Introduction to MicroPower/Pascal

Order No. AA–M388C–TK

digital

software

Introduction to MicroPower/Pascal

Order No. AA–M388C–TK

June 1987

This manual introduces the MicroPower/Pascal software development toolset.

This manual supersedes the *Introduction to MicroPower/Pascal,* AA–M388B–TK.

Operating System and Version:	Micro/RSX Version 3.0 RSX–11M Version 4.2 RSX–11M–PLUS Version 3.0 RT–11 Version 5.2 VAX/VMS Version 4.0
Software Version:	MicroPower/Pascal–Micro/RSX Version 2.4 MicroPower/Pascal–RSX Version 2.4 MicroPower/Pascal–RT Version 2.4 MicroPower/Pascal–VMS Version 2.4

Digital Equipment Corporation Maynard, Massachusetts

First Printing, January 1982
Revised, June 1985
Revised, June 1987

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

Contents

Chapter 6 Development Cycle Overview

Chapter 7 Building and Running the CARS Program Examples

Glossary

Index

Examples

Figures

Tables

Preface

This manual introduces the basic concepts and components of MicroPower/Pascal. This book gives you a general understanding of MicroPower/Pascal's capabilities and uses and enables you to build and run a sample MicroPower/Pascal application. This introduction provides you with an overall perspective in preparation for your independent use of the MicroPower/Pascal software and documentation set.

Structure of This Document

This manual consists of seven chapters and a glossary as follows:

- Chapter 1 provides a general overview of the MicroPower/Pascal product. The chapter introduces concurrent-programming concepts and describes the MicroPower/Pascal software components.
- Chapter 2 describes host and target system environments and the features, options, and requirements of the host operating systems.
- Chapter 3 defines the process concept as used in MicroPower/Pascal and provides detailed information on declaring and creating processes within MicroPower/Pascal applications. Examples illustrating the concepts are included.
- Chapter 4 discusses both concurrency concepts, as they relate to processes and procedures, and functions that operate on processes. This chapter describes process synchronization and communication using semaphores and ring buffers and introduces procedures that operate on semaphores and ring buffers. Connecting processes to interrupts and exception-handling processes and procedures are also covered.
- Chapter 5 presents a sample program, developed in three stages, to illustrate the use of specific MicroPower/Pascal features.
- Chapter 6 discusses the steps necessary to build and run a MicroPower/Pascal application.
- Chapter 7 uses the design example developed in Chapter 5 to build and run a MicroPower/Pascal application. The automatic build procedure is introduced as a tool to facilitate application building.

- The Glossary defines terms used throughout the MicroPower/Pascal documentation set. You should find the Glossary a valuable aid in understanding new terms specific to MicroPower/Pascal.

Intended Audience

This manual assumes that you have a general understanding of your host system hardware and that you are familiar with your host operating system concepts. In addition, this manual assumes your familiarity with the Pascal language. Although this introduction presents language features specific to the MicroPower/Pascal implementation of Pascal, it does not offer a tutorial on programming in Pascal. For operating system information, RT–11 users may refer to the RT–11 documentation supplied in their MicroPower/Pascal–RT distribution kit. Other users are referred to the documentation supplied with their host operating systems.

Conventions Used in This Document

The following conventions are used in this manual:

- Unless otherwise noted, MPBUILD is used as the generic name of the supplied command file generator, known as MPBUILD to RSX and VMS users or as MPBLD to RT–11 users.
- Unless otherwise noted, user input in command examples is shown in **boldface** type to differentiate it from computer output.
- All commands or command strings terminate with a carriage return.
- To produce certain characters in system commands, you must type a combination of keys simultaneously. For example, while holding down the CTRL key, type C to produce the CTRL/C character. Such key combinations are documented as <CTRL/C> , <CTRL/O> , and so forth.
- In examples, you must distinguish between the capital letter O and the number 0. Examples in this manual represent these characters as follows:

  Letter O: O

  Number 0: 0
- The sample terminal output in this manual contains version numbers where they would normally appear. The version numbers include xx in those fields that may vary from installation to installation.
- The RSX system prompt is shown as a right angle bracket (>). The RT–11 system prompt is shown as a period (.). If you are using a DCL interface, you should see the dollar sign prompt ($) rather than a right angle bracket.

Associated Documents

The following manuals and reference card are included in the MicroPower/Pascal documentation set for all users of MicroPower/Pascal:

- *Introduction to MicroPower/Pascal*
- *MicroPower/Pascal Debugger User's Guide*
- *MicroPower/Pascal I/O Services Manual*
- *MicroPower/Pascal Language Guide*
- *MicroPower/Pascal Pocket Guide*
- *MicroPower/Pascal Run-Time Services Manual*
- *MicroPower/Pascal Master Index*
- *MicroPower/Pascal Release Notes*
- *PDP-11 MACRO-11 Language Reference Manual*
- *PDP-11 Programming Card*

For RT-11 host system users, these additional manuals are included in your documentation set:

- *MicroPower/Pascal-RT Installation Guide*
- *MicroPower/Pascal-RT Messages Manual*
- *MicroPower/Pascal-RT System User's Guide*

For RSX and VAX/VMS host system users, these additional manuals are included in your documentation set:

- *MicroPower/Pascal-RSX/VMS Messages Manual*
- *MicroPower/Pascal-RSX/VMS System User's Guide*
- *MicroPower/Pascal-RSX Installation Guide* (for RSX host system users only)
- *MicroPower/Pascal-VMS Installation Guide* (for VAX/VMS host system users only)

Document Descriptions

A brief overview of the MicroPower/Pascal and PDP-11 manuals in your documentation set follows. The *MicroPower/Pascal-RT Installation Guide* describes the RT-11 and PDP-11 documentation included only in the MicroPower/Pascal-RT documentation set.

MicroPower/Pascal Debugger User's Guide

This manual describes the MicroPower/Pascal symbolic debugger, PASDBG. Chapter 1 introduces PASDBG, describes its operating environment, and outlines steps to be taken before you attempt to debug a MicroPower/Pascal application. Chapter 2 describes PASDBG's features and command set by functional group. Chapter 3 contains detailed descriptions of each PASDBG command, including the action performed by the command and correct syntax. Examples accompany most command descriptions. The commands are listed in alphabetical order. Chapter 4 contains two PASDBG tutorials. Each tutorial stresses the use of different

PASDBG commands and is based on the CARS3 example program described in the *Introduction to MicroPower/Pascal.*

The appendixes describe set-up requirements for the host/target communication line, target interface specifications, debugging hints, and a recursive Pascal program example that uses the SET SCOPE command.

MicroPower/Pascal I/O Services Manual

This manual describes I/O services provided by MicroPower/Pascal system processes. Chapter 1 presents an overview of MicroPower/Pascal I/O services (file system services, task-to-task communication, and device drivers). This chapter lists the supported devices and protocols, then describes the basic mechanisms of MicroPower/Pascal I/O, the I/O system architecture, and the available I/O interfaces. Subsequent chapters detail the device drivers and other system processes that provide I/O services. Addressed topics include the features and capabilities of each driver and its supported hardware device or protocol; build information particular to the driver; user interfaces to the driver, including (as applicable) the standard Pascal I/O interface, the Pascal support routine interface, and the device-level (send/receive) interface; the status completion codes returned by the driver; and the driver prefix file. The manual concludes with a guide to writing a device driver and a chapter describing useful driver macros and subroutines.

MicroPower/Pascal Language Guide

This manual presents the Pascal programming language and its extensions for use in microprocessor application programming. The manual covers Pascal's format and structure, basic concepts, data types, statements, procedures, functions, and specific MicroPower/Pascal extensions.

Part One of the manual describes the standard Pascal language, as defined by Niklaus Wirth, and the DIGITAL-created extensions not related to real-time programming. Part Two describes the real-time programming requests—predeclared programming procedures and functions that extend the capabilities of standard (sequential) Pascal to allow access to the real-time concurrent programming services of the MicroPower/Pascal kernel.

Chapter 1 provides an overview of the MicroPower/Pascal language and describes the structure of a MicroPower/Pascal program. Subsequent chapters in Part One include discussion of standard data types, declaration section statements, process invocation (creation) and procedure call statements, compilation units and independent compilation, predeclared data manipulation functions and procedures supplied with the MicroPower/Pascal software, the syntax and use of the predeclared input/output procedures, and the syntax and use of the MicroPower/Pascal attributes. Chapters in Part Two discuss real-time programming requests conventions, process management requests, semaphore management requests, queue semaphore management requests, ring buffer management requests, interrupt management requests, exception condition management requests, memory allocation and mapping requests, timer requests, and miscellaneous requests, including logical name assignments and creating mutual exclusion (mutex) structures.

MicroPower/Pascal Pocket Guide

This manual summarizes reference information for experienced MicroPower/Pascal users. The manual contains Pascal language syntax, Pascal real-time requests, MACRO-11 real-time requests, utility program commands, PASDBG symbolic debugger commands, and other useful reference information.

MicroPower/Pascal Run-Time Services Manual

This manual describes the entire MicroPower/Pascal programming environment and, in particular, the services and functions supplied by the MicroPower/Pascal kernel.

This manual provides details of the system services provided by the MicroPower/Pascal run-time system modules that make up a part of your application. Chapter 1 presents an overview of the MicroPower/Pascal run-time system. Kernel organization is described in general terms, as are primitive services and system processes. Chapter 2 describes processes in the MicroPower/Pascal context and system data structures. Chapter 3 gives detailed descriptions for each MACRO-11 primitive service request. Chapter 4 describes system configuration macros. Chapter 5 describes dynamic region allocation and shared regions. Chapter 6 describes MicroPower/Pascal exception processing. Chapter 7 describes kernel interrupt dispatching and ISRs.

MicroPower/Pascal Master Index

This manual is a summary of the indexes of the manuals in the MicroPower/Pascal documentation set.

MicroPower/Pascal Release Notes

This manual contains changes and restrictions to MicroPower/Pascal software not reflected in other documents. Chapter 1 lists restrictions you must observe when using the software and clarifies some obscure error conditions. Chapter 2 lists documentation corrections to and additions for the other manuals in your documentation set. Chapter 3 contains an annotated listing of the contents of your distribution kit.

PDP-11 MACRO-11 Language Reference Manual

This manual explains how to use the MACRO-11 relocatable assembler to develop applications in PDP-11 assembly language. The manual gives detailed descriptions of MACRO-11's features, including source and command string control of assembly and listing functions, directives for conditional assembly and program sectioning, and user-defined and system macro libraries.

PDP-11 Programming Card

This card is a compact reference for the PDP-11 instruction set numerical operation codes and mnemonics.

MicroPower/Pascal Installation Guides

A separate installation guide exists for each host operating system (RSX, RT-11, and VMS). Each guide includes the hardware configuration information you need to connect the host (development) system to the target (application) system hardware by means of a serial communication line.

The *MicroPower/Pascal-RSX Installation Guide* contains the information you need when installing MicroPower/Pascal-RSX software on your host system. Complete software installation procedures are given for magnetic tape, RK07, and RX02 distribution kit media.

The *MicroPower/Pascal-RT Installation Guide* contains the information you need when installing MicroPower/Pascal-RT software on your host system. Complete software installation procedures are given for RL02, RX02, and RX50 distribution kit media.

The *MicroPower/Pascal–VMS Installation Guide* contains the information you need when installing MicroPower/Pascal–VMS software on your host system. Complete software installation procedures are given for magnetic tape, RK07, and RL02 distribution kit media.

MicroPower/Pascal Messages Manuals

These manuals (one for RSX/VMS host users and one for RT–11 host users) list and describe the MicroPower/Pascal utility program messages and the compiler messages. Chapter 1 describes the order and format of messages, hard error conditions, and memory information. Chapter 2 lists messages issued by the utility programs COPYB, MERGE, MIB, PASDBG, and RELOC. In addition, Chapter 2 describes the cause of the error and the most likely recovery procedure. Chapter 3 details the compiler command line, compile-time, run-time, and compiler malfunction error messages. The command line messages are described in the same way as utility program messages. Chapter 4 describes the exception codes generated by run-time errors in the application and reported by PASDBG.

MicroPower/Pascal System User's Guides

These manuals (one for RSX/VMS host users and one for RT–11 host users) explain how to build programs by using the MicroPower/Pascal compiler and utility programs. The manuals also contain detailed information on linking, loading, and debugging a MicroPower/Pascal application.

Chapter 1 provides an overview of the MicroPower/Pascal development tools and run-time software and of the MicroPower/Pascal development process. Chapter 2 describes the operation of the command file generator, MPBUILD, used to build most MicroPower/Pascal applications. Chapter 3 explains how to build a MicroPower/Pascal kernel image by using the individual build utilities. Chapter 4 explains how to build system processes by using the individual build utilities. Chapter 5 explains how to build user processes by using the individual utilities. Chapter 6 discusses the use of supervisor-mode and user-mode shared libraries and explains how to build applications with instruction (I-) and data (D-) space separation. Chapter 7 describes methods of loading application images. Chapters 8 through 11 describe the compiler and the utilities MERGE, RELOC, and MIB individually. Chapter 12 describes the COPYB utility. Chapter 13 of the RSX/VMS version describes how to down-line load an application image into a target system over the DECnet/DDCMP.

Chapter 1
Introducing MicroPower/Pascal

This chapter describes features that distinguish MicroPower/Pascal from other development systems. The host/target environment, concurrent program design, and interprocess communication are unique MicroPower/Pascal features. These concepts are explained briefly in this chapter and are expanded later in this manual. Next, the chapter describes the MicroPower/Pascal software components that make up the development system and the stand-alone run-time support. The chapter concludes with an overview of the application build cycle.

1.1 What Is MicroPower/Pascal?

MicroPower/Pascal is a software development toolkit for creating real-time applications that run in dedicated target systems. Dedicated real-time MicroPower/Pascal applications can include instrumentation, materials handling, process control, and robotics. You develop MicroPower/Pascal applications with two systems: a host and a target. You use the full capabilities of the host operating system for compiling, building, and debugging. You tailor your application to contain only the specific MicroPower/Pascal operating system services it needs to run in the target system.

You code MicroPower/Pascal applications in a version of Pascal that includes extensions for real-time and multitasking support. You may also use MACRO–11 to code part or all of an application. (This manual provides an overview of the extended features that MicroPower/Pascal offers to enhance your real-time programming tasks. Although MACRO–11 is also available for programming applications, examples in this manual use Pascal.)

MicroPower/Pascal code is ROMable. Portions of the application that do not change value during the operation (code and read-only data, such as constants) can be placed in read-only memory in the target.

MicroPower/Pascal offers these advantages over traditional software development systems:

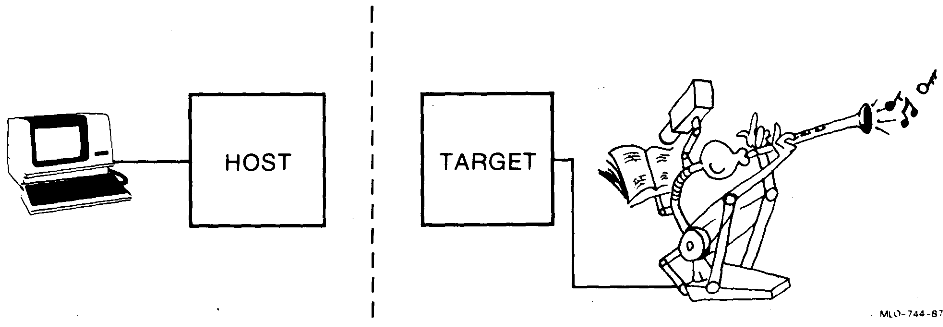
- Simpler real-time programs, with separate processes for each real-time event
- Better program structure, as each process concentrates on its task
- Shorter debug time, resulting from simpler code and symbolic debugging capability

1.1.1 Host/Target Development Environment

MicroPower/Pascal uses both a host system environment and a target system environment for application development. With MicroPower/Pascal, you develop fast, powerful stand-alone run-time systems, using the full capabilities of your host system to design, build, and debug the application while it runs on the target system. A serial line connects the host system and the console port of the target system so you can use the MicroPower/Pascal symbolic debugger on the host to control and track execution of the application program on the target system.

Your host system may be a Micro/RSX, RSX-11M/M-PLUS, RT-11-based, or VAX/VMS computer. Your applications execute on separate target Q-bus-based PDP-11 systems.

Each application is constructed specifically for its target system, with the exact set of operating system services it needs. This customized set of routines is called the kernel. When you include only the required system services in the modular, tailored kernel, the target run-time environment can minimize physical memory requirements and avoid the overhead of a traditional, general-purpose operating system. Chapter 2, Host and Target, provides a more detailed discussion of the host and target environments.



1.1.2 Concurrent Program Design

A MicroPower/Pascal application consists of user processes, system processes, and a kernel of required system services. A MicroPower/Pascal process is a program unit that may operate in parallel with other program units. MicroPower/Pascal applications typically contain multiple processes—a separate process is programmed for each task. The multiple processes in your program appear to execute simultaneously. Such execution is known as multitasking, or concurrent execution.

The concurrent approach to software design provides a simpler conceptual approach to solving real-time problems than does one that uses sequential programming techniques. Concurrent design encourages well-structured applications by dividing both a user program and the entire application into multiple tasks.

Concurrent processes make efficient use of the target system. Your concurrent design coordinates all the processes in one or several user programs. Such coordination keeps the size of the application to a minimum, since no general-purpose operating system is required to referee processes.

1.1.3 Interprocess Communication and Synchronization

Each process communicates with the others to synchronize execution. MicroPower/Pascal contains mechanisms for synchronizing executing processes and mutually excluding processes from shared resources. Semaphores are mechanisms for synchronization and mutual exclusion. Priority-based scheduling may also be used to synchronize process execution.

1.1.3.1 Semaphores

A semaphore is a data structure that implements interprocess communication and synchronization. Manipulated by two or more processes, a semaphore may be used to block the execution of one process until another process sends a signal to proceed. Semaphores may be used so their states at any given time correctly guide the responses of the entire application to external real-time events. Semaphores are also used to provide mutual exclusion for data or devices that must be protected from concurrent access.

MicroPower/Pascal defines three types of semaphores: binary, counting, and queue. Chapter 4 discusses semaphores and their use in detail.

1.1.3.2 Priority-Based Scheduling

You can determine the order in which processes gain access to the CPU by assigning a priority to each process. Processes that respond to critical external events typically receive higher priorities.

The occurrence of a real-time event, such as a signal from an external monitoring device, causes the target system hardware to interrupt the microprocessor. The microprocessor stops work and notifies the device driver process, which becomes ready to compete for control of the CPU in response to the interrupt. A relatively high priority ensures timely response to the real-time interrupt.

1.2 MicroPower/Pascal Software Components

Your MicroPower/Pascal software is packaged to correspond to the host system environment that you plan to use for application development. (Although applications may be developed in different host system environments, the choice of host system does not influence subsequent execution on the target run-time system.) You may choose from the following environments: MicroPower/Pascal–Micro/RSX, MicroPower/Pascal–RSX, MicroPower/Pascal–RT, and MicroPower/Pascal–VMS. For the MicroPower/Pascal–RT user, all necessary RT–11 operating system software elements are provided for the single-user environment. The other choices are layered products for multiuser environments.

MicroPower/Pascal software comprises two categories: application development tools and run-time system software. The following sections introduce major components of each category.

1.2.1 Development Tools

The following components are the major MicroPower/Pascal development tools:

- Extended Pascal compiler
- The automated build procedure: MPBUILD
- The application build utilities: MERGE, RELOC, MIB
- The PASDBG symbolic debugger

1.2.1.1 Extended Pascal Compiler

The MicroPower/Pascal compiler operates on a superset of the standard Pascal language to produce optimized code. MicroPower/Pascal enables you to use a high-level language to program complex real-time applications. An extended version of the ISO-Standard Pascal language, the MicroPower/Pascal language can handle interprocess communication, process synchronization, and multitasking (tasks normally associated with system-level operations). That capability limits the need for system services, since tasks normally handled by a traditional operating system may be handled by the processes themselves.

1.2.1.2 Automated Build Procedure

This program, known as MPBUILD (or MPBLD for RT–11 users), automates much of the process of building an application. Through a question-and-answer dialog, MPBUILD constructs a command file that, when executed, produces the files for loading into the target system and debugging.

1.2.1.3 Application Build Utilities

The three build utilities (MERGE, RELOC, and MIB) let you build an application memory image for the target system. These build utility programs transform compiled or assembled object modules into a loadable memory image. MERGE combines individual object modules into a single, merged object module. RELOC transforms virtual addresses into actual memory addresses, allocates memory, and sorts programs into read-only and read-write sections. MIB creates the final memory image.

Creating an application is a multistep, multiphase process. Each component of the final application (the kernel, system processes, and user processes) goes through the build utility cycle. This modular approach makes testing and debugging easier and simplifies the job of updating and expanding applications. Although you may run the build utilities directly, you usually use MPBUILD to invoke them indirectly when constructing your application.

1.2.1.4 PASDBG Symbolic Debugger

The debugger resides on the host development system and communicates over a serial line with a debugger service module in the application. You use interactive debugging commands to track the execution of your application and to locate and correct bugs. Because it is a symbolic debugger, PASDBG permits debugging using the Pascal source program's variables, scopes, labels, and identifiers. Information is presented to the programmer in the same form as described to the compiler in the source program.

1.2.2 Run-Time Support

Run-time software consists of the kernel and system processes, including device drivers, the Ancillary Control Process (ACP), and network support.

