Introduction to MicroPower/Pascal

digital

Introduction
to
MicroPower/Pascal™

AA-M388A-TC

digital equipment corporation · maynard, massachusetts

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

The following are trademarks of Digital Equipment Corporation:

DEC	**digital**	RSTS
DECnet	Edusystem	RSX
DECsystem-10	IAS	UNIBUS
DECSYSTEM-20	MASSBUS	VAX
DECUS	MicroPower/Pascal	VMS
DECwriter	PDP	VT
DIBOL	PDT	

M15900

Contents

Chapter 4 Processes

Chapter 5 MicroPower/Pascal and Concurrent Programming

Chapter 6 Application-Development Example

Appendix A Source Program for Application Example

Glossary

Index

Figures

Preface

Objectives and Assumptions

This manual introduces the basic concepts and components of MicroPower/Pascal. After reading this book, you will know the capabilities of the product and its uses. This introduction provides you with an overall perspective; the other books in the manual set describe each aspect of MicroPower/Pascal.

We assume that you are familiar with the programming language Pascal or with MACRO-11. We also assume that you are acquainted with the RT-11 operating system. The MicroPower/Pascal manual set does not describe RT-11 in detail; nor does it present a tutorial on programming in Pascal.

Structure of this Manual
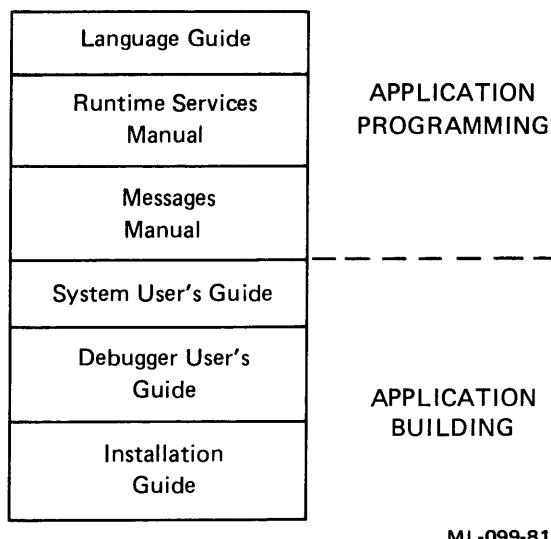
This *Introduction to MicroPower/Pascal* comprises six chapters and a glossary:

- Chapter 1 sketches a general description of MicroPower/Pascal and introduces concurrent programming concepts.

- Chapter 2 describes the two computer systems involved with MicroPower/Pascal: the host and the target.

- Chapter 3 presents the five steps involved in using MicroPower/Pascal.

- Chapter 4 describes MicroPower/Pascal processes.

- Chapter 5 illustrates some specifics of concurrent programming with MicroPower/Pascal.

- Chapter 6 demonstrates how to create and run a concurrent sample program.

The Glossary lists MicroPower/Pascal terms that appear throughout the manual set and is a valuable aid to understanding MicroPower/Pascal. Use the Glossary whenever you are in doubt about the meaning of a term.

The Rest of the Manual Set

In addition to this manual, the MicroPower/Pascal documentation set consists of the manuals shown in the following figure:

```
┌─────────────────────────┐
│     Language Guide      │
├─────────────────────────┤        APPLICATION
│   Runtime Services      │        PROGRAMMING
│        Manual           │
├─────────────────────────┤
│       Messages          │
│        Manual           │
├─────────────────────────┤  ── ── ── ── ── ──
│   System User's Guide   │
├─────────────────────────┤
│    Debugger User's      │
│        Guide            │        APPLICATION
├─────────────────────────┤        BUILDING
│     Installation        │
│        Guide            │
└─────────────────────────┘
                                    ML-099-81
```

```
┌─────────────────────────┐
│     MicroPower/Pascal   │
│    System User's Guide  │
└─────────────────────────┘
```

This manual describes the operation of MicroPower/Pascal utility programs in detail, including writing and creating target system applications and loading the application into the target microcomputer. Chapter 2 presents a brief overview of the RT–11 operating system used as the host system. Chapter 3 covers RT–11 text editors. Chapter 4 explains how to invoke the Pascal compiler and use its options. Chapters 5 to 9 describe the utility programs and explain how they create target system applications. Chapter 10 covers creating and maintaining libraries of macros and object modules. Chapters 11 to 14 present three methods of loading an application program into the target system.

Refer to the *MicroPower/Pascal System User's Guide* for a description of the utility programs used to develop the application. The *MicroPower/Pascal System User's Guide* contains detailed information on linking, loading, and debugging a MicroPower/Pascal application.

```
┌─────────────────────────┐
│     MicroPower/Pascal   │
│      Language Guide     │
└─────────────────────────┘
```

This manual presents the programming language Pascal and its special extensions for use in microprocessor application programming. The manual covers

Pascal's format and structure, basic concepts, data types, statements, procedures, and functions, as well as specific MicroPower/Pascal compile-time aids and error handling.

As a MicroPower/Pascal programmer, you must understand the real-time extensions of the MicroPower/Pascal language in order to design and code a successful application. Refer to the *MicroPower/Pascal Language Guide* for keywords and Pascal statements that have been added by MicroPower/Pascal specifically for programming concurrent processes in a real-time application. Chapters 5 and 6 of this *Introduction to MicroPower/Pascal* present a brief overview of concurrent programming.

```
MicroPower/Pascal
Runtime Services Manual
```

This manual describes the services and functions supplied by the MicroPower/Pascal runtime system. The manual describes the kernel services and services provided by system processes such as device drivers and the clock process. Also included is a guide to writing a device driver. The *MicroPower/Pascal Runtime Services Manual* covers MACRO-11 programming for MicroPower/Pascal applications.

Refer to the *MicroPower/Pascal Runtime Services Manual* for details of the system services provided by the MicroPower/Pascal runtime system (whose modules will make up a part of your application). This manual explains how the MicroPower/Pascal application program supplies basic functions such as interrupt dispatching and process scheduling, as well as how applications handle concurrent processes.

```
MicroPower/Pascal
Debugger User's Guide
```

This manual describes the MicroPower/Pascal symbolic debugger, PASDBG. Chapter 1, an overview of PASDBG's features, describes the target system hardware and software environment necessary to use PASDBG. Chapter 2 presents general techniques for using PASDBG and specifics on debugging MicroPower/Pascal programs. Chapter 3 lists PASDBG commands, organized by function. Each command is explained with illustrations.

```
MicroPower/Pascal
Messages Manual
```

This manual lists and describes the MicroPower/Pascal utility program messages and the Pascal messages. Chapter 2 covers the utility programs included in the MicroPower/Pascal distribution kit. It lists messages for COPYB,

DLLOAD, MERGE, RELOC, MIB, and PASDBG. In addition, Chapter 2 describes the cause of the error and the most likely recovery procedure. Chapter 3 details the Pascal command line, compile-time, runtime, and compiler malfunction error messages. The command line messages are described in the same way as utility program messages.

```
MicroPower/Pascal
Installation Guide
```

The *MicroPower/Pascal Installation Guide* covers installing MicroPower/Pascal software on the RT–11 V4.0 system to be used for developing MicroPower/Pascal application software.

```
MicroPower/Pascal
Reference Card
```
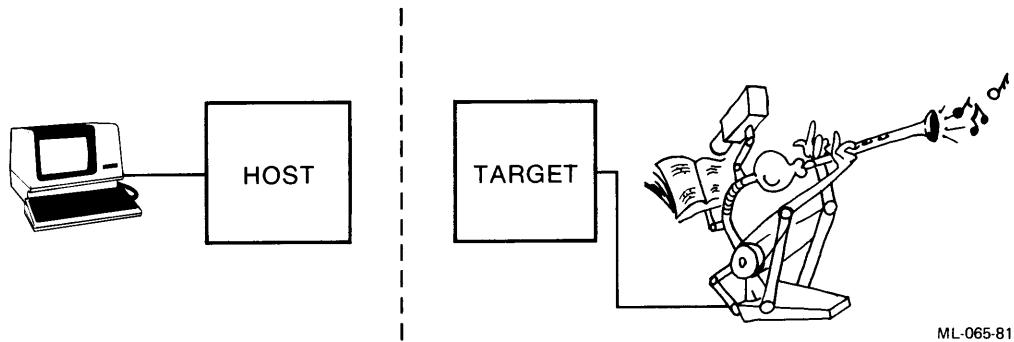
The *MicroPower/Pascal Reference Card* contains tables of kernel service requests, system process requests, start-up commands, utility program commands, PASDBG symbolic debugger commands, and other useful reference information.

Chapter 1
This Is MicroPower/Pascal

1.1 Introducing MicroPower/Pascal

MicroPower/Pascal is a package of software used to create concurrent real-time application programs. You develop these programs, or applications, on a bounded RT-11 XM host system. These applications execute on a separate target microcomputer — an LSI-11 or SBC-11/21 processor in a dedicated computing environment.

ML-065-81

Each application is constructed especially for its target system, with the exact set of operating system services needed. All code is optimized to execute efficiently and to minimize the amount of memory required. In addition, portions of the application that do not change value during the operation can be placed in read-only memory (ROM) in the target.

1.1.1 High-Level Programming Language

You can write MicroPower applications by using an extended version of the Pascal language. MACRO-11 assembly language can also be used; this manual, however, emphasizes Pascal as the programming language appropriate for most applications. MicroPower/Pascal is a superset of standard Pascal, with added data types, functions, and language constructs especially suited to microcomputer programming.
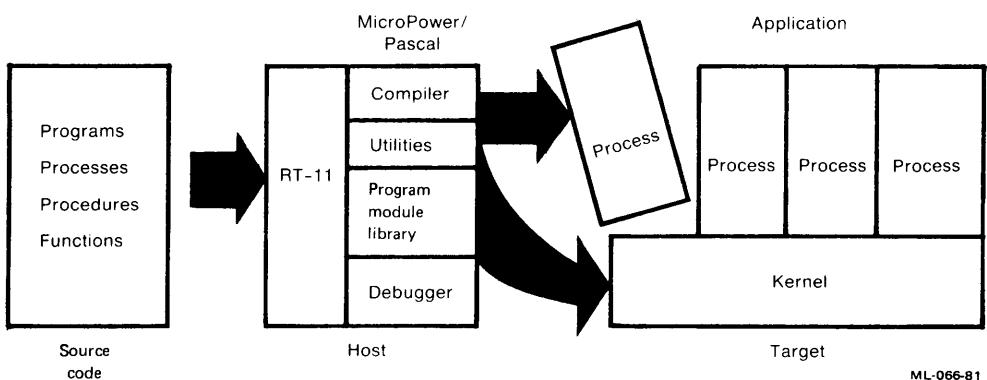
1.1.2 Concurrent Programming Capability

Concurrent programming is the structuring of your application into pieces, or processes, that appear to execute simultaneously. Concurrent processes make efficient use of the target microcomputer. MicroPower/Pascal processes are not supported by a conventional operating system; instead, every application contains its own customized set of supporting routines, called the kernel.

1.1.3 Development Tools

The MicroPower/Pascal compiler transforms your source code into a set of optimized machine instructions. Utility programs link the compiled code with a customized kernel to create the application. The kernel contains only those operating system services needed to execute your application.

The MicroPower/Pascal utility programs construct your application one piece at a time (see Figure 1-1). This modular approach makes testing and debugging easier and simplifies the job of updating and expanding applications. The resulting application can be placed into the target system by one of several different loading methods, including PROM chip transfer.

Figure 1-1:   Constructing a MicroPower/Pascal Application

1.1.4 Symbolic Debugger

You can debug the application interactively over a communication link by using the MicroPower/Pascal symbolic debugger software running under RT-11 in the host. An application constructed for debugging does not contain optimized code. After debugging, you can redevelop the code into an optimized application of greater efficiency and smaller size.

1.1.5 Device and File Support

MicroPower/Pascal provides precompiled driver processes that act as interfaces between your application and various DIGITAL devices (such as the RX02 floppy disk drive). You can also include a file system process and modules that allow you to create, access, and maintain data on target mass storage devices in RT-11-compatible format.
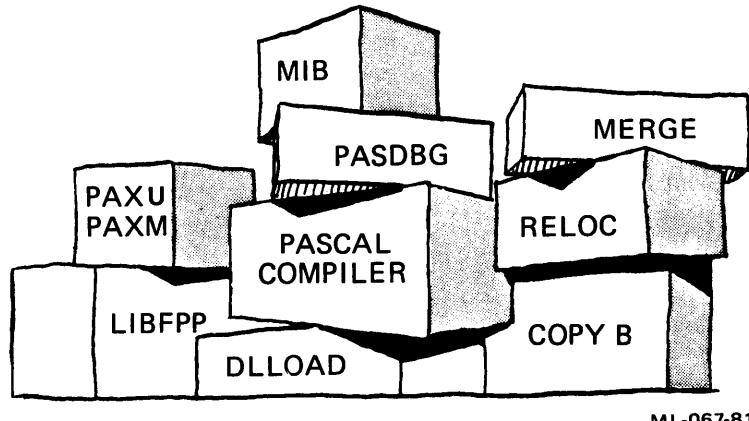
Driver processes and the file system, which are fully accessible from your Pascal source code, greatly simplify I/O operations.

1.2 MicroPower/Pascal Software Product

MicroPower/Pascal is shipped as files on either RL02 disks or RX02 floppy diskettes, to be installed in the host RT-11 operating system. The *MicroPower/Pascal Installation Guide* explains how to copy these files and set the host system to begin application development.

The MicroPower/Pascal software product consists of the following:

1. A Pascal compiler that operates on a superset of the standard Pascal language to produce optimized machine code. (MicroPower/Pascal adds extra data types and functions to standard Pascal as well as special language constructs that support concurrent programming.)

2. Several utility programs that construct the MicroPower/Pascal application and load it into target system memory. These utilities run on the RT-11 operating system in the host.

3. A library of object (program) modules, some or all of which will be included in a software kernel in each application. This ROMable kernel supports the many features and extensions of MicroPower/Pascal and supplies the target system with basic services.

4. A symbolic debugger for testing and correcting the installed application, using source code scopes, labels, and identifiers.

