 and that employs the QIO$ and/or PARSE$ macros must be assembled with the FSMAC.MLB macro library in addition to COMM.MLB or COMU.MLB. That process must also be merged with the MACFS.OLB object library, as shown in the following sample command sequences:

- Assemble user-process source module with FSMAC.MLB to provide file-system macro definitions:

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

- Merge user-process object module(s) with the MACRO-11 file system support library MACFS.OLB:

  ```
  >[MCR] MRG
  MRG>PROCESSX=PROCESSX.OBJ,KERNEL.STB,mpp-lib:MACFS.OLB/LB
  ```

These forms of assembly and merge command lines would be used in a user-process build step of a "manually" performed build cycle, as described in Chapter 3. MACRO-11 file system support can be automatically and conveniently included, however, where required in a build cycle generated by the MPBUILD procedure.

APPENDIX B

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 3 of this manual for a general understanding of the application build cycle and Chapter 4 for information about system-configuration and prefix-module files.

B.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 "bits and pieces" of your responses. (MPBUILD produces as its direct output a command file that in turn invokes the requested build cycle.)

More specifically, MPBUILD allows for the following possibilities:

- A total build cycle in which you construct a complete application image "from scratch," 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 outputs of both cycles are the corresponding symbol table and debug files, as applicable.)

- A partial build cycle in which you build just the kernel and, optionally, some system processes -- device handlers and file system process (DSP). 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 in which you build only the user processes, using an existing kernel image file as input. (Optionally, you can add system processes to the kernel image file at the same time.) The primary output is an application image file.

- A partial build cycle in which you 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 B.3 gives directions for performing this form of build operation, which involves special interpretation of certain MPBUILD questions.)

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. (Those capabilities are the ones described in Chapter 3 of this manual.) For some applications, however, you may need to use the individual MERGE, RELOC, and MIB utilities 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. Alternatively, you may be able to modify certain commands in the command file generated by MPBUILD to overcome a specific limitation.

The significant limitations of MPBUILD are the following:

1. You cannot specify a user-supplied macro library for a MACRO-11 assembly step. You can, however, use the MACRO-11 assembler to "preassemble" 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.)

2. You cannot build a MACRO-11 user process for debugging purposes -- that is, with a name known to PASDBG -- if the static process module does not contain a .TITLE statement that specifies the static process name defined by the PID parameter of the DFSPC$ macro call. See the explanation of question 32 in the dialog description, Section B.2.3.5. You can use MERGE, RELOC, and MIB separately for the process in question, specifying the /NM (module name) option in the RELOC step, after using MPBUILD to install all other processes in the application image. Alternatively, you can edit the command file generated by MPBUILD to add that RELOC option switch, which is described in Chapter 6.

3. You cannot build a process that needs special, user-controlled address relocation, such as for a target system with an unmapped, interspersed ROM/RAM/ROM memory configuration -- an unusual situation. You must use MERGE, RELOC, and MIB individually in such cases.

4. In general, you cannot explicitly request any MACRO, MERGE, RELOC, or MIB options. (MPBUILD automatically generates all commonly required option switches, however, as needed.)

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

B.2 THE MPBUILD DIALOG

On a PDP-11 RSX-11M/M-PLUS development system, you invoke the MPBUILD MCR command procedure as follow:

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

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

On a VAX-11 RSX-11 compatibility-mode development system, you can invoke MPBUILD.COM simply by entering its name at system level, as follow:

```
$ 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.

In both cases, MPBUILD issues an identifying message and then initiates its system/user dialog. The format of the dialog produced by the two versions of MPBUILD differs in minor respects; such incidental, system-dependent format differences are mostly ignored in the following description.

B.2.1 Dialog Structure

The MPBUILD dialog consists of a highly variable sequence of questions. That is, many of the questions are conditional, either asked or bypassed depending on your responses to prior questions. For descriptive purposes, the dialog is divisible into five logical sections:

1. Kernel and global information
2. System processes
3. Pascal user processes
4. MACRO user processes
5. Optional bootstrap

The questions in the kernel and global-information 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/handler 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: mapped or unmapped target, ROM/RAM or

RAM-only target, hardware instruction options, inclusion of debugging support, and so on

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 handlers that are to be built and installed in the kernel image file and determines whether the Directory Service Process (DSP) is to be installed.

The Pascal user-process section requests file specifications for the Pascal PROGRAM and MODULE files, 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 MACRO user-process section requests file specifications for the MACRO-11 program modules, 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 determines whether a device bootstrap is to be installed in the application image and, if so, for which device. (The section is entered only for an all-RAM application that does not include PASDBG support.)

B.2.2 Options, Usage Rules, and Defaults

Many MPBUILD questions require a file specification as a response. The option switches /LBR, /LIST, /OBJ, and /MAP can be appended to many user input file specifications, as appropriate to the context. (The detailed dialog description in Section B.2.3 indicates where the option switches are applicable.)

The meaning and effects of the switches are as follows:

- The /LBR switch indicates that a given input file is a user-supplied object library to be used in a process merge step. If /LBR is specifed, the file type default is OLB instead of PAS or MAC. Note that it is not sufficient, or even necessary, to specify file type OLB in order to indicate an object library when a source file is assumed by MPBUILD. If /LBR is specified, no other file option is valid.

- The /LIST switch 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 switches are mutually exclusive.)

- The /OBJ switch 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. The effect of this option is to bypass 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 or MAC. Note that it is not sufficient, or even necessary, to specify file type OBJ in order to indicate an object file when a source file is assumed by MPBUILD. The /OBJ switch must be used to suppress the implied compilation or assembly, regardless of file type specification.

- The /MAP switch causes a relocation map to be produced by the RELOC utility. RELOC maps can be generated for the kernel and for user processes, depending on where the /MAP switch is used. The .MAP file is generated in the user's default directory. RELOC maps are primarily of use when debugging a process implemented in MACRO-11.

The /LIST switch name can be abbreviated to /LIS.

General usage rules are as follows:

1. In response to any question, you may request an explanation of the question by entering a question mark (?).

2. Y or N is sufficient for an affirmative or a negative response, respectively.

3. You can give your responses in either lower- or uppercase; case is not significant. Literal responses are indicated in the dialog description in uppercase, however, for clarity.

4. The MPBUILD command procedure does not verify the existence of any input file. Therefore, to preclude any nonexistent-file errors during the build cycle, you must ensure that all input files exist as specified in the dialog before the build cycle is executed. (You can verify all file specifications through inspection of the generated command file.)

Many MPBUILD questions have a default response, which is indicated within square brackets following the question text. (The format varies slightly, depending on your host system.) For example:

```
     Do you wish to build a kernel? [yes]:              (VAX/VMS dialog)
or
     Do you wish to build a kernel? [S D:"yes"]:        (RSX-11 dialog)
```

The default response to this question is indicated as the affirmative, 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 in 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.

B.2.3 Dialog Description

The following conventions are used in the description of the MPBUILD dialog presented below:

- Conditional questions are preceded by an "If ... go to question n" statement indicating the condition that causes the question to be bypassed.
- The user's response to question n is signified by (n), where n is a question number -- as in "If (2) was no, ..."
- In descriptions of user responses, a RETURN-only response is denoted by <RET>.

B.2.3.1 **Kernel and Global-Information Section** - The following 17 questions comprise 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]:

The long form of dialog provides an explanation -- similar to the explanatory text presented here -- preceding each question. You might want that explanation the first few times you use the command procedure. If you respond with no, you can still get an explanation of any subsequent question by typing "?".

Response: Y, N, or <RET>

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

You can either build a new kernel/handler image or begin the build cycle with an existing -- previously built -- kernel image file. A response of no implies a partial build cycle -- a new application image file only, consisting of the existing kernel/handlers plus newly built user processes. (When you build only a new application image, you also have the option of extending the existing kernel image file with additional handlers and/or the DSP; see question 4.)

Response: Y, N, or <RET>

Question 3: Kernel memory image file name? :

You must provide a file name, with optional directory or logical device information, for the kernel memory image (.MIM) file and, by implication, for the associated .STB and .DBG files also. A file type or version number is not applicable in the response. (The application image file, if any, is named later.) If (2) was yes, the response will be used to name the new kernel/handler MIM, STB, and DBG files to be created. If (2) was no -- a kernel is not being built -- the response must identify the existing kernel/handler MIM, DBG, and STB files t be used in the build cycle. The existing files must all res' in the same directory.

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

If (2) was yes, go to question 5.
Question 4: Do you wish to add any handlers and/or the DSP? [no]:

You may want to add one or more DIGITAL-supplied system processes to the existing kernel/handler image file before building or rebuilding the application image file.

Response: Y, N, or <RET>

If (2) was no, go to question 9.
Question 5: System config file spec? [default]:

You must identify the system configuration file to be used for building the new kernel. The default is a file in the user's default directory with type MAC and the name specified for the kernel image in (3). If /MAP is appended to the specification, a kernel relocation map is produced.

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

If (5) is object file, go to question 8.
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>

If (6) was no, go to question 8.
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 originally specified file as input -- the modified version if the source was your default directory. (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/handlers? [no]:

You may want to create a kernel image file only (yes) or also create an application image file (perform a total build cycle).

Response: Y, N, or <RET>

If 8 was asked and the response was yes, go to question 10.
Question 9: Application memory image file name? :

You must provide a file name, with optional directory or logical device information, for the application memory image (.MIM) file to be created. The response will also name the corresponding

application debug-symbol (.DBG) file, if one is being created (see question 12). A file type or a version number is not applicable in the response.

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

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

You must provide a file specification -- usually a name only -- for the command file that MPBUILD will generate to perform the build cycle. The default specification, based on previous responses, has the same name as either the kernel or the application image file and has the CMD or COM file type, depending on your host system.

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

Question 11: Mapped image? [no]:

Your target system may or may not include a memory-management -- or mapping -- option (LSI-11/23-type processors). You indicate here whether the memory image is to be built to use the hardware mapping capability if it exists on your target. The response must agree with the specification in the configuration file for "MMU=" in the PROCESSOR macro. (The response establishes the version of the system macro library, kernel library, and driver library to be used in the build cycle, as well as the required form of many build utility commands.)

Response: Y, N, or <RET>

If (11) was yes, go to question 12.
Question 11a: Is the target a KXT11-C? [no]:

You indicate here if your unmapped target system is a KXT11-C peripheral processor. The response must agree with the specification in the configuration file for "TYPE=" in the PROCESSOR macro. (The response determines whether the DRVU or DRVK unmapped driver library is to be used in the system-process build phase.)

Response: Y, N, or <RET>

Question 12: Debug support required? [yes]:

You may or may not want the kernel and/or application to be built with PASDBG symbolic debugging support. The response determines compilation and assembly command options and governs the creation/use of .DBG and user process .STB files in the build cycle. If the response is no, debug files are not created. A yes response implies -- but does not ensure -- that the Debug Service Module (DSM) is included in the kernel. (You must specify "DEBUG=YES" in the SYSTEM macro of the system configuration file to cause the DSM to be included when the kernel is built.)

Response: Y, N, or <RET>

If (12) was yes, go to question 15.
Question 13: Does this system contain any ROM? [no]:

Your target system memory may consist of RAM only or may contain both programmable ROM and RAM. The response determines many build utility options and must reflect the target memory configuration specified in the MEMORY macro(s) of the configuration file used to build the kernel.