1.2.2.1 MicroPower/Pascal Kernel

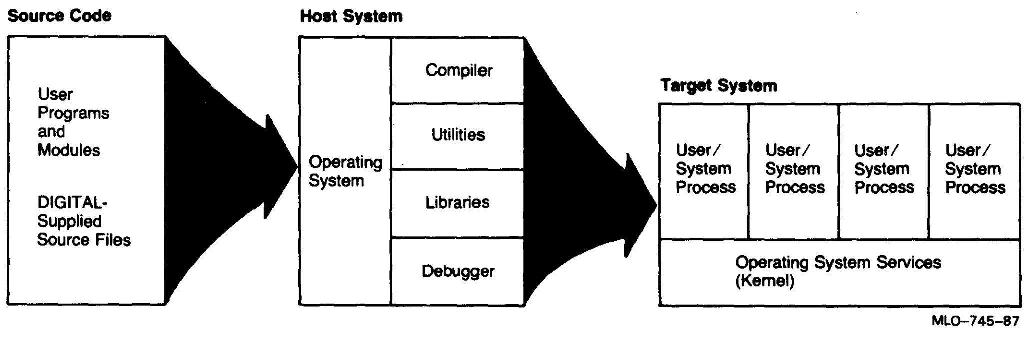
The kernel is the modular executive portion of the target run-time system. The kernel supplies the target system with basic services, handles interrupts and traps, and schedules process execution. On request from processes, the kernel provides the interprocess synchronization and communication services, called primitives, that let processes interact in a real-time environment.

1.2.2.2 System Processes

System processes include driver processes, the ACP, and the network service process (NSP). MicroPower/Pascal provides precompiled driver processes that interface between your application and a variety of DIGITAL devices. These drivers are supplied in object libraries. You include in your application only those drivers needed for your target system.

The ACP interfaces between user processes and device handlers. The ACP provides file system support that lets any process create, maintain, and delete file directory entries on target mass-storage devices. The ACP and NSP let you create network links for task-to-task communication and provide DECnet task-to-task end-node support on Ethernet. Figure 1–1 provides a synopsis of application building, including source code, host development tools, and the final target application composed of the kernel and multiple processes.

Figure 1–1: Constructing a MicroPower/Pascal Application



1.3 MicroPower/Pascal Development Cycle

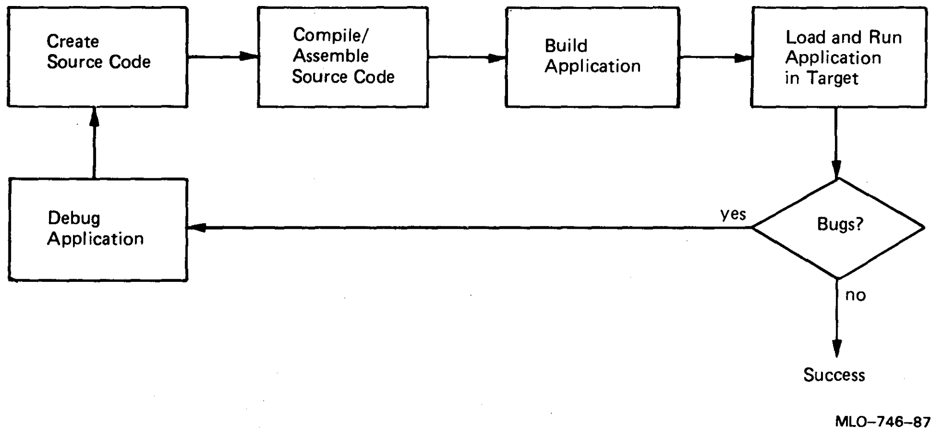
The following list summarizes the steps required to develop a MicroPower/Pascal application:

1. Design the application and write Pascal or MACRO–11 source code.
2. Compile and/or assemble the source programs into object code.
3. Use MPBUILD or the MicroPower/Pascal build utility programs on the host to build the full application by linking the output from step 2 with kernel functions and services.
4. Load the completed application program into the target system, using PASDBG.

5. Test and debug the application executing in the target under control of the PASDBG symbolic debugger program residing in the host system.

Figure 1–2 shows the development cycle of a MicroPower/Pascal application.

Figure 1–2: MicroPower/Pascal Application Development



Chapter 6 explains the development cycle in greater detail.

Chapter 2
Host and Target

Two systems are used in MicroPower/Pascal application development: the host system and the target system. You develop application programs on a PDP–11 or VAX host system and load the resulting memory image into a low-end, PDP–11-family target system for debugging and execution.

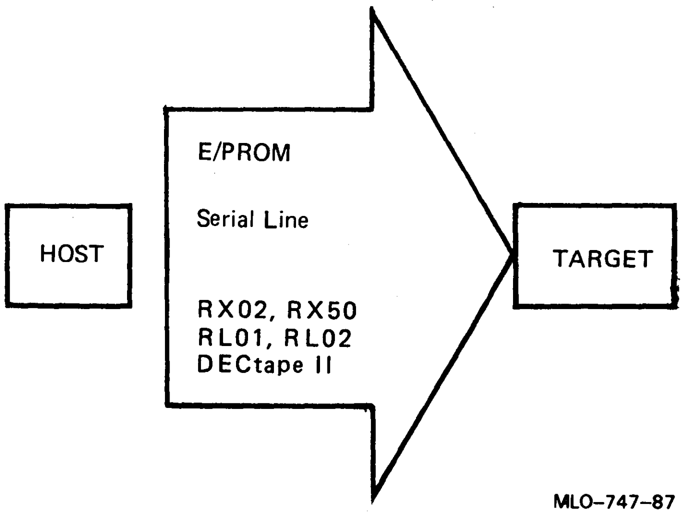
This chapter provides an overview of the host/target development environment and the target run-time environment. The host operating systems available for MicroPower/Pascal development are described in terms of hardware requirements, options, and features available for each. This information is of interest to you if you are to select the host operating system for MicroPower/Pascal development or are considering migrating from one host system to another. If you want more specific hardware information, consult the MicroPower/Pascal software product description for the host operating system that interests you. Target system characteristics are also discussed in this chapter.

2.1 The Host System

You use the facilities of your host system to design and build your application software. Application software is tested and debugged in the run-time environment under control of PASDBG, which resides on the host. During the debugging stages of application software development, an asynchronous serial communications line connects the target system to your host system.

You use the serial line to transfer applications to target system memory when debug support is required or when you want to use PASDBG to load the application without debug support. Other means are available for applications that do not require debug support. You may include a bootstrap in the application image and place it on disk or TU58 DECtape II for booting from a target device. Your application may also be placed into programmable read-only memory (PROM). See Figure 2–1.

Figure 2-1: Transferring Application to Target System Memory



2.1.1 General Host System Requirements

You need an asynchronous serial line for down-line loading or debugging a MicroPower/Pascal application on each target system. For required host/target serial line unit characteristics for down-line loading or debugging MicroPower/Pascal applications, see the MicroPower/Pascal installation guide for your host system.

2.1.2 Your Micro/RSX or RSX-11M/M-PLUS Host System

RSX offers a multiprogramming environment for application development on PDP-11 systems. You use the capabilities and utility programs provided by your host system to develop your applications, then transfer the application to the target for debugging and execution. RSX host environments may include the EDI line-oriented editor and the EDT character-oriented editor to create and modify source programs, as well as the LBR (Librarian) and PIP (Peripheral Interchange Program) utility programs.

The minimum host system requirements include the following:

- Any Micro/RSX system with at least 256K words of memory running on a MicroPDP-11 with a 10MB (or larger) disk, RSX-11M system with at least 124K words of memory, or any RSX-11M-PLUS system with at least 256K words of memory
- An asynchronous serial line for host/target system communication

For additional information, refer to the Micro/RSX or RSX-11M/M-PLUS software documentation for your host system.

2.1.3 Your RT–11 Host System

RT–11 is an interactive, single-user operating system. MicroPower/Pascal users who choose to work in the RT–11 environment are given all the RT–11 software components that are necessary to develop and run target applications. Those components include the extended memory (XM) monitor and the following utility programs:

- KED and EDIT—Editors for creating and modifying text files. EDIT includes character- and line-oriented commands. KED is a keypad editor for use with video terminals.
- LIBR—RT–11 librarian utility, which creates and modifies library files of commonly used object modules.
- PIP—RT–11 peripheral interchange program, which lets you transfer files from one area of the system to another or delete files.
- DIR—RT–11 directory program, which displays the file directory of any file-structured device.

The *RT–11 System Utilities Manual* and the *PDP–11 Keypad Editor User's Guide* describe these utilities and their use. See the *Guide to RT–11 Documentation* for information on these and other RT–11 manuals provided in your documentation set.

The host system hardware generally consists of an LSI–11 or PDP–11 processor with memory-management hardware, a console terminal, mass-storage devices, and peripheral devices, such as a printer.

The minimum host system hardware requirements for developing MicroPower/Pascal–RT application software include the following:

- Any PDP–11 or LSI–11 CPU with EIS, memory-management hardware, and a line time clock
- 128K bytes of memory
- An RL02 or RX02 drive for media installation and one of the following random-access, mass-storage device drives (RK06, RK07, RL01, or RL02)

  or

  any MicroPDP–11 system that includes a 10MB (or larger) Winchester disk and an RX50 diskette drive
- Two DLV11-compatible serial line units for the console terminal and the host/target system communications
- A VT52, VT100, VT220, VT240, LA34, LA36, or LA120 console terminal

2.1.4 Your VAX/VMS Host System

VAX/VMS is a comprehensive multiuser operating system. MicroPower/Pascal is installed as a layered product on VAX/VMS systems. That arrangement permits you to use your VAX/VMS system, with its fast execution speed and large mass-storage capabilities, to develop applications for your PDP–11-based target run-time system. VAX/VMS offers powerful text editors for creating and modifying your source code and Digital Command Language (DCL) for communication with the VMS operating system.

VAX/VMS host system development requirements include the following:

- A VAX 11/750, 11/780, 11/782, 11/785, or 8600 configuration that contains an RL02 or RK07 disk drive or a 9-track 1600 bpi magnetic tape (PE) and two megabytes of memory

  or

  a VAX 11/730 that contains both an RL02 and an RA80 disk drive and two megabytes of memory
- A VAX/VMS-supported asynchronous serial line (DZ11, DZ32, or DMF32) for host/target system communication
- At least 10,000 free blocks on system disk for installation, after which 5500 blocks are used for storage of MicroPower/Pascal files

For additional information, refer to the VAX/VMS software documentation for your host system.

2.2 The Target System

MicroPower/Pascal target systems function in dedicated real-time environments such as:

- Computer-assisted manufacturing
- Ethernet server networks
- Materials handling
- Monitoring and testing
- Process control
- Robotics

MicroPower/Pascal supports application execution on component and packaged microcomputer systems using SBC–11/21, SBC–11/21–PLUS, LSI–11, LSI–11/2, LSI–11/23, LSI–11/73, KXT11–C (IOP), KXJ11–C (IOP), and LSI–11/23–PLUS processors. PDP–11/03, PDP–11/23, PDP–11/23–PLUS, PDP–11/73, PDP–11/83, MicroPDP–11, MicroPDP–11/53 (RAM only), and CMR21 packaged systems are also supported.

The following target system hardware is required:

- Memory consisting of any combination of RAM and PROM, including at least 4K bytes of RAM
- DLV11-compatible serial line unit for down-line loading and debugging application software
- Interface hardware for your own target devices

2.2.1 Target System Memory

Target system memory may be unmapped or mapped. An unmapped memory system has a total address space of 64KB. A mapped memory system increases the supported physical address space to 256KB with 18-bit addressing and 4096KB with 22-bit addressing. In a target system with memory-management hardware, memory can be split into independent sections known as virtual address spaces. The largest possible virtual address space is 64KB. Each virtual address space in the mapped target system contains one process family—one static process and its dynamic process descendants. (See Chapter 3 for a discussion of static and dynamic processes.) Details of memory mapping are presented in the *MicroPower/Pascal Run-Time Services Manual* and the *Microcomputer and Memories Handbook.*

2.2.2 Supported Devices

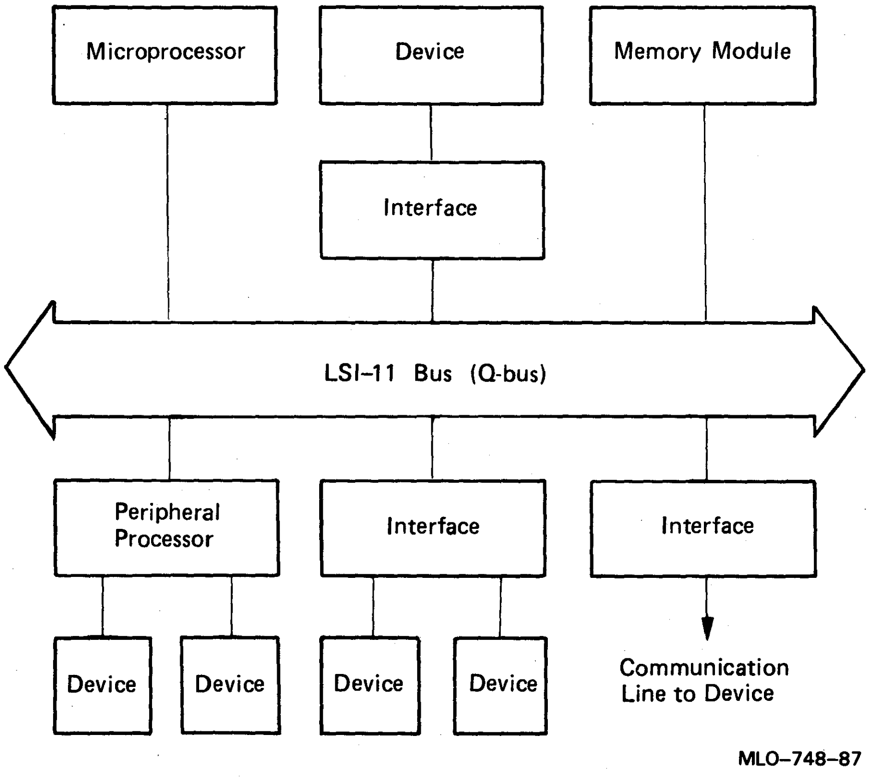
The following DIGITAL devices and interfaces are supported by device-driver software supplied with MicroPower/Pascal:

- ADV11-C, AAV11-C, and AXV11-C A/D and D/A modules.
- DELQA-M Ethernet controller, supported in DEQNA mode only.
- DEQNA Ethernet controller.
- DHQ11-M cost-reduced DHV11, dual height.
- DHV11 serial line unit.
- DLV11, DLV11-E, DLV11-F, and DLV11-J serial line units.
- DPV11 synchronous serial line unit.
- DRV11, DRV11-B, DRV11-J parallel line units.
- DZQ11-M cost-reduced DZV11, dual height.
- DZV11 serial line unit.
- FALCON, FALCON-PLUS parallel port.
- FPJ floating-point accelerator—for KDJ11.
- IEQ-11 IEEE-488 bus interface.
- KEF11, FPF11 floating-point options—for LSI-11/23 microcomputers.
- KEV11 EIS/FIS arithmetic option—for LSI-11 or LSI-11/2 microcomputers.
- KWV11-C programmable real-time clock.
- KXT11-A2 Macro-ODT ROM option, which must be installed on the target system when you are debugging FALCON applications, or KXT11-A5 Macro-ODT ROM, which must be installed on the target system when you are debugging FALCON-PLUS applications.
- KXT11-CA and KXJ11-CA single-board computers' parallel, asynchronous, and synchronous ports, DMA capabilities, and 2-port RAMs.
- Line clocks for the LSI-11, LSI-11/2, LSI-11/23, FALCON SBC-11/21, and FALCON-PLUS.
- MRV11-C, -D PROM module.

- MSV11-D, -L, -P, -S RAM module.
- MXV11-A or MXV11-B multifunction module, which includes PROM, RAM, two serial lines, and 50/60 Hz clock. If you are bootstrapping your application software from RL01 or RL02 disk, DECtape II cartridge, or RX02 diskette, you must use the MXV11-A2 or MXV11-B2 bootstrap ROM option, respectively, with the MXV11-A or MXV11-B.
- RLV12 RL01/RL02 disk cartridge controller.
- RQDX1, RQDX2, RQDX3 controller for RD3n, RD5n Winchester disk and RX50 diskette drives used in the MicroPDP-11 target system.
- RX33 flexible disk drive (with RX50 media only).
- RXV21 dual-density diskette.
- TK50 streaming tape cartridge (non-file-structured).
- TU58 DECtape II cartridge tape.

The processor, memory, and devices attach to the LSI-11 bus. Figure 2-2 shows a possible LSI-11 bus configuration for MicroPower/Pascal target applications.

Figure 2-2: Interfaces to LSI-11 Bus (Q-bus)



2.2.3 Describing the Target System

The system configuration file describes the hardware of the target system and its memory characteristics. The configuration file is one of your inputs to the build cycle during application development. The configuration file is used to tailor the memory image in general and the kernel in particular.

The configuration file consists of a series of MACRO–11 calls that define certain hardware characteristics, including the type of target processor, mapped or unmapped memory, floating-point hardware, device information, and kernel requirements. MicroPower/Pascal provides several prototype configuration files that describe typical configurations. You may choose a configuration file to use as is or to edit. Configuration files are discussed in more detail in Chapter 6, Development Cycle Overview.

Chapter 3

A MicroPower/Pascal Process

The fundamental unit of a MicroPower/Pascal application is a process. Each application is composed of user and system processes. This chapter describes the process concept, differentiates between static and dynamic processes, and discusses the types of processes that may make up a MicroPower/Pascal application. The chapter then shows you how to declare, create, and refer to processes in a Pascal environment and provides an example that uses the language features that are introduced. The chapter concludes with a discussion of the kernel's role in process synchronization and scheduling—the basis for the concurrent-programming concepts and techniques discussed in Chapter 4.

3.1 Process Definition

A process is a program unit that operates in parallel with other program units. The application is organized as if to perform many activities at once. Although processes appear to execute simultaneously, only one process has control of the CPU at any time. That is, the concurrency is virtual, not actual. A process executes a particular sequence of instructions to perform a distinct task. (The same set of instructions might be executed by several processes.) Processes both compete and cooperate with one another for control of the CPU and processes share other common resources (for example, data areas and peripheral devices) while appearing to execute simultaneously. Processes are supported by a software kernel that supplies basic system services to the concurrent processes (see Section 3.3).

3.1.1 Static Versus Dynamic Processes

A typical MicroPower/Pascal application includes several independent processes, both static and dynamic. A static process exists in the application after initialization; that is, a static process is always present after power is turned on or system-reset processing is completed. A static process corresponds to a Pascal program. Your MicroPower/Pascal application contains one or more user static processes and at least one system static process. A dynamic process is created by the action of another process during execution of the application. Each process family in the application consists of one static process and all dynamic processes created by the static process. Processes may be referenced within the static-process family to which they belong and, under certain conditions described later in this chapter, by another static process.

3.1.2 System Processes

System processes are provided as part of the MicroPower/Pascal software product. They furnish commonly required services and are, in general, privileged processes.

System processes that are provided include device drivers for particular devices and interfaces, including DLV11, DRV11, RX02, TU58; the ancillary control process (ACP), which provides file system support; and the network service process (NSP), for communications support.

3.1.3 Process Types

Processes may be categorized into four types: device-access, privileged, driver, and general. In unmapped systems, the entire application exists in a mutually accessible address space, and mapping types are not applicable. In mapped systems, the process types determine which areas of memory processes can access.

- Device-access processes access the I/O page, which contains I/O device addresses. A device-access process can manipulate the control and status registers (CSRs) and data buffers of target devices, such as a DLV11 interface.
- Privileged processes access the kernel data area, where data structures such as semaphores and process control blocks reside, and the I/O page.
- Driver processes, like privileged processes, access both kernel data area and the I/O page. Driver processes also contain interrupt service routines (ISRs), independent sections of code designed to execute in response to interrupts from target devices.
- General processes do not access the I/O page or kernel data area directly.

3.1.4 Process Components

Each active process within a MicroPower/Pascal application consists of a process control block (PCB), associated data structures, the process stack, and an instruction sequence.

3.1.4.1 Process Control Block

A PCB is a kernel data structure that identifies the process to the kernel. The kernel creates a PCB whenever a process is created. Information in the PCB describes a process completely and includes:

- Process name (optional)
- Process priority
- Process state
- Process type
- Saved process context

The *MicroPower/Pascal Run-Time Services Manual* contains complete information on the content and use of the PCB.

3.1.4.2 Data Structures for the Process

Two types of data structures are associated with every process. Although created by the process, system data structures reside in the kernel memory and include semaphores and ring buffers. The target system's configuration file specifies the amount of memory needed for system data structures.

Other data structures (for example, arrays, records, and integers) are local to the process and reside in data areas associated with the process. To provide those data areas, each process must have its own memory associated with it. Section 3.4.1 explains how to allocate process data areas by using the DATA_SPACE attribute.

3.1.4.3 Process Stack

The process stack is an area of memory associated with each process for the creation of data structures. Section 3.4.1 explains how to allocate process stack area from the process data area by using the STACK_SIZE attribute.

3.1.4.4 Instruction Sequence

Each process has an associated instruction sequence. That code may be placed in ROM.

3.2 How Static Processes Run Concurrently in Your Target Application

When the target system is powered up and the application loaded into memory, the kernel readies each static process for execution. Then, an initialization procedure, if any, runs for each static process. Named system data structures, such as semaphores and ring buffers, are typically created in the initialization procedure. You declare this procedure in the source code for each static process and identify the procedure with the INITIALIZE attribute:

```
[INITIALIZE] PROCEDURE Do_first;
```

All initialization procedures within the total application execute first, before the main program of any static process is activated. In that way, all needed semaphores and other structures are created and initialized before any other process can start. Thus, the necessary synchronization mechanisms are in place when the static processes begin to compete for control of the CPU.