5. A subset of the RT-11 operating system, supplied as a pre-SYSGENed RT-11 XM monitor and utility programs for use as the host operating system.

ML-067-81

1.3 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.

The following questions and answers introduce basic concepts of concurrent programming in MicroPower/Pascal.

Question: What is a concurrent program?

Answer: A MicroPower/Pascal application program contains separate sequences of instructions, called processes, which perform distinct tasks. These independent activities are carried out in parallel (concurrently) by a single processor. Thus, the application is organized as if to perform many activities at once; you treat each activity separately in your source code. Chapter 4 describes MicroPower/Pascal application processes.

Key to this compartmentalized application design is the way in which processes share control of the CPU and other common resources (data areas and peripheral devices, for example). The MicroPower/Pascal language contains mechanisms for synchronizing executing processes and mutually excluding processes from shared resources. These mechanisms, called semaphores, are described in Chapter 5.

In summary, MicroPower/Pascal processes can be thought of as running simultaneously, even though there is only one target system CPU. (The lifespans of processes overlap, but only one process controls the CPU at a time.) The size of the application is kept to a minimum, since no general-purpose operating system is required to referee processes; instead, your concurrent design coordinates them.

Question: How do I plan and write concurrent programs?
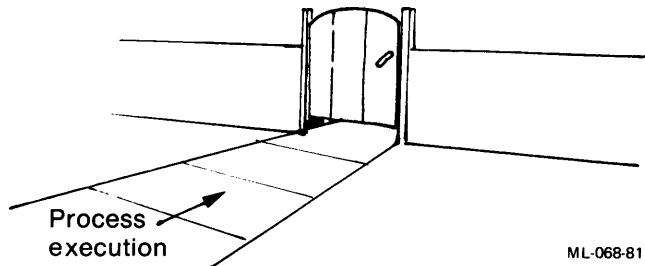
Answer: Chapters 5 and 6 give examples of the concurrent approach to program design. A programmer usually constructs a program from a flowchart of sequential activities. By contrast, a programmer constructs a concurrent program from several parallel flowcharts, each of which communicates with the others to synchronize execution. The most important part of concurrent programming is coordinating processes to avoid interference.

Question: How do I coordinate my processes?
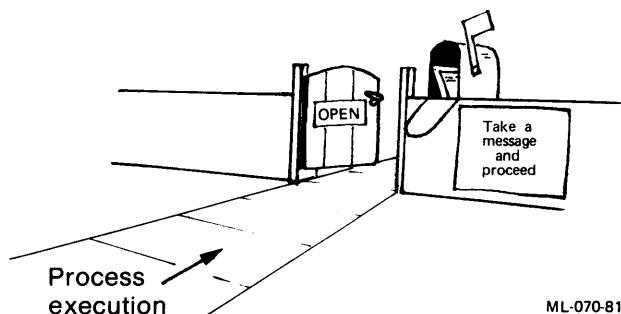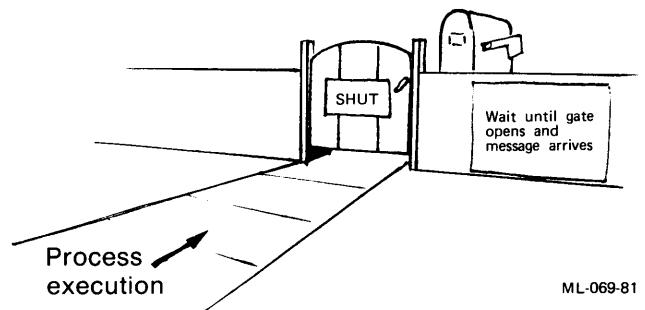
Answer: MicroPower/Pascal provides several mechanisms, called semaphores, for process synchronization and communication. A semaphore is a global data structure that is manipulated by two or more processes. A semaphore is used to halt execution of one process until another process sends a signal to proceed. There are many variations on this basic wait/signal concept, as described in Chapter 5.

You set up semaphores in the source program so that their states at any given time correctly guide the responses of the entire application to external real-time events. For instance, if more than one process must access a data area or other system resource, a point of communication among the processes (semaphore) should be created to guard that resource by ensuring sequential access. In this case, the semaphore acts like a garden gate; the semaphore (gate) protects the resource (garden). When a process accesses the resource, it will close the gate, resetting the semaphore to prohibit access by another process. Any process seeking access will have to wait until the resource becomes available. The resource is freed by signaling the proper semaphore, opening the
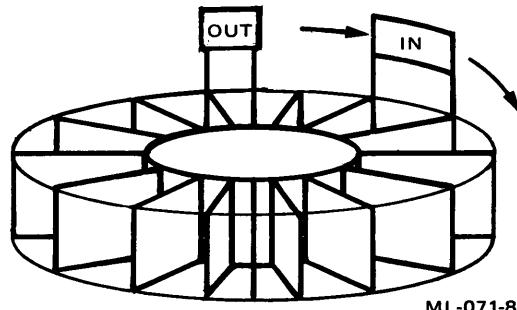
gate to other processes. Without this protection, several concurrent processes might have access to a resource at the same time, interrupting one another to modify the resource unpredictably. Simultaneous access could destroy the integrity of the shared resource.

ML-068-81

There are three types of semaphores: binary, counting, and queue. The binary semaphore represents the open/closed garden gate. The counting semaphore acts like a gate that stays open until a number of processes have passed, then prevents any more from entering. The queue semaphore can pass information to a waiting process, as if the garden gate had a mailbox with a message from the process responsible for opening the gate.

ML-069-81

ML-070-81

Another communication structure that can be shared among processes is the ring buffer. A ring buffer can be thought of as a circle of compartments filled clockwise with data by one process. The series of filled compartments can be emptied by another process in the same sequence: first in, first out. (See Chapter 5.)

ML-071-81

Question: How does a concurrent program deal with real-time events?

Answer: 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-handler process. This process becomes ready to compete for control of the CPU in response to the interrupt.

The order in which processes act in response to external events can be modified by assigning a priority to each process. Processes that respond to important events receive higher priorities.

1.4 Key Terms

Some important terms introduced in Chapter 1 are defined below, in the order in which they appear in the text.

Host	The RT-11 V4 operating system on which MicroPower/Pascal applications are developed.
Target	The LSI-11 or SBC-11 system on which MicroPower/Pascal applications run.
Real time	Related to a physical activity as it happens.
Process	One task-performing piece of an application.
Concurrency	Parallel design of processes that execute in a single CPU.
Synchronization	Coordination of processes.
Semaphore	A MicroPower/Pascal data structure; the basic mechanism for synchronizing processes.
Interrupt	A signal that alters the sequence of instruction execution in the processor.

Chapter 2
Host and Target

We call the two systems involved in MicroPower/Pascal application develop-
ment the host system and the target system. You use the facilities of the host
system to develop your application program, which is designed to run in the
target system. A serial data link between target and host enables you to install
the target system processor in its workplace (lab or workshop, for example),
then load, test, and debug the application program there. (Three other means
of transferring the application to target system memory are: PROM chip,
RX02 diskette, and DECtape II.) See Figure 2-1.

ML-072-81
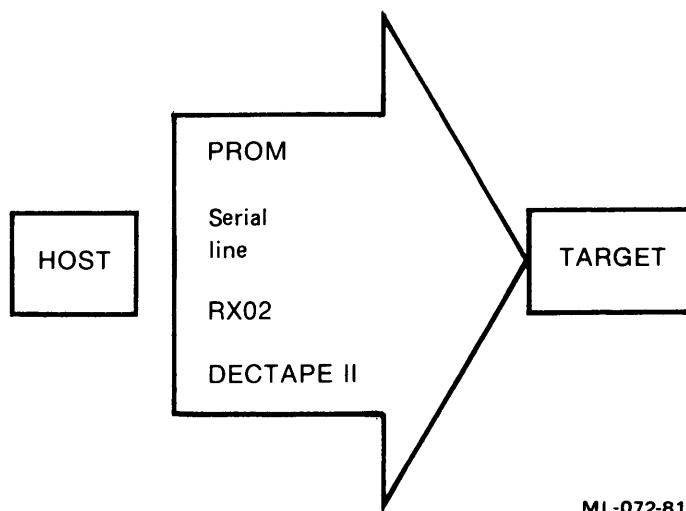
Figure 2-1: Transferring Application to Target System Memory

2.1 The Host System (RT-11)

The host system hardware generally consists of an LSI-11 or PDP-11 proces-
sor with memory-management hardware, a console terminal (which allows the
user to communicate with the system), mass storage devices, peripheral de-
vices such as printers, as well as the RT-11 extended-memory operating sys-
tem to control this hardware. The RT-11 XM operating system includes an

extended-memory (XM) monitor, several device-handling programs (one for each supported hardware device), and utility programs for data entry and manipulation.

MicroPower/Pascal software is added to the RT–11 operating system. This software includes a compiler that recognizes an extended version of Pascal and utility programs that construct the application.

To write MicroPower/Pascal applications, you use either the Pascal language or the MACRO–11 language. You can mix modules containing Pascal and MACRO–11 instructions as sources in your MicroPower/Pascal application program.

For MicroPower/Pascal application development, the host system must include the following:

- Clock option

- Memory-management unit

- 128KB of main memory

- Serial line to system console device

- Serial line to target system (for debugging)

During application development, you use RT–11 utility programs to create files of source code and to manipulate other files created by the MicroPower/Pascal utilities while building an application. The RT–11 utility programs needed to create a MicroPower/Pascal application program are:

- DIRectory — lists the files on the RT–11 system and lets you view their status.

- KED (Keypad EDitor) or EDIT program — provides a facility for entering information in a new file or changing information in an existing file.

- LIBR — creates and modifies files to contain macros (definitions of additional MACRO–11 source statements) or object modules (compiled subroutines) that will be used often in the application program.

- PIP (Peripheral Interchange Program) — provides commands so that you can copy files from one area of the system to another or delete files.
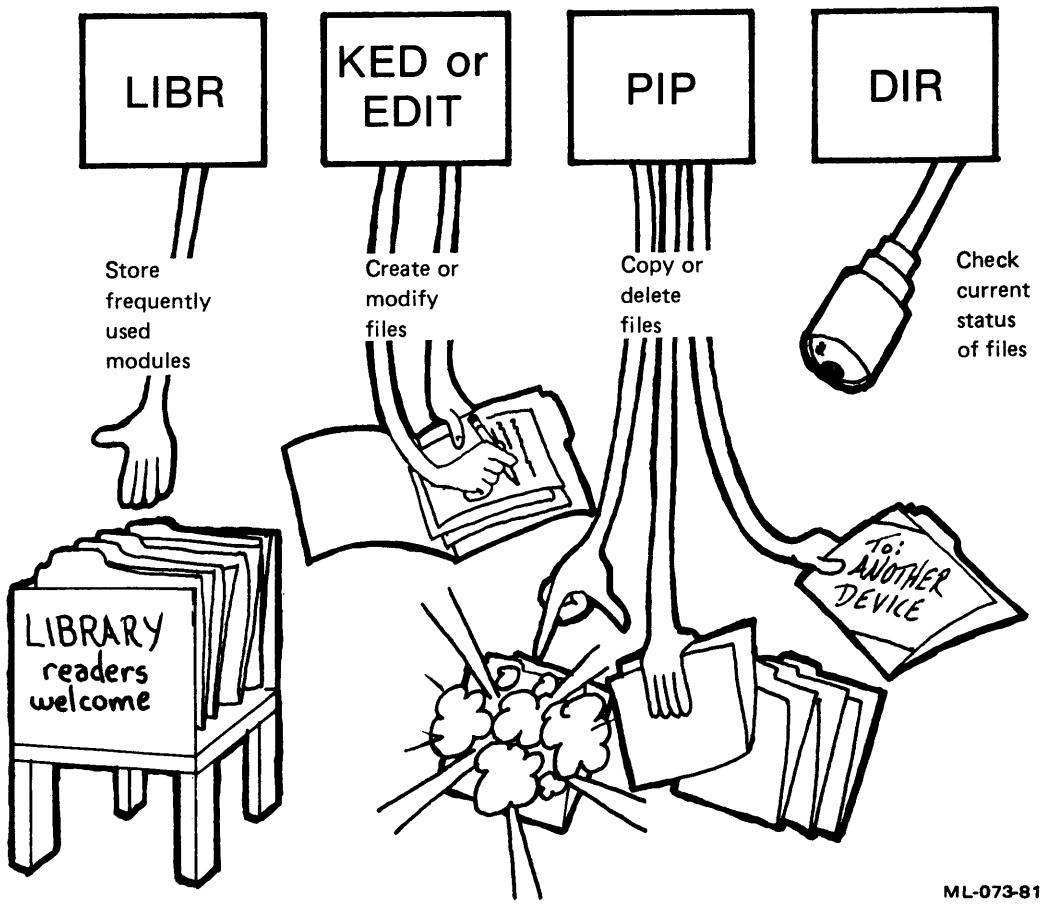
See Figure 2–2.

The RT–11 system documentation fully describes RT–11 utilities. Refer to the *RT-11 Documentation Directory* for information on these manuals. Note, however, that Chapter 3 of the *MicroPower/Pascal System User's Guide* introduces the RT–11 editing utility (helpful for new users).

2.2 Target Systems

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

- Computer-assisted manufacture

Figure 2-2: RT-11 Utilities

- Materials handling

- Monitoring and testing

- Process control

- Robotics

A MicroPower/Pascal target system consists of:

- Interface hardware for your own target devices

- An LSI-11/2, LSI-11/23, or SBC-11/21 microcomputer

- Main memory consisting of RAM and optional ROM large enough for the application; more memory is required during debugging

In addition, a target system may include the following:

- Clock

- Mass storage for reading and writing files

- Parallel line units

- Serial line units for terminals and other data transfer requirements

If the application is not transferred from host to target via PROM chip, one of the following is required to load the application:

- RX02 floppy disk drive

- Serial communication line

- TU58 DECtape II drive

The LSI-11/2, LSI-11/23, or SBC-11/21 microcomputer plus memory, I/O devices, and interconnection hardware form a powerful computer system. You specify the configuration of this system as one step in creating a working application. Following are some of the features of the target system to be configured:

- LSI-11 microprocessor — The target system CPU is one of three microprocessors: LSI-11/2, LSI-11/23, or SBC-11/21. Depending on the type of microcomputer, an extended instruction set (KEV11 for the LSI-11/2) or floating-point hardware option may be configured into the target. (More information on these processors can be found in the *Microcomputer Processor Handbook*.)