Response: Y, N, or <RET>

If (2) or (13) was no or (11) was yes, go to question 15.
Question 14: What is the base of RAM (octal address)? :

You must specify the base address of the first -- low address -- RAM segment in an unmapped ROM/RAM target memory configuration. The RELOC utility uses that value to locate the kernel's initial read/write data section at the proper physical address.

Response: Positive octal integer

Question 15: Instruction set hardware? {NHD,FPP,EIS,FIS} [default]:

You must identify the hardware instruction-set option, if any, to be assumed when compiling and merging a Pascal program. The default is NHD for an unmapped image or EIS for a mapped image. The response determines the /IN:xxx compilation option and the corresponding OTS library (LIBxxx.OLB) to be used when building any Pascal user process. (See Chapter 2.) The response also determines the proper object library (DSPxxx.OLB) to be used when the DSP process is built.

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

If (11) or (12) was yes or (13) was no, go to question 17.
Question 16: LSI-11/2 mode on compilations? [no]:

The /IN:LS2 compilation option is required for LSI-11 or LSI-11/2 ROM applications if the target system has the EIS or FIS hardware option. (See Chapter 2.) Answer yes only if your target is a ROM-based LSI-11 or LSI-11/2 and your response to question 15 was EIS or FIS.

Response: Y, N, or <RET>

B.2.3.2 **System-Process Section** - This section consists of eight questions about device handlers and the DSP (file system) process. The four questions specific to handlers are asked repeatedly in a loop. The section is bypassed if the response to both questions 2 and 4 was no.

The dialog is as follows:

Beginning system-process section.

If (2) and (4) were no, go to question 25.
Question 17: Handler prefix file spec? :

You may supply a file specification for a DIGITAL-supplied device handler prefix file. The file is used to build the corresponding device handler process, which is installed in the kernel/handler image. If the response is <RET> only -- signifying "no more prefix files" -- the dialog passes to question 21.

Response: File specification with a .MAC or .PAS default (see 18), or <RET>. (/OBJ is valid. If it is used, the type default becomes OBJ.)

Question 18: Is this a PASCAL-implemented handler? [no]:

Here you must indicate whether the specified prefix file is for a handler implemented in Pascal. (The Pascal device-handler prefix files are identified in the long-dialog explanation provided for this question.) A yes response causes the prefix module to be compiled rather than assembled, provided that /OBJ was not specified in (17). The response also affects the object libraries used in the corresponding merge step. Thus, the response is significant whether or not /OBJ was specified for the file. If the response is yes and /OBJ was not specified, the default type for question 17 is .PAS; if no, the default is .MAC.

Response: Y, N, or <RET>

If (17) is an object file, return to question 17.

Question 19: Do you wish to modify <file specified in 17>? [no]:

You may want to edit the specified prefix module 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 specified file is used as is.

Response: Y, N, or <RET>

If (19) was no, return to question 17.

Question 20: Satisfied with edit? [yes]:

At this point, upon exit from the editor, you can either proceed with the dialog (yes) or repeat the editing session (no). (This question is asked each time you leave the editor.) If the response is no, you are returned to EDT with the originally specified file as input -- the modified version if the source was your default directory. If yes, the dialog resumes at question 17.

Response: Y, N, or <RET>

Question 21: Install the Directory Service Process (DSP)? [no]:

Here you indicate whether the Directory Service Process (DSP) is to be installed in the kernel MIM file. The DSP constitutes a portion of MicroPower/Pascal file system support. In general, the DSP is needed if the application utilizes any block-structured storage device such as a disk or a TU58 tape unit.

Response: Y, N, or <RET>

If (21) was no, go to question 25.
Question 22: DSP (file system) prefix file spec? [default]:

Provide a file specification for the DIGITAL-supplied Directory Service Process (DSP) prefix file. The file is used to build the process, which is then installed in the kernel/handler image. The default file specification is mpp-lib:DSPPFX.MAC, where the substitution for mpp-lib: is host-system and installation dependent.

Response: File specification; .MAC default. (/OBJ is valid. If it is used, the type default becomes OBJ.)

If (22) is an object file, go to question 25.
Question 23: Do you wish to modify <file specified in 22>? [no]:

You may want to edit the specified prefix module 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 specified file is used as is.

Response: Y, N, or <RET>

If (23) was no, go to question 25.
Question 24: Satisfied with edit? [yes]:

At this point, upon exit from the editor, you can either proceed with the dialog (yes) or repeat the editing session (no). (This question is asked each time you leave the editor.) If the response is no, you are returned to EDT with the originally specified file as input -- the modified version if the source was your default directory. If yes, the dialog resumes at question 25.

Response: Y, N, or <RET>

B.2.3.3 **Beginning of User-Process Build Phase** - The user-process build phase comprises the Pascal and MACRO user-process sections of the dialog. (The 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.

The beginning of the user-process build phase is marked by a single question that pertains to both Pascal and MACRO user processes.

User Process Build Phase.

If (8) was yes, go to question 38.
Question 25: Does any user process require file system support? [no]:

Here you indicate whether any of the user processes to be installed in the application image performs file-oriented I/O and therefore needs to be built with file system support libraries. (The libraries in question are FSLIB.OLB for a Pascal process or FSMAC.MLB and MACFS.OLB for a MACRO process.) See questions 29 and 34 for further details; your response here determines whether those questions are asked for each user process.

Response: Y, N, or <RET>

B.2.3.4 **Pascal User-Process Section** - This section consists of five questions, four of which are asked repeatedly in two loops. The section is bypassed if only a kernel/handler image is being built.