Each process may have a termination procedure. Execution shifts to this procedure when the process is stopped. (A process can be stopped by the STOP request, issued by itself or by another process, as the result of an unhandled exception condition, or by normal completion.) You specify the termination procedure in the source code and identify it with the TERMINATE attribute:

```
[TERMINATE] PROCEDURE Do_last;
```

Typically, the termination procedure performs any general clean-up steps, deletes kernel structures created by the process, and releases any allocated resources, such as message packets.

3.3 Process Scheduling and Synchronization

Processes are supported by a modular executive, known as the kernel. The kernel provides rudimentary, or primitive, services on demand (as well as other services implicitly). The kernel performs all process synchronization in response to requests by processes.

3.3.1 Kernel's Role

The kernel responds to demands for service but has no independent control in areas other than scheduling and interrupt/trap handling. For example, a process may request the kernel to assign it a packet in kernel-controlled memory or to change the value of a semaphore.

The kernel scheduler determines which process gains control of the processor after a significant event, based on the priorities of the currently eligible processes and the running process. (For a definition of significant event, see Section 3.3.2.)

The kernel supplies basic system services, such as:

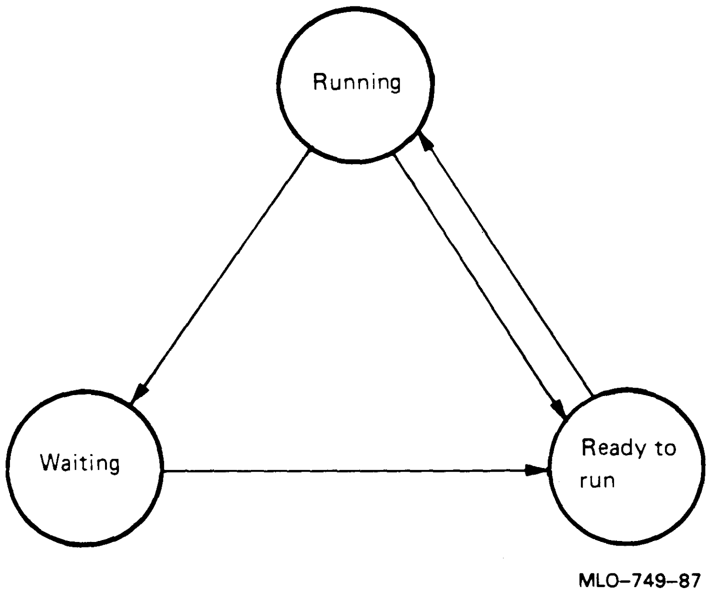
- Exception dispatching
- Interprocess communication, including primitives required to create, operate on, and destroy queue semaphores and ring buffers, and primitives to operate on message packets
- Interrupt dispatching and trap handling
- Process creation and deletion

The *MicroPower/Pascal Run-Time Services Manual* contains complete information on services provided by the kernel.

3.3.2 Process States and State Transitions

As shown in Figure 3–1, an existing process may be in one of three state categories at any one time—running, ready-to-run, or waiting for an event. The arrows represent transitions from one state category to another.

Figure 3–1: State Category Diagram



Every existing MicroPower/Pascal process is in one of seven states at any time. Each state belongs to one of the state categories shown in Figure 3–1. Ready and waiting processes may be suspended explicitly and must be explicitly resumed before reentering the ready or waiting state they were in. Separate wait states exist for processes waiting for an exception-handling process to execute. (In MicroPower/Pascal, an abnormally terminated process is considered to be in the inactive state. The inactive state is not discussed in this manual. See the *MicroPower/Pascal Run-Time Services Manual* for information about this auxiliary state.)

A process in the run state has control of the processor. Only one process may be running at any one time. A process leaves the run state when it gives up control of the CPU or is preempted by another process. A ready-to-run process is eligible to run when the processor is available and the kernel scheduler determines that it may run, based on the process's priority relative to other eligible processes. The MicroPower/Pascal ready-to-run process states are:

- Ready active
- Ready suspended

A process in a wait state can enter a ready-to-run state only after an event that it has been waiting for occurs. The MicroPower/Pascal wait states are:

- Wait active
- Wait suspended
- Exception-wait active
- Exception-wait suspended

When a process is created, it is in the ready-active state. Events occur to cause a process in one state to undergo a transition to another state. Such an event might be waiting on a semaphore or waiting for a device. A process waiting for an event is known to be blocked. A blocked process is unblocked when the signal or resource for which it is waiting is provided. Unblocking implies a transition from either the wait-active or wait-suspended state to the corresponding ready state.

Table 3–1 summarizes the seven states and their characteristics.

Table 3–1: MicroPower/Pascal Process States

State Category	State Name	Description
Running	Run	This process is currently executing. Only one process may be in the run state at a time.
Ready to run	Ready active	Eligible to run. When the running process gives up control or is preempted, the highest-priority ready-active process gains the CPU.
	Ready suspended	Ready but has been explicitly suspended by a suspend request and needs to be "resumed" before reentering the ready-active state.
Waiting	Wait active	Waiting for a particular event or resource. When unblocked, moves to ready-active state.
	Wait suspended	Waiting and suspended. "Resume" will return it to the wait-active state. If unblocked before resumed, returns to the ready-suspended state.
	Exception-wait active	Waiting for action of an exception-handling process. Returns to ready-active state when exception handler done.
	Exception-wait suspended	Explicitly suspended while in exception-wait active state.

Any event that causes a process to move into the ready-active state or out of the run state is known as a significant event. The occurrence of a significant event invokes the kernel scheduler. Whenever a process is eligible to take control of the CPU (joins the ready-active state) or whenever the running process becomes ineligible (leaves the run state) the kernel scheduler takes control. The scheduler compares the priority of the running process, if one is running, with that of the highest-priority ready-active process. If the priority of the highest-priority ready-active process is higher than that of the currently running process, that process moves into the run state, thereby gaining control of the CPU.

Figure 3–2 shows state changes that invoke the kernel scheduler and that consequently may cause control of the CPU to shift from one process to another. Figure 3–3 shows changes in and out of the run state, and Figure 3–4 shows all state changes.

Figure 3–2: State Changes That Invoke Kernel Scheduler

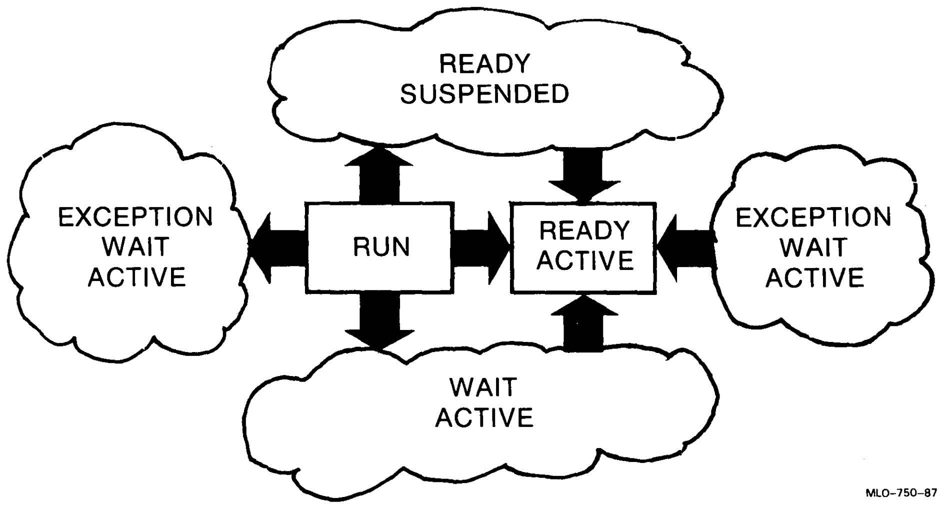
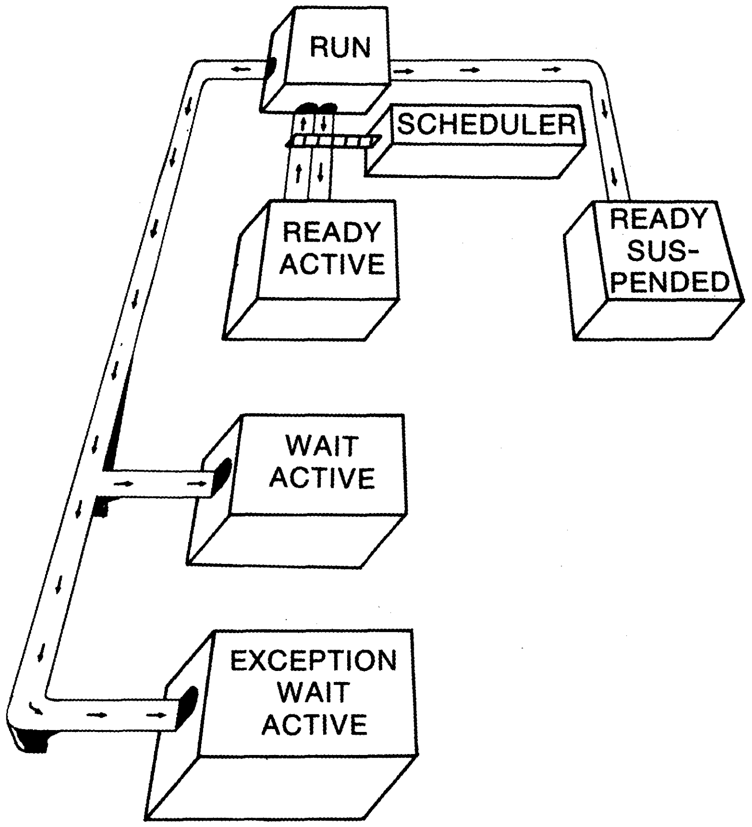
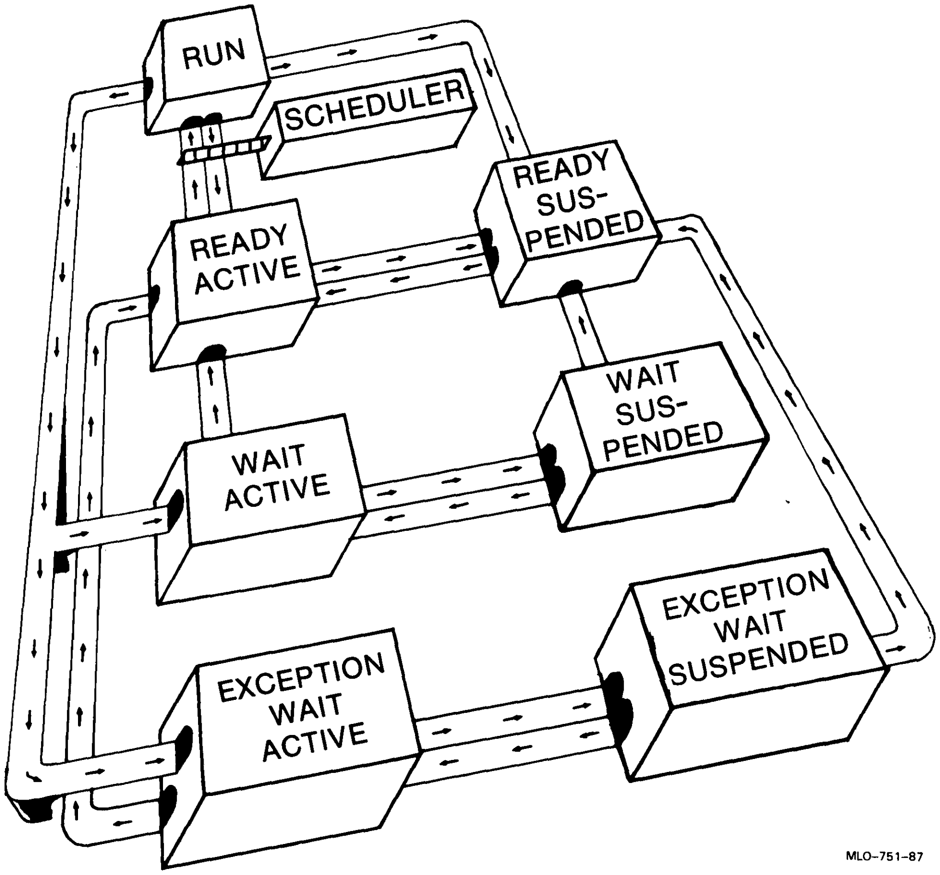


Figure 3–3: State Changes Involving the Run State



MLO-752-87

Figure 3–4: Summary of All State Changes



3.4 Using Processes in Your Application

The following sections describe how to declare, invoke, and reference MicroPower/Pascal processes within your application.

3.4.1 Declaring a MicroPower/Pascal Process

A process declaration consists of the process header and block. The header may contain attributes, the reserved word PROCESS (or PROGRAM, if it is the static process), and the process identifier or compile-time name. The header is followed by process declarations and the process statements delimited by the reserved words BEGIN and END.

The following example shows a process declaration:

```
PROCESS  RACE;

BEGIN
WRITELN ('This is the process Race.');
END;
```

When you declare a process, MicroPower/Pascal allows you to specify attributes that control certain properties associated with the process. If an attribute is unspecified, Pascal may supply a default value. Three attributes are specified in the following program header (note that CAR is a static process):

```
[DATA_SPACE(2000), STACK_SIZE(400), PRIORITY(1)] PROGRAM CAR;
```

These attributes are described in the following sections.

3.4.1.1 DATA_SPACE Attribute

This attribute may be specified for static processes only. The DATA_SPACE attribute specifies the amount of storage space to be used for all dynamically allocated program (static process) data and for the stack space for all dynamic processes in this family.

DATA_SPACE is a keyword included in the Pascal main program header. All process stacks, including the main program's stack, are allocated from the data space assigned to the main program. Thus, the DATA_SPACE size must be as large as the sum of the greatest number of stack sizes that can exist at any time. The data space also includes the heap, an area in which memory is allocated by means of the Pascal NEW procedure during execution for dynamic data structures.

To specify DATA_SPACE, include the keyword along with a constant value in the program header.

3.4.1.2 STACK_SIZE Attribute

This attribute specifies the amount of stack space that is used for program and process stacks. This space is allocated from the storage space declared by the DATA_SPACE attribute, which may be specified in program declarations and process declarations. For each process, data structures, such as arrays, constants, and variables, are allocated in data areas associated with the process. Therefore, each process must have an amount of memory associated with it for the creation of structures. For help in determining the appropriate stack size to declare for a process, see Chapter 10, Attributes, in the *MicroPower/Pascal Language Guide.*

3.4.1.3 PRIORITY Attribute

This attribute may be specified in program declarations and process declarations. The scheduler, a component of the kernel, compares priorities to determine which process gains control of the CPU.

You assign process priorities to specify the relative importance of MicroPower/Pascal processes. Certain processes in your application may be more time critical than others. A process that responds to real-time events may need to execute at a higher priority than a process that updates a terminal screen display. You assign relative priorities by including a priority attribute at process declaration or creation time. A process priority may be any integer between 0 and 255. Whenever two processes of different priorities are eligible to control the target CPU, the higher-priority process takes precedence.

The following example creates the process CAR and includes priority as a parameter:

```
CAR (PRIORITY := 2);
```

Generally, you should not use priority as a mechanism for synchronizing the execution of two processes. If the higher-priority process blocks, the synchronization is no longer correct. Priorities should be assigned to processes only to enhance the efficiency of the application.

3.4.1.4 NAME Attribute

You may assign a run-time name to a process by including the NAME attribute in the process declaration. Alternatively, you may assign a run-time name to a process by specifying a name parameter in the invocation statement.

Run-time names must consist of six characters within parentheses. Shorter names must be padded with trailing blanks. This example includes the NAME attribute:

```
[NAME ('JALOPY')] PROCESS CAR;
```

3.4.2 Invoking a MicroPower/Pascal Process

You invoke, or create, a process with its compile-time name followed by a parameter list. You may specify a run-time name, a process descriptor, both a run-time name and a process descriptor, or neither. The process invocation associates the actual parameters in the list with the formal parameters in the heading of the process declaration. (The predeclared parameters NAME, DESC, STACK_SIZE, and PRIORITY establish the particular identification and environment for each invocation of a process.) Each time a process is invoked, a replication of it is created, using the data specified by the actual and predeclared parameters in the invocation statement. The following program example invokes Process CAR four times:

```
[SYSTEM(MICROPOWER), DATA_SPACE(3000)] PROGRAM RACE;{static process}

VAR

  Car_desc, Auto_desc : PROCESS_DESC;

PROCESS CAR; {declares dynamic process CAR}

BEGIN
.
.
.
END;
```

```
BEGIN
❶ CAR;                                          {creates dynamic process }
❷ CAR (DESC := Car_desc);                       {creates dynamic process }
❸ CAR (NAME := 'ROBERT');                       {creates dynamic process }
❹ CAR (DESC := Auto_desc, NAME := 'JUDITH');{creates dynamic process }
END.
```

In the previous program example, the process block CAR is created four times; that is, four separate invocations of the process block CAR may run. The name and descriptor information is used to identify a particular invocation of the process to the kernel. The first process ❶ has no specified parameters and no default run-time name. (A NAME attribute was not specified in the process declaration.) No other process may directly control the execution of ❶; that is, it may not be suspended, resumed, or stopped directly by another process through a kernel request, since the process ❶ cannot be controlled either by name or by descriptor. The second process ❷ may be identified by a process descriptor (the variable Car_desc) which is supplied as a parameter in the invocation statement. The process descriptor allows a process to be referenced directly by another process within its static process family. The third process ❸ may be identified by its run-time name, ROBERT, supplied as a parameter in the invocation statement. This process may be referenced both within a static process and across static process boundaries. The fourth process ❹ has two parameters, name and descriptor, and may be referenced by either.

3.4.3 Referencing MicroPower/Pascal Processes Within Your Application

Because run-time names and descriptors identify processes to the kernel, you can manipulate processes by referencing either the descriptors or the run-time names. Commands to manipulate processes include such requests as STOP, SUSPEND, and RESUME—for example:

```
STOP (NAME := 'ROBERT');
```

Inserting this line in the previous example will stop the execution of the Process CAR identified by the run-time name "ROBERT" ❸.

A process with a run-time name can be referenced both within its own static process and by another static process within the application. However, descriptors are known only within the static process in which the descriptor variable is declared, unless the value of the descriptor is passed to another process by means of the SEND request or the descriptor is shared between two processes by means of the GLOBAL and EXTERNAL attributes.

3.4.4 Declaring, Invoking, and Referencing MicroPower/Pascal Processes

The following example illustrates how processes may be declared, created, and referenced:

```
[SYSTEM(MICROPOWER), DATA_SPACE(5000)] PROGRAM STATIC_PROCESS_1;

VAR
P1_desc,
P1_B_desc,
P2_desc,
P2_D_desc : PROCESS_DESC;
```

```
PROCESS P1;
BEGIN
.
.
.
END;

[NAME ('P2P2P2')] PROCESS P2;
BEGIN
.
.
.
END;

BEGIN

❶ P1;   { Cannot reference this process directly. }
❷ P1 (DESC := P1_desc);
❸ P1 (NAME := 'AAAAAA');
❹ P1 (DESC := P1_B_desc, NAME := 'BBBBBB');
❺ P2;   { Can reference this process by name. }
❻ P2 (NAME := 'CCCCCC');

  STOP (DESC := P1_desc);
  STOP (NAME := 'AAAAAA');
❼  STOP (NAME := 'P2P2P2');
❽  STOP (NAME := 'CCCCCC');

  WRITELN ('Done');
  END.
```

This example declares two dynamic processes: P1 and P2. P1 is declared without a name attribute; P2 is given the default name "P2P2P2." The first invocation of P1 ❶, without a run-time name or descriptor parameter, cannot be suspended, resumed, or stopped either within this static process or by another static process. The second invocation of P1 ❷, with a descriptor "P1_desc," may be stopped by a process within its own static process (in this case, the program itself) because it can be identified by its descriptor. The third invocation of P1 ❸, with the run-time name "AAAAAA," may also be referenced because of its run-time name "AAAAAA." The fourth invocation of P1 ❹ may be referenced by either its run-time name or its descriptor.

P2 has been declared with a name attribute. An invocation without a name parameter causes that invocation to have the default name "P2P2P2." The STOP procedure ❼ causes ❺ to stop.

The run-time name "CCCCCC" ❻ can be stopped by STOP ❽.

Chapter 4

MicroPower/Pascal and Concurrent Programming

This chapter describes Pascal language extensions for process synchronization and communication. It begins with a general discussion of concurrent programming and then shows how to synchronize processes by using data structures known as binary and counting semaphores. Other synchronization methods are also briefly discussed. The chapter continues with a discussion of process communication using ring buffers and queue semaphores. Interrupt-handling and exception-handling techniques are discussed. The chapter concludes with an overall design strategy for a hypothetical real-time problem.

4.1 Concurrent Application Design

The efficiency and compactness of MicroPower/Pascal applications result from a concurrent program design that eliminates the need for a traditional operating system in the target. Concurrent programming structures an application into independent parts designed to execute simultaneously. These parts compete for control of the target CPU and other target system resources. MicroPower/Pascal contains mechanisms for synchronizing the executing processes and mutually excluding processes from shared resources. Semaphores (see Section 4.2) and ring buffers (see Section 4.3) are data structures that implement process synchronization.