- Memory — The target system can contain up to 248K bytes of physical memory in 18-bit mode or 252K bytes when optionally expanded with any of the MSV11-Ex or MSV11-Dx memory cards.

Microcomputer	Maximum Memory		
	16-bit	18-bit	22-bit
LSI-11/2	60KB	—	—
LSI-11/23	60KB	248KB	4MB
SBC-11/21	60KB	—	—
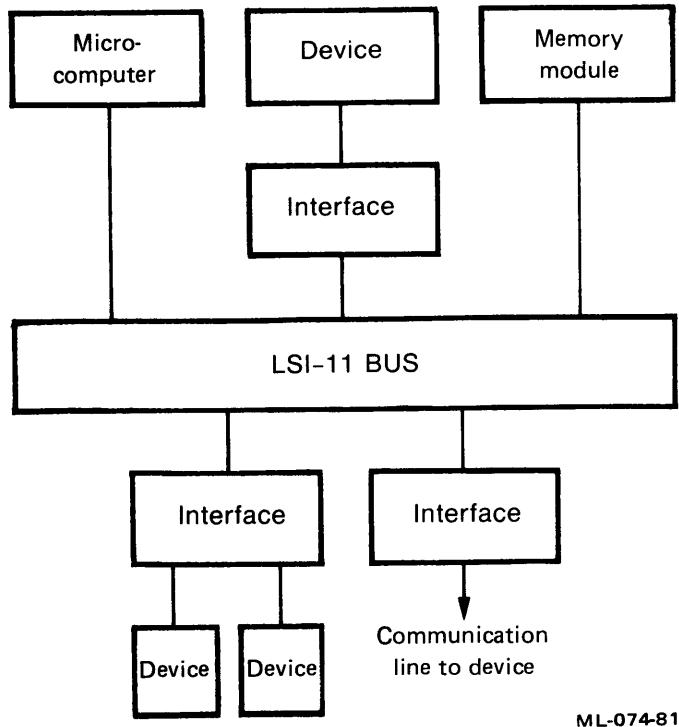
System memory sizes of more than 60K bytes require special memory mapping hardware available on the LSI-11/23. Furthermore, the MicroPower/Pascal target system can contain both random-access memory (read/write, or RAM), and read-only memory (ROM).

2.2.1 I/O Devices

The following Digital Equipment Corporation I/O devices and interfaces are supported by device-handler software supplied with MicroPower/Pascal:

- DLV11/MXV11 serial device interface

- DRV11, DRV11-J parallel device interfaces

- RX02 floppy disk drive

- TU58 DECtape II

All devices and the processor attach to the LSI-11 bus. Digital Equipment Corporation offers different interfaces for attaching devices to the LSI-11 bus, including interfaces with added memory and multiport interfaces (Figure 2-3).

Figure 2–3: Interfaces to LSI–11 Bus

2.2.2 The Configuration File

One of the first steps in creating a MicroPower/Pascal application is editing and assembling the configuration file. The configuration file, which describes the target system, comprises a series of macro calls. You modify (edit) the parameters of these macro calls to describe the characteristics of your target system hardware. The RT–11 utilities use the resulting object module to tailor both the memory image in general and the kernel in particular to your specifications.

For example, one of the macros in the configuration file is the processor macro. To specify that the target contains memory-management hardware, but does not have floating-point capability or the memory-parity option, you would modify one particular line of the configuration file to read:

PROCESSOR mmu = YES, fpp = NO

The configuration file is described in detail in the *MicroPower/Pascal System User's Guide.*

Figure 2–4 is a configuration worksheet listing the items in a MicroPower/Pascal configuration. Chapter 6 discusses the configuration worksheet in detail and shows how to use it for editing configuration files.

Configuration name _____

SYSTEM: Debugger support _____ | Y | N | [No]

           Optimize _____ | Y | N | [No]

INCLUDE ONLY IF SYSTEM OPTIMIZE = Y

RESOURCES: Kernel stack size _____ bytes RAM
                              [predefined minimum]
            Number of system packets _____
                                            [20.]
            Kernel pool for system data structures _____ bytes RAM
                                          [1000.]
            Free RAM table size _____ bytes
                                      [20.]

TRAPS: ___ ___ ___ ___ ___ ___ ___ ___
            [All]

            All = | LSI | SBC-11/21 | T10 EMT TRP |
                   | | | TR4 BRK |

            Other traps: FIS FPP MMU MPT

PROCESSOR:

Memory-management unit	Y	N		No
Floating-point unit	None	FP11	FIS	None
Type	T11	L112	L1123	L1123
Vector	_____			1000(octal) (400octal SBC-11/21)

KXT11
(for SBC-11/21)

nxm	break	syshalt	level7
__HALT	TRAP	HANG__	ODT __
__TRAP	HALT	ODT __	TRAP __
[HALT]	[TRAP]	ODT ROM __	ODT ROM __
		USER__	IGNORE__

level7 only if
    nxm = HALT
break = TRAP
syshalt = HANG

                    [ODTROM]   [TRAP]

MEMORY:	Base address	Size (64-byte segment)	Type	Parity	Parity controller CSR	Volatile
Segment 1			RO RW	Y N		Y N
Segment 2			RO RW	Y N		Y N
Segment 3			RO RW	Y N		Y N
Segment 4			RO RW	Y N		Y N
			[RW]	[No]	[0]	[Yes]

DEVICES: ___ ___ ___ ___ ___ ___ ___ ___

Figure 2-4: A MicroPower/Pascal Configuration Worksheet
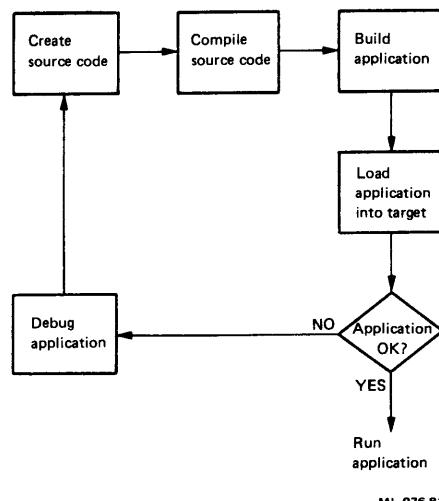
Chapter 3
Real-Time Application Development

There are five steps in developing a MicroPower/Pascal application, as follows:

1. Design the application and write Pascal source programs and procedures.

2. Compile the source program statements into machine instructions. (This work is done on a host operating system, the RT–11 system.)

3. Build the full application program by linking the output from step 2 with kernel functions and services (using MicroPower/Pascal utility programs in the host).

4. Load the completed application program into the target microprocessor.

5. Test and debug the application in the microprocessor under control of a debugger program residing in the host system.

3.1 Development Cycle

Figure 3–1 shows the development cycle of a MicroPower/Pascal application.

ML-076-81

Figure 3–1: MicroPower/Pascal Application Development

The following sections explain the development steps.
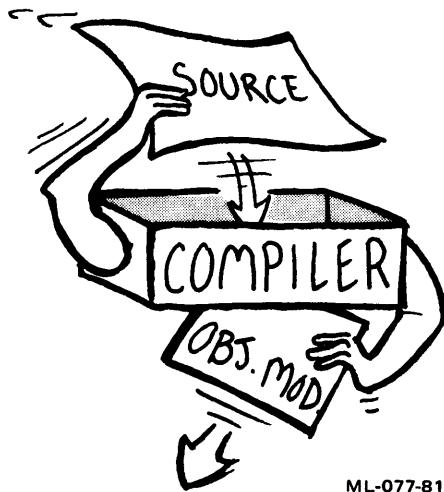
3.1.1 Step 1: Design and Code Source Programs

First, you define precisely the functions of your application system, software requirements, and function priorities. Written specifications and/or flowcharts help you keep the functions in perspective. See Chapter 5 for an example of designing a concurrent application.

After setting the design, you write source code for the application by using a text editor and the file-management facilities of the host. Both the Pascal and MACRO–11 languages can be used to write parts of a MicroPower/Pascal application. Information for MACRO–11 programmers is contained in the *MicroPower/Pascal Runtime Services Manual*. Pascal programming is covered in the *MicroPower/Pascal Language Guide*.

3.1.2 Step 2: Compile Source Code

Each Pascal program statement expands into many machine instructions. The MicroPower/Pascal compiler translates your code into machine language and checks for program syntax errors and improperly declared or mismatched program variables. The result of compilation (or assembly, for MACRO–11 code) is a machine language code module, called an object module. The MicroPower/Pascal utilities link object modules for inclusion in the application. MicroPower/Pascal allows you to construct application programs individually, compiling (or assembling) portions of source code and building a subset of your planned application program for debugging and testing.

ML-077-81

3.1.3 Step 3: Build the Application

The MicroPower/Pascal utilities need more than object modules to construct a proper application. The utilities require information describing the hardware of the target system and its memory characteristics. You provide this information in a configuration file.

A prototype configuration file already exists as part of your software distribution package. You edit a copy of this file and then use it as preliminary input to the MicroPower/Pascal utilities with the object modules produced in step 2. (This configuration file, like files of source code written in the MACRO-11 language, must be assembled rather than compiled, to be proper input for the utilities. The output of the assembler program is a file of object code, ready for MicroPower/Pascal utilities.)
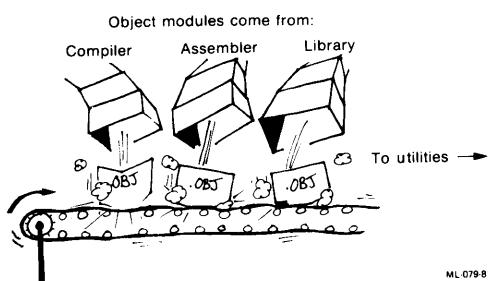
ML-078-81

As input, MicroPower/Pascal utilities accept object modules of compiled or assembled source code contained in files. (You identify the appropriate files by name when you invoke the utility programs.)
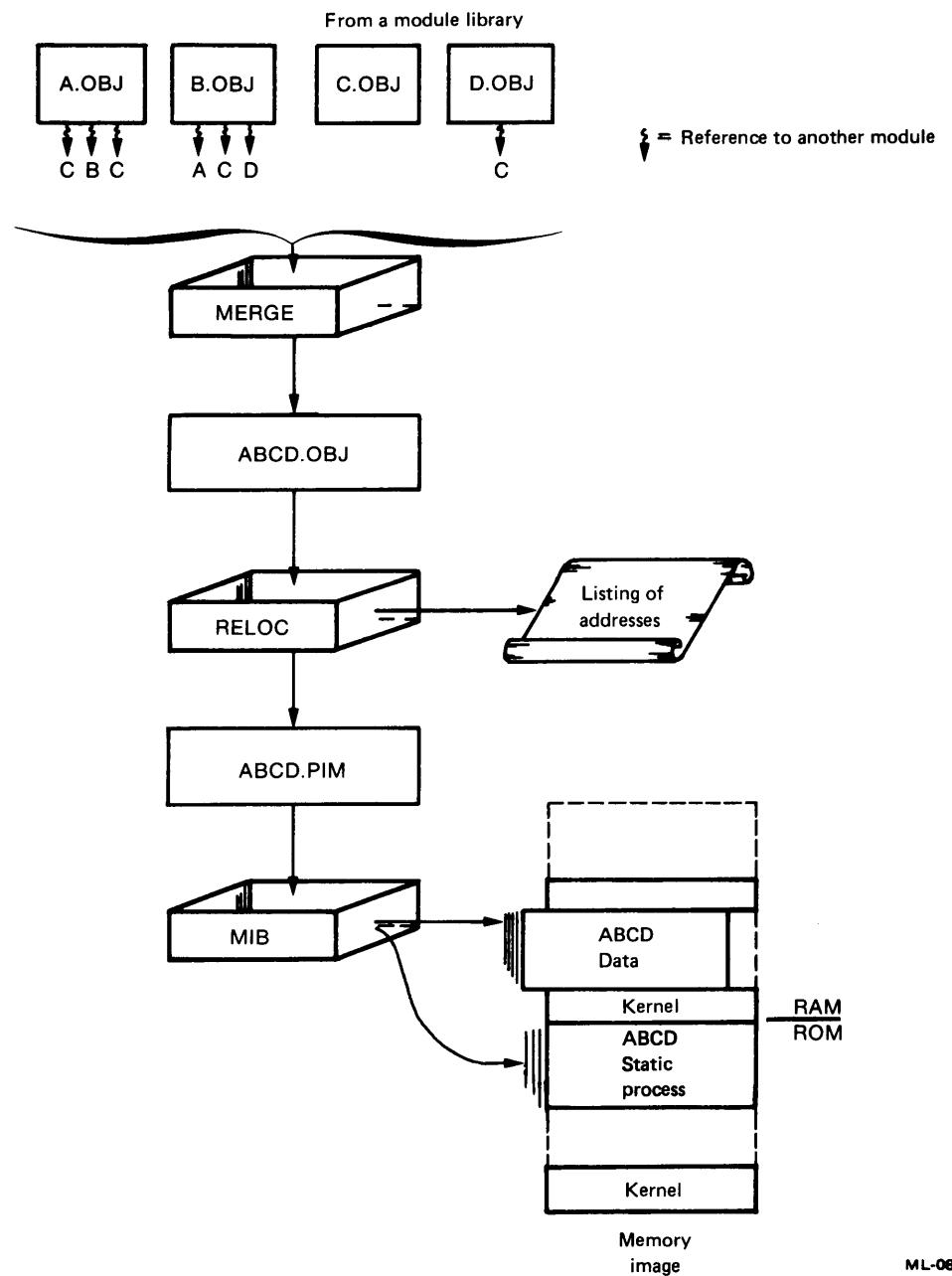
MicroPower/Pascal utilities perform several functions, as follows:

1. Resolve references from one input module to another or to program modules contained in module libraries

2. Combine the object modules into an executable unit

3. Relocate the addresses of separate sections of code in the object modules and allocate sufficient target system memory to each part of the application

ML-079-81

4. Set up debugging information (names, addresses, and relationships of program variables) in the host and target for later use

5. Produce listings and maps

Together, the MicroPower/Pascal utilities MERGE, RELOC, and MIB perform the same function as the linker in a traditional development system. As separate utilities, MERGE, RELOC, and MIB allow you to develop your application module by module. See Figure 3–2.
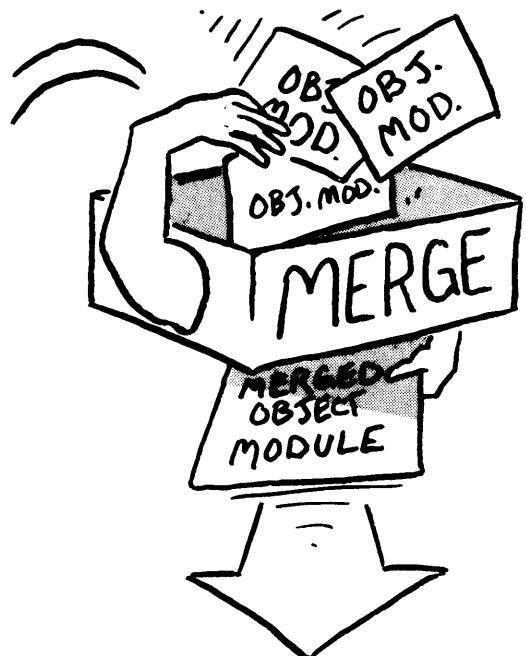
Figure 3–2: MicroPower/Pascal Utilities

3.1.3.1 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 automatically by the MicroPower/Pascal compiler or specified by the MACRO-11 programmer. These p-sects contain all code and data. MERGE combines p-sects of identical names from all input object modules. The output of MERGE is 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 system libraries of object modules. References to these modules made in the configuration file are resolved, and the selected kernel becomes ready for relocating.