The dialog is as follows:

Beginning Pascal process section.

Question 26: Any Pascal processes? [yes]:

Here you indicate whether any user processes written in Pascal are to be built and installed in the application image file.

Response: Y, N, or <RET>

If (26) was no, go to question 31.
Question 27: User Pascal process file spec? [default, if any]:

Here you supply a file specification for a user-written static process (PROGRAM) implemented in Pascal. The file can be either a source or an object file (/OBJ). The first time this question is asked, a default corresponding to the application-image file name specified in (9) is provided. In repetitions of the question, no default is provided. If the response is <RET> only with no default -- signifying "no more Pascal processes" -- the dialog passes to the next section.

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

If (27) was <RET> only, go to question 31.
Question 28: Additional module or library? :

You may specify an additional source or object MODULE file that must be merged with the static process (PROGRAM) file. You may also specify a user-supplied object library, which must be identified as such by the /LBR switch. (If the process uses the Pascal Analog System Interface to A/D devices or to the real-time clock (KW), the process must be merged with the RHSLIB.OLB library, which you specify as mpp-lib:RHSLIB/LBR.) If you specify a module or a library, the question is repeated. If you respond with <RET> only -- signifying "no more inputs" -- the dialog passes conditionally to question 29.

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

If (25) was no -- no file system support -- go to question 30.
Question 29: Does this process use the file system (OPEN statement)? [no]:

Here you indicate whether the process opens any files or devices and thus needs to be built with file system support. The response is used to determine whether the file-system support library FSLIB.OLB is to be supplied in the MERGE step for this process. If the process performs only terminal I/O -- reads and writes to a terminal via the implicit INPUT/OUTPUT variables -- with no OPENs, you must respond with no.

Response: Y, N, or <RET>

If (11) was no -- unmapped image -- return to question 27.
Question 30: Is this process a device handler? [no]:

You must indicate whether the user process performs interrupt-driven device handling. If so, it must be built with driver/ISR mapping. (Yes implies that you have specified the DRIVER attribute in the PROGRAM heading.) Driver mapping is required in a mapped application for any process that contains an interrupt service routine. The response determines whether special relocation options are to be used in the RELOC step for the current process.

Response: Y, N, or <RET>. The dialog returns to question 27 in all cases.

B.2.3.5 **MACRO User-Process Section** - This section consists of five questions, four of which are asked repeatedly in two loops. The section is bypassed if only a kernel/handler image is being built.

The dialog is as follows:

Beginning MACRO process section.

Question 31: Any MACRO processes? [no]:

Here you indicate whether any user processes written in MACRO-11 are to be built and installed in the application image file.

Response: Y, N, or <RET>

If (31) was no, go to question 36.
Question 32: User MACRO process file spec? [default, if any]:

Here you supply a file specification for a static-process module implemented in MACRO-11. The file can be either a source or an object file (/OBJ). In order to provide an "external" static process name to PASDBG for debugging purposes, the module specified here must contain the DFSPC$ macro call and a .TITLE statement matching the PID parameter of DFSPC$. The first time this question is asked, a default corresponding to the application-image file name specified in (9) is provided. In repetitions of the question, no default is provided. If the response is <RET> only with no default -- signifying "no more MACRO processes" -- the dialog passes to the next section.

Response: File specification; .MAC default. (Either /LIST or /OBJ, and /MAP are valid.) You may specify /MAP either here or in response to question 33, but it need be specified only once for each static process.

If (32) was <RET> only, go to question 36.

Question 33: Additional module or library? :

You may specify an additional source or object module file that must be merged with the static process module. You may also specify a user-supplied object library, which must be identified as such by the /LBR switch. If you specify a module or library, the question is repeated. If you respond with <RET> only -- signifying "no more inputs" -- the dialog passes conditionally to question 34.

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

If (25) was no -- no file system support -- go to question 35.

Question 34: Does this process use the file system (QIO$)? [no]:

Here you indicate whether the process executes any QIO$ or $PARSE macro calls and thus needs to be built with file system support. The response is used to determine whether the file-system support library MACFS.OLB is to be supplied in the MERGE step for this process.

Response: Y, N, or <RET>

NOTE

If (25) was yes -- include file system support -- the file-system macro library FSMAC.MLB is automatically included in the assembly step for every MACRO-11 user process.

If (11) was no -- unmapped image -- return to question 32.

Question 35: Is this process a device handler? [no]:

You must indicate whether the user process performs interrupt-driven device handling. If so, it must be built with driver/ISR mapping. (Yes implies that you have specified the driver mapping type, symbol PT.DRV, in the DFSPC$ macro call.) Driver mapping is required in a mapped application for any process that contains an interrupt service routine. The response determines whether special relocation options are to be used in the RELOC step for the current process.

Response: Y, N, or <RET>. The dialog returns to question 32 in all cases.

B.2.3.6 **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:

If (8), (12), or (13) was yes, the dialog ends.
Question 36: 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 using PASDBG to down-line "load and go" with the LOAD/EXIT command.

Response: Y, N, or <RET>

If (36) was no, the dialog ends.
Question 37: Bootstrap device? {DY,DD,DL,DU}:

> Here you supply the bootstrap device name -- DY for an RXØ2 device, DD for a TU58 device, DL for an RLØ2 device, or DU for a RX5Ø or RD51 device. The response is used in constructing the appropriate bootstrap file name -- DYBOTM.BOT, for example.

Response: DY, DD, DL, or DU

B.2.3.7 **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 will leave 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/handler 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 will 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 will sometimes cause some intermediate files to be left in your directory that otherwise would be deleted.

B.3 ADDING A PROCESS TO AN EXISTING APPLICATION IMAGE

Assume that you have already built a kernel and application image and have the following set of build-cycle output files:

- KERNL1.MIM, KERNL1.STB, and KERNL1.DBG, from the kernel/handler image building phase

- APPLC1.MIM, APPLC1.STB, and APPLC1.DBG, from the application image building phase.

(The kernel and application STB files are identical in content; the only purpose of APPLC1.STB is to facilitate the strategy described below.) Assume further that after some testing, you want to add one or more user processes to the application MIM and DBG files without modifying the kernel, handlers, or any of the existing user processes. You can do that with MPBUILD and avoid having to rebuild all of the existing processes, by employing the following sample strategy:

1. Invoke MPBUILD and request an application-only partial build cycle (NO response to question 2), as if you were starting the build from an existing kernel/handler image.

2. Specify APPLC1 as the name of the "kernel" memory image file in response to question 3.

3. Respond NO to question 4 (additional system processes) and specify APPLC2 as the name of the application image file in response to question 9.

4. Answer the global-information questions (11 through 16) exactly as you did when building the previous application image and then specify the new user process(es) to be installed in the extended application image.

Using this approach, you end up with the original APPLC1 files plus the new APPLC2.MIM and .DBG files with the additional user process installed. (You also have an APPLC2.STB file that, again, is identical in content to APPLC1.STB and KERNL1.STB.)

Note that, using MPBUILD, you cannot replace a process in an existing application image with a modified, identically named version. You must rebuild the entire application image, that is, create a new application MIM file, to effectively "replace" a static process with a debugged version.

B.4 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

The MPBUILD procedure has terminated due to an unrecoverable error condition encountered during the build dialog.

<comfile-name>-F-Build error; build cycle aborted

The generated build cycle has terminated abnormally due to an error in one of the build steps. This message is preceded at some point by an error message from the program that detected the error condition.

MPBUILD-W-Conflicting switches

You specified two mutually exclusive option switches: /LIS and /OBJ, or /LBR and any other switch. Enter your response again.

%MPBUILD-W-Illegal file specification

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

Self-explanatory; enter your response again.

?MPBUILD-W-Invalid character in <filespec>

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

?MPBUILD-W-Invalid device spec

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

?MPBUILD-W-Invalid file name

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

?MPBUILD-W-Invalid file type

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

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

Self-explanatory; enter your response again.

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

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

MPBUILD-W-Invalid or ambiguous switch

You used either an undefined option switch -- something other than /LBR, /LIS, /OBJ, or /MAP -- or an option switch that is invalid for the context. Enter your response again.

?MPBUILD-W-Invalid UFD

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

?MPBUILD-W-Invalid version number

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

%MPBUILD-W-Merge command line may need editing

Due to 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 that utility (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 inputs to MPBUILD for the process in question.

MPBUILD-W-Must specify positive octal value

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

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

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

?MPBUILD-W-Switches not allowed on file spec

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

AAPFX.PAS, prefix module for
ADV11-C/AXV11-C, 4-29
/AB (alphabetical list) option,
for RELOC, 6-9
ABPFX.PAS, prefix module for
AAV11-C/AXV11-C, 4-31
. ABS. p-sect
excluding symbols from map,
6-13
symbols in .STB file, 6-5
/AL (align RW segment) option,
for RELOC, 6-9, 7-18, 7-21
.ALST. p-sect, 6-11, 6-12, 7-15,
7-21
Applications
building, 3-1
automated, 1-8, 3-26, B-1
via MPBUILD, 1-8, 3-26, B-1
overview, 1-3
utility program summary, 3-3
components of, 3-1
development overview, 1-6
loading into PROM, 8-2
loading/debugging, 8-1
overview of, 1-8
.AUX file, output from MERGE, 3-3,
5-2, 5-7, 5-10

$BHUSR kernel symbol, FALCON,
4-12, 4-14
Boot-device CSR, modifying, 9-3
Bootable volume
copying .MIM file to, 9-1
prepared by COPYB, 9-2
Bootstraps
installing in .MIM file, 3-14,
7-6, 7-11
loading application by, 8-2
memory required for, 7-12
in MPBUILD dialog, B-15
processed by COPYB, 8-2, 9-1
supplied files, 3-14, 7-12
/BS (install bootstrap) option,
for MIB, 7-11
Build cycle, 1-3, 3-2, B-1
automated, 3-26, B-1
debugging and rebuilding, 3-27
details of, 3-7
kernel phase, 1-6, 3-5, 3-7
load/debug phase, 1-8
overview of, 1-6, 3-4
system-process phase, 1-6, 3-15
user inputs for, 1-8
user-process phase, 1-7, 3-21
Build-utility programs
functional summary, 3-3
overview, 1-3

.CDAT. p-sect (Pascal), 7-15
CFDxxx.MAC, working configuration
files, 4-5
/CH:xxx compilation option, 2-7
CKPFX.MAC, prefix module for line
clock, 4-32
on KXT11-C target, 4-47
Clock handler (CK) prefix module,
4-32
for KXT11-C target, 4-47
/CM (compact image) option, for
MIB, 7-12
Command files, generated by
MPBUILD, 1-8, B-8, B-15
Command symbols, user-defined,
for VMS, 1-10
Compilation commands
examples, 2-3, 2-5
rules and restrictions, 2-4
syntax, 2-2
Compilation options, 2-4, 2-7
in command line
/CH:xxx switch, 2-7
/DE switch, 2-7
/EX switch, 2-8, 2-11
/FI switch, 2-8
/IN:xxx switch, 2-9
/MA switch, 2-10
/NO switch, 2-10