The concurrent design of your application coordinates all processes in a program. This keeps the size of the application to a minimum, since no general-purpose operating system is needed to coordinate processes.

The following three conditions must be built into any real-time system of concurrent processes:

- Processes that must respond to an event should have priority over less important processes.

  When you design a MicroPower/Pascal application, you assign a priority to each process to determine the relative importance of all processes in the application.

- Shared resources, such as devices and data, must be protected by mutual-exclusion mechanisms.

  Access to each shared resource in the application must be mutually excluded from competing processes. Binary semaphores and mutual exclusion structures establish mutual exclusion between processes, thus protecting the shared resource.

- Real-time interrupts must be handled immediately. Interrupted processes must be guaranteed safe storage until they resume later.

  MicroPower/Pascal provides two mechanisms for handling interrupts. These mechanisms are discussed in Section 4.4.

4.2 Process Synchronization

The kernel performs process synchronization in response to requests from processes. Your processes request kernel services to create, operate on, and destroy semaphores and ring buffers.

The following sections discuss process synchronization and mutual-exclusion techniques. Mutual exclusion protects shared resources, preventing simultaneous access that could destroy the integrity of the shared resource.

4.2.1 Using Semaphores for Process Synchronization

Operations on semaphores control the execution of concurrent processes. A semaphore is a data structure that is manipulated by two or more processes. Any process can create a semaphore by issuing a request to the kernel. A semaphore is used to block execution of one process until another process sends a signal to proceed. Processes can wait for specific events, such as the ringing of alarms, by waiting on a semaphore that is signaled by the key event. If more than one process is waiting on the semaphore, the highest priority or oldest process unblocks and proceeds. The remaining processes are still blocked on the semaphore.

You may create three types of semaphores: binary, counting, and queue. (Queue semaphores are discussed in Section 4.3.2.) A binary semaphore is a gate variable that allows one process to proceed and exclude the next process. Binary semaphores may provide mutual exclusion to shared data resources. Thus, the "gate" is like a garden gate, protecting the shared resource ("garden") from access by another process.

In addition, MicroPower/Pascal offers another data structure, called a mutex, to use in place of a binary semaphore for mutual exclusion within the same program. A mutex is an optimization of a binary semaphore. See the *MicroPower/Pascal Language Guide* for information about the mutex structure.

A counting semaphore allows as many processes to proceed as specified by the value parameter of the CREATE_COUNTING_SEMAPHORE request and then excludes the next process. Counting semaphores may be used in applications that contain multiple servers.

The following sections describe the Pascal language statements you use in your source code to access semaphore data structures. The procedures and functions used to operate on binary and counting semaphores are requests to the primitive services provided by the MicroPower/Pascal kernel. See the *MicroPower/Pascal Language Guide* for information about the Pascal language interface to the MicroPower/Pascal kernel. See the *MicroPower/Pascal Run-Time Services Manual* for information about kernel primitive services.

4.2.1.1 CREATE_BINARY_SEMAPHORE Function

This function requests the kernel to allocate and initialize a binary semaphore structure in system-common memory. You use the name parameter to specify a unique name, if desired, and the descriptor parameter to specify the semaphore's structure identifier. This function permits a process to create a binary semaphore that can be manipulated by the semaphore management requests. The following statement creates a binary semaphore:

```
Result := CREATE_BINARY_SEMAPHORE (DESC := R1,  NAME := 'BISEM1');
```

This statement creates a binary semaphore with two parameters: a structure descriptor "R1" and the run-time name "BISEM1". This statement returns the value TRUE or FALSE for the Boolean variable Result, indicating whether or not the semaphore was created.

4.2.1.2 CREATE_COUNTING_SEMAPHORE Function

This function requests the kernel to allocate and initialize a counting semaphore structure in system-common memory. You use the name parameter to specify a unique name, if desired, and the descriptor parameter to specify the semaphore's structure identifier.

The value parameter tells how many processes can perform WAITs before the semaphore closes. The value parameter may be in the range 0 to 65,535. A value of 0 specifies that the semaphore is closed. A positive value specifies how many processes the semaphore is open to. The following statement creates a counting semaphore:

```
Result := CREATE_COUNTING_SEMAPHORE (VALUE := 4, DESC := CS1);
```

This statement creates a counting semaphore referenced by the structure descriptor "CS1." This semaphore is open, and four WAITs can proceed before the semaphore is closed to other processes.

4.2.1.3 SIGNAL Procedure

This procedure signals a specified binary or counting semaphore, unblocking the first process waiting on that semaphore. This procedure permits the calling process to signal another process that an event has occurred.

The SIGNAL procedure increments (opens) a binary or counting semaphore's gate variable if the semaphore's current value is 0 (closed). If a binary semaphore is open, the procedure returns control to the caller, with no other action. If a counting semaphore is open, its value is incremented if its maximum value has not yet been reached; and, if its value was 1 or more before the signal, control returns to the caller. If the signal causes the semaphore to open, and if at least one process is waiting on the semaphore, the procedure unblocks the first waiting process and calls the kernel scheduler. This may cause the calling process to be preempted—lose control of the CPU—depending on the relative priority of the process at the head of the semaphore's queue of blocked processes. If the semaphore value changes from closed to open, and if no process is waiting, control returns to the calling process.

The actions resulting from a call to the SIGNAL procedure for binary and counting semaphores can be summarized as follows:

Initial Conditions	Resulting Action
Semaphore is open	No effect (binary) Increment count if less than maximum (counting)
Semaphore is closed; no processes waiting	Semaphore opens
Semaphore is closed; one or more processes waiting	First waiting process unblocks (moves from a wait to a ready state) Scheduler is invoked if unblocked process has a higher priority than that of currently running process

COND_SIGNAL, the conditional form of SIGNAL, permits the calling process to signal a semaphore if at least one process is waiting on that semaphore. If no process is waiting, the semaphore is not signaled.

4.2.1.4 WAIT Procedure

A WAIT on a semaphore tests the specified semaphore for a signal (positive value). The calling process is blocked if the semaphore had not been signaled. When a process waits on a semaphore, the value associated with the semaphore is checked. If the semaphore value is 0, the process blocks (waits) and is placed in the wait-active state. Each semaphore may have a queue of waiting processes. When the semaphore is signaled, the process at the head of the queue enters the ready-active state and is eligible to resume execution according to its priority. If the semaphore value is not 0 when checked, it is decremented, and the process continues.

The results from a call to the WAIT procedure can be summarized as follows:

Initial Conditions	Resulting Action
Semaphore is closed	Process blocks Scheduler is invoked
Semaphore is open	Close semaphore (binary) Decrement count (counting)

COND_WAIT, the conditional or nonblocking form of the WAIT procedure, permits the calling process to test for the arrival of a signal from another process without being blocked if the signal has not yet occurred.

Together, the SIGNAL and WAIT requests provide a means for two cooperating processes to implement a variety of synchronization and mutual-exclusion mechanisms. The following example, using a binary semaphore, illustrates the role of SIGNAL and WAIT in process synchronization.

```
PROGRAM Binary_Semaphore_Example;

VAR
  Gate : SEMAPHORE_DESC;

PROCESS Me_Too;

BEGIN
  WRITE ('X');
  SIGNAL (Gate);
END;

BEGIN
  IF CREATE_BINARY_SEMAPHORE (DESC := Gate)
   THEN
     BEGIN
       Me_Too;
       WAIT (Gate);
       WRITELN (' marks the spot.');
     END
   ELSE
     WRITELN ('Failed');
END.
```

This example uses one binary semaphore, and the two processes synchronize by means of SIGNAL and WAIT operations on the semaphore.

The program first creates the semaphore with a default value of 0 (closed). The program creates process Me_Too and then waits on the semaphore. Because the semaphore is closed, the main program blocks, allowing process Me_Too to run. Me_Too writes one character, then signals the semaphore. Signaling the semaphore unblocks the main program, allowing it to finish execution. Output of this example is:

```
X marks the spot.
```

4.2.2 Using SUSPEND and RESUME for Process Synchronization

The kernel can also achieve process synchronization by using the SUSPEND and RESUME functions.

4.2.2.1 SUSPEND Function

This function gives to the running process power over other processes waiting for control of the CPU. Specifically, SUSPEND allows the running process to suspend execution of another process (or itself) directly. The suspended process may not execute until resumed by another process.

A process issuing a SUSPEND on another process specifies a process descriptor or process name.

```
Result := SUSPEND (NAME := 'JALOPY');
```

This statement suspends the process with the run-time name JALOPY and, if successful, returns the value TRUE to the Boolean variable Result. The state of the suspended process is changed to ready suspended, wait suspended, or exception-wait suspended, depending on its state at the time of suspension. If neither a name nor a descriptor is specified as a parameter, the process is itself suspended:

```
Result := SUSPEND;
```

The running process suspends itself, and the highest-priority process in the ready-active state is placed in the run state.

4.2.2.2 RESUME Function

This function gives to the running process power over other processes waiting for control of the CPU. A resume request is required to return a suspended process to one of the active states. Specifically, RESUME allows the running process to resume a suspended process. The following statement resumes the suspended process JALOPY and, if successful, returns the value TRUE to the Boolean variable Result:

```
Result := RESUME (NAME := 'JALOPY');
```

4.3 Interprocess Communication

The kernel uses two data structures to transmit messages between processes: ring buffers and queue semaphores. Ring buffers are a means of asynchronous data transfers, usually involving interrupt-driven input/output devices. Queue semaphores provide tighter synchronization of the message transfer than is possible with ring buffers.

4.3.1 Using Ring Buffers to Send Messages

Ring buffers are kernel data structures that transmit variable-length messages between processes. Data is input by one process and may be removed by another process in the same sequence: first in, first out. Two ring buffer operations may take place concurrently; one process may put data into the buffer at the same time another process removes data.

4.3.1.1 CREATE_RING_BUFFER Function

This function creates a ring buffer data structure and sets up a descriptor for referring to it.

4.3.1.2 GET_ELEMENT Procedure

This procedure extracts a specified number of bytes of data from a ring buffer and transfers them to the calling process. If an insufficient number of bytes is present, the calling process blocks. COND_GET_ELEMENT, the conditional or nonblocking form, does not cause the receiving process to block if the request cannot be satisfied. Instead, the request may implement a partial transfer or return without a transfer.

4.3.1.3 PUT_ELEMENT Procedure

This procedure copies a specified number of bytes from the calling process to a ring buffer; the calling process is blocked if the ring buffer has insufficient space. COND_PUT_ELEMENT, the nonblocking form, attempts to copy the specified number of bytes to the ring buffer but does not block if the request cannot be satisfied.

The following example creates a ring buffer structure, puts data into the buffer, removes data from the buffer, and writes it.

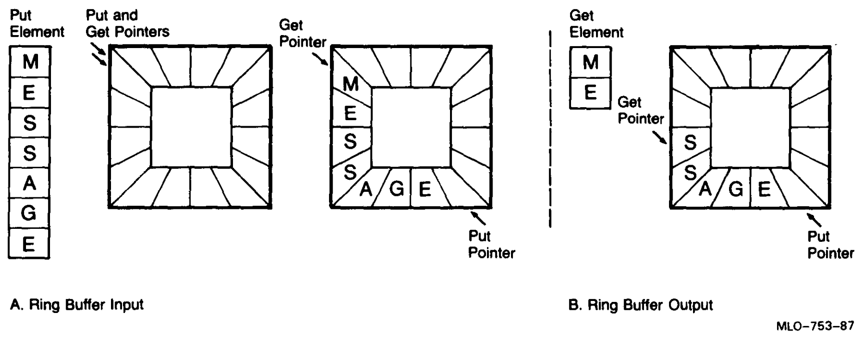
```
PROGRAM Ring_Buffer_Example;

VAR
  Bye   : RING_BUFFER_DESC;
  Chars : PACKED ARRAY [1..2] OF CHAR;

BEGIN
  IF CREATE_RING_BUFFER (DESC := Bye, SIZE := 16)
    THEN
      BEGIN
        PUT_ELEMENT (DESC:= Bye, DATA := 'Message', LENGTH := 7);
        GET_ELEMENT (DESC:= Bye, DATA := Chars, LENGTH := 2);
        WRITE (Chars);
      END
    ELSE
      WRITELN ('Failed');
END.
```

The output of this program is:

```
Me
```

Figure 4–1 illustrates the use of the ring buffer in this example.

Figure 4–1: Using a Ring Buffer

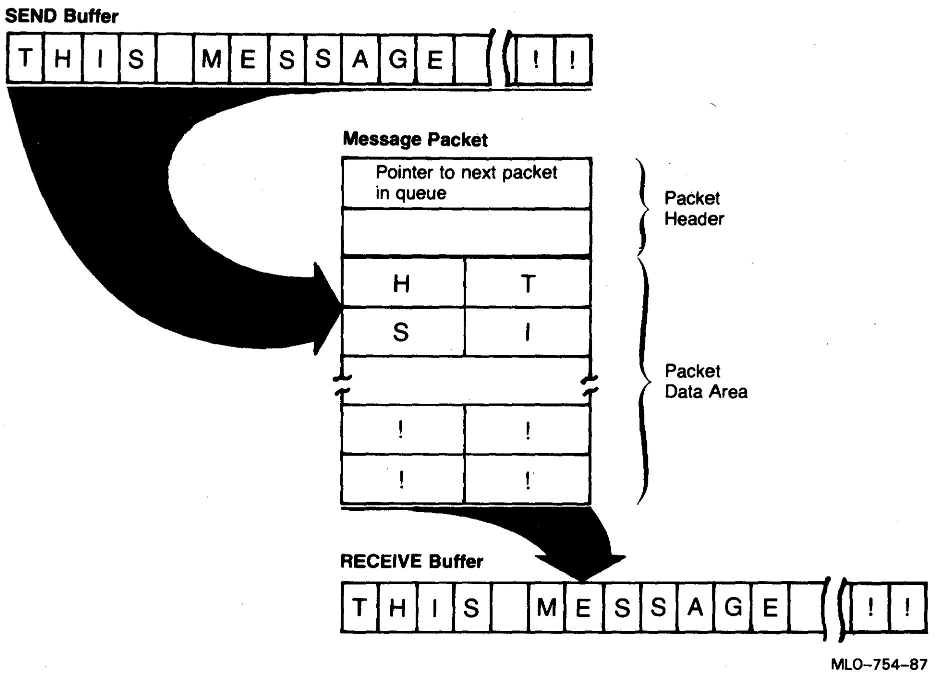


4.3.2 Using Queue Semaphores to Send Messages

You may want to pass a message from one process to another by using a queue semaphore. The queue semaphore is an extension of a counting semaphore. A queue semaphore associates a counting semaphore with a queue of message packets to implement synchronized message transfer.

The message packets are composed of a packet header and a data area. The header contains such information as the pointer to the next packet in the queue and a reference data flag. The data area portion of the packet may contain up to 34 bytes of data. Message transmission may be by value, by reference, or by a combination of both means. When data is sent by value, the data to be transmitted is copied into the message packet. When data is sent by reference, the packet tells the kernel to look for a pointer to the sender's message buffer. This pointer is contained in the kernel data structure INFO_BLOCK. INFO_BLOCK is a record that contains information about the packet. Figure 4–2 shows transmission of value data by using a message packet.

Figure 4–2: Using a Message Packet



4.3.2.1 CREATE_QUEUE_SEMAPHORE Function

This function permits a process to create a queue semaphore that can be manipulated by the various queue semaphore management requests. Like binary and counting semaphores, queue semaphores are created by specifying a descriptor parameter and an optional name parameter.

4.3.2.2 SEND Procedure

This procedure requests the kernel to allocate a packet, fill it with the specified data, and signal a queue semaphore. Data may be sent by value (up to 34 bytes), by reference if the message is larger, or by both means. If a packet is not available, the process blocks on the queue semaphore until the packet becomes available. See the *MicroPower/Pascal Language Guide* for more information.

COND_SEND is the conditional form of the SEND procedure. COND_SEND determines if a process is waiting on the queue semaphore. If a receiving process is not waiting for the packet, COND_SEND returns control to its process.

4.3.2.3 RECEIVE Procedure

The receiving process issues the RECEIVE statement, naming the queue semaphore. When the sending process sends a message to that queue semaphore, the receiving process gets the data in the packet.

COND_RECEIVE, the conditional form of the RECEIVE procedure, checks for an available packet. If no packet is available, execution returns to the process without blocking.

Both the sender and the receiver must specify the size and configuration of the message. Processes can designate optional reply (acknowledgment) semaphores to report successful transmission. For more information about queue semaphore requests, see the *MicroPower/Pascal Language Guide.*

The following example uses the SEND and RECEIVE procedures to transfer data by means of a message packet.

```
[SYSTEM (MICROPOWER)]  PROGRAM Queue_Semaphore_Example;

TYPE
  Buffer = PACKED ARRAY [1..34] OF CHAR;

VAR
  Queue     : QUEUE_SEMAPHORE_DESC;
  Send_Buff : Buffer;

PROCESS Get_Message;

VAR
  Info     : INFO_BLOCK;
  Counter  : INTEGER;
  Rec_Buff : Buffer;

BEGIN
  RECEIVE ( DESC       := Queue,
            VAL_DATA   := Rec_Buff,
            VAL_LENGTH := SIZE(Rec_Buff),  (* size of message received *)
            RET_INFO   := Info);  (* records information about what is sent *)

  FOR Counter := 1 TO Info.VAL_XMIT_LEN DO
    WRITE (Rec_Buff[Counter]);
  WRITELN;
END;

BEGIN (* Main Program *)
  Send_Buff := 'This message is 34 bytes long!!!!!';
  IF CREATE_QUEUE_SEMAPHORE (DESC := Queue)
   THEN
     BEGIN
       Get_Message;
       SEND (DESC       := Queue,
             VAL_LENGTH := SIZE(Send_Buff), (* size of message sent *)
             VAL_DATA   := Send_Buff)
     END
   ELSE
     WRITELN ('Failed');
END.
```

The main program SENDs a message to Process Get_Message, which prints it out after receiving it. (When the message packet is created, Get_Message blocks on the queue semaphore, Queue, until the complete message has been transmitted by the sending process.)

VAL_LENGTH is a parameter used by both the SEND and RECEIVE procedures. For SEND, the VAL_LENGTH parameter specifies the size of the message being sent. For RECEIVE, VAL_LENGTH specifies how many bytes are accepted. The number of bytes may be equal to the size of the buffer declared to accept the message, or it may be fewer bytes. (For data sent by value, the limit is 34 bytes for both the SEND and RECEIVE procedures.)

VAL_XMIT_LEN, a field of INFO_BLOCK, contains the size of the data transferred by value. The size is determined by the VAL_LENGTH parameter of the RECEIVE or the VAL_LENGTH parameter of the SEND, whichever is smaller.

4.4 Connecting Processes to Interrupts

An interrupt, or signal from a device, automatically causes a change in the flow of instruction execution within the processor. In a MicroPower/Pascal target system, interrupts can arrive unpredictably and, therefore, are called asynchronous. When an interrupt occurs, control of the CPU transfers to an appropriate interrupt service routine (ISR).

After the ISR runs, the kernel assigns control of the CPU to the highest priority process. That process may or may not be the same process that was in control when the interrupt occurred. (Actions taken by the ISR responding to the interrupt may have enabled a waiting process or affected the eligibility of processes in some other way.) ISRs are a part of any process written to handle devices, usually a process of type DRIVER.

4.4.1 CONNECT_INTERRUPT Procedure

This procedure associates an interrupt vector with an ISR to establish a process as a device driver. CONNECT_INTERRUPT is used to handle interrupts that occur at a relatively high rate for the amount of data being transmitted.

4.4.2 CONNECT_SEMAPHORE Procedure

This procedure associates an interrupt vector with a semaphore. The specified semaphore is signaled each time an interrupt occurs. CONNECT_SEMAPHORE is used to handle interrupts that occur at a relatively low rate for the amount of data being transmitted, with slow devices, and when millisecond response times are adequate.

4.5 Exception-Handling Processes and Procedures

An exception is an event that alters the normal flow of execution. An exception condition is a consequence of the execution of the current instruction. An exception condition may or may not represent a fatal execution error in the application.

The target system hardware and software can detect 16 types of exceptions to normal application execution. Three examples of exception conditions are:

- Accessing a nonexistent I/O device (Type: Hard_IO)
- Executing an illegal instruction (Type: Illegal_Operation)
- Insufficient space for stack (Type: Resource)

In addition to the exception conditions recognized by MicroPower/Pascal software, you can define your own category of exceptions that are specific to your application.

Either processes or procedures may be declared as exception handlers. An exception-handling procedure is typically used when you want to log exception occurrences within a particular process. An exception-handling process is used to detect exception conditions for groups of processes and when process control block (PCB) information is required to take corrective action. You include the appropriate system exception declaration file(s) with the source modules that contain your exception handlers. See the *MicroPower/Pascal Run-Time Services Manual* and the *MicroPower/Pascal Language Guide* for further information on exception handling in MicroPower/Pascal.

4.5.1 Exception-Handling Processes

You can set up processes to take control when the system detects an exception condition. You can design each exception-handling process to respond to exceptions of certain types. Also, you can specify one process to handle the exceptions of a group of processes. To establish an exception-handling process, use the CONNECT_EXCEPTION and WAIT_EXCEPTION statements within the process.

The CONNECT_EXCEPTION statement identifies that process to the kernel as the exception handler. The WAIT_EXCEPTION statement causes the process to block on the specified queue semaphore. When an exception occurs, the kernel signals the queue semaphore and returns the address of the PCB of the offending process. Upon examining the information provided by the PCB, the exception-handling process can use this information to determine its corrective action.