ML-081-81

3.1.3.2 RELOC — The RELOC utility assigns addresses to program sections within a merged object module. The result is a process image module (PIM). You can use RELOC to assign base addresses to individual program sections.

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

3.1.3.3 MIB — The MIB (memory image builder) utility creates the executable application by placing all its components into one structure, called the memory image file. You use MIB to insert each process image module into the memory image.

Input to MIB is a process image module (PIM). One process image module is the kernel of basic services, or "kernel." Other process image modules result from Pascal main programs or MACRO–11 programs.

You first invoke MIB to create a memory image file and install in it the kernel process image (PIM) file. You signify this invocation of the MIB utility with a /K in the command. Once the new memory image is created with the kernel in place, you invoke MIB to include each successive process image module in the application.

MIB allows you to 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 leading the application into the target system.

3.1.4 Step 4: Load the Application into the Target

There are three ways to transfer the application to the target system processor, as follows:

1. Down-line load if a communication link exists between the two systems

2. Media and hardware-boot on the target system

3. Program into a PROM chip for installation into the target

ML-082-81
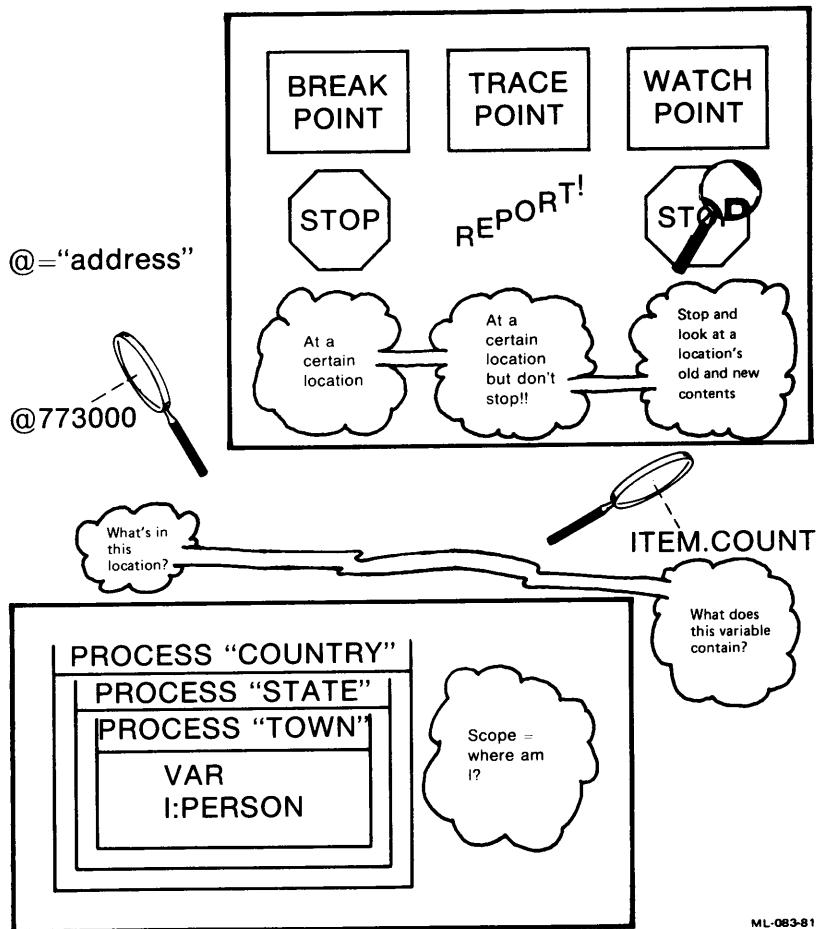
3.1.5 Step 5: Test and Debug the Application

Debugging is done from the host over a serial communication line to the application running in the target system. You can create the application one piece at a time, debugging each portion separately in the target system, then recreating the entire application as each piece is tested.

The MicroPower/Pascal debugger is symbolic; it recognizes the names of entities in the application. Using the debugger program in the RT-11 system, you can down-line load the application into the target, then control its execution using debugger commands. These commands will:

- Deposit values into memory locations

- Determine the scope of a process or a variable

- Examine the contents of memory locations

- Reveal the location and value of named data structures

- Set breakpoints, or numbered "stop signs," throughout the program

- Step through program execution one statement at a time

The Pascal symbolic debugger uses a symbol table created by MIB. The symbol table contains the names defined in the original MicroPower/Pascal code for the application. This table represents the relationships among all symbols as well as their addresses. The MicroPower/Pascal debugger features are shown in Figure 3-3.

Figure 3-3: MicroPower/Pascal Debugger Features

Chapter 4
Processes

Every MicroPower/Pascal application comprises many independent, but coordinated, processes. These processes are supported by a software kernel supplying basic system services such as:

- Interprocess communication, including functions required to create, operate on, and destroy semaphores, ring buffers, and message packets

- Interrupt dispatching

- Process creation and deletion

- Process scheduling, to enforce the order of execution of processes

- Responding to certain conditions, called exceptions, detected during normal operation of the system

This chapter focuses on processes and mentions the kernel in relation to processes. Services provided by the kernel are described in the *MicroPower/Pascal Runtime Services Manual.*

All process synchronization in MicroPower/Pascal programming is performed by the kernel in response to requests by processes. Thus, the kernel responds to all demands for services according to strict guidelines, but makes no decisions. For instance, a process may request the kernel to assign it a packet of kernel-controlled memory or to change the value of a semaphore. The kernel will comply if conditions allow, but responsibility for these resources is with the process. In short, the kernel is an indispensable body of software that provides rudimentary services on demand. See Figure 4-1.

ML-084-81
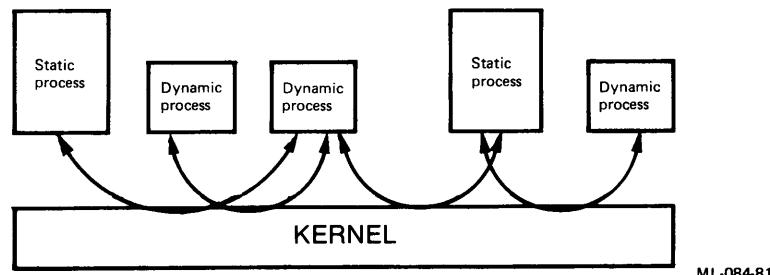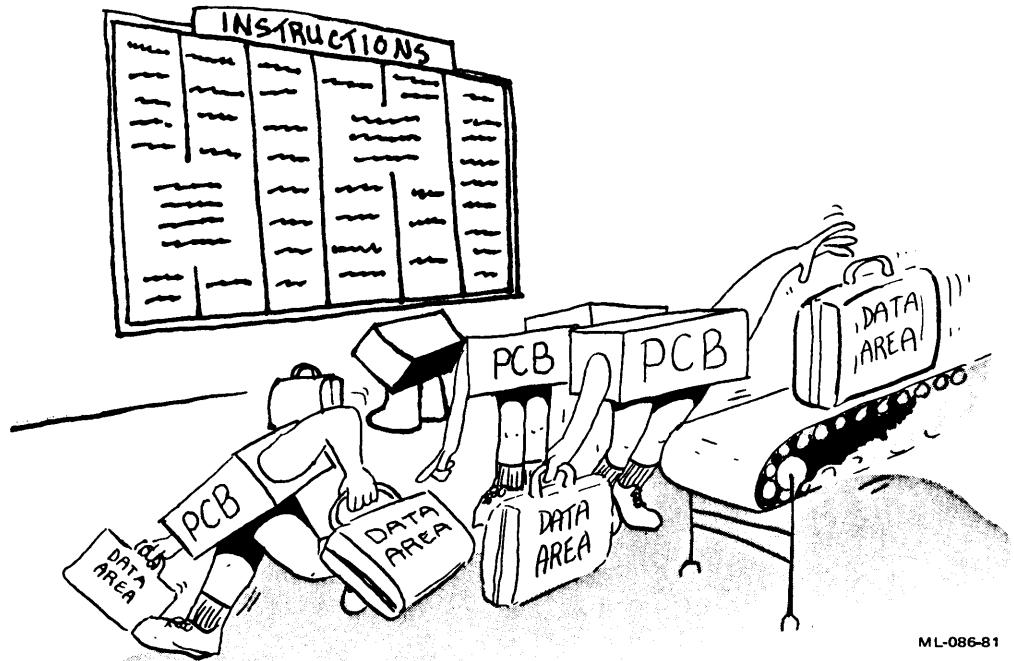
Figure 4-1: The Kernel and Interprocess Communication

4.1 Makeup of a Process

A process is a program unit that may operate in parallel with other program units. Each process within the MicroPower/Pascal application consists of:

1. A process control block (PCB)

2. Associated data structures

3. A named sequence of instructions

ML-086-81

The following sections describe these three parts of a process.

4.1.1 Process Control Blocks

A process control block (PCB) is a small block of memory that contains information about one process called the context. Understanding the use of PCBs will help you create an application.

Each PCB consists of from 24 to 79 words of memory, depending on the type of process and memory configuration of the target system. Each PCB contains the following:

- Process name
- Process priority
- Process state   (Described later in this chapter.)
- Process type

- Saved process context (Described in the *MicroPower/Pascal Runtime Services Manual.*)

This information describes a process completely. By manipulating the process control block, you effectively control the execution of the process.

4.1.2 Process Data Areas and Structures

Process data structures (arrays, constants, and variables) for each process are allocated in data areas associated with the process. To provide these data areas, each process must have an amount of memory associated with it for the creation of structures. Two Pascal keywords, STACK__SIZE and DATA__SPACE, are associated with the allocation of data areas.

Each process, including the main program, may have a STACK__SIZE attribute included in its Pascal process or program header. Stack size defines the amount of memory allocated to the process stack.

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 associated with the main program. Therefore, 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 via the Pascal NEW procedure during execution for temporary data structures.

Semaphores, PCBs, and other system data structures are created out of the kernel's data area. You specify the size of this area in the target system configuration file.

4.1.3 Process Names and Process Descriptor Blocks

In the MicroPower/Pascal source program, the code for a process is called the process body and defines the instructions to be executed by the process. Each process body in the MicroPower/Pascal source code is headed by a name in the PROCESS declaration statement:

```
PROCESS Harcourt;

BEGIN

  WRITELN ('This is the process Harcourt');
END;
```

Multiple references in the source code to the name Harcourt will cause several processes to be created, each using the process body named

```
Harcourt;
```

If a specific invocation of Harcourt is to be referenced, however, the process body name will not work, since Harcourt can be invoked from the source code any number of times (creating a new process each time). There must be an individual identifier for every process.

Within the application, an identification number is used for this purpose. The number for each new process can be copied into a separate variable in the creating process, called a process descriptor block (PDB).

A process descriptor block is declared in the source code as follows:

```
VAR

Mudd: PROCESS_DESC;
```

A process descriptor block named Mudd can be filled in with the identification number of process Harcourt when the process is created, as follows:

```
Harcourt (DESC := Mudd);
```

The application's kernel also associates a unique runtime name with each process identification number. Runtime names can be used by other processes to unambiguously select any invocation of the process body named Harcourt.

Runtime names are specified in the source code when a process is created. (Remember, processes are created by reference to a process body name.) For example:

```
Harcourt (NAME := 'FENTON');
```

The process body is named Harcourt, and the runtime name for a particular invocation of Harcourt is named FENTON.

Finally, both a runtime name and a process descriptor block can be specified during process creation:

```
Harcourt (NAME := 'FENTON', DESC := Mudd);
```

When both a name and a process descriptor block are given in the statement, the name is included in the PDB, along with the identification number. The process created in the statement above can be referenced by its name (FEN-TON), which is unique and valid throughout the application, or, in the creating process, by its process descriptor block (Mudd).

4.2 Process Scheduling and Synchronizing

The key to efficiency in MicroPower/Pascal applications is coordinated execution of processes. Chapter 5 explains how semaphores in the Pascal source code determine the flow of control among different processes. To fully understand the concurrent organization of an application, however, it is helpful to know how process scheduling and synchronizing are implemented during execution.

4.2.1 Kernel's Role

Process scheduling is a service provided by software routines in the kernel. The kernel manipulates process control blocks (and therefore processes) in response to requests by processes. Scheduling is invoked by certain process state changes.