/ST switch, 2-10
in source code
examples, 2-6
INDEXCHECK, 2-6
LIST/NOLIST, 2-6, 2-10
MATHCHECK, 2-6
POINTERCHECK, 2-6
RANGECHECK, 2-6
STACKCHECK, 2-6
STANDARD, 2-6
specified in MPBUILD, B-9
Compilation source listings, 2-11
Compiler. See MPPASCAL
COMx.MLB, system macro libraries,
3-8, 3-16, 3-22, 4-6, 4-60

CONFIG.MAC, prototype
configuration file, 4-3
Configuration file, 1-4, 3-5, 4-1
assembling, 4-6
building kernel with, 3-5
CFDxxx.MAC, 4-5
CONFIG.MAC, 4-3
creating and assembling, 3-8
function of, 4-2
input to MERGE, 5-7
macros
CONFIGURATION, 4-7
DEVICES, 4-8
ENDCFG, 4-8
KXT11, 4-9
KXT11C, 4-15
MEMORY, 4-18
order of, 4-5
PRIMITIVES, 4-19
PROCESSOR, 4-20
RESOURCES, 4-21
summary of, 4-6
SYSTEM, 4-23
TRAPS, 4-24
merging with kernel module
library, 3-9
modifying macros, 4-7
prototype, 4-3
specified in MPBUILD, B-7
CONFIGURATION macro, 4-7
COPYB utility program, 9-1
command line
examples, 9-3
syntax, 9-2
/CS option, 9-2
invocation
CPB command (RSX), 9-2
MPCOPYB command (VMS), 9-2
processes bootstrap, 8-2, 9-1
CPB command (RSX), 9-2
/CS (CSR address) option, for
COPYB, 9-2
CSR address, modifying in
bootstrap, 9-2

.DBG file
output from MIB, 3-2, 3-3, 7-1,
7-10
produced via MPBUILD, B-8
DDPFX.MAC, prefix module for TU58,
4-34
DDPFXK.MAC, prefix module for
TU58 (KXT11-C), 4-47
/DE (debug) option
for compiler, 2-7
for MERGE, 5-10, 5-13
for RELOC, 6-5, 6-10
Debug symbol (.DBG) file, details,
7-6, 7-10
Debug symbol information. See
also ISD records
collected in .DBG file, 7-6
deleting from .DBG file, 7-6
for MACRO-11 programs, 5-1,
5-14
for Pascal programs, 2-7, 5-1,
5-13
generated by MERGE, 5-14
in .STB file, 6-10, 7-6
processed by MIB, 7-6, 7-10
propagated
by MERGE, 5-13
by RELOC, 6-5, 6-10
provided via MPBUILD, B-8
Debugger Service Module (DSM),
3-13, 3-14, 4-23
Debugger. See PASDBG
Debugging
as part of build cycle, 3-27
overview of, 1-8
serial line for, 10-1
errors associated with, 10-5
setting characteristics of,
10-2, 10-4
Decimal vs octal radix, in RELOC
switches, 6-8
Development process. See also
Build cycle
overview, 1-6
Development-tools overview, 1-2
build-utility programs, 1-3
COPYB utility, 1-4
MERGE utility, 1-3
MIB utility, 1-3
MPBUILD procedure, 1-3
MPPASCAL compiler, 1-2
PASDBG symbolic debugger, 1-2
RELOC utility, 1-3
Device handlers
building, 3-6, 3-15
installing in memory image,
3-20
mapped, relocating, 7-20
object libraries, 1-4, 4-25
overview, 1-4
prefix modules, 4-25
relocating, 3-19
DEVICES macro, 4-8
Directory Service Process, A-1
assembling prefix for, 3-17,
A-3
editing prefix for, A-2
merging, 3-18, A-1
object libraries, 1-5, A-1
overview, 1-5
prefix module for, 1-5, A-2
relocating, 3-20
DLPFX.MAC, prefix module for
RL01/RL02, 4-34
DLV11 (XL) prefix module, 4-39
Down-line loading, via PASDBG,
8-1
DPV11 (XP) prefix module, 4-43
Driver. See Device handlers

DRV11 (YA) prefix module, 4-45
DRV11-J (XA) prefix module, 4-38
DRVM.OLB, DRVU.OLB, DRVK.OLB, 1-4, 4-26
/DS (disable sort) option, for RELOC, 6-10
DSM, debugger support in kernel, 3-13, 3-14, 4-23
DSP. See Directory Service Process
DSPNHD.OLB, DSPEIS.OLB, DSPFIS.OLB, DSPFPP.OLB, 1-5, 5-4, A-1
DSPPFX.MAC prefix module, 1-5, A-2
DUPFX.MAC, prefix module for MSCP disks (RX50, RD51), 4-35
DYPFX.MAC, prefix module for RX02, 4-35

ENDCFG macro, 4-8
/EX (delete process/symbols) option, for MIB, 7-13
/EX (extend section) option, for RELOC, 6-11
/EX (extra stats) compilation option, 2-8, 2-11
Exception group code, modifying, 7-13

FALCON kernel
  modules, user supplied, 4-12
  symbols $BHUSR, $NHUSR, $NUHNG, 4-12
FALCON PIO port (YF) prefix module, 4-46
FALUSR.MAC, for FALCON kernel, 4-12, 4-13
/FI (filter) compilation option, 2-8
File options in MPBUILD, /LBR, /LIST, /OBJ, /MAP, B-4
File system process (DSP). See also Directory Service Process
  details, A-1
  libraries, 1-5, A-1
  overview, 1-5
File system support
  libraries, 1-5, 3-22, A-3
  per process, 1-5, A-3
  requested in MPBUILD, B-11, B-13
File types, summary, 3-3
FSLIB.OLB, Pascal I/O support library, 1-5, 2-1, 5-4, A-3
FSMAC.MLB, file-system macro library, 3-22, A-3

/GC (exc. group code) option, for MIB, 7-13
Global symbol, 5-1
  references resolved by MERGE, 5-3
  values adjusted by RELOC, 6-2
GSD records, 5-1, 5-3
  for module name, 5-16, 6-11
  in .STB file, 6-5

Handler. See Device handlers
Host systems, 1-1
  logical devices for, 1-9, 10-2
  relationship between RSX-11 and VMS, 1-2
  terminal line used for debugging, 10-1
Host/target serial line
  assignment of, 10-2, 10-3
  errors associated with, 10-5
  requirements of, 10-1, 10-3
  setting characteristics of, 10-2, 10-4

.IDENT directive, overriding, 5-17, 6-13
/IN (include module) option, for MERGE, 5-15
/IN:xxx (instructions) compilation option, 2-9
INCLUDE files, Pascal
  PREDFL.PAS, 2-10
  RHSDSC.PAS, 5-4
INDEXCHECK source option, 2-6
Internal Symbol Directory (ISD) records, 5-13
ISD records, 5-1
  details, 5-13
  in .STB file, 6-5, 6-10, 7-6

Kernel, 3-1
  functions of, 1-4
  installing in memory image, 3-12, 7-14
  object libraries, 1-4, 3-9
  optimizing with MERGE, 5-7
  relocating, 3-10
  steps in building, 3-5, 3-7
  symbols, merging with processes, 3-17, 3-22
Kernel module library, merging with configuration file, 3-9
/KI (kernel init) option, for MIB, 7-10, 7-14
KKPFXK.MAC, prefix module for KXT11-C slave, 4-48

KWPFX.PAS, prefix module for KWV11-C, 4-36
KWV11-C (KW) prefix module, 4-35, 4-36
KXPFX.MAC, prefix module for KXT11-C arbiter, 4-37
KXT11 macro, 4-9
KXT11-C Q-BUS arbiter (KX) prefix module, 4-37
KXT11-C target
  async. SLU (XL) prefix module, 4-49
  DTC (QD) prefix module, 4-48
  kernel parameters for, 4-15
  libraries for, 4-25
  parallel port (YK) prefix module, 4-51
  prefix modules for, 4-28, 4-46
    CKPFX.MAC, 4-47
    DDPFXK.MAC, 4-47
    KKPFXK.MAC, 4-48
    QDPFXK.MAC, 4-49
    XLPFXK.MAC, 4-49
    XSPFXK.MAC, 4-51
    YKPFXK.MAC, 4-51, 4-59
  Q-BUS slave (KK) prefix module, 4-47
  sync. SLU (XS) prefix module, 4-50
  using PASDBG with, 4-17
KXT11C macro, 4-15

/LB (library file) option, for MERGE, 5-16
/LB:name (extract from library) option, for MERGE, 5-16
/LBR switch, in MPBUILD dialog, B-4
LIBNHD.OLB, LIBEIS.OLB, LIBFIS.OLB, LIBFPP.OLB, 2-1, 5-3
Libraries
  COMM.MLB and COMU.MLB, 3-8, 3-16, 3-22, 4-6, 4-60
  DRVM.OLB, DRVU.OLB, DRVK.OLB, 1-4, 4-26
  DSPxxx.OLB, 1-5, 5-4, A-1
  file system support, A-3
  FSLIB.OLB, 1-5, 5-4, A-3
  FSMAC.MLB, 3-22, A-3
  LIBxxx.OLB, 2-1, 5-3
  MACFS.OLB, 1-5, 5-4, A-3
  PAXM.OLB, PAXU.OLB, 1-4, 3-9
  RHSLIB.OLB, 5-4
Library file switch, /LB, for MERGE, 5-16
Library references, resolution of, 5-3
LINDF$ macro, for serial lines, 4-40, 4-50
/LIST switch, in MPBUILD dialog,B-4