The following code fragment sets up an exception-handling process for SYSTEM_SERVICE exceptions and waits on the queue semaphore Exception.

```
PROCESS  Exception_Handler;
         .
         .
         .
VAR
  Exception :  QUEUE_SEMAPHORE_DESC;
  Ptr :  PCB_POINTER;
BEGIN
         .
         .
         .
CONNECT_EXCEPTION (DESC := Exception, EXC_TYPE := [SYSTEM_SERVICE]);
WAIT_EXCEPTION (DESC := Exception, PCB_PTR := Ptr);
         .
         .
         .
END;
```

4.5.2 Exception-Handling Procedures

Exceptions for processes in one address space can be handled by procedures nested directly inside the Pascal main program (static process). To identify a procedure to handle exceptions for the program, you include an ESTABLISH statement, which has the form:

```
ESTABLISH (EXC_PROCEDURE := Resource_Errors,
EXC_TYPE := [RESOURCE]);
```

In the previous statement, the Pascal procedure Resource_Errors is designated to handle exceptions of type RESOURCE that arise during execution.

4.5.3 Generating an Exception

When testing an exception handler, you may want to generate an exception. You can generate an exception by using the REPORT statement, which has the form:

```
REPORT (EXC_TYPE := [RESOURCE], EXC_CODE := ES_$NMP );
```

The exception defined by type RESOURCE and code ES_$NMP is generated wherever this statement is included in the process.

If you define your own category of exceptions specific to your application, you must use the REPORT statement to declare exceptions of that type at the appropriate places in your application.

4.6 Concurrent Design Solution: Bottle-Corking Machine

Process synchronization in concurrent application design can be illustrated by the operation of a hypothetical bottle-corking machine. This example divides a problem into separate tasks, assigns processes to correspond to the separate tasks, and coordinates the processes by using queue semaphores.

In this example, a high-speed bottling plant uses conveyors to transport bottles among automated bottling stations. Those stations clean, fill, cork, and package bottles. The hypothetical bottle-corking machine is controlled by a MicroPower/Pascal application.

4.6.1 Target Hardware

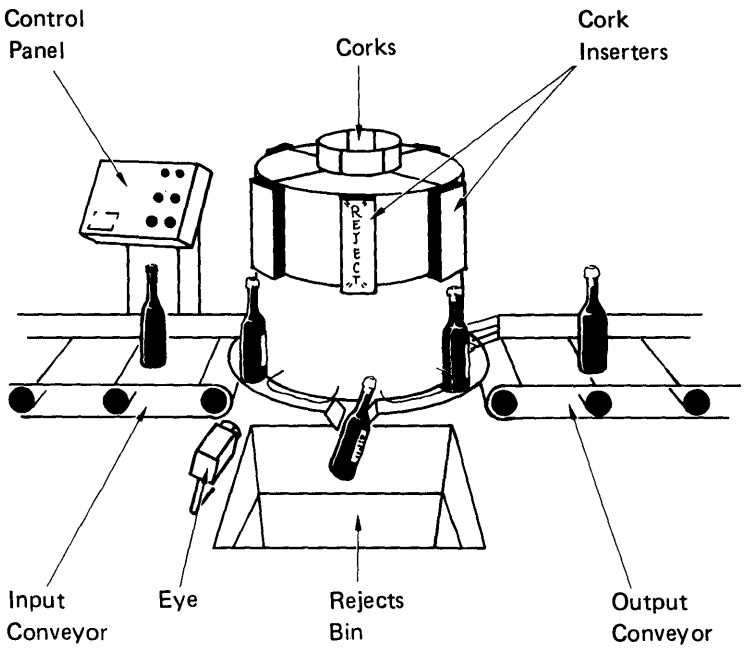
The corking machine consists of a vertical, rotating drum with a rounded ledge on which the bottles sit in discrete slots. A cork-inserter assembly is attached to the drum above each slot. As it moves around the drum, a bottle is corked by the inserter positioned above it. The inserters are reloaded from a cork hopper above the drum.

Two conveyor belts service the machine. The input belt brings bottles to the machine, and the output belt takes bottles away. The two belts run at the same speed as the drum, preventing the bottles from jamming as they enter or leave the workspace.

An electric eye positioned near the input conveyor senses the filled bottles as they occupy slots on the drum. The eye detects vacant slots and slots containing broken or unfilled bottles.

A large bin for bad bottles is in front of the drum. Slots containing bad bottles are emptied into this rejects bin. That arrangement purges the line of bad bottles and allows glass to be recycled. See Figure 4–3.

Figure 4-3: Bottle-Corking Machine



MLO-755-87

4.6.2 Operating Characteristics

The bottle-corking machine performs the following functions:

- Inserts corks into bottles, ignoring any slots that do not contain bottles, so as not to waste corks
- Rejects broken or unfilled bottles from the line
- Senses a low cork supply and notifies the operator
- Senses a depleted cork supply and stops the input conveyor until the corks are restored
- Senses a breakdown of the input conveyor and termination of incoming bottles and shuts down the machine
- Senses a breakdown of the output conveyor and stops the input conveyor until the output conveyor restarts

A concurrent MicroPower/Pascal application can be designed to run this bottle-corking machine.

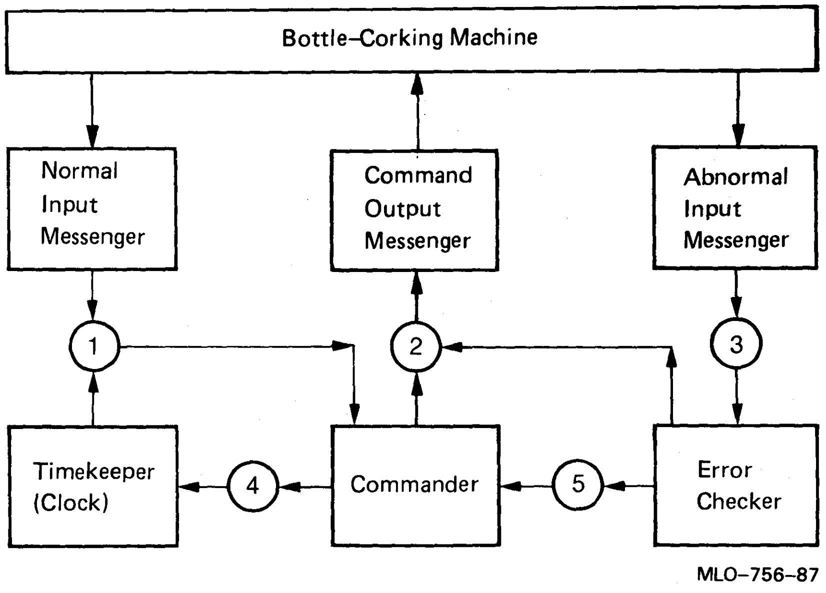
4.6.3 Designing a Concurrent Solution

Concurrent design lets you divide the problem into separate tasks. You can define six parallel processes to run the machine. Three device driver processes handle I/O between the application and the machine. Another process, Commander, sends normal, repetitive operating commands to the machine. Error Checker receives notice of abnormal conditions from the machine. (This process may transmit a message to Commander, which in turn alters the command sequence.) Finally, Timekeeper interfaces with the system clock to provide timings.

The processes communicate by means of five queue semaphores, which act like mailboxes, holding messages from one process to another. Queue semaphores are used rather than binary semaphores, because each semaphore may be signaled for several reasons. In every case, some information about the reason must pass to the waiting process. Note that only one arrow leaves each queue semaphore; in this application, only one process waits on any one queue semaphore. Finally, more than one arrow may leave the boxes (processes), since one process can signal more than one queue semaphore during execution.

Figure 4–4 shows how the processes are related. The six processes are represented by boxes. The five queue semaphores are represented by circles.

Figure 4–4: Processes Comprising the Concurrent Solution



4.6.3.1 Device Driver Processes

Serial lines transmit electrical signals to and from the machine. Three device driver processes handle the lines. The two input messenger processes translate incoming signals into symbolic codes and pass them on to Commander and Error Checker. On the output side, Command Output Messenger translates line signals for the outgoing wire.

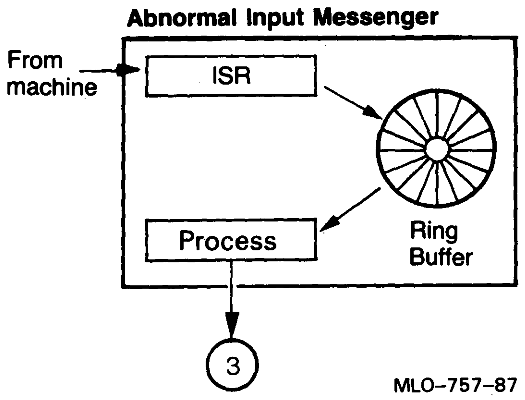
The three device driver processes are named for the type of information they convey: Abnormal Input Messenger, Command Output Messenger, and Normal Input Messenger. The three processes are as follows:

- Abnormal Input Messenger encodes messages from the station error sensors and passes codes to the error-handler process.
- Command Output Messenger decodes messages from Commander and places appropriate signals on the output line to the machine. Command Output Messenger can receive input from two sources: Commander and Error Checker. Under normal operation, only Commander sends messages to Command Output Messenger.
- Normal Input Messenger encodes signals received during the bottle-corking operation.

The application must not miss any incoming signals and should send outgoing signals without long gaps. Ideally, part of the application should attend to the I/O lines regardless of coding and decoding operations. Therefore, each device driver is composed of two independent parts: the process and an ISR. (ISRs and device driver processes, including a guide to writing your own device drivers, are described in detail in the *MicroPower/Pascal I/O Services Manual.*)

Each device driver uses a ring buffer to pass information between the independent ISR and the rest of the device driver process (see Figure 4–5). Upon receipt of a line signal, the ISR uses the PUT_ELEMENT statement to put data into the ring buffer. As the signal requests arrive, the ISR may do several PUT_ELEMENT operations in a short time. The ISR then waits for more signals while the process removes each element in turn from the ring buffer (with the GET_ELEMENT statement) and performs the encoding operation.

Figure 4–5: The Device Driver



Prompt response to signals from the real-time environment is critical. So, MicroPower/Pascal ISRs run at priorities higher than those of normal processes, although priorities may vary among ISRs. Because quick response is so important, ISRs usually are responsible for little more than attending to the I/O lines. Thus, ISRs receive information from or put information on the lines and interact with temporary buffer storage, such as ring buffers.

For instance, on the input side, the ISR is ready to respond to an interrupt from the machine, grabbing signals off the wire and storing them for the encoder part of the process. At the lower process priority, the process translates the messages into codes to be sent to the Error Checker.

4.6.3.2 Commander Process

As shown in Figure 4–4, each arrow entering a box marks a waiting point for the process represented by that box. One box, Commander, has two arrows leading into it. Commander must pay attention to acknowledgments from the machine and messages passed from Error Checker. If made to wait for input from Error Checker by Queue_Semaphore (5), Commander cannot run the machine. Ideally, Commander should attend to Error Checker if a message is forthcoming immediately. This situation calls for use of the RECEIVE_ANY function. RECEIVE_ANY lets Commander wait on more than one queue semaphore.

The other incoming arrow, Queue_Semaphore (1), to Commander passes acknowledgments from Normal Input Messenger and timeout signals from Timekeeper. If the machine does not acknowledge orders from Commander within a specified interval, the clock process notifies Commander of a timeout, and the command stream to the machine is interrupted by a halt sequence. Because at least one of these messages must arrive during each cycle of the machine, this queue semaphore is vital to proper synchronization. Commander must block itself from executing and wait for a message before continuing to issue orders to the machine.

4.6.3.3 Using Process Priorities to Gain Efficiency

In this application, if each process executed in a separate CPU, overall performance would be synchronized. In reality, however, only one process runs at a time. This application may be defined for greater efficiency by assigning relative priorities to processes.

If an error requiring a stop-at-once command comes in from the machine, it should receive prompt attention. Therefore, Error Checker may be given higher priority than Commander. Timekeeper needs to work only when its clock ticks, but that work is very important, because timings depend on it. Therefore, Timekeeper may be given a high priority.

Chapter 5

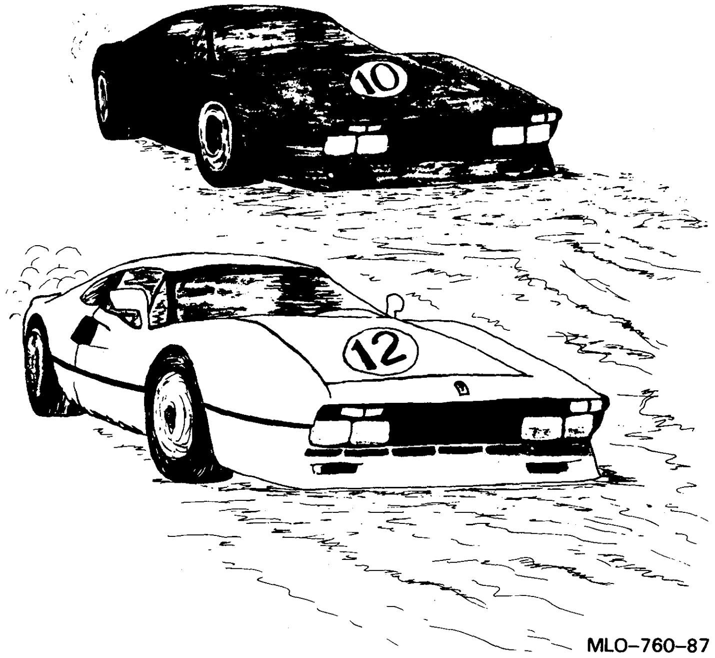
Application Development: Designing the Source Code

This chapter presents a 3-stage overview of concurrent programming in the MicroPower/Pascal environment. Three application development programs introduce concepts fundamental to real-time MicroPower/Pascal application development. The programs also are the basis for the application development examples you can build and run in Chapters 6 and 7.

5.1 Introducing the CARS Program

Three examples of the CARS program are presented. The programs run on a target system that includes video terminal output. In the first two examples, one car, represented by the symbol #, travels across the terminal screen. When it reaches the right edge, the car starts again at the left edge, giving the illusion of continuous movement. Example 5–1 uses a monolithic approach, resembling a traditional Pascal program. Example 5–2 performs the same task but introduces a MicroPower/Pascal feature, a dynamic process. The process concept is used to an advantage in Example 5–3, which expands the program to include two cars traveling across the terminal screen. Each car is directed by its own dynamic process. This program introduces the real-time features of binary semaphores, which coordinate interprocess synchronization. SIGNAL and WAIT requests on the two binary semaphores implement a handshake mechanism that makes sure that the two cars run concurrently across the screen.

The following sections describe these three program examples in detail. Chapters 6 and 7 give you an opportunity to build and run a version of the CARS program for yourself.



5.2 First CARS Program Example: CARS1

Example 5–1 of the CARS program, included to show a typical Pascal program, moves one car across the terminal screen. This program uses MicroPower/Pascal syntax but does not include any special real-time features. The program is included as a stepping stone to the MicroPower/Pascal features introduced in Examples 5–2 and 5–3.

Example 5–1: Program CARS1

```
[SYSTEM(MICROPOWER), PRIORITY(1),
 DATA_SPACE(2000), STACK_SIZE(400)] PROGRAM CARS1;

CONST
  Line = 10;

VAR
  Column : INTEGER;

PROCEDURE Clear_Screen;
BEGIN
  WRITE(''(27)'[2J');
END;
```

```
PROCEDURE Move_Car_Right;
BEGIN
  WRITE (''(27)'[', Line:1, ';', Column:1, 'H'); (* position cursor *)
  IF Column < 77
   THEN
     BEGIN
       WRITE (' #');
       Column := Column + 1;
     END
   ELSE
     BEGIN
       WRITE (' '); (* blank last column *)
       Column := 1; (* position to Column 1 *)
       WRITE (''(27)'[', Line:1, ';', Column:1, 'H#'); (* write car *)
     END;
END;

BEGIN
  Clear_Screen;
  Column := 1;
  WHILE TRUE DO
    Move_Car_Right;
END.
```

5.2.1 Program Heading

The heading specifies the program name and attributes declared at the global level. The attributes SYSTEM, PRIORITY, DATA_SPACE, and STACK_SIZE are declared. The attributes are enclosed in brackets and precede the reserved word PROGRAM. Attribute values are enclosed in parentheses.

5.2.2 Procedures

The program contains two procedures:

- Clear_Screen
- Move_Car_Right

The procedure Clear_Screen writes the escape sequence that clears the terminal screen. Clear_Screen is invoked by the main program.

The procedure Move_Car_Right advances a "car" one space to the right by using escape sequences that perform cursor positioning and by writing the symbol "#" to the screen.

5.2.3 Execution

The program invokes the procedure Clear_Screen. Column is initialized to 1. The program next invokes the procedure Move_Car_Right from within a continuous WHILE loop.

5.3 Second CARS Program Example: CARS2

Example 5–2 introduces a dynamic process, CAR, to the basic program. This example shows the basic structure of a MicroPower/Pascal program. The components, general format, and correct language syntax of a MicroPower/Pascal program are illustrated.

Example 5-2: Program CARS2

```
[SYSTEM(MICROPOWER), PRIORITY(1),
 DATA_SPACE(2000), STACK_SIZE(400)] PROGRAM CARS2;

PROCEDURE Clear_Screen;
BEGIN
  WRITE(''(27)'[2J');
END;

[ PRIORITY(2), STACK_SIZE(400) ] PROCESS Car (Line : INTEGER);
VAR
  Column : INTEGER;

PROCEDURE Move_Car_Right ;
BEGIN
  WRITE (''(27)'[',line:1,';',column:1,'H');
  IF column < 77
   THEN
     BEGIN
       WRITE (' #');
       Column := Column + 1;
     END
   ELSE
     BEGIN
       WRITE (' ');
       Column := 1;
       WRITE (''(27)'[', Line:1,';', Column:1,'H#');
     END;
END;

BEGIN    (* Process Car *)
  Column := 1;
  WHILE TRUE DO
    Move_Car_Right;
END;

BEGIN
  Clear_Screen;
  Car (Line := 10);     (* create one car on line 10 *)
END.
```

5.3.1 Procedures

As in CARS1, this program uses the Clear_Screen and Move_Car_Right procedures.

5.3.2 Processes

The program declares one process: CAR. Since Move_Car_Right is invoked only by the process CAR, the procedure can be declared within the process.

The variable COLUMN is a local variable declared within CAR. Column is used in cursor positioning.

The purpose of this process is to move the "car" across the terminal screen by calling the procedure Move_Car_Right. This car "runs" on line 10 of the terminal screen. The process has a priority of 2.

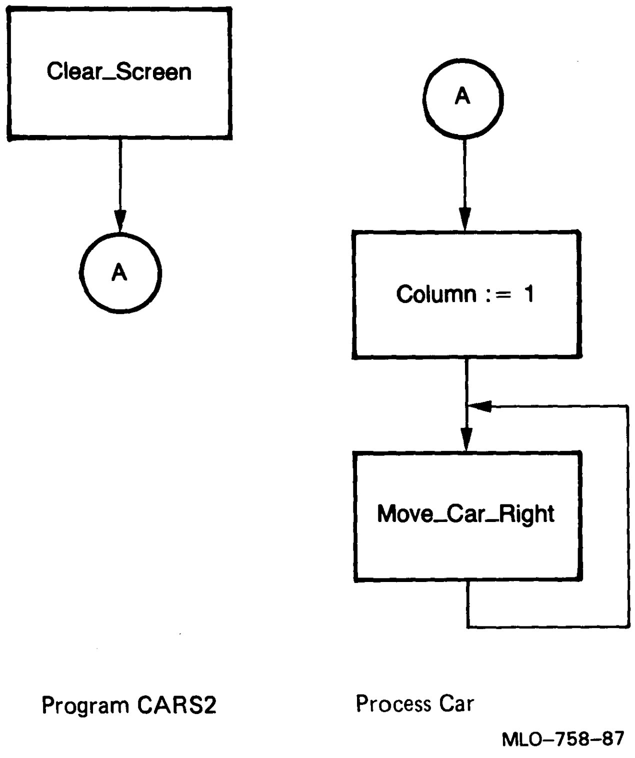
5.3.3 Execution

The main program's code is the static process. The main program invokes the procedure Clear_Screen. The main program then invokes (creates) the process CAR. The result is one "car" traveling across the terminal screen on line 10.

When CAR is invoked (created), its priority is compared with that of the currently running static process (Program CARS2). Since the dynamic process CAR has a higher priority, the process CAR runs.

Figure 5-1 shows the logic of this program.

Figure 5-1: Program CARS2 Flowchart



5.4 Third CARS Program Example: CARS3

Example 5-3 expands the program to include two dynamically created processes. Process CAR is created twice, to allow two cars to race across the screen. This program uses the following real-time features: binary semaphores and the SIGNAL and WAIT primitives. The processes use the semaphores to implement a handshake mechanism that coordinates their concurrent execution. The effect on your terminal is two cars traveling across the screen, one on line 10 and the other on line 12.