4.2.2 Process States

Processes are scheduled to control the CPU, and their execution is synchronized by means of seven process states. Every process is considered to be in one of these states at any time. The state of a process indicates its execution status:

- Executing (only one process can be in this state at a time)

- Ready for execution, but not currently running

- Suspended by another process, but ready

- Blocked from execution because of the value of a semaphore

- Blocked and suspended

- Waiting for resolution of an exception condition

- Waiting for resolution of an exception and suspended

These seven states reflect the execution statuses for any process, as determined by the values of semaphores. For instance, all processes disabled from execution by semaphores are in either the wait-active or wait-suspended state. Processes waiting for the assistance of an exception-handler process are either exception wait active or exception wait suspended.

The first three process states above are represented by specific queues, as follows:

- Run

- Ready active

- Ready suspended

The run queue consists of at most one element, the process in control of the CPU.
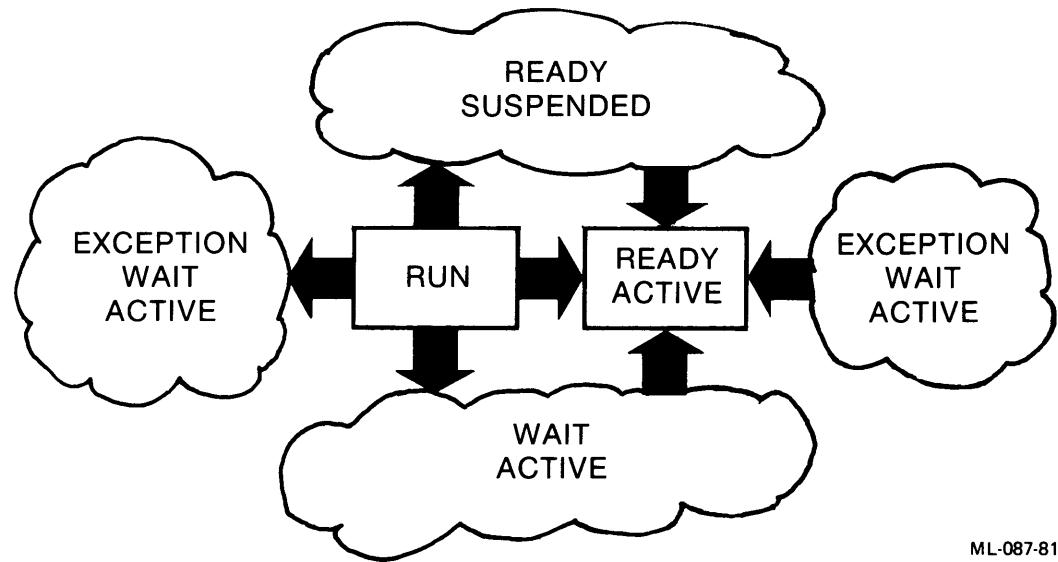
Two ready queues reflect the states of processes that are not blocked from execution because of semaphores, but must still wait for control of the CPU. The kernel schedules these processes according to priorities individually assigned to them in the source code. Processes of the same priority are scheduled on a first-come, first-served basis.

MicroPower/Pascal allows the process in control of the CPU to affect the state of another process by altering a semaphore; or, an external interrupt may cause a state change. A state change is called an event. Whenever a process is eligible to take control of the CPU (joins the ready-active queue) or whenever the running process becomes ineligible (leaves the run queue), the kernel scheduler takes control. Any movement, either into the ready-active state or out of the run state, invokes the scheduler. The scheduler compares the priority of the running process (if there is one) with that of the highest-priority ready-active process. The winner moves into the run state and gains control of the CPU.

In summary, process states will be updated to reflect any changes whenever a process:
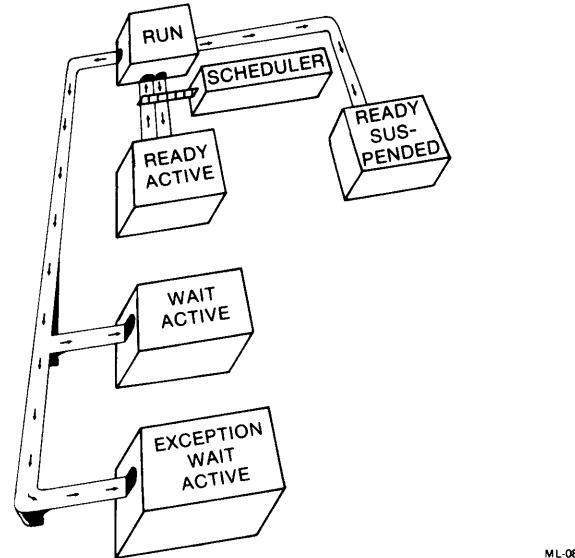
- Cannot continue execution because of a semaphore

- Enables another process by means of a semaphore

- Enables a suspended process

Figure 4-2 shows state changes that may cause control of the CPU to shift from one process to another.

ML-087-81
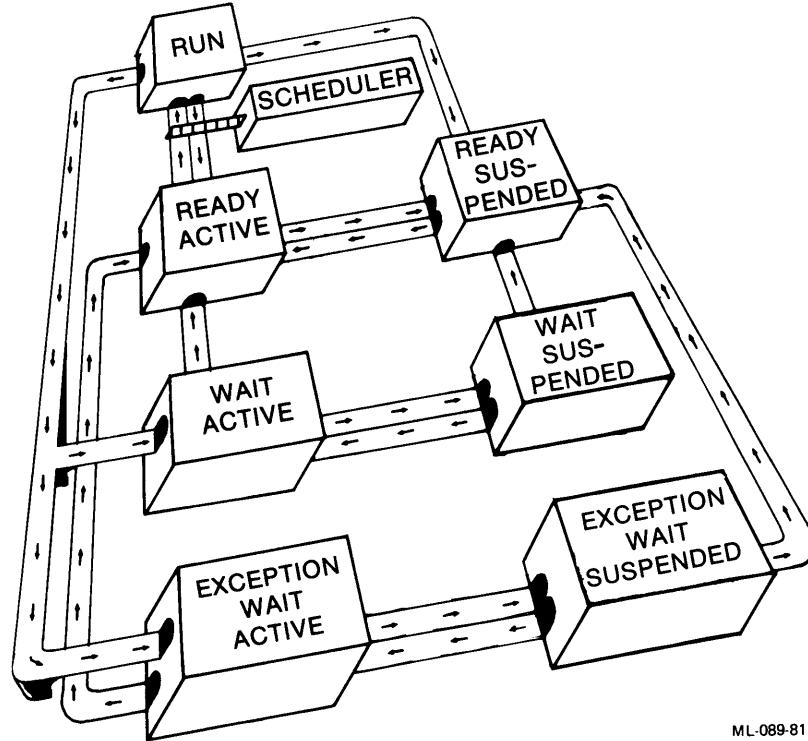
Figure 4-2: State Changes that May Affect Control of the CPU

Figure 4-3 shows changes into and out of the run state.

ML-088-81

Figure 4-3: State Changes Involving the Run State

Figure 4-4 shows all possible state changes.

ML-089-81

Figure 4-4: Summary of All State Changes

4.3 Process Families

Inside the application, processes are gathered into groups, or families. Each family consists of one static process and one or more dynamic processes.

4.3.1 Static Versus Dynamic Processes

Static processes derive from MicroPower/Pascal main programs, the basic units operated on by MicroPower/Pascal utilities. A static process — that is, a static process control block — exists from the moment the application starts in the target system. Dynamic processes are created by other processes during execution, and their data areas are allocated as they are created.

Static processes also differ from dynamic processes in that only one static process results from a MicroPower/Pascal main program. By contrast, several dynamic processes can result from one MicroPower/Pascal process body.

Each process family in the application contains one static process and all dynamic process descendants created during the lifespan of the static process.

The scopes of nested process bodies, procedures, and functions in source code are defined by the standard Pascal scope rules. No process, static or dynamic, can be deleted from the application until all processes it spawned have first been deleted.

4.3.2 Mapped Memory Processes

In a target system with memory-mapping hardware, memory can be split into independent sections (address spaces) in order to use a larger total amount of physical memory. (The largest possible address space is 64KB.) Details of memory mapping are presented in the *MicroPower/Pascal Runtime Services Manual* and the *Microcomputer Processor Handbook*.

Each address space in the mapped target system contains one process family — one static process and its dynamic process descendants. In addition, there is a kernel address space containing the kernel and its related data areas. Figure 4-5 shows process families in address spaces.
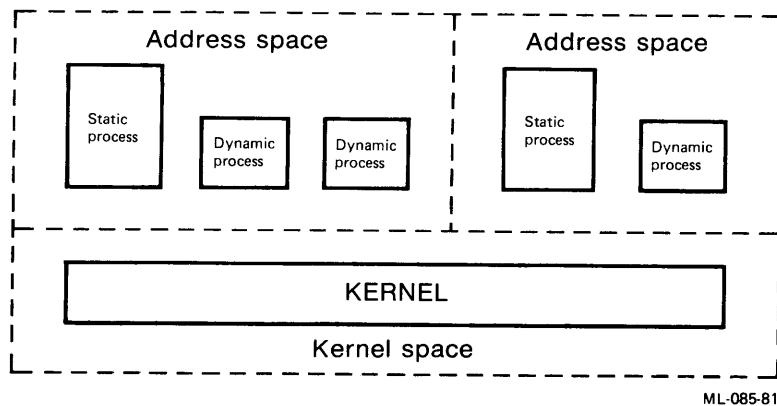
Figure 4-5: Address Spaces

4.3.3 Process Types

In mapped target systems, processes can differ as to which areas of memory they have access to. (In unmapped systems, the entire application exists in a mutually accessible address space.)

There are four process types, as follows:

• Device access

• Privileged

• Driver

• General

Device-access processes have access to the portion of target system memory containing I/O device addresses. A device-access process can manipulate the control and status registers (CSR) and data buffers of target devices.

Memory for target system data structures, such as semaphores and process control blocks, comes from an area associated with the kernel called system-common memory. Privileged processes have access to this kernel data area as well as the I/O addresses section of memory.

Driver processes, like privileged processes, have access to both kernel data and I/O addresses. Driver processes generally contain independent sections of

code (called interrupt service routines), designed to execute quickly in response to interrupts from target devices.

General processes do not access I/O addresses or kernel data directly.

Each process family belongs to one of the four types above.

4.3.4 Initializing and Terminating Components of the Application

When power is on in the target system and when the application is loaded into memory, a sequence of steps readies the static processes for execution. As part of this sequence, a special initializing procedure runs for each static process; named system data structures, such as semaphores and ring buffers, can be created in this initialization procedure. You provide this procedure in the source code by using the INITIALIZE attribute:

```
[INITIALIZE] PROCEDURE Dofirst;
```

This procedure can be included in the Pascal main program, just like any other procedure.

After the initializing procedure is executed, control for every static process shifts to that static process's transfer address (usually the beginning of its Pascal main program). However, the static process does not begin running immediately. Instead, execution begins in the next static process's initializing procedure.

Each static process executes its initializing procedure and becomes ready to run before beginning to work. This ensures that all needed semaphores and other structures can be created and initialized before concurrent execution starts.

The static processes of the application vie for control of the CPU. If initialization has been done correctly, it does not matter which static process takes control first, since the necessary synchronization mechanisms are set.

<div align="center">

NOTE

</div>

> Normally, you create named synchronization mechanisms for processes during initialization.

Each process can have a termination point. Execution shifts to this termination point when the process is stopped. (A process can be stopped by the STOP request — issued by it or by another process — or as the result of an exception condition being detected. See Section 4.5.)

The termination point of a process is a nested procedure with the [TERMINATE] attribute. After a process executes its termination procedure, it is destroyed.

A destroyed process no longer exists in any state. Dynamic processes created by this defunct process are independent and can continue, and data structures declared in a defunct process are still accessible to them. To fully delete a process — including its stack of data structures — its dynamic process descendants must also be terminated.

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 at unpredictable moments and therefore are called asynchronous. When an interrupt occurs, control of the CPU automatically transfers to an appropriate interrupt service routine (ISR). Interrupt service routines are a part of any process written to handle devices, usually of type DRIVER. ISRs immediately respond to interrupts. After the ISR has run, the kernel intervenes and assigns control of the CPU to the most appropriate eligible process. This process may or may not be the same one 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.)

A process can most easily be alerted to the occurrence of a pertinent interrupt by creating a semaphore to be signaled by the kernel when such an interrupt is detected. This is accomplished by using the CONNECT_SEMAPHORE statement. When CONNECT_SEMAPHORE is included in a process, the kernel automatically signals a specified semaphore in response to interrupts from a certain device or devices.

4.5 Exception-Handling Processes and Procedures

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

1. Accessing an I/O device where none exists

2. Executing an illegal instruction

An exception condition may or may not represent a fatal execution error in the application.

Processes can be set up to take control whenever the CPU detects one of these exception conditions. Each exception-handling process can be designed to respond to exceptions of certain types. In addition, one process can be specified to handle the exceptions of a group of processes.

Alternatively, exceptions for processes in one address space can be handled by procedures nested directly inside the Pascal main program (static process) for that address space. Such a procedure is associated with its client process by an ESTABLISH statement in that process. This statement has the form:

```
ESTABLISH(EXC_PROCEDURE := ohoh , EXC_TYPE := [resource]);
```

In this case, the Pascal procedure named ohoh is designated to handle exceptions of type [resource] that arise during execution.

An exception can be artificially provoked by using the REPORT statement. This statement has the form:

```
REPORT(EXC_TYPE := [resource] ,EXC_CODE := ES$NMP );
```

Exception code ES$NMP of type [resource] is generated wherever this statement is included in the process.

4.6 System Processes

Finally, the application may contain several processes supplied directly by the MicroPower/Pascal utilities in the host during creation of the application. Unlike user-created application processes, these system processes:

- Are provided as part of the MicroPower/Pascal package

- Furnish commonly required services

- Are (in general) privileged processes

Services provided by system processes include interfacing with the clock to provide timings, as well as driving (handling) certain devices and interfaces, including TU58, RX02, DRV11, and DLV11.

A system process called the directory structure process (DSP) is part of the MicroPower/Pascal file system. The DSP allows any process to create, maintain, and delete file directory entries on target mass storage devices.