LIST/NOLIST source option, 2-6, 2-10
Listings, compilation, 2-11
Load map, produced by MIB, 7-24
Loading applications, 8-1
  bootstrapping, 8-1
  down-line loading, 8-1
  overview, 1-8
  into PROM, 8-2
Logical devices
  MICROPOWER$LIB, for VMS, 1-10
  MP1:, for RSX-11, 1-9
  MP:, for RSX-11, 1-9
  TD:, for debugging, 10-2, 10-3
    errors associated with, 10-5
LOGIN.COM file (for VAX/VMS), 1-10

/MA (Macro) compilation option, 2-10
MACFS.OLB, Macro I/O support library, 1-5, 3-23, 5-4, A-3
Macro libraries
  COMM.MLB and COMU.MLB, 3-8, 3-16, 3-22, 4-6, 4-60
  FSMAC.MLB, 3-22, A-3
  MPBUILD restriction on, B-2
Macros for configuration file, 4-6
  CONFIGURATION, 4-7
  DEVICES, 4-8
  ENDCFG, 4-8
  KXT11, 4-9
  KXT11C, 4-15
  MEMORY, 4-18
  PRIMITIVES, 4-19
  PROCESSOR, 4-20
  RESOURCES, 4-21
  SYSTEM, 4-23
  TRAPS, 4-24
Map
  produced by MERGE, 5-12
  produced by MIB, 7-6, 7-24
  produced by RELOC, 6-7, 6-13, 6-14
.MAP file
  output from MERGE, 3-3, 5-2, 5-10
  output from MIB, 3-3, 7-1, 7-10
  output from RELOC, 3-3, 6-1, 6-5
/MAP switch, in MPBUILD dialog, B-5
MATHCHECK source option, 2-6
.MDAT. p-sect (MACRO-11), 7-21
Memory image (.MIM) file
  bootstrap format, 7-3
  compaction of, 7-7, 7-12
  copying to boot volume, 9-1
  "holes" in, 7-12
  initializing, 7-2, 7-14

installing kernel in, 7-14
installing processes in, 3-26, 7-5
PASDBG load format, 7-3
PROM format, 7-4
"short" vs full size, 7-7
MEMORY macro, 4-18
Memory map, produced by MIB, 7-24
MERGE utility program, 3-2, 5-1
command line
examples, 5-11
syntax, 5-10
configuration file as input, 5-7
detailed description, 5-2
functions, 3-2
invocation
MPMERGE command (VMS), 5-9
MRG command (RSX), 5-9
merges static process modules, 5-7
optimizes the kernel, 5-7
options
/DE, 5-10, 5-13
/IN, 5-15
/LB, 5-16
/LB:name, 5-16
/NM, 5-15
summary of, 5-13
switch syntax, 5-10
/VR, 5-17
order of library inputs, 5-4
resolves global symbols, 5-3
satisfies library references, 5-3
section map, 5-11
summary of input and output files, 3-3
Merged object module (MOB), 5-1
MIB command (RSX), for MIB, 7-8
MIB utility program, 3-2, 7-1
command line
examples of, 7-16
syntax, 7-9
compacts mapped .MIM file, 7-7
creates .DBG file, 7-6
creates .MIM file, 7-2
deletes debug symbols, 7-6
deletes static processes, 7-5
detailed description, 7-2
functions, 3-2
installs bootstrap, 7-6
installs debug symbols, 7-6
installs static processes, 7-5
interaction with RELOC options, 7-18
invocation
MIB command (RSX), 7-8
MPMIB command (VMS), 7-9
memory map, 7-6, 7-24
.MIM file
bootstrap format, 7-3
PASDBG load format, 7-3
PROM format, 7-4
options
/BS, 7-11
/CM, 7-12
/EX, 7-13
/GC, 7-13
/KI, 7-10, 7-14
/PR, 7-14
/QB, 7-15, 7-21
/SM, 7-7, 7-15
summary of, 7-11
switch syntax, 7-10
strategy of use, 7-7
summary of input and output files, 3-3
MICROPOWER$LIB logical device, for VMS, 1-10
MicroPower/Pascal
compiler. See MPPASCAL
development process, 1-6
file system, A-1
host systems, 1-1, 1-2
product components, 1-2
product features, 1-1
product objectives, 1-1
target systems, 1-1, 1-2
.MIM file
copying to boot volume, 9-1
creating
for bootstrapping, 3-13
for debugging or down-line loading, 3-13
from kernel, 3-12
for ROM/RAM, 3-15
input to COPYB, 9-2
input to MIB, 3-3, 7-1, 7-10
input to RELOC, 3-3, 6-1, 6-5
output from MIB, 3-2, 3-3, 7-1, 7-9
.MOB file
input to RELOC, 3-3, 6-1, 6-5
output from MERGE, 3-2, 3-3, 5-2, 5-10
Module name GSD record, in object module, 5-16, 6-11
MP: and MP1: logical devices, for RSX-11, 1-9
MPBUILD procedure, 1-8, 3-1, B-1
detailed description, B-1
dialog defaults, B-5
error messages, B-16
errors from, B-15
invocation, B-3
limitations of, B-2
long form of dialog, B-6
macro-library restriction, B-2
versus "manual" build, 3-26
special usage, B-15
.TITLE requirement, B-2, B-13
MPCOPYB command (VMS), 9-2
MPMERGE command (VMS), 5-9
MPMIB command (VMS), 7-9
MPP command (RSX), 2-2
MPPASCAL command (VMS), 2-2

MPPASCAL compiler, 1-2, 2-1
  command-line syntax, 2-2
  compilation options, 2-3, 2-4, 2-7
  invocation
    MPP command (RSX), 2-2
    MPPASCAL command (VMS), 2-2
  language extensions, 2-1
  OTS libraries, 2-1, 5-3
  PREDFL.PAS file, 2-10
  source listings, 2-11
  source-code options, 2-6
  temporary file space, 2-1
MPRELOC command (VMS), 6-4
MPSETUP.COM file (for VAX/VMS), 1-10
MPxxxx symbol definitions, for VMS, 1-10
MRG command (RSX), for MERGE, 5-9
MSCP (DU) prefix module, 4-33

$NHUSR kernel symbol, FALCON, 4-12, 4-13
/NM (module name) option
  for MERGE, 5-15
  for RELOC, 6-11
/NO (no PREDFL) compilation option, 2-10
$NUHNG kernel symbol, FALCON, 4-11, 4-14

.OBJ file, input to MERGE, 3-3, 5-2, 5-5, 5-10
/OBJ switch, in MPBUILD dialog, B-4
Object libraries
  DRVM, DRVU, and DRVK, 1-4, 4-26
  DSPNHD, DSPEIS, DSPFIS, DSPFPP, 1-5, 5-4, A-1
  FSLIB (Pascal I/O support), 1-5, 5-4, A-3
  identification of, 5-16
  LIBNHD, LIBEIS, LIBFIS, LIBFPP, 2-1, 5-3
  MACFS (Macro I/O support), 1-5, 5-4, A-3
  order of input to MERGE, 5-4
  PAXM and PAXU, 1-4, 3-9
  RHSLIB (special I/O support), 5-4
  summary of, 5-4
Object module contents
  GSD records, 5-1, 5-3, 6-5
    for module name, 5-16, 6-11
  ISD records, 5-1, 6-5, 6-10
    details, 5-13
  RLD records, 5-1, 5-3
  TXT records, 5-1
Object-time support, Pascal, 2-1, 5-3
Octal vs decimal radix, in RELOC switches, 6-8
.OLB file, input to MERGE, 5-2, 5-5, 5-10
Options, summary of
  MERGE, 5-13
  MIB, 7-11
  MPPASCAL, 2-4
  RELOC, 6-8
OTS libraries, Pascal, 2-1, 5-3

P-sects. See Program sections
Pascal. See also MPPASCAL
  file system support, A-3
  INCLUDE files
    PREDFL.PAS, 2-10
    RHSDSC.PAS, 5-4
  language extensions, 2-1
  OTS libraries, 2-1, 5-3
PASDBG symbolic debugger, 1-2, 10-1
  .DBG symbol file for, 7-6
  host-specific errors from, 10-5
  LOAD/EXIT command, 8-1
  serial-line requirements
    under RSX, 10-1
    under VMS, 10-3
  SET PROGRAM command, 5-16, 6-11
PAXM.OLB, PAXU.OLB, 1-4, 3-9
.PIM file
  input to MIB, 3-3, 7-1, 7-10
  output from RELOC, 3-2, 3-3, 6-1, 6-4
POINTERCHECK source option, 2-6
/PR (process priority) option, for MIB, 7-14
PREDFL.PAS file, 2-10
Prefix module, 3-6, 4-1, 4-25
  assembling or compiling, 3-16, 4-60, A-3
  editing, 3-16, 4-29, A-2
    for KXT11-C, 4-46
  file for DSP, 1-5, A-2
  files for handler processes, 1-4, 4-27
  function, 4-26
  summary of supplied modules, 4-27
  supplied
    AAPFX.PAS, 4-29
    ABPFX.PAS, 4-31
    CKPFX.MAC, 4-32, 4-47
    DDPFX.MAC, 4-34
    DDPFXK.MAC (for KXT11-C), 4-47
    DLPFX.MAC, 4-34
    DSPPFX.MAC, A-2
    DUPFX.MAC, 4-35
    DYPFX.MAC, 4-35
    KKPFXK.MAC (for KXT11-C), 4-48
    KWPFX.PAS, 4-36

KXPFX.MAC, 4-37
QDPFXK.MAC (for KXT11-C), 4-49
XAPFX.MAC, 4-39
XLPFX.MAC, 4-39
XLPFXD.MAC, 4-39