Example 5-3: Program CARS3

```
[SYSTEM(MICROPOWER), PRIORITY(1),
 DATA_SPACE(2000), STACK_SIZE(400)] PROGRAM CARS3;

VAR
  S1, S2 : SEMAPHORE_DESC; (* Semaphores for the "handshake" mechanism *)
  OK     : BOOLEAN;        (* Indicates successful semaphore creation *)

[INITIALIZE] PROCEDURE Setup;
BEGIN
  OK := CREATE_BINARY_SEMAPHORE ( DESC := S1, VALUE := 0 ) AND
        CREATE_BINARY_SEMAPHORE ( DESC := S2, VALUE := 0 );
END;

PROCEDURE Clear_Screen;
BEGIN
  WRITE (''(27)'[2J');
END;

[STACK_SIZE(400)] PROCESS Car (Line : INTEGER; VAR Start, Done : SEMAPHORE_DESC);

VAR
  Column : INTEGER;

  PROCEDURE Move_Car_Right;
  BEGIN
    WRITE (''(27)'[', Line:1, ';', Column:1, 'H');
    IF Column < 77
     THEN
       BEGIN
         WRITE (' #');
         Column := Column + 1;
       END
     ELSE
       BEGIN
         WRITE (' ');
         Column := 1;
         WRITE (''(27)'[', Line:1, ';', Column:1, 'H#');
       END;
  END;
```

```
BEGIN   (* Process Car *)
  Column := 1;
  WHILE TRUE DO
    BEGIN
      WAIT (DESC := Start);
      Move_Car_Right;
      SIGNAL (DESC := Done);
    END;
END;   (* Process Car *)

BEGIN
  IF OK
   THEN
    BEGIN
      Clear_Screen;
      (* create first car on line 10 *)
      Car (Line := 10, Start := S1, Done := S2,
          PRIORITY := 2, NAME := 'LANE10');
      (* create second car on line 12 *)
      Car (Line := 12, Start := S2, Done := S1,
          PRIORITY := 3, NAME := 'LANE12');
      SIGNAL (DESC := S1);
    END;
END.
```

5.4.1 Global Variable Declarations

This program declares three variables: two binary semaphores and a Boolean. The binary semaphores S1 and S2 are used to implement a "handshake mechanism" between the two dynamic processes. OK, a Boolean variable, is used to indicate the successful creation of the binary semaphores. The Boolean variable OK receives the value TRUE when the semaphores are successfully created.

5.4.2 Procedures

In addition to the procedures declared in CARS2, CARS3 contains the Setup procedure. Setup is the initialization procedure, identified by the attribute [INITIALIZE]. Setup is invoked by the system startup routine and executes first, before any other part of the program. The INITIALIZE procedure typically creates any structures needed by the program. During initialization, the INITIALIZE procedure creates both semaphores and designates S1 and S2 as the program's structure descriptor variables that hold the names and identification numbers of the semaphores.

The semaphores are created with an initial value of 0. (The semaphore value is specified for clarity; if no value is specified, 0 is also the default value.) Thus, the semaphores are closed. The CREATE_BINARY_SEMAPHORE functions are part of an assignment statement to the Boolean variable OK. The assignment statement returns a value of TRUE if the semaphores are successfully created and a value of FALSE if they are not.

5.4.3 Processes

As in CARS2, one dynamic process, CAR, is declared. The semaphores are passed as VAR parameters.

The process CAR first initializes Column to 1, then enters a WAIT/SIGNAL loop. When it performs a WAIT, the process blocks itself and joins the wait queue if the semaphore is closed. (That happens the first time the instruction executes for each invocation of the process.) However, if the process performs a WAIT and the semaphore is open, the process continues executing. When the process performs the SIGNAL, it opens the semaphore. That causes the scheduler to compare the priority of the currently running process with that of the first waiting process.

This WAIT/SIGNAL loop uses the two semaphores passed as parameters. A WAIT is performed on the semaphore passed to the parameter Start, and a SIGNAL is performed on the semaphore passed to Done.

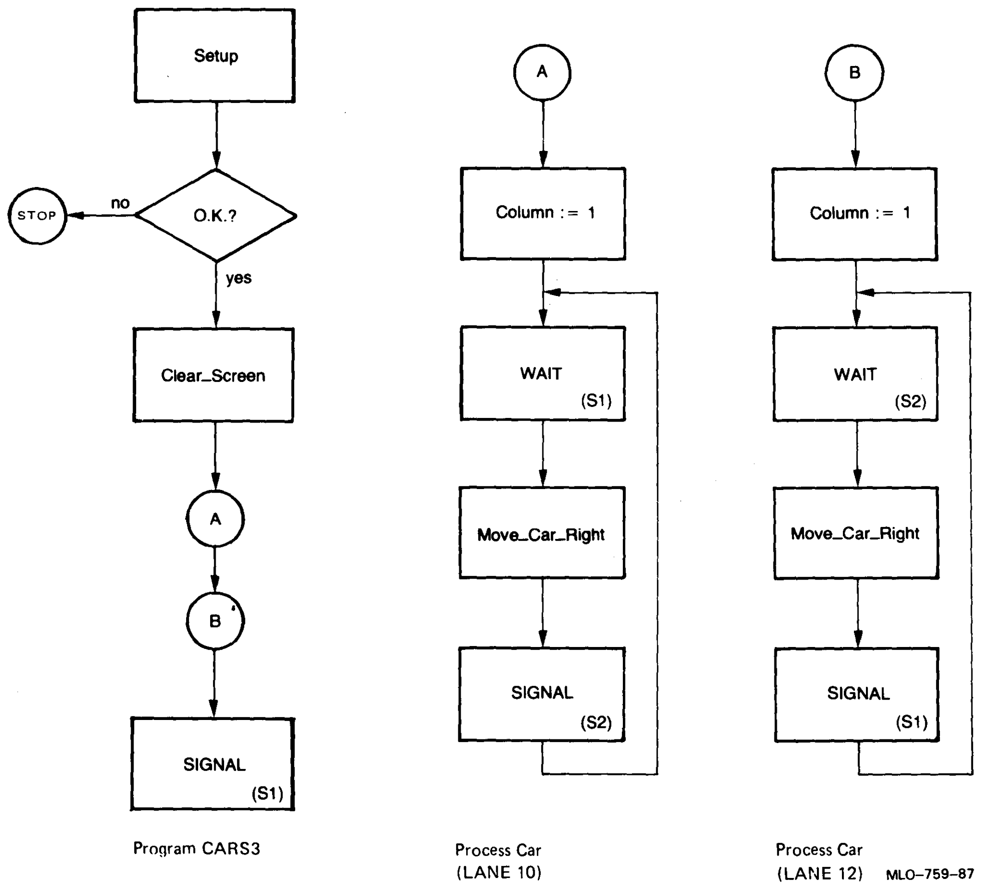
5.4.4 Execution

The following steps describe the execution of CARS3:

1. The INITIALIZE procedure (Setup) is invoked by the system startup routine, and the two binary semaphores are created.
2. The Boolean variable OK is checked to make sure the semaphore creation was successful.
3. If the semaphore creation was successful, program execution begins with a call to the Clear_Screen procedure.
4. Next, the process CAR is created, with the run-time name LANE10 and a process priority of 2.
5. Since the priority of the newly created process is higher than that of the presently running (static) process, LANE10 now runs. LANE10 initializes its Column variable and does a WAIT on semaphore S1. Finding the semaphore closed, the process blocks.
6. Execution returns to the static process (main program). The process CAR is again created, this time with the run-time name LANE12. LANE12 runs, initializes its Column variable, and WAITs on semaphore S2. Finding the semaphore closed, the process blocks.
7. Execution again returns to the static process (main program). The static process SIGNALs semaphore S1. That signal opens S1, and LANE10, which is waiting on S1, now runs. (The priority of LANE10 is higher than that of the currently running (static) process.)
8. LANE10 invokes Move_Car_Right, then SIGNALs S2. That signal allows LANE12, which is waiting on S2, to run. (LANE12 has a higher priority than LANE10.)
9. LANE12 invokes Move_Car_Right, then SIGNALs S1, making LANE10 eligible to run. Because LANE 12 has a higher priority than LANE10, LANE12 continues to run. However, on its next instruction LANE12 WAITs on S2 and blocks itself on the wait, having found the semaphore closed. This gives LANE10 a chance to run. LANE10 moves a car and signals S2, unblocking LANE12. This handshake mechanism permits each process to move its car one space. Using this mechanism overrides the effects that priority assignments would otherwise have.

Figure 5–2 shows the logic of this program.

Figure 5–2: Program CARS3 Flowchart



Chapter 6
Development Cycle Overview

The steps needed to prepare a MicroPower/Pascal application to run on your target system are:

1. Design and code source programs.
2. Compile or assemble source code.
3. Build the application image.
4. Load the application into the target.
5. Test and debug the application.
6. Rebuild the debugged application.

Figure 6–1 shows the development cycle of a MicroPower/Pascal application. Shaded areas indicate steps that are typically performed by the automatic command file generator, MPBUILD. The dotted line indicates that invoking the build utilities is an iterative procedure to be repeated several times during a typical application build. The following sections describe these steps in greater detail.

Figure 6–1: MicroPower/Pascal Application Development

Use a text editor to create Pascal and/or MACRO–11 source files, and edit hardware configuration file and driver prefix files.

Create object modules (and optional listings) from Pascal or MACRO–11 source code.

Input to the build utilities are the kernel, DIGITAL-supplied static processes, and user static processes. MERGE resolves library and intermodule references and combines individual object modules into a single, merged object module. RELOC sorts programs into read-only and read/write sections, transforms virtual addresses into actual memory addresses, and allocates memory. MIB creates the final memory image.

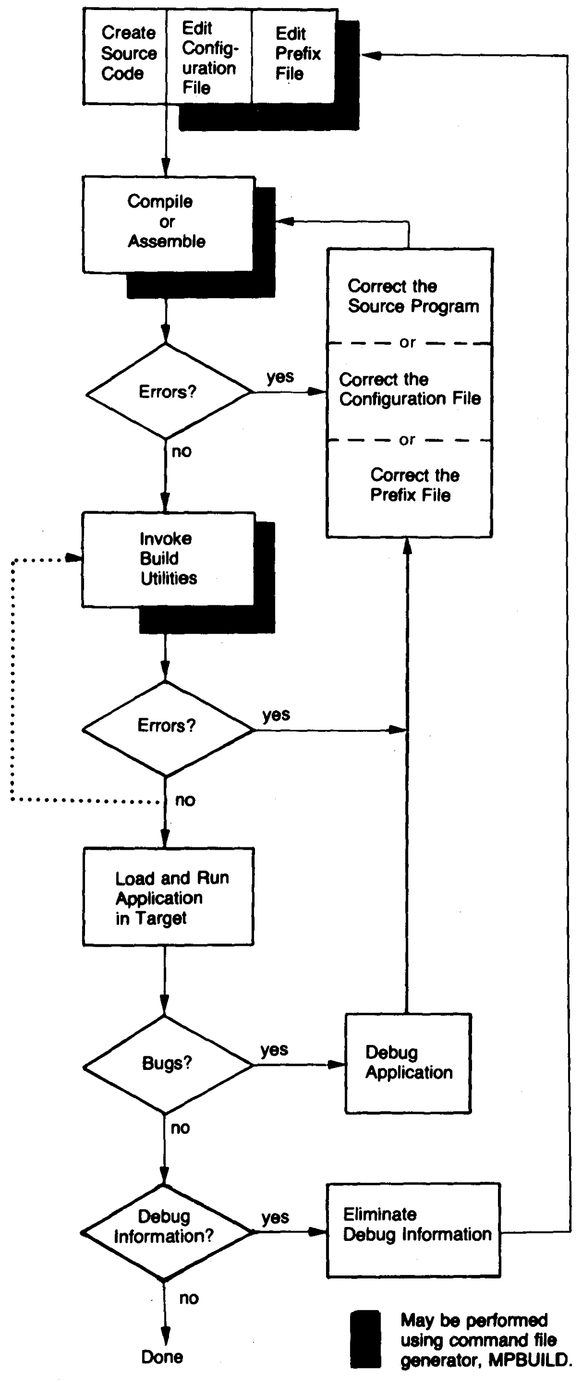
The build utilities issue diagnostic messages. You may need to correct the configuration or prefix file or correct source code.

Transfer application to target:

1. Down-line load with PASDBG.
2. Place into PROM chips.
3. Copy to bootable disk or tape.

A serial communications line connecting host and target computers allows you to use PASDBG, the symbolic debugger running in the host, to debug the application running in the target system under control of its own software.

Rebuild to eliminate debug information. Run completed application in target system.



MLO–761–87

6.1 Design and Code Source Programs

You may design your application by dividing your problem into individual tasks and coding a separate process for each task, using the techniques introduced in Chapter 4 for process synchronization and communication. Designing your program corresponds to the state reached by the Bottle Corking example (see Chapter 4), in which the job requirements were divided into separate tasks and a separate process was designed to perform each task. Flowcharts, as used in Chapter 5 for the CARS examples, can be useful in creating your concurrent design. You can use the editor available on your host system to create source programs.

You can use both the Pascal and MACRO–11 languages to write parts of a MicroPower/Pascal application. The *MicroPower/Pascal Language Guide* contains information for Pascal programmers. The *MicroPower/Pascal Run-Time Services Manual* and the *MicroPower/Pascal I/O Services Manual* cover MACRO–11 programming.

6.2 Compile or Assemble Source Code

Each Pascal program statement expands into one or more machine instructions. The MicroPower/Pascal compiler translates your Pascal source code into machine language and checks for program syntax errors and improperly declared or misused program variables. Any errors are reported to you for corrective action. If your program is error-free, the compiler produces an optimized object module.

The MACRO–11 assembler translates the MACRO–11 program into machine language. Compilation or assembly produces an object module. The automatic command procedure, MPBUILD, may perform this step for you. Note that MPBUILD accepts source or object code.

6.3 Build the Application Image

You use MPBUILD and the MicroPower/Pascal utility programs to build the application image. But, before you can use these programs, you need to define your system configuration.

6.3.1 Configuration File

The configuration file, consisting of a series of MACRO–11 instructions, provides information describing the hardware of the target system and its memory characteristics. That information is needed by the build utilities. The configuration file is assembled, and the resulting object file is input to the build utilities. The utilities use the resulting object module to tailor both the memory image in general and the kernel in particular to your specifications.

Your MicroPower/Pascal distribution kit contains several versions of a configuration file. You select the one that most closely matches your target system characteristics and, if necessary, edit that file to conform to your target requirements.

As in the compile step, this step may be accomplished during the MPBUILD process. The configuration file is described in detail in the *MicroPower/Pascal Run-Time Services Manual.*

6.3.2 MPBUILD

This section introduces the command file generator, MPBUILD, which you use for most application building. MPBUILD generates a command file that, when run, invokes the individual build utilities.

MicroPower/Pascal provides a command file generator, MPBUILD (abbreviated to MPBLD for RT–11 users), that facilitates the application-building process. Through a question-and-answer dialog, MPBUILD solicits information that enables it to create a command file containing all MACRO, Pascal, and build utility (MERGE, RELOC, and MIB) command lines needed to build your application. MPBUILD automatically includes the DIGITAL-supplied system macro and object libraries your application needs.

MPBUILD provides a fast, efficient way to build most applications. By using MPBUILD, you do not need to invoke each build utility independently. MPBUILD provides the most widely needed capabilities of the build utilities, allowing you to use it to build most applications. By executing the command file produced by MPBUILD, you create the memory image (.MIM) file that you load into your target system. (See Chapter 2 of the appropriate system user's guide for more information on MPBUILD's capabilities and limitations.)

MPBUILD is a versatile tool. It can perform an entire application build cycle or part of a build cycle. How you answer a question determines the next question you are asked. An MPBUILD cycle may include these four sections corresponding to the steps needed to build a complete application:

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

6.3.3 MicroPower/Pascal Build Utilities

The MicroPower/Pascal build utilities (MERGE, RELOC, and MIB) perform the same function as a linker in a traditional development system. In most of your application building, you do not invoke these utilities directly. The command file that MPBUILD generated for you invokes them.

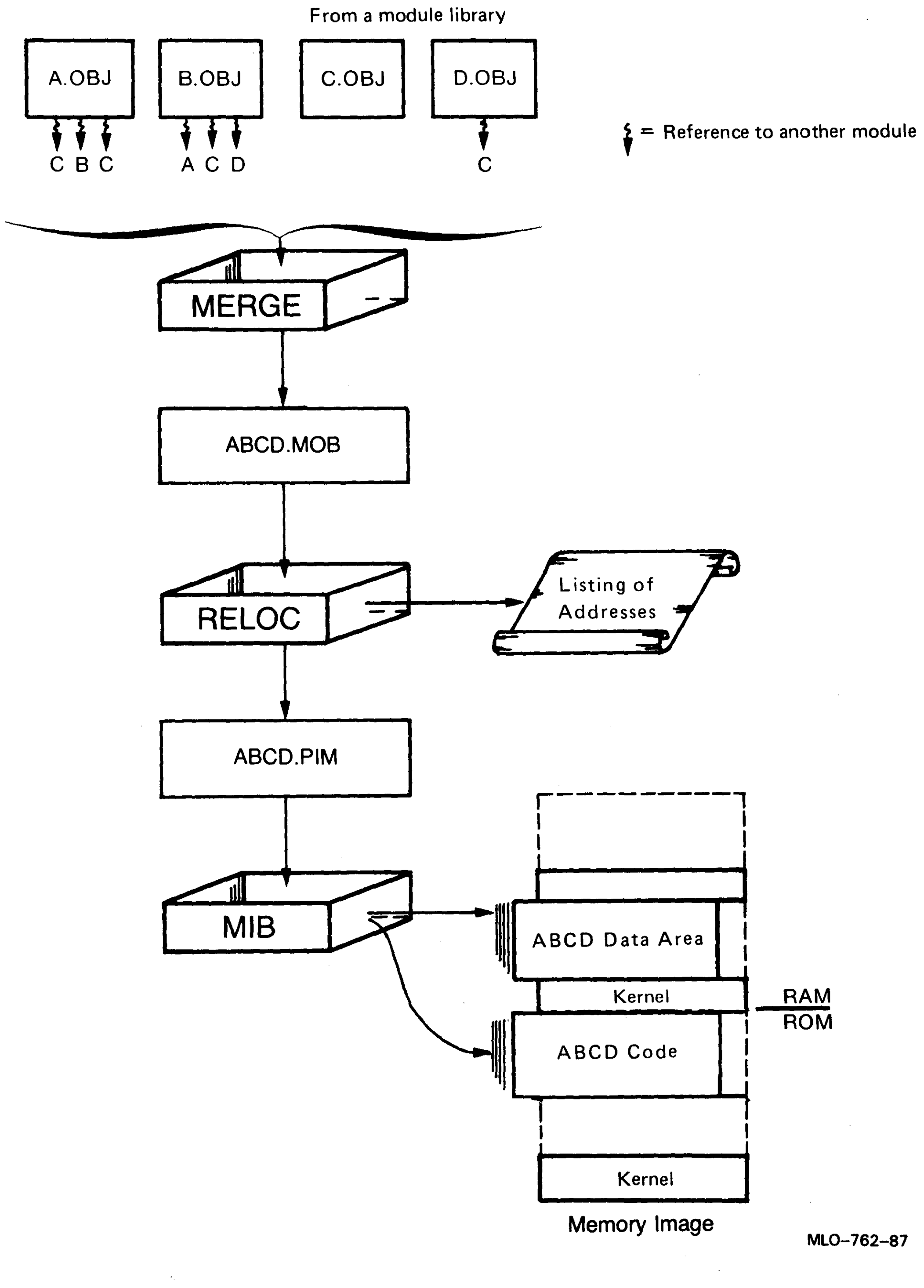
MicroPower/Pascal utilities accept as input object modules of compiled or assembled source code contained in files. You specify these files as prompted by MPBUILD or in command lines when invoking the utility programs yourself.

MicroPower/Pascal utilities perform the following functions:

- Resolve references from one input module to another or to program modules contained in module libraries
- Combine the object modules into an executable unit
- Relocate the addresses of separate sections of code in the object modules, and allocate sufficient target system memory to each part of the application
- Set up debugging information (names, addresses, and relationships of program variables) in the host and target for later use
- Produce listings and maps

Figure 6-2 follows the path of four object files as they are processed by the build utilities and become part of the target application. A description of each build utility follows.

Figure 6-2: MicroPower/Pascal Utilities



6.3.3.1 MERGE

The MERGE utility accepts multiple object modules containing compiled/assembled source code and data as input. Each object module contains program sections (p-sects) created by the MicroPower/Pascal compiler or specified by the MACRO-11 programmer. MERGE combines p-sects of identical names from all input object modules, producing a merged object module (.MOB).

MERGE also uses the symbol tables created in each object module during compilation to resolve intermodule references. For every reference to a declared external name, MERGE looks for a declared global definition in the other object modules. MERGE flags references that cannot be resolved because the referenced symbol is not defined.

The first component of the application constructed by MERGE is the kernel of basic services required to support your processes. In this case, inputs to MERGE are the object module containing the configuration information and the kernel system library of object modules. References to these modules made in the configuration file are resolved, and the selected kernel becomes ready for relocating.

6.3.3.2 RELOC

The RELOC utility assigns addresses to program sections within a merged object module. The result is a process image module (.PIM). RELOC may be used to assign base addresses to individual program sections.

RELOC separates program sections according to their read-only/read-write attributes and modifies the p-sects to execute properly at their assigned addresses. RELOC optionally creates a symbol table file containing debugging information.

6.3.3.3 MIB

The MIB (memory image builder) utility creates the executable application by placing all its components into one structure, called the memory image file. MIB inserts each process image module—kernel, system, and user—into the memory image.