Chapter 5
MicroPower/Pascal and Concurrent Programming

5.1 Pascal and MACRO–11 Languages

MicroPower/Pascal applications can be written in two programming languages: Pascal and/or MACRO–11. Pascal was developed in the late 1960s as a teaching tool and is used in schools and industry. A versatile, readable programming language, Pascal encourages a straightforward logical structure.

MACRO–11 is an assembly language; each symbolic MACRO–11 instruction corresponds to one LSI–11 machine instruction. In MACRO–11, symbols are used in place of numerical machine code. These MACRO–11 symbols consist of names for instructions and memory locations as well as special characters for indicating addressing modes. MACRO–11 instructions can evoke every function of the LSI–11 processor.

Each Pascal statement is translated by the compiler program into several LSI–11 machine instructions. Translation limits slightly the dexterity of Pascal, but makes programming faster and easier. Indeed, you rarely need the extra range and control of MACRO–11. For instance, small portions of MicroPower/Pascal device handlers must be written in MACRO–11, although most of the driver is written in Pascal.

This manual set does not teach Pascal or MACRO–11 programming. Once you are familiar with the Pascal language, you can use the *MicroPower/Pascal Language Guide* to learn DIGITAL's version of Pascal and the real-time extensions required for concurrent application development. MACRO–11 programming extensions are documented in the *MicroPower/Pascal Runtime Services Manual*. This chapter explains the basic concepts of concurrent programming with MicroPower/Pascal.

5.2 Concurrency Concepts

We will use an analogy — The Simultaneous Chef — to illustrate concurrency. In this analogy, we will present concurrent directions for preparing spaghetti sauce with meat. To make spaghetti sauce with meat, you must add browned hamburger to simmering tomato sauce.

The meat and the sauce can be cooked at the same time. Therefore, we can write this recipe concurrently — as two separate but related tasks — rather than as one complex sequential procedure. A concurrent design is both more efficient and easier to understand.

ML-090-81

To write this recipe concurrently, we prepare separate directions for performing each task: sauce steps and meat steps, as follows:

Sauce	**Meat**
1. Start simmering the sauce.	1. Start browning some meat.
2. After a minute, stir the sauce.	
3. Add salt to the sauce.	
	2. After a couple of minutes, add a pinch of salt to the meat.
4. Stir the sauce every minute or so until it is time to add the meat.	
	3. When the meat looks brownish, add it to the sauce.
5. After adding the meat, stir the meat sauce every minute or so until it is hot and tasty. Then remove it from the heat and serve.	

The two independent recipes, or processes, of five and three steps, respectively, create an efficient program for cooking spaghetti sauce with meat. The two processes communicate at one critical point; both direction sets specify that we wait for the arrival of the browned meat before finishing up the sauce.

By contrast, if we wrote our directions as a sequential procedure, they might look as follows:

1. Start simmering the sauce.

2. Start browning some meat.

3. After a minute, stir the sauce.

4. Add salt to the meat.

5. After a couple of minutes, stir the sauce.

6. After a minute, add salt to the sauce.

7. Continue to stir the sauce every minute or so until the meat looks brownish.

8. Add the browned meat to the sauce.

9. Continue to stir the sauce occasionally until it is hot and tasty; then remove it from the heat and serve.

When the tasks of cooking sauce and meat are combined, the additional steps and ingredients make the directions more difficult to follow.

In addition to being easier to follow, the concurrent design is easier to expand. Assume, for instance, that we want to guard against possible kitchen fires while making our sauce. We can do this by equipping our kitchen with a smoke detector and a fire extinguisher. We will also add a set of instructions telling us what to do when the smoke alarm rings. The directions for fire fighting will have a higher priority than those for cooking. If a fire starts, we will immediately stop cooking and extinguish the flames.

Adding such instructions to our sequential recipe would make it more difficult to understand how the cooking tasks are accomplished. Our concurrent design accommodates such expansion, however.

The Simultaneous Chef highlights three features that must be a part of any system of concurrent processes and real-time interrupts:

1. Important processes should have priority over less important processes.

2. Processes that work on related tasks must not interfere with one another when manipulating shared resources such as devices and data.

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

5.2.1 Processes Manage Shared Resources

Note that both sets of directions above use salt. We can assume that salt is a shared resource in our cooking application. A hurried chef who tries to follow several sets of cooking directions at once may prematurely stop salting one food in favor of another. In other words, the chef may allow one set of directions to steal the salt resource from another set of directions. The chef who does not deliberately salt one food at a time may become confused and may lose track of how much salt has been added to the dish.

In a MicroPower/Pascal application, processes must manage shared resources carefully. Processes can be interrupted by external events and be superseded by other processes because of the interruption. In this case, control of shared resources could pass haphazardly from process to process, with unpredictable results.

To avoid this problem, access to each shared resource in the application must be mutually excluded from competing processes. (For instance, we must finish salting the meat before we salt the sauce.)

Semaphores establish mutual exclusion between processes. When access to more than one resource is mutually excluded, the concurrent design must be sophisticated enough to avoid deadlock, which is an infinite stalemate between two processes that exclude each other from a needed resource.

5.2.2 Semaphores Synchronize Concurrent Processes

Operations on semaphores control the execution of concurrent processes. A semaphore is a data structure in kernel address space that is operated on by MicroPower/Pascal routines such as WAIT and SIGNAL. Any process can create a semaphore by a request to the kernel.

In MicroPower/Pascal, you use built-in functions to create the three types of semaphores (binary, counting, and queue) and ring buffers. The following is a statement for creating a binary semaphore:

```
Result := CREATE_BINARY_SEMAPHORE (NAME := ' RENE ', DESC := artes);
```

This statement returns a true or false value for the Boolean variable Result.

The following is also a legal statement:

```
IF CREATE_BINARY_SEMAPHORE (NAME := ' RENE ', DESC := artes) THEN;
```

Conversely, the order to destroy this semaphore is:

```
DESTROY (DESC := artes);
```

This order is not a function call and need not have the form of an assignment or logic statement.

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. If the semaphore value is nonzero when checked, it is decremented, and the process continues.

As long as the semaphore is 0, any process that executes a WAIT on the semaphore will enter the wait-active state. Each semaphore has a list of waiting processes.

A process that has blocked by waiting on a semaphore is placed in the semaphore's list and enters the wait-active state. When the process reaches the top of the semaphore's list and another process signals the semaphore, the first process changes state to ready active and is eligible to resume execution according to its priority.

Processes can wait for specific events, such as the ringing of alarms, by waiting on a semaphore that is signaled by the key event. In addition, you can use a semaphore to ensure that access to a data area by more than one process is sequential. In this case, two or more processes wait on the same semaphore, ensuring that only one process at a time will execute its section of code. (When a process has finished its section, it signals the semaphore, enabling another waiting process to continue according to its priority.)

A counting semaphore may keep track of the number of resources available to any requesting process. As resources are accessed and released, the counting semaphore is incremented and decremented. A process attempting to access the controlled resources will be forced to wait on the counting semaphore only when the semaphore is 0. As long as at least one resource is available, there will be no waiting.

A queue semaphore is a counting semaphore with an associated queue of data elements; the value of the semaphore indicates the number of queue elements. When a process waits on the semaphore, it will either gain access to the next queue element (semaphore > 0) or block itself until an element is present. When the semaphore is signaled (attains a value greater than 0), a data element is present. Three things happen, as follows:

1. The waiting process gains access to the element.

2. The semaphore is decreased by 1.

3. The waiting process becomes eligible to run again.

The kernel is in charge of associating data elements with queue semaphores. The kernel also increments and decrements the semaphores in response to requests from processes.

We can use an analogy — The Doctor's Waiting Room — to explain how semaphores can protect shared resources. Suppose that several patients are in a doctor's waiting room. These patients represent processes in the application; the doctor is a resource they share. The doctor resource is protected by a binary semaphore — the office door, which may be open or closed. The doctor's receptionist acts as the kernel, managing business by responding to requests from the patients.
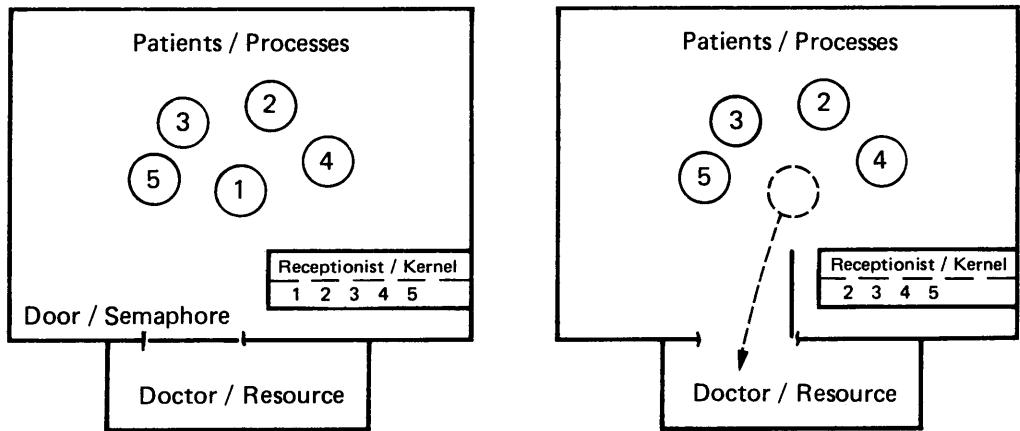
Each patient in the waiting room has stopped other activity to wait for access to the doctor. Each of these individuals requires access to a shared resource; each has asked the receptionist for access to the doctor. At the beginning of the day, the doctor's door was open, and the receptionist allowed the first patient to go right in, then shut the door. As subsequent patients enter and ask to see the doctor, the receptionist tells them to wait and adds their names to the list of waiting patients.

After consulting with the doctor, the first patient asks the receptionist to open the door. The receptionist does so, and the patient leaves.

The opening of the door makes the next waiting patient eligible for access to the doctor. The receptionist responds by granting the patient access to the shared resource and closing the door.

This doctor sees patients on a first-come, first-served basis. Alternatively, the doctor might have decided to see patients on a highest-priority, first-served basis.

ML-100-81

We can expand our analogy — The Medical Center Waiting Room — to explain how a counting semaphore works. Now we have a clinic, with one receptionist and one waiting room serving a dozen doctors. A set of flip cards is on the receptionist's desk. The cards, numbered 1 to 12, indicate the number of doctors free. As many as 12 patients can go into doctors' offices before anyone has to wait.

When more than 12 patients desire access to doctors, they are listed on either a first-come, first-served or a priority basis. When finished with a doctor, the patient asks the receptionist to open the door. The receptionist does so and flips a card to indicate that another doctor is free. If there are waiting patients, the receptionist opens the door for the next patient on the list.

5.2.2.1 Processes Use Semaphores to Send Packets, Messages — Semaphores can transmit data from one process to another. Processes, both static and dynamic, send data back and forth by using chunks, or packets, of memory managed by the kernel rather than by altering a shared data structure. The sender places a message in a packet; the receiver removes the message; the kernel acts as the go-between.

This type of transmittal is like two excited college professors who communicate with written notes (packets) instead of the blackboard (shared data area) in their office. They pass notes because one professor is a slow reader and the other is a fast writer. The blackboard is an unacceptable communication medium for them; once it is full, the fast writer must stand idle while the slow reader laboriously studies the chalk symbols. Instead, the fast writer stacks notes in a pile beside the slow reader. Each note in the stack is a message analogous to a MicroPower/Pascal packet.

Another advantage of passing notes is that the fast reader can dynamically allocate just enough paper to hold the information to be passed to the slow writer. This action is analogous to the dynamic allocation of packets by the MicroPower/Pascal kernel. Any shared data area declared in the source code must be large enough to hold the largest single piece of data sent among processes. Waste is avoided by allocating (via packets) just enough memory during execution to handle the communication needs. Packets of messages are the only way to transfer data between processes of different scopes, since no shared data areas exist between them.

In MicroPower/Pascal, the kernel allocates packets in response to requests from running processes. The configuration of each message is spelled out during the request.

A packet is requested from the kernel as follows. A process can issue a GET_PACKET request specifically, or it can ask for a packet by the SEND command. The SEND statement (with its numerous arguments) asks the kernel to allocate a packet, fill it with certain data, and signal a queue semaphore. The receiving process will issue the RECEIVE command, naming the same queue semaphore. When that semaphore is signaled (by the sender), the receiver gathers the data in the packet.

Both the sender and the receiver must specify the size and configuration of the message and can designate optional reply (acknowledgment) semaphores to report successful transmission. It is important to note that SEND and RECEIVE protect the integrity of processes by filtering messages through a third party, the kernel.

5.2.2.2 Race Conditions — A race condition arises when the relative speeds of execution of processes affect the behavior of an application. Race conditions among interdependent processes usually result from insufficient communication and can be disastrous. For instance, two device-access processes racing to set up registers in a peripheral device for subsequent I/O operations are likely to produce bizarre results. If the execution speeds of the processes vary each time they run, the results may appear different each time. The remedy for unwanted race conditions is prudent use of semaphores to eliminate the effects of unequal speeds of execution.

5.2.2.3 Critical Sections — A critical section is a sequence of instructions in one process that must finish executing before a particular section of another process can execute. In mutual exclusion, for example, the portion of each process concerned with accessing a shared resource must execute uninterrupted by similar portions of other processes. These critical sections can be prevented from interfering with each other by using WAIT and SIGNAL operations in the source code.

It is possible to serialize the execution of critical sections according to the ordering of semaphore waiting lists (process priority or first in, first out). This procedure is called establishing precedence. You can also design your own precedence mechanisms by using multiple semaphores.

5.2.3 Process Priorities Affect Concurrency

You can specify the importance of MicroPower/Pascal processes by assigning process priorities. A process priority can be any integer between 0 and 255; the higher the number, the higher the priority. Whenever two processes of different priorities are eligible to control the target CPU, the higher-priority process takes precedence.

You can assign process priorities to synchronize concurrent processes. However, priorities will not solve most synchronization problems, because priority assignments are a relatively inflexible mechanism, not a shortcut to process synchronization. Assignments of process priority should be used to fine-tune an application whose processes already cooperate at the same priority.