XLPFXF.MAC, 4-39
XLPFXK.MAC (for KXT11-C), 4-49
XPPFX.MAC, 4-44
XSPFXK.MAC (for KXT11-C), 4-51
YAPFX.PAS, 4-45
YFPFX.MAC, 4-46
YKPFXK.MAC (for KXT11-C), 4-51, 4-59
PRIMITIVES macro, 4-19
Process image (.PIM) file, 6-3
Processes
changing startup priority of, 7-14
installing in memory image, 3-26
relocating, 3-19, 3-24
details, 6-2
PROCESSOR macro, 4-20
Program "ident" (/VR) option, 5-17, 6-13
Program name, for debugging, 5-16, 6-11
Program relocation, details, 6-1
Program sections
.ALST., 6-11, 6-12, 7-15, 7-21
.CDAT. (Pascal), 7-15, 7-21
combined by MERGE, 5-2, 5-3
extending with RELOC, 6-11
first read-only, 6-12, 7-15, 7-21
first read/write, 6-9, 6-13, 7-15, 7-21
mapped, setting physical address of, 7-15, 7-21
.MDAT. (MACRO-11), 7-21
MERGE map of, 5-11
order requirements, 6-10
relocation map of, 6-6
relocation of, 6-2
rounding up size of, 6-13
setting base address of, 6-12, 7-15, 7-21
sorted by RELOC, 6-2, 6-10
PROM, loading applications in, 8-2

/QB (p-sect address) option
for MIB, 7-15, 7-21
for RELOC, 6-12
QDPFXK.MAC, prefix module for KXT11-C, 4-49

Radix representation, in option switches, 6-8
RANGECHECK source option, 2-6
RD51 (DU) prefix module, 4-33
REL command (RSX), for RELOC, 6-3
RELOC utility program, 3-2, 6-1
command line
examples, 6-5
syntax, 6-4
detailed description, 6-2
functions, 3-2
inputs and outputs, 6-1
interaction with MIB, 7-18
invocation
MPRELOC command (VMS), 6-4
REL command (RSX), 6-3
options
/AB, 6-9
/AL, 6-9, 7-18, 7-21
/DE, 6-5, 6-10
/DS, 6-10
/EX, 6-11
/NM, 6-11
/QB, 6-12
/RO, 6-12, 7-19, 7-21
/RW, 6-13, 7-19, 7-21
/SH, 6-13
summary of, 6-8
switch syntax, 6-5
/UP, 6-13
/VR, 6-13
/WI, 6-14
/ZR, 6-14
relocation map, 6-6, 6-13, 6-14
summary of input and output files, 3-3
Relocating processes
details, 6-2
device handler, 3-19
mapped device handler, 7-20
system, 3-19
user, 3-24
Relocation map, produced by RELOC, 6-6, 6-13, 6-14
RESOURCES macro, 4-21
RHSDSC.PAS file, 5-4
RHSLIB.OLB, special I/O support library, 2-1, 5-4, B-12
RL01/RL02 (DL) prefix module, 4-33
RLD records, 5-1, 5-3
/RO (first RO p-sect) option, for RELOC, 6-12, 7-19, 7-21
in unmapped case, 7-19
ROM/RAM target environment, 1-9
creating .MIM file for, 3-15
/IN:LS2 compiler switch, 2-9
interspersed, restriction on, B-2
mapped, relocating for, 6-9, 7-18, 7-22
relocating for, 3-11, 3-19, 3-25, 7-22
sorting p-sects for, 6-10

Runtime software overview, 1-2, 1-4
  device handlers, 1-4
  DSP, 1-5
  file system process, 1-5
  file system support, 1-5
  kernel, 1-4
/RW (first RW p-sect) option, for RELOC, 6-13, 7-19, 7-21
  in mapped case, 7-19
RX02 (DY) prefix module, 4-33
RX50 (DU) prefix module, 4-33

SBC-11/21. See FALCON
/SH (short map) option, for RELOC, 6-13
/SM (small image) option, for MIB, 7-7, 7-15
/ST (standard) compilation option, 2-10
STACKCHECK source option, 2-6
STANDARD source option, 2-6, 2-10
Static process. See also Processes; System processes; User processes
  deleting from image, 7-5
  first RO p-sect in, 6-12, 7-15, 7-21
  first RW p-sect in, 6-9, 6-13, 7-15, 7-21
  installing in image, 7-5
  list element (.ALST.), 6-11, 7-15, 7-21
  modules, input to MERGE, 5-7
  runtime name, for debugging, 5-16, 6-11
  startup priority, modifying, 7-14
  system, 3-1
  user, 3-1
.STB file
  input to MERGE, 3-3, 5-2, 5-5, 5-10
  input to MIB, 3-3, 7-1, 7-10
  output from RELOC, 3-2, 3-3, 6-1
    details, 6-5
Symbol definition file, MPSETUP.COM, 1-10
System configuration file. See Configuration file
SYSTEM macro, 4-23
System processes, 3-1
  building, 3-15
  DSP, details, A-1
  installing in memory image, 3-20
  merging with kernel, 3-17
  need prefix module, 3-16
  object libraries, 4-25
  overview, 1-4, 1-5
  relocating, 3-19

Target systems, 1-2, 4-20
TD: logical device, for debugging
  assignment of, 10-2, 10-3
  errors associated with, 10-5
Temporary files, compiler, 2-1
.TITLE directive
  overriding, 5-15, 6-11
  required for MPBUILD, B-2
TRAPS macro, 4-24
TU58 (DD) prefix module, 4-33
  for KXT11-C, 4-47
TXT records, 5-1

Undefined locations, value of, 6-14
/UP (round up p-sect) option, for RELOC, 6-13
User modules, for FALCON kernel, 4-12
User processes
  building, 3-21
  compiling or assembling, 3-22
    with file system support, A-3
  installing in memory image, 3-26
  Macro, MPBUILD requirements for, B-2, B-13
  merging with kernel STB, 3-22
    and file system support, A-3
  Pascal, merging with OTS, 3-23
  relocating, 3-24
Utility programs
  COPYB, 1-4, 9-1
  MERGE, 1-3, 3-2, 5-1
  MIB, 1-3, 3-2, 7-1
  overview, 1-3, 3-2
  RELOC, 1-3, 3-2, 6-1

/VR (version number) option
  for MERGE, 5-17
  for RELOC, 6-13

/WI (wide map) option, for RELOC, 6-14
WRITE_ANALOG_WAIT procedure, 4-28, 4-31

XAPFX.MAC, prefix module for DRV11-J, 4-39
XLPFX.MAC, prefix module for DLV11, 4-39
XLPFXD.MAC, prefix module for DLV11, 4-39
XLPFXF.MAC, prefix module for FALCON SLU, 4-39

XLPFXK.MAC, prefix module for
KXT11-C SLU, 4-49
XPPFX.MAC, prefix module for
DPV11, 4-44
XSPFXK.MAC, prefix module for
KXT11-C SLU, 4-51

YAPFX.PAS, prefix module for
DRV11, 4-45

YFPFX.MAC, prefix module for
FALCON PIO, 4-46
YKPFXK.MAC, prefix module for
KXT11-C parallel port, 4-51,
4-59

/ZR (zero locations) option, for
RE'OC, 6-14

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Chicago	312–640–5612 8:15 AM to 5:00 PM CT	Digital Equipment Corporation Accessories & Supplies Center 1050 East Remington Road Schaumburg, IL 60195
San Francisco	408–734–4915 8:15 AM to 5:00 PM PT	Digital Equipment Corporation Accessories & Supplies Center 632 Caribbean Drive Sunnyvale, CA 94086
Alaska, Hawaii	603–884–6660 8:30 AM to 6:00 PM ET *or* 408–734–4915 8:15 AM to 5:00 PM PT	
New Hampshire	603–884–6660 8:30 AM to 6:00 PM ET	Digital Equipment Corporation Accessories & Supplies Center P.O. Box CS2008 Nashua, NH 03061
Rest of U.S.A., Puerto Rico*	1–800–258–1710 8:30 AM to 6:00 PM ET	
Canada		
British Columbia	1–800–267–6146 8:00 AM to 5:00 PM ET	Digital Equipment of Canada Ltd 940 Belfast Road Ottawa, Ontario K1G 4C2 Attn: A&SG Business Manager
Ottawa–Hull	613–234–7726 8:00 AM to 5:00 PM ET	
Elsewhere	112–800–267–6146 8:00 AM to 5:00 PM ET	
Elsewhere		Digital Equipment Corporation A&SG Business Manager*

*Prepaid orders from Puerto Rico must be placed with the local DIGITAL subsidiary (call 809–754–7575)

*c/o DIGITAL's local subsidiary or approved distributor

MicroPower/Pascal–RSX/VMS
System User's Guide
AA-AK13A-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 _____ Telephone _____

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

SSG/ML PUBLICATIONS, ML05–5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MA 01754

— — Do Not Tear — Fold Here —

Cut Along Dotted Line