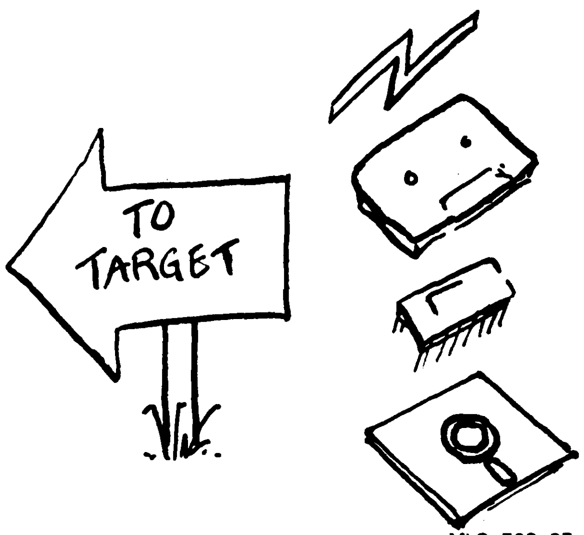
MIB first creates a memory image file and installs the kernel process image (.PIM) file in it. Once the new memory image is created with the kernel in place, MIB is invoked to include each successive process image module in the application.

MIB lets you control the placement of process images in memory. MIB can also create an optional symbol file used by the debugger program and/or include a bootstrap program for installing the application into the target system.

6.4 Load the Application into the Target

You can transfer the application to the target system in three ways:

- Down-line load if a communication link exists between the two systems
- Media and hardware-boot on the target system
- Program into a PROM chip for installation on the target



MLO–763–87

6.5 Test and Debug the Application

The MicroPower/Pascal debugger, PASDBG, is an interactive debugging tool that allows you to monitor application execution on the target system from your host system terminal. To use PASDBG, specify the DEBUG = YES option in your configuration file. This adds the debugger service module (DSM) to your application. You must also answer YES to the debug question in the MPBUILD dialog.

MicroPower/Pascal allows you to create the application one piece at a time, debugging each portion separately in the target system, then re-creating the entire application as each piece is tested.

A serial communication line connects the host system to the application running in the target system. The debugger in the host system communicates with the DSM residing in the target system. Using the debugger program residing on your host system, you can down-line load the application into the target, then using PASDBG commands, control its execution. The *MicroPower/Pascal Debugger User's Guide* includes a tutorial debugging session that features basic debugging commands.

The MicroPower/Pascal debugger is symbolic; it recognizes the names of entities in the application. PASDBG uses a symbol table created by the MIB utility. The symbol table contains the names defined in the original MicroPower/Pascal code for the application. That table represents the relationships among all symbols as well as their addresses. You determine the mode of data interpretation and the scope of symbols.

You can enter and/or display memory contents in any of the following representations:

- ASCII
- Binary
- Decimal
- Hexadecimal
- Octal
- PDP–11 instruction
- RAD50

Your choice determines the mode of the debugger's presentation of memory contents. In addition, PASDBG can interpret data in bytes, words, records, or arrays.

Any symbol used in the source code can potentially apply to more than one location in memory, because of duplicate naming or multiple uses of the same section of code during execution. As a result, for debugging, you use a path name to distinguish each instance of a symbol from all other references to that name. A typical path name may look like the following:

$$\text{program} \setminus \left\{ \begin{array}{l} \text{process} \\ \text{procedure} \\ \text{function} \end{array} \right\} \ldots \setminus \text{symbol}$$

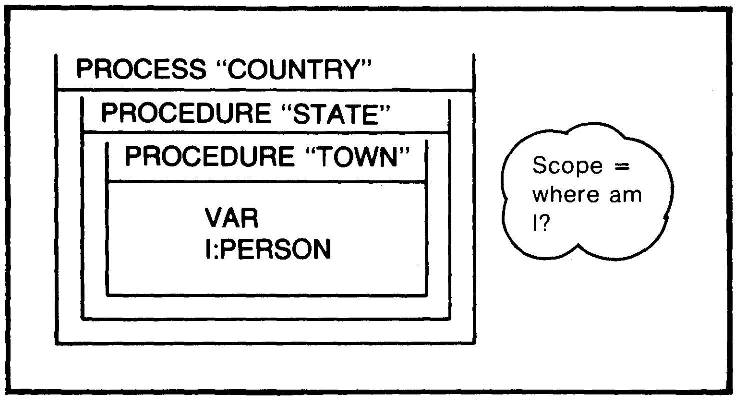
Each part of the path name leads to the next part, narrowing the scope, until the symbol has been identified. (Path names may contain more than one process, procedure, or function.) When the scope of symbols and the mode of data interpretation are clear, you can easily use the PASDBG commands to check on the progress of your application.

PASDBG gives you a range of tools to use in testing your application for errors. When PASDBG is running, you can use specific debugging commands to:

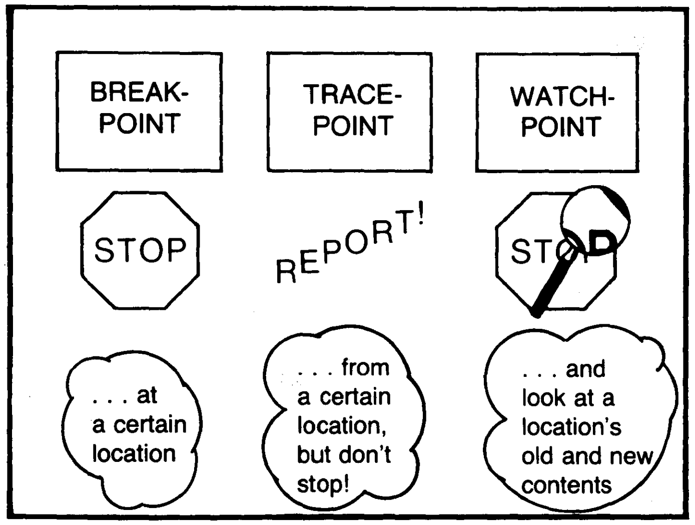
- Examine/modify memory locations
- Determine the scope of a process or a variable
- Reveal the location and value of named data structures
- Set breakpoints, tracepoints, or watchpoints throughout the program
- Set different mapping modes so you can access the entire memory space used by your application (mapped systems)
- Step through program execution one Pascal statement or one MACRO–11 instruction at a time
- Create DO lists—groups of commands that execute in sequence at breakpoints or tracepoints or as you step through an application
- Display lines of source code and display source code corresponding to the current instruction

Figure 6–3 shows MicroPower/Pascal debugger features.

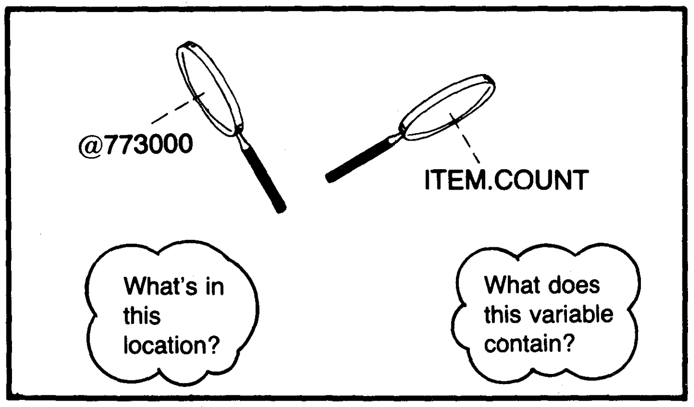
Figure 6-3: MicroPower/Pascal Debugger Features



(A) Set Scope


BREAK-POINT
TRACE-POINT
WATCH-POINT
STOP
REPORT!
STOP
. . . at a certain location
. . . from a certain location, but don't stop!
. . . and look at a location's old and new contents

(B) Control Execution



(C) Examine by Address or Variable Name

MLO-764-87

6.6 Rebuild the Debugged Application

An application constructed for debugging does not contain optimized code and does contain information required for debugging. When you are satisfied with the application, you may rebuild it to obtain an application of greater efficiency and smaller size. You do so by editing the configuration and prefix files to eliminate debugging information and then rebuilding your application, using MPBUILD.

If you choose to transfer your application to the target system by means of bootable media, add the bootstrap option (/BS or /B) to the MIB command line.

Chapter 7

Building and Running the CARS Program Examples

This chapter guides you through two examples that use the MPBUILD command procedure to build, down-line load, and run the CARS program examples. The first example steps through a complete build cycle. The second example is a partial build using an existing kernel: the one built for the first example. This example shows the use of the build command file generator for both a complete and a partial build. The programs selected for these examples are CARS2 and CARS3, the same program development examples described in Chapter 5.

7.1 Sample Build Session: CARS2 Program

To create the memory image (.MIM) and optional debug information (.DBG) files needed to run the application on your target, follow these steps:

1. Prepare source files.
2. Configure host/target and know target characteristics.
3. Invoke MPBUILD dialog.
4. Run resulting command file.
5. Use PASDBG to load resulting .MIM file into your target.

7.1.1 Prepare Source Files

Use the file CARS2.PAS contained in your MicroPower/Pascal distribution kit. On RSX systems, copy that file from directory [2,10] to your working directory. On MicroRSX systems, copy that file from directory [MPPKIT] to your working directory. On VAX/VMS systems, copy that file from MICROPOWER$LIB to your working directory. On RT–11 systems, copy that file from the LB: area to DK:.

Copy the configuration file (CFDUNM.MAC) and the prefix file (TTPFX.MAC, located in the same directory as CARS2.PAS) to your working directory. Rename the configuration file you copied to KERNUM.MAC. That unique name identifies the specific configuration file for this particular application and leaves the original configuration file intact.

7.1.2 Configure the Hardware

The host and target hardware must be configured to use PASDBG, the symbolic debugger, for down-line loading the application examples. PASDBG uses a serial line to communicate between the host and target systems. PASDBG down-line loads your application through this serial line.

The target system must include a DIGITAL VT100/200 terminal with a baud rate set to 9600 as console device and 16K words of RAM. See your MicroPower/Pascal installation guide for detailed information on using the PASDBG symbolic debugger to configure your system's hardware for down-line loading.

7.1.3 Invoke the Automatic Build Command Procedure

The MPBUILD dialog question sequence is partially determined by your answers. Y or N is sufficient to answer YES or NO questions, and bracketed answers indicate the default. You may press the RETURN key to accept the default response. A long form of dialog, providing help information for each question, is available; alternatively, you may get help on a specific question by typing a question mark (?). The examples in this chapter use the short form of dialog, since necessary explanations are provided in the accompanying text as you go along. The MPBUILD dialog questions are printed in bold type to distinguish them from the explanatory text.

Depending on which host system you use, type the appropriate command at your system-level prompt:

```
RSX users:      >MPBUILD

RT-11 users:    .IND LB:MPBLD

VAX/VMS users:  $ MPBUILD
```

LB: is the RT–11 logical disk device containing MPBLD.COM.

Note

Device:[directory] represents your default VMS or RSX device and directory. If you are working in an RT–11 environment, you should see DK:, the default RT–11 logical disk. This book indicates device:[directory].

MPBUILD responds with messages similar to the following:

```
MicroPower/Pascal-VMS Vx.xx Build Command Procedure Generator

Type "?" for help at any question.

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

Since this example uses the short form, press the RETURN key.

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

Press the RETURN key, since you are building a complete application, which includes the kernel image file as well as the application image file.

```
Kernel memory image file name ? :
```

Type the name of a file that identifies the kernel for this application. MPBUILD lets you choose any name you like. Assume you are building an unmapped kernel and want to name it KERNUM.MIM. Do not specify a file type in your answer. Type KERNUM.

```
System config file spec ? [device:[directory]KERNUM.MAC;]:
```

The system configuration file is indicated in the default response. Press the RETURN key.

```
Do you wish to modify device:[directory]KERNUM.MAC; ? [no]:
```

Do not modify the existing kernel/driver configuration file; that is, you do not want to add any DIGITAL-supplied system processes (device drivers and/or file system processes). Press the RETURN key.

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

Press the RETURN key.

```
Application memory image file name ? :
```

Type the name of the application source file. Type CARS2. (The file type is supplied by default.)

```
Output command file spec ? [device:[directory]CARS2.COM;]:
```

When executed, that command file produces the desired .MIM file needed for down-line loading. Press the RETURN key.

```
Mapped image ? [no]:
```

You are building an unmapped image. Press the RETURN key.

```
Debug support required ? [yes]:
```

You want debug support. Press the RETURN key.

```
Optimize the kernel ? [no]:
```

You do not need to optimize the kernel. Press the RETURN key.

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

Press the RETURN key.

```
Build a shared library ? [no]:
```

This application does not use shared libraries. Press the RETURN key.

```
Beginning system process section.
Driver prefix file spec ?
```

This application requires the terminal driver. Specify the TTPFX prefix file by typing TTPFX.

```
Do you wish to modify device:[directory]TTPFX.MAC; ?  [no]:
```

Press the RETURN key.

```
Driver prefix file spec ? :
```

The question is repeated in case other drivers are to be built. Press the RETURN key to indicate a null response and terminate this single-question loop.

```
User process build phase.
```

```
Beginning user process section.
```

```
User process file spec ? [device:[directory]CARS2.PAS;]:
```

MPBUILD assumes that the Pascal process you want to build is called CARS2.PAS. You can press the RETURN key or, if you want to produce a listing file, type /LIST and then press the RETURN key.

```
Additional module or library ? :
```

No additional modules or libraries are required. Press the RETURN key to indicate a null response and terminate this single-question loop.

```
User process file spec :
```

No additional user processes are to be built and installed in this memory image. Press the RETURN key to terminate this single-question loop and conclude this MPBUILD session.

MPBUILD informs you of successful completion with this message to VMS users:

```
%MPBUILD-S-Command procedure generated - DEVICE:[DIRECTORY]CARS2.COM;
```

RSX users receive this message:

```
MPBUILD-I-Command procedure generated - DEVICE:[DIRECTORY]CARS2.CMD
```

RT–11 users receive this message:

```
?MPBLD-I-Command procedure generated - DK:CARS2.COM
@EOF
```

7.1.4 Execute the Command File

Your directory now contains CARS2.COM, the command file that, when executed, produces the files needed to run your application. The VMS version of that file is reproduced here:

```
$
$! MicroPower/Pascal V2.4 Build Command Procedure
$! Application: CARS2
$! Created on 9-AUG-1987 10:16:55.55 by USER
$
$! Command file:        DISK:[USER]CARS2.COM;
$! Kernel build:        Yes
$! Application build:   Yes
$! Optimize kernel:     No
$! Debug support        Yes
$! Code type:           NHD
$! Shared library:      None
$! Unmapped Ram application
$
```

```
$ verify$ = f$verify(0)
$ @MICROPOWER$LIB:MPSETUP
$ if verify$ .eq. 1 then set verify
$
$ set on
$ on warning then goto abort$
$ on control_y then goto exit$
$ old$dir = f$logical("sys$disk") + f$dir()
$ set default DISK:[USER]
$
$ assign 'f$logical("MICROPOWER$LIB")' LB$
$ assign DISK:[USER] DF$
$
$! Build kernel and create memory image
$
$ RUN SYS$SYSTEM:MAC
  KERNUM = LB$:COMU/ML,DF$:KERNUM.MAC;
$ MPMERGE
  KERNUM.KMO = KERNUM/DE,LB$:PAXU/LB/DE
$ MPRELOC
  KERNUM,,KERNUM = KERNUM.KMO
  ,,KERNUM.DST = KERNUM.KMO/DE
$ MPMIB
  KERNUM,,KERNUM = KERNUM,,KERNUM.DST/KI/SM
$ DELETE KERNUM.OBJ;*
$ DELETE KERNUM.KMO;*
$ DELETE KERNUM.PIM;*
$ DELETE KERNUM.DST;*
$
$! Build and install device drivers
$
$ RUN SYS$SYSTEM:MAC
  TTPFX = LB$:COMU/ML,DF$:TTPFX.MAC;
$ MPMERGE
  TTPFX = TTPFX,DF$:KERNUM.STB,LB$:DRVU/LB
$ MPRELOC
  TTPFX = TTPFX,KERNUM
$ MPMIB
  KERNUM = TTPFX,KERNUM/SM
$ DELETE TTPFX.OBJ;*
$ DELETE TTPFX.MOB;*
$ DELETE TTPFX.PIM;*
$
$! Create application memory image
$
$ COPY KERNUM.MIM CARS2.MIM
$ COPY KERNUM.STB CARS2.STB
$ COPY KERNUM.DBG CARS2.DBG
$ if f$search("KERNUM.MIM;-1") .nes. "" then -
   PURGE/KEEP=1 KERNUM.MIM
$ if f$search("KERNUM.STB;-1") .nes. "" then -
   PURGE/KEEP=1 KERNUM.STB
$ if f$search("KERNUM.DBG;-1") .nes. "" then -
   PURGE/KEEP=1 KERNUM.DBG
$
$  Build and install user processes
$
$ MPPASCAL/INSTRUCTION=(NHD)/OBJECT=CARS2/LIST=CARS2 DF$:CARS2.PAS;/DEBUG
$ MPMERGE
```

```
  CARS2 = DF$:CARS2.OBJ/DE,DF$:CARS2.STB,LB$:FILSYS/LB,LB$:LIBNHD/LB
$ MPRELOC
  CARS2,,CARS2.PST = CARS2/DE,CARS2
$ MPMIB
  CARS2,,CARS2 = CARS2,CARS2,CARS2.PST/SM
$ DELETE CARS2.PST;*
$ DELETE CARS2.MOB;*
$ DELETE CARS2.PIM;*
$ DELETE CARS2.OBJ;*
$ if f$search("CARS2.MIM;-1") .nes. "" then -
   PURGE/KEEP=1 CARS2.MIM
$ if f$search("CARS2.STB;-1") .nes. "" then -
   PURGE/KEEP=1 CARS2.STB
$ if f$search("CARS.DBG;-1") .nes. "" then -
   PURGE/KEEP=1 CARS2.DBG
$
$exit$:
$
$ set noon
$ set default 'old$dir'
$ deassign LB$
$ deassign DF$
$ exit 1 .or. %x10000000
$
$abort$:
$
$ set noon
$ write sys$output ""
$ write sys$output "%CARS2.COM-F-Build error; build cycle aborted"
$ write sys$output ""
$ set default 'old$dir'
$ deassign LB$
$ deassign DF$
$ exit 2 .or. %x10000000
```

At your system prompt, type the command @CARS2 to invoke the command file and produce the .MIM and .DBG files you need to down-line load and debug your application.

7.1.5 Load and Run the Application

Depending on which host system you use, type the following command to run the symbolic debugger, PASDBG:

RSX:	**>PDB**
RT-11:	**.RUN LB:PASDBG**
VAX/VMS:	**$ PASDBG**

PASDBG responds with a debugger version number, a message reporting the state of the target system, and the following prompt:

```
PASDBG>
```

Down-line load the program into the target by typing the following command:

```
PASDBG>LOAD CARS2
```

Start program execution by typing the following command:

```
PASDBG>GO
```

You should see the following message, which indicates that the program has started:

```
[Target execution resumed - type  RETURN to stop target]
```

To stop the application, press the RETURN key. To restart the application after stopping it, type the following lines on the host terminal:

```
PASDBG>INIT/RESTART
PASDBG>GO
```

7.2 Sample Build Session: CARS3 Program

To build CARS3, you may repeat the build procedure as previously described, substituting CARS3 each time CARS2 is indicated. (CARS3.PAS may be copied from the same directory as CARS2.PAS. See Section 7.1.1 for the location of these files.) However, rather than the complete build performed in the previous build, you need only perform a partial build by using the kernel you built for the previous application.

At your system prompt, invoke MPBUILD with the appropriate command as previously described. You should see the identical system messages as before:

```
MicroPower/Pascal-VMS Vx.xx Build Command Procedure Generator

Type "?" for help at any question.

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

Press the RETURN key.

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

Type NO, since you are using KERNUM.MIM, which is already built.

```
Input memory image file name ? :
```

Type KERNUM to identify the kernel for this application. Do not specify a file type in your answer.

```
Application memory image file name ? :
```

Type CARS3. (The file type is supplied by default.)

```
Output command file spec ? [device:[directory]CARS3.COM;]:
```

Press the RETURN key.

```
Mapped image ? [no]:
```

Press the RETURN key.

```
Debug support required ? [yes]:
```

Press the RETURN key.

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

Press the RETURN key.

```
Build a shared library ? [no]:
```

This application does not use shared libraries. Press the RETURN key.

```
Beginning system process section.
Additional driver prefix file spec ? :
```

Press the RETURN key.

```
User process build phase.
Beginning Shared Library section.
Beginning user process section.
User process file spec
```

You must supply the file specification to this question. Type CARS3. Add the /LIST option if you want to produce a listing file.

```
Additional module or library ? :
```

Press the RETURN key.

```
User process file spec :
```

Press the RETURN key to terminate this single-question loop and conclude this MPBUILD session.

MPBUILD informs you of successful completion with the appropriate message.

At your system prompt, type the command @CARS3 to invoke the command file and produce the .MIM and .DBG files needed to down-line load and debug this application.

To down-line load and run this application, use the procedures described in Section 7.1.5.

7.3 Future Use of MPBUILD

When using MPBUILD again on your own, you may want to refer to the appropriate host system user's guide for information about the capabilities and limitations of MPBUILD.

Glossary

Terms in the Glossary are used throughout the documentation set.

Absolute section (A-sect)
A section of code that must reside in specific memory locations; not relocatable.

Application
A MicroPower/Pascal memory image consisting of the kernel, system processes, and one or more application programs.

Application program
In MicroPower/Pascal, a MACRO–11 or Pascal program that was developed by the user under a host operating system and that runs on a stand-alone target system. Compare with System process.

A-sect
See *Absolute section*

Asynchronous
Not operating in exact time coincidence.