NOTE

Be wary of using assignments of process priority to solve concurrent synchronization problems. Priorities should be assigned to processes only to make the application more efficient.

5.2.4 SUSPEND and RESUME Affect Other Processes

Semaphore-based synchronization mechanisms are passive; that is, a process waiting on a semaphore may only stop itself from executing. SUSPEND and RESUME, on the other hand, give the running process power over those processes waiting for control of the CPU. These active instructions can be used to block execution of another process directly. Blocking may occur anywhere in the course of execution, not just at clearly defined WAIT instructions.

NOTE

SUSPEND and RESUME are not alternatives to the use of semaphores for synchronizing processes. Instead, SUSPEND and RESUME should be used only when a particular process must be disabled for some time, regardless of its progress.

5.3 Concurrency in Designing a Sample Application

We can use an example — The Bottle Corker Machine — to explain the synchronization of processes in concurrent application design. In this example, a high-speed bottling plant uses conveyors to transport bottles among automated workstations. These stations clean, fill, cork, and package bottles. Our hypothetical corking machine is controlled by a MicroPower/Pascal application.

5.3.1 The Target Hardware
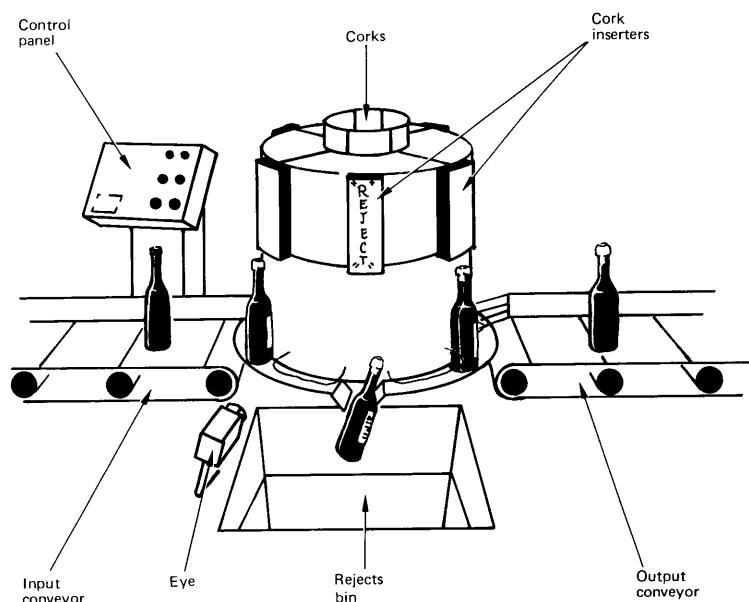
The corking machine consists of a vertical, rotating drum with a rounded ledge on which the bottles sit in discrete slots. A cork inserter assembly is above each slot, attached to the drum. As bottles move around the drum, they

are corked by inserters positioned above them. The inserters are reloaded from a cork supply in a hopper at the top of the machine

Two conveyor belts, 180° apart, service the machine. The input belt brings bottles to the machine, and the output belt takes away bottles. These two belts run at the same constant 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. This eye detects vacant slots and slots containing broken or unfilled bottles.

A large bin for bad bottles is in front of the drum. Slots with bad bottles are emptied into this rejects bin. This arrangement allows glass to be recycled and purges the line of bad bottles.

ML-091-81

5.3.2 Operating Characteristics

The bottle corker workstation performs the following tasks:

1. Inserts corks into bottles, ignoring any slots that do not contain bottles, so as not to waste corks

2. Rejects broken or unfilled bottles from the line

3. Senses when the cork supply is low and notifies the operator

4. Senses when the cork supply is depleted and rejects bottles until the corks are restored

5. Senses when the input conveyor breaks and shuts down the workstation when bottles are corked

6. Senses when the input conveyor starts and immediately starts the workstation
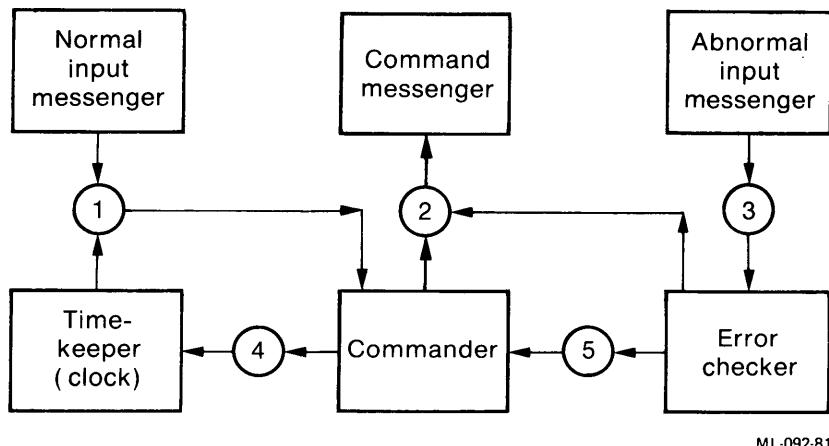
7. Senses when the output conveyor breaks and rejects bottles (uncorked) until the output conveyor restarts or the input conveyor stops

We will design a concurrent MicroPower/Pascal application to run this workstation.

5.3.3 Designing a Concurrent Solution

Concurrent design allows us to divide the problem into separate, smaller tasks. We will define six parallel tasks to run the machine and a process for each task. Three processes handle I/O between the application and the machine. Another process, the commander, sends normal, repetitive operating commands to the machine. The error-checker process receives notice of abnormal conditions from the machine. (This process may transmit a message to the commander, which in turn alters the command sequence.) Finally, the sixth process interfaces with the system clock to provide timings.

Figure 5-1 shows how the processes are related. There are six boxes (processes) and five points of communication (queue semaphores), denoted by circles.

ML-092-81

Figure 5-1: Tasks Comprising the Concurrent Solution
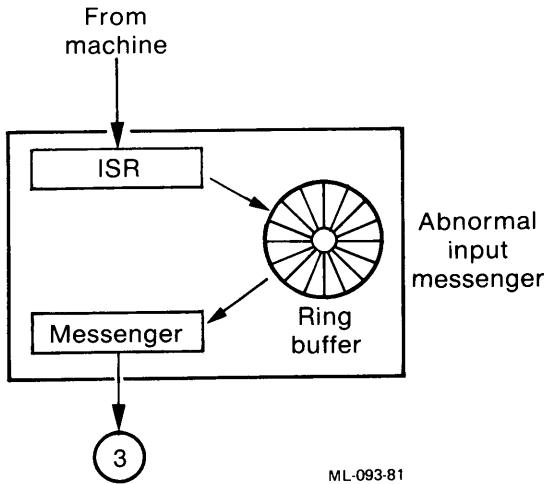
NOTE

When signaling and waiting on queue semaphores to pass messages between processes, remember to use MicroPower/Pascal SEND and RECEIVE statements, not SIGNAL and WAIT.

Serial lines with three device-access processes to handle the lines bring electrical signals to and from the machine. Incoming signals must be translated into symbolic codes and passed on to the commander and error-checker processes. On the output side, one coded command must be translated into line signals for the outgoing wire.

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 messenger is composed of two independent parts: the process and an interrupt service routine (ISR). (Interrupt service routines and device-handler processes are described in detail in the *MicroPower/Pascal Runtime Services Manual*, with a complete guide to writing device handlers.)

From
machine

Figure 5-2: The Messenger Process

Associated with each messenger process is a ring buffer, used to pass information between the independent interrupt service routine and the rest of the device-handler process (messenger). Upon receipt of a line signal, the ISR uses the PUT_ELEMENT statement to put an element into the ring buffer. As a clump of signals arrives, the ISR may do several PUT_ELEMENT operations in a short time. The ISR then waits for more signals while the messenger process removes each element in turn from the ring buffer (with the GET_ELEMENT statement) and performs the encoding operation.

Responding promptly to signals from the real-time environment is so critical that MicroPower/Pascal interrupt service routines automatically 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 pluck information from or put information on the lines and interact with temporary buffer storage, such as a ring buffer.

For instance, on the input side, the interrupt service routine is ready to respond to an interrupt from the hardware, grabbing signals off the wire and storing them for the encoder part of the messenger process. At the lower process priority, the messenger translates the messages into codes to be sent to the error-handler process.

We will name the three device-handler processes according to the type of information they convey: abnormal input messenger, command output messenger, and normal input messenger. The three processes can be described as follows:

1. The abnormal input messenger encodes messages from the error sensors in the workstation and passes codes to the error-handler process.

2. The command output messenger decodes messages from the commander and places appropriate signals on the output line to the machine. The command output messenger can receive input from two sources: the commander and the error-handler process. Under normal operation, only the commander sends messages to this process.

3. The normal input messenger encodes signals received during the workstation operation. The normal input messenger performs one other task. Before it sends each code from the hardware to the commander, it sends a message that cancels the countdown.

As shown in Figure 5-1, each arrow entering a box marks a waiting point for the process represented by that box. One box, the commander, has two arrows leading into it. The commander process must pay attention to acknowledgments from the machine and messages passed from the error-checker process. If the commander is made to wait for input from the error-checker process, the commander cannot run the machine. Ideally, the commander should pay attention to the error checker if a message is forthcoming immediately. This situation calls for use of a conditional receive, specified with the COND_RECEIVE statement. If there is a message, the commander will receive it; if not, the commander will continue without waiting. A process performing a conditional receive will never block its own execution because of the semaphore.

The other incoming arrow, or semaphore, passes acknowledgments from the normal input messenger and time-out signals from the timekeeper. If the hardware does not acknowledge orders from the commander process within a specified interval, the clock process notifies the commander of a time-out, and the command stream to the hardware is interrupted by a halt sequence. Because at least one of these messages must arrive during each cycle of the machine, this semaphore is vital to proper synchronization. The commander process must block itself from executing and wait for a message before continuing to issue orders to the machine.

The other processes wait on only one semaphore. Each semaphore is a queue semaphore, which acts like a mailbox, holding messages from one process to another. Queue semaphores rather than binary semaphores are used 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 semaphore; in this application only one process will wait on any semaphore. (This is not true for all applications, of course.) Finally, more than one arrow may leave the boxes (processes), since one process can signal more than one semaphore during execution.

So far, we have used the idea of concurrent execution of tasks to design our workforce of independent processes. In our application, if each process executed in a separate CPU, overall performance would be synchronized. In reality, however, only one process will run at a time. We can refine our application for greater efficiency by assigning relative priorities to workforce members.

If an error requiring a stop-at-once command comes in from the machine, we want it to receive prompt attention. Therefore, we give the error-checker process higher priority than the commander. The timekeeper process needs to work only when its clock ticks, but that work is very important, because timings depend on it. Therefore, we may want to give the timekeeper process a high priority.

Chapter 6
Application-Development Example

6.1 Overview

In this chapter we will describe the program for a simple application, prepare a memory image, load it into the target system, and run it. You need only minimal system knowledge to benefit from the following sections.

6.1.1 Requirements

In order to develop a MicroPower/Pascal application, you must be able to run the host RT-11 operating system and manipulate and edit files. The target system must include a DIGITAL VT100 or VT52 terminal as console device and 16K words of RAM memory.

6.1.2 Steps

The steps in creating a MicroPower/Pascal application are as follows:
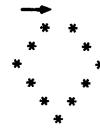
1. Design and code the source.

2. Compile the source code.

3. Build the memory image, using MicroPower/Pascal utilities.

4. Load the application into the target LSI-11.

5. Debug/run the application.

Our sample application will be concurrent and simple enough for you to enter the source code and perform the steps described below. (However, source code is also included as a file in the distribution kit and is used in the *MicroPower/Pascal Installation Guide* for testing.)
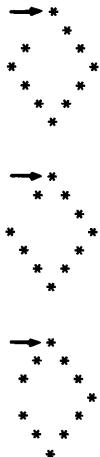
6.2 Application Example: The Shooting Gallery

We will use a simple application — The Shooting Gallery Computer Game — as our example. The program for this game spins a wheel containing 11 asterisks on the target system's DIGITAL VT100 or VT52 terminal screen. You, the player, try to eliminate all 11 asterisks, one by one, as they whirl by.

You accomplish this by taking potshots at one spot on the wheel. For example, at first the wheel looks like this:

```
   →
   *  *
 *      *
*        *
 *      *
   *  *
    *
```

(One asterisk has been knocked out to show the rotation of the wheel.) The wheel turns counterclockwise, and you see:

```
  → *
      *
 *      *
*        *
  *      *
    *  *
      *
```

```
  → * *
     * *
        *
 *        *
  *      *
    *  *
      *
```

```
  → *
    * *
 *      *
*        *
 *      *
   *  *
     *
```

In your subsequent shots, you try to eliminate the remaining asterisks as they pass by the arrow.

This sample program demonstrates basic concurrent programming techniques for synchronizing processes with semaphores. It also illustrates correct language syntax and the general format of a MicroPower/Pascal program. Finally, this program will serve as the basis for a step-by-step exploration of the MicroPower/Pascal utilities later in the chapter. (The source code for this application example is given in Appendix A.)

We will describe the way in which the processes communicate by using semaphores so as not to ruin shared data. We will also outline the general approach used to write concurrent code.

6.2.1 The Concurrent Program

In designing the game program, we first divided the game into two concurrent tasks to be performed by processes. The program consists of two Pascal processes and two sequential Pascal procedures:

- Entry — a dynamic process
- Example — the main program, a static process
- Setup — a procedure
- Dofirst — the initialization procedure

The main program, called Example, creates the process called Entry. Setup and Dofirst are nested procedures in Example. Example and Entry, the two processes, communicate via three semaphores:

'SHOOT!'

'SCREEN'

'SPIN!!'

These three semaphores are created within the initialization procedure, Dofirst. (Another procedure in the program, $TTYST, will be discussed later.)

6.2.1.1 Main Program Declarations — The variable declaration section (VAR) of the main program declares an array to represent the 12 locations on the asterisk wheel and the 3 variables, called structure descriptor blocks, that will hold the names and identification numbers of the semaphores. Note that because the code for Entry is nested inside Example, both processes will have access to these structure descriptor blocks.

This section of the program also declares character and integer variables needed throughout the program. There is one Boolean variable, Result. Some MicroPower/Pascal functions are designed to return a true or false value whenever called in order to notify the calling code of successful/unsuccessful completion of the function. In our program, we will use the Boolean variable Result to receive the value true or false when creating semaphores.

```
[SYSTEM(MICROPOWER),PRIORITY(200),
 DATA_SPACE(1000),STACK_SIZE(200)] PROGRAM Example;

  CONST
    Escapecode = 155;

  VAR
    Spinner, { Semaphore for }
    Onoff, { Semaphore for }
    Screen { Semaphore for serializing access to the screen }
         : STRUCTURE_DESC;
    Ar : ARRAY [0..11] OF CHAR;
    Bullet,
    Play,
    Esc : CHAR;
    Indx,
    Miss,
    Asterctr : INTEGER;
    Firstime, Result : BOOLEAN; { for use with MicroPower/Pascal functions }

  [EXTERNAL ($TTYST)] PROCEDURE No_echo (value : INTEGER); EXTERNAL;
```

6.2.1.2 Initialization Procedure (Dofirst) — The initialization procedure executes first, before any other part of the program. During initialization we create the three needed semaphores and designate which structure descriptor blocks in the main program will hold the name and identification number of each semaphore. Each semaphore-creating statement in Dofirst has the form of an assignment. The CREATE_BINARY_SEMAPHORE function is designed to return a true/false value to indicate success or failure (like other MicroPower/Pascal functions).

```
[INITIALIZE]   PROCEDURE Dofirst;
{ Create the binary semaphores. }
BEGIN
   Result := TRUE;
   IF NOT CREATE_BINARY_SEMAPHORE (NAME := 'SHOOT!', DESC := Onoff)
    THEN Result := FALSE;
   IF NOT CREATE_BINARY_SEMAPHORE (NAME := 'SCREEN', DESC := Screen)
    THEN Result := FALSE;
   IF NOT CREATE_BINARY_SEMAPHORE (NAME := 'SPIN!!', DESC := Spinner)
    THEN Result := FALSE;
   Esc := CHR(Escapecode);
   Firstime := TRUE;
END; { Dofirst }
```

6.2.1.3 Setup Procedure — The sequential procedure Setup runs before the Entry process is created and then after each round of the game. Setup fills the wheel array with asterisks, then initializes the asterisk and miss counters. Setup also signals each of the semaphores, so that initially all gates built into this program are open.

```
PROCEDURE Setup;
VAR
   n : INTEGER;

BEGIN
{ Initialize the array, leave target spot blank. }
   FOR n := 1 TO 11 DO
      Ar[n] := '*';
   Ar[0] := ' ';
   Indx := -1;
{ Initialize the counters. }
   Miss := 0;
   Asterctr := 11;
{ Signal all the semaphores. }
   SIGNAL (DESC := Onoff);
   SIGNAL (DESC := Screen);
   SIGNAL (DESC := Spinner);
END;   { Setup }
```

6.2.1.4 Main Program's Executable Block — The main program's executable block spins the asterisk wheel and presides over the time between rounds of the game. If a game player answers y (yes) to the question "Shall we begin?", the main program enters a loop. This loop sets up the initial conditions for the start of a round (via Setup), creates the Entry process, and starts spinning the asterisk wheel on the terminal screen. Keep in mind that because the main program executes concurrently with Entry, Entry may interrupt its execution to gain control of the CPU.
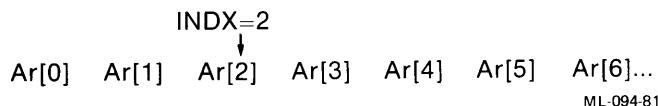
```
BEGIN
   WRITE (Esc, 'H', Esc, 'J');      { Erase terminal screen. }
   WRITE (Esc, 'Y', CHR(40), CHR(32));
   WRITE ('Shall we begin (y/n) ? ');
   READ  (Play);
   WRITELN;
   WHILE (Play = 'y') OR (Play = 'Y') DO
      BEGIN  { Main loop of game. }
         Setup;
         IF Firstime
          THEN Entry (NAME := 'ENTRY!')      { Create process }
          ELSE WRITE (Esc, 'Y', CHR(40), CHR(32));
         WRITELN ('Press any letter to shoot...Fire Away!');
```

Note that the main program's executable block includes a reference to the Entry process and to the Setup procedure. The two references look similar, but have different results.

When a reference to a process occurs during program execution, the kernel readies the new process for concurrent execution in the target processor. (The kernel places its process control block on the appropriate state queue.) Having been created, this process will run only according to the rules for MicroPower/Pascal processes. Its priority and the statuses of related semaphores will determine if and when it runs. Pascal procedures, by contrast, are entered directly when referenced by name.

Example contains a loop that spins the asterisk wheel by incrementing the variable Indx so that it points to each element of the array in turn. Then Indx wraps from the eleventh element back to element zero (using the MOD function, which returns the value of the remainder of a division operation). This loop continues as long as there are asterisks in the wheel and is the only section of Example that competes with the Entry process for control of the CPU. The loop contains two critical sections, protected by two separate binary semaphores. The loop ends when Asterctr attains a value less than 1 — when all the asterisks have been eliminated.

INDX=2
↓
Ar[0]   Ar[1]   Ar[2]   Ar[3]   Ar[4]   Ar[5]   Ar[6]...

ML-094-81

The two critical sections are protected by the semaphores 'SPIN!!' and 'SCREEN.' Example waits on the semaphore 'SPIN!!' (named in structure descriptor block Spinner) whenever it comes to the point of modifying the variable Indx.
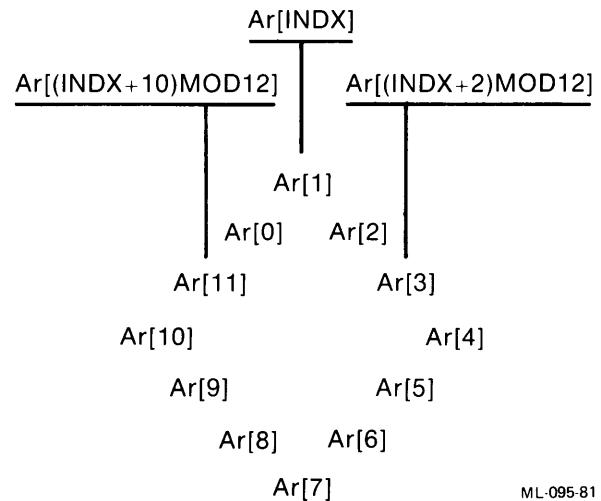
Indx is shared by Example and Entry. Example changes the value of Indx regularly; Entry uses Indx at unpredictable intervals as a subscript of the asterisk wheel array. When Example modifies Indx, there is a point at which Indx briefly has a value of 12, which is outside the array range. Entry must not cut in and try to use Indx as a subscript of the array before Example has finished updating it.

When Example has updated the value of Indx, it signals 'SPIN!!', allowing Entry to cut in (Entry may already be waiting for this chance). No harm can be done now that Example is finished with the shared resource.

NOTE

The variable Asterctr is also shared by the processes. Like Indx, Asterctr can be read by one process and changed by the other. Yet it is not necessary to protect Asterctr with a mutual-exclusion semaphore, because Asterctr cannot have a value that would harm the progress of the two processes.
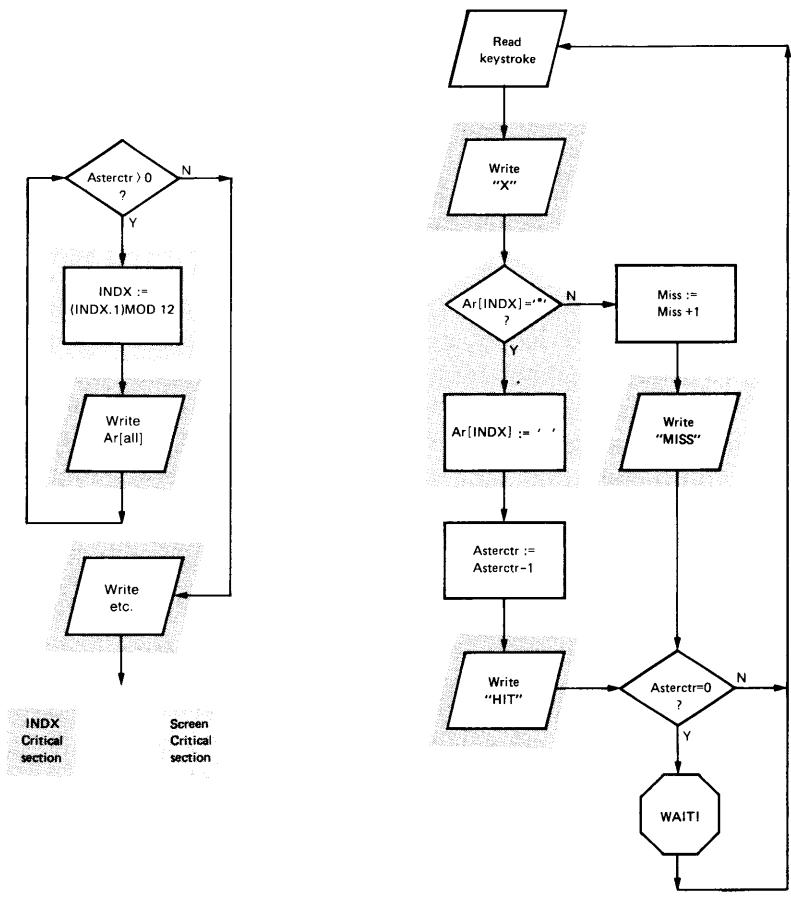
After Indx has been updated, Example writes the asterisk wheel array to the terminal screen (see Figure 6-1).

Ar[INDX]

Ar[(INDX+10)MOD12]   Ar[(INDX+2)MOD12]

Ar[1]

Ar[0]   Ar[2]

Ar[11]   Ar[3]

Ar[10]   Ar[4]

Ar[9]   Ar[5]

Ar[8]   Ar[6]

Ar[7]   ML-095-81

Figure 6-1:   Example of Array Positions on Terminal Screen

Since both processes write to the terminal, this shared resource is also pro-
tected by a mutual-exclusion semaphore called 'SCREEN'. If both processes
were allowed to interrupt each other's write statement sequences, the output
on the terminal screen would be bizarre (see Figure 6-2).

Figure 6-2:   Critical Sections in Example and Entry

The asterisk wheel loop in Example follows:

```
WHILE (Asterctr > 0) DO
   BEGIN      { Update the asterisk wheel. }
      WAIT (DESC := Spinner);
      Indx := Indx + 1;
      Indx := Indx MOD 12;
      SIGNAL (DESC := Spinner);      { Now print out the wheel. }
      WAIT (DESC := Screen);
      WRITE (Esc, 'H');
      WRITELN (' ->', Ar[Indx], '            ');
      WRITELN ('   ',Ar[(Indx + 11)MOD 12], '   ', Ar[(Indx + 1) MOD 12]);
      WRITELN ('  ',Ar[(Indx + 10)MOD 12], '     ', Ar[(Indx + 2) MOD 12]);
      WRITELN (Ar[(Indx + 9)MOD 12], '        ', Ar[(Indx + 3) MOD 12]);
      WRITELN ('  ',Ar[(Indx + 8)MOD 12], '     ', Ar[(Indx + 4) MOD 12]);
      WRITELN ('   ',Ar[(Indx + 7)MOD 12], '   ', Ar[(Indx + 5) MOD 12]);
      WRITELN ('     ',Ar[(Indx + 6)MOD 12]);
      WRITELN;
      SIGNAL (DESC := Screen);
   END;
```

After Asterctr reaches 0, the main program writes an appropriate message, asks whether the player wishes to continue, and loops or not, according to the answer.

The remainder of the main program executable block follows:

```
WAIT (DESC := Screen);
WRITE (Esc, 'H', Esc, 'J');      { Clean the terminal screen.}
WRITE (Esc, 'Y', CHR(40), CHR(32));
WRITELN ('You shot out all the asterisks!');
WRITELN ('...while missing ', miss:2, ' times...');
CASE Miss OF
   0: WRITELN ('You are obviously a visitor from an alien race.');
   1, 2: WRITELN ('You must work out with Gary Cooper.');
   3, 4, 5: WRITELN ('Don''t go anywhere near Wyatt Earp.');
   6, 7, 8, 9: WRITELN ('Sugarfoot you ain''t.');
   12, 13, 14: WRITELN ('You missed more than you hit!');
   15, 16, 17: WRITELN ('You have poor...................timing.');
   10, 11, 20: WRITELN ('Take some advice-stay out of the O.K.corral.');
   21, 22, 23: WRITELN ('Crazy glue in the ol'' holster, hum?');
   24, 18, 19: WRITELN ('My horse can shoot better than that!');
   25, 26, 27: WRITELN ('Targets moving too fast for you, Cowboy?');
END; { Case }
IF Miss > 27
   THEN WRITELN ('You can do better than that, can''t you, Tenderfoot?');

WRITELN;
WRITELN ('Press <ret>');
READLN;
WRITE    ('How about another round?  (y/n) ? ');
READ (Play);
IF (Play = 'y') OR (Play = 'Y')
   THEN Firstime := FALSE;
WRITE (Esc, 'H', Esc, 'J');      { Erase terminal screen. }
SIGNAL (DESC := Onoff);
END;      { Main game loop. }
```

```
WRITELN;
WRITELN     ('OK...Game cancelled due to lack of interest.');

DESTROY (DESC := Onoff)
{ Delete structures } ;
DESTROY (DESC := Screen);
DESTROY (DESC := Spinner);
STOP (NAME := 'ENTRY!');
WRITELN;
WRITE ('Thanks for the game, Sport.');

END                                    { of main program example } .
```

6.2.1.5 Dynamic Process (Entry) — Entry is the shooting mechanism of the game and runs concurrently with Example. Synchronization between the processes comes from the strategically placed WAIT and SIGNAL operations on semaphores 'SPIN!!', 'SHOOT!', and 'SCREEN'. Except for the first statement, a procedure call to $TTYST, Entry is an endless loop. It examines one element of the asterisk wheel — the element pointed to by Indx. If that element contains an asterisk, Entry