Attribute
A MicroPower/Pascal language feature that facilitates control over certain aspects of a program. For example, attributes allow you to determine allocation characteristics of data structures and describe the real-time properties of processes.

Binary semaphore
A variable whose value may be 0 or 1. Binary semaphores are managed by the kernel in response to requests from processes and interrupt service routines. Each binary semaphore can have a queue of waiting processes. See also *Semaphore, Counting semaphore, Queue semaphore.*

Block

- A predefined extent of data. On mass storage devices, a block is a group of logically adjacent words or bytes. The block size for most DIGITAL mass storage devices is 512 bytes. A block of data is normally the smallest system-addressable segment of a mass storage device involved in I/O.
- In Pascal, a statement sequence delimited by the reserved words BEGIN and END; can be used anywhere a statement is used.
- In MicroPower/Pascal, to inhibit the execution of a process until some condition is met.

Blocked (process)

A process waiting for an event to occur (for example, a specific semaphore to be signaled) before continuing execution. A blocked process is in the wait-active, wait-suspended, exception-wait-active, or exception-wait-suspended state.

Block I/O

The transfer of a predefined amount (block) of data to or from a peripheral device. In block data transfers, bytes are loaded into consecutive storage locations. Only the address of the first byte needs to be specified.

Bootstrap

A short program or routine whose first instructions are sufficient to start a more complex system of programs. Bootstraps are generally used to load programs into memory from I/O devices.

Breakpoint

A location in a MicroPower/Pascal program marked for debugging with the PASDBG symbolic debugger. When a breakpoint is reached, program execution stops, and the debugger displays a program status message and may optionally execute a DO list.

Build utilities

A set of three utility programs that transform an object module produced by either the compiler or assembler into a loadable memory image and optionally create symbolic information files for debugging. See *MERGE, MIB, RELOC.*

Compilation unit

Pascal source code that is either a program or a module. See *Module, Program.*

Communication link

A physical connection between two or more processors used for transferring data. For example, a serial I/O interface is a communication link through which data is sent one bit at a time. The bit sequence is organized by prearranged protocol rules.

Concurrent

Occurring at the same time. In MicroPower/Pascal, concurrency indicates the sharing of the CPU resource by cooperating processes whose lifespans overlap. Processes appear to execute simultaneously.

Configuration file

A sequence of MACRO–11 instructions that describes the hardware of the target system and its memory characteristics.

Context

The set of data defining the environment, both hardware and software, in which a process or other code entity executes. Hardware context may include the contents of the general registers, memory-mapping registers, floating-point registers, and a processor status word. Software context includes the contents of various flags and pointers maintained by the kernel.

Context switching

Saving the hardware and software environment of a process that has lost control of the CPU and establishing similar information for a new process.

Control and status register (CSR)

A single interface register that monitors the status of an I/O device and controls its operation. Some devices have more than one CSR.

COPYB

A MicroPower/Pascal utility program that prepares a memory image for bootstrapping into the target system memory.

Counting semaphore

A variable whose value varies between 0 and 65535. Counting semaphores are managed by the kernel in response to requests from processes. Each counting semaphore can have a queue of waiting processes. See also *Semaphore, Binary semaphore, Queue semaphore*

Critical section

A portion of a process that must complete before a specific portion of another process can execute.

CSR

See *Control and status register*

Deadlock

The condition of two or more processes preventing each other from accessing needed resources. Deadlocked processes block each other, waiting for resources that are never available.

Debugger

See *PASDBG*

Declaration (section/entry)

A code fragment that describes a subprogram, process, or data item. In MicroPower/Pascal, labels and identifiers for constants, functions, procedures, processes, types, and variables must be declared. The part of a program or subprogram block that contains the declarations is called the declaration section.

Device driver

A process that drives or services an I/O device and controls the operation of the device. Synonymous with device handler.

Device register

A register associated with a hardware device. Device registers store information about the status and control of the associated device or exchange data with a device.

Down-line loading

Loading a program into the target processor's memory over a serial-line communications link from the host operating system.

Dynamic allocation

The granting of a resource to a process during execution. In dynamic allocation, the needed resource comes from a pool and may not be available when requested. In static allocation, access is established during system startup.

Dynamic process

A process not defined during target system initialization but created by the action of another process during operation of the application.

EPROM (Eraseable programmable read-only memory)

A kind of PROM that can be erased, thereby returning the device to a blank state. See also *PROM, ROM*

Ethernet

A local area network component that provides a high-speed communications channel, optimized to connect information-processing equipment in a limited geographic area.

Exception condition

An event detected by hardware or software that alters the normal flow of instruction execution. An exception condition is associated with the execution of an instruction and occurs synchronously with process execution. Examples are arithmetic overflow or underflow, illegal address references, and insufficient stack space.

Executable image

See *Memory image*

Extended address

Memory or device addresses in excess of 16 bits. Mapped memory systems use extended addresses in order to access more than 64KB of address space.

External symbol

A link between independently compiled or assembled programs or modules. An external symbol in one module represents a symbol globally defined in another module.

Flag

A variable or register used to record the status of a program or device.

Function

A Pascal program unit that returns a value when executed. A function consists of a heading, which includes the function's name and result variable type, and a block.

General process

In mapped target systems, a process without special mapping to kernel or device register areas of memory.

Global symbol

A link between independently compiled or assembled program or module. A global symbol is defined in one module and can be referenced from other modules.

Handler

See *Device driver*

Heap

An area of memory in a Pascal-implemented MicroPower/Pascal application program for dynamic allocation of pointer objects. Dynamic processes' stacks are allocated from the heap.

Host processor

A computer running an RSX, RT–11, or VMS operating system on which MicroPower/Pascal application programs are developed.

Intermodule reference

A reference made in one module or program to a symbol defined in another module or program.

Interrupt

A signal from a device to the processor that changes the flow of instruction execution on the interrupted processor. Interrupts occur asynchronously with respect to the execution of processes.

Interrupt service routine (ISR)

A routine designed to execute at a CPU priority level greater than 0 when a particular device signals the processor with an interrupt. The processor locates ISRs in memory, using an address vector triggered by the interrupting device. ISRs are also called interrupt-handling routines or interrupt handlers.

ISR

See *Interrupt service routine*

Kernel

The processor-specific portion of the MicroPower/Pascal run-time support software that manages the processor state, handles interrupts, and provides interprocess synchronization and communication services. The kernel provides basic, real-time control and service functions for all processes in the target system. Kernel components include the system scheduler and dispatcher and many service functions that can be invoked by the user. Categories of kernel services include:

Creating/deleting processes
Dispatching exceptions
Dispatching interrupts to the appropriate interrupt service routines
Managing (allocating) resources
Scheduling processes
Synchronizing processes

Library file

A file containing one or more relocatable object modules used to incorporate other programs. These program modules might be used repeatedly in a program or by more than one program. Library file modules are merged with user-program modules during MicroPower/Pascal development.

Library module

A module from a library file.

Linking

Converting object modules to a format suitable for loading and executing. Linking object modules:

- Assigns absolute addresses
- Produces a load map and creates a symbol table
- Relocates the program sections within the object modules
- Resolves global symbols that are defined in one module and referenced by external symbols in another
- Searches library files to locate unresolved global symbols

In MicroPower/Pascal development, the linking functions are performed by executing the MERGE, RELOC, and MIB utilities.

Load map

A table produced during creation of a MicroPower/Pascal application program that provides information about the load module's (memory image's) characteristics; for example, the transfer address, the global symbol values, and the low and high address limits of the relocated code.

Load module

A program in a format for loading and execution (relocated, with references to labels and identifiers resolved). A completed memory image file is the load module for a MicroPower/Pascal application.

Mapped memory

Memory that is divided into virtual segments, or pages, each located separately in (mapped into) physical storage. Mapping translates the 16-bit virtual addresses used with LSI family processors into 18-bit or 22-bit physical memory addresses. Specifically in the LSI–11/23, up to four 64K-byte virtual address spaces can be mapped into noncontiguous 8K-byte segments (18-bit mode) or 64 different spaces (22-bit mode).

Memory image

The file resulting from running the MIB utility that contains the image of the application program as it appears in the target system memory. This file can be down-line loaded, bootstrapped from bootable media, or placed in ROM for execution in the target. The memory image file name has the extension .MIM.

Memory image builder (MIB)

The MicroPower/Pascal utility program that combines the following components into a memory image file:

- Bootstrap loader (if needed)
- Kernel
- Relocated process image file (file containing an image of the program as it appears in its portion of the target system memory)

This memory image file is loaded into the target processor. MIB optionally creates a debug symbol table file (.DBG).

MERGE

The MicroPower/Pascal utility program that combines two or more object modules, resolving intermodule references, if possible, and updating the relocation directories.

MIB

See *Memory image builder*

Modular programming

A method of constructing a program from several programming units called modules. Modular programming helps organize the program concepts. Code sections can be written either as separate parts of one source program (procedures in MicroPower/Pascal) or as distinct modules compiled into separate, cross-referenced object modules to be linked into one load module.

Module

In MicroPower/Pascal, a compilation unit with a header containing the reserved word "MODULE" and one or more declarations. Compare with *Program*.

MPBUILD

A tool that you use to build most applications. A command file generator that uses a question/answer dialog to build a command file that, when invoked, builds your application. May perform an entire build cycle or part of a build cycle.

Multiprocessing

The simultaneous execution of two or more parts of the same program by two or more processors.

Multiprogramming

Apparently simultaneous execution of two or more programs or portions of a program by a single processor. Since these programs execute instructions alternately in the processor, more than one program is in progress at one time.

Mutex

In MicroPower/Pascal, a mechanism to enable efficient mutual exclusion in the same program. A Mutex is an optimization of a binary semaphore.

Mutual exclusion

A mechanism that protects the integrity of shared resources, preventing simultaneous access.

Object module

The primary output of an assembler or compiler, which can be linked with other modules and loaded into memory as part of an executable program. The object module is composed of the relocatable machine language code, relocation information, and a global symbol table that defines the labels and identifiers meant to be referenced by other parts of the program. The object module may also contain optional debug symbol information.

Object time system (OTS)

The MicroPower/Pascal library of object modules that is called by compiled or assembled code to perform predefined operations.

Option

An element of a command or command string that enables the user to select alternatives associated with the command. In MicroPower/Pascal, an option consists of a slash character (/) followed by the option name and, optionally, a colon and an option value. Synonymous with "qualifier" and "switch."

OTS

See *Object time system*

OVERLAID attribute

In MicroPower/Pascal language, an attribute applied to a program data area or module data area to be shared with another program or module during execution.

Packet

A communication message that is passed between processes.

Packet Queue

A synchronization mechanism for serializing the arrival of packets for a process. The process pulls one packet at a time from the queue.

Page address register (PAR)

A register containing the base address of one of eight 8K-byte blocks of physical memory.

Page descriptor register (PDR)

A register containing access information about each 8K-bytes memory page whose base is described by the corresponding PAR (length, R/O versus R/W, etc.).

PAR

See *Page address register*

PASDBG

The symbolic debugger for MicroPower/Pascal applications. PASDBG allows symbolic, interactive access and control of a target system. PASDBG recognizes Pascal data types, user-defined data types, kernel structures and can report the target system state.

PCB

See *Process control block*

PDR

See *Page descriptor register*

Physical address

The hardware address of a main memory location. Physical addresses in the LSI–11/23 are from 0 to 4M bytes (in 22-bit mode with optional MSV–11L). Virtual addresses of up to 16 bits (64K bytes) can be relocated into the larger physical address space by memory mapping. See also *Virtual address*

Primitive

A fundamental operation performed by the kernel when requested by a process in the application. Primitive operations are indivisible and must complete; they do not block themselves. In the MicroPower/Pascal language, primitives are invoked implicitly by calls to predefined real-time procedures.

Privileged process

In mapped target systems, a process with access to both kernel and device-register areas of memory.

Procedure

A Pascal program unit that consists of a procedure heading and a block. When called, the procedure is executed as a unit.

Process

A program unit that may operate in parallel (concurrently) with other program units. Processes may be implemented on multiprocessors or, through interleaved execution, on a single processor. Specifically, a process is an independent scheduling unit that represents an asynchronous CPU activity relative to other processes for the purposes of the MicroPower/Pascal kernel. Synchronization among processes is achieved by primitive operations provided by the kernel. A process is similar to a task in other programming contexts.

Processes are the basic, logical entities of the MicroPower/Pascal application. Rates of progress may vary, since processes execute cooperatively in the target processor, affecting one another's execution by operations on semaphores. A process is defined by hardware and software context information stored in process control blocks. Four types of processes exist in a mapped environment: general, device-access, driver, and privileged.

Process control block (PCB)

The activation record of a MicroPower/Pascal process. The PCB preserves the software and hardware context of the process and reflects the state of the process (see *Process state*). The PCB contains:

- Hardware context of the process (including the contents of the general registers, the FPU registers, and the PSW.)
- Process priority
- Software context of the process (contents of associated flags and pointers maintained by kernel operations)
- State code and substates
- State queue pointers

Process name

A 6-character alphanumeric string that identifies a process. When a process is created, the user specifies its name, which is stored in the kernel's system name table. The process name can also be kept in a process descriptor block allocated from the address space where that process resides.

Process state

Every process exists in one of the possible process states at any time. These states are:

- Exception-wait $\left\{\begin{array}{l}\text{Active}\\ \text{Suspended}\end{array}\right\}$
- Ready $\left\{\begin{array}{l}\text{Active}\\ \text{Suspended}\end{array}\right\}$
- Run
- Wait $\left\{\begin{array}{l}\text{Active}\\ \text{Suspended}\end{array}\right\}$

Process synchronization

In MicroPower/Pascal, coordinating the execution of interrelated processes. Semaphores and ring buffers are basic mechanisms that synchronize MicroPower/Pascal processes.

Program

In MicroPower/Pascal, a compilation unit composed of a header containing the reserved word "PROGRAM," a declaration section, and an executable section. When acted on by the MicroPower/Pascal utilities, a program results in a static process within the application. Compare with *Module*.

Program section (P-sect)

One of four named units created by the MicroPower/Pascal compiler from Pascal source code. These units are used to apportion the target system memory into sections for:

- Executable code
- Memory space for stack and heap
- Storage for constants
- Storage for global variables

The memory for the stack is dynamically allocated during execution and has the read/write (R/W) attribute. Others are read only (RO).

Programmable read-only memory (PROM)

A type of read-only memory on a silicon chip that is manufactured in the blank state (zeros or ones). You give the bit pattern for your application program by formatting the chip in a PROM formatter. The bit pattern is permanent. See also *ROM, EPROM*

PROM

See *Programmable read-only memory*

P-sect

See *Program section*

Queue elements

Areas of data managed by the kernel, allocated from the kernel pool, and used for communication between processes.

Queue semaphore

An extension of a counting semaphore. In addition to its own integer value, a queue semaphore has queue elements associated with it. The number of queue elements equals the value of the semaphore. When a process signals the queue semaphore, it increments the semaphore by 1 and adds one element to the queue. When a process waits on the queue semaphore, it removes one queue element and decrements the semaphore. If the semaphore is 0 (and the associated queue is empty), the waiting process blocks itself and cannot resume until another process signals the semaphore and adds an element to the queue. See also *Semaphore, Binary semaphore, Counting semaphore*

Radial-serial protocol

A prearranged sequence of signals on a communication line. The TU58 device communicates with its device driver process, using radial-serial protocol over the serial line that connects it to the processor.

RAM

See *Random-access memory*

Random-access memory (RAM)

A read/write memory device. Application programs that require storage space for variables and buffers can write data into RAM locations and can read the contents of RAM locations. Compare with *ROM.*

Read-only memory (ROM)

A memory device manufactured with binary values placed in each addressable location. The contents of ROM locations cannot be changed after they are manufactured. (ROM chips are purchased from an integrated circuit manufacturer who has placed a purchaser-supplied program on the chips.) Compare with *RAM.* See also *PROM, EPROM*

RELOC

A utility that produces a process image file by associating each program section in a merged object module to a specific set of virtual addresses. This is one step in the process of linking a MicroPower/Pascal application program.

Ring buffer

A system data structure designed primarily for character-oriented data communication between processes. Both input and output operations can be performed simultaneously on the same ring buffer.

ROM

See *Read-only memory*

Scheduling

Determining which process is allocated control of the processor after a significant event. In MicroPower/Pascal applications, scheduling is performed by the kernel, based on the priorities of the currently eligible (ready-active) processes and the running process.

Scope

The portion of the program in which an identifier has a particular meaning.

Semaphore

A nonnegative integer variable on which two types of operations, wait and signal, are defined. For the semaphore variable S, the operations are:

- Signal(S): S is incremented by 1.
- Wait(S): If S is greater than 0, S is decremented by 1, and the process continues execution. If S is 0, the process is blocked until S is greater than 0. S is then decremented by 1, and the process continues execution.

The previously defined operations are indivisible. Processes use semaphores to coordinate their concurrent execution and to protect shared resources from destructive alteration. A process waiting on a semaphore is able to resume execution only after the semaphore has been signaled to a nonzero value by another process.

Shared Library

A library of subroutines or OTS routines that exist once in physical memory but can be shared (called) by different processes in different address spaces.

Significant event

Any event that causes a process to move into the ready-active state or out of the run state, thus invoking the kernel scheduler to determine which process controls the CPU. A significant event may occur synchronously with process execution (a primitive operation) or asynchronously (an external interrupt). Examples are:

- Creating or deleting a process
- Occurrence of clock interrupt
- A process blocking itself by waiting on a semaphore
- Resuming a suspended process
- Signaling a semaphore on which a process is waiting
- Suspending a running or ready-active process

Stack

For MicroPower/Pascal application programs, the area of memory within the heap that is allocated for each dynamic process within the program to hold nonstatic process variables.

Static allocation

Dedicating a resource to the process that allocated it. Static allocation occurs during application building (compiling and linking).

Static process

A process that exists in the application after initialization. (The process is always present after power is on or system-reset processing is completed.) A static process corresponds to a MicroPower/Pascal program. In MACRO–11, a static process is defined by the DFSPC$ macro.

Stepping

A debugging technique in which a process stops after executing each statement or instruction.

Stopped

A process can be forced to reenter itself at its termination entry point when it or another process executes the STOP procedure or STPC$ macro. Either action stops the process. Exceptions for which no exception-handler request exists and which are not handled by the process also stop the process.

Structure ID

A 48-bit value assigned to a structure when the structure is created. This value consists of the structure index and structure serial number.

Suspend, suspended

A process is suspended when it is placed in suspend state by the Pascal SUSPEND function or the MACRO–11 SPND$ macro. A suspended process can resume only after another process has performed the RESUME primitive operation.

Symbol file

A file containing a symbol table. The Debugger Symbol Table (.DBG) file contains the information that PASDBG needs to reference addresses and modify program variables symbolically rather than by virtual or physical addresses.

Symbol table

A list of names that can be referenced in a module. Symbol tables link calls from other sources to the named entities within a module.

System data structures

Data structures dynamically allocated by the kernel in system-common memory to service process requests. Examples of system data structures include message packets, process control blocks, ring buffers, semaphores, and system queues.

System process

A process supplied as part of the MicroPower/Pascal package for inclusion in user-created applications. System processes furnish commonly needed services and are usually privileged in mapped targets.

Target processor

A microcomputer in which the MicroPower/Pascal application is intended to run, once developed on the host processor.

Termination point

The location in a process where execution begins when that process is stopped. A termination point is not required for every process. The END statement of the process is the default. A procedure can be declared as the termination point of a process by using the TERMINATE attribute.

Tracepoint

Reports when a certain program statement is executed but does not cause PASDBG to halt the application.

Trap

An exception condition caused by executing a trap instruction such as the EMT, TRAP, BPT, and IOT instructions. Trace traps (T-bit traps) are also included. The exception, which occurs after execution of the trap instruction, is therefore synchronous with process execution.

Unmapped memory

Contiguous physical memory that is not managed by memory-management hardware; unmapped virtual and physical addresses are identical.

Virtual address

An octal value from 0 to 177777; a 16-bit address in a program's (maximum) 64K-byte address space. In unmapped systems, virtual addresses and physical addresses have a one-to-one relationship. In mapped systems with multiple address spaces, virtual addresses and physical addresses have a one-to-many relationship.

Wait, waiting

A process in the wait state. A process waits on a ring buffer or semaphore, unable to change states or to resume execution until the ring buffer or semaphore has been signaled by another process.

Watchpoint

Stops execution when a certain memory location is modified.

Word

Two bytes; 16 bits.

Index

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603-884-6660	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Rest of U.S.A. and Puerto Rico*	800-258-1710	
Canada	800-267-6219 (for software documentation) 613-592-5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order desk
Internal orders (for software documentation)	—	Software Distribution Center (SDC) Digital Equipment Corporation Westminster, MA 01473
Internal orders (for hardware documentation)	617-234-4323	Publishing & Circulation Serv. (P&CS) NR03-1/W3 Digital Equipment Corporation Northboro, MA 01532

* Prepaid orders from Puerto Rico must be placed with DIGITAL's local subsidiary (809-754-7575)

Introduction to MicroPower/Pascal
AA–M388C–TK

READER'S COMMENTS

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

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

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

Please indicate the type of user/reader that you most nearly represent:

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

Name ______ Date ______

Organization ______

Street ______

City ______ State ______ Zip Code or Country ______

— — Do Not Tear — Fold Here and Tape — — — — — — — — — — — — — — — —

digital

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
MLO5–5/E45
146 MAIN STREET
MAYNARD, MA 01754–2571

— — Do Not Tear — Fold Here — — — — — — — — — — — — — — — — — —

Cut Along Dotted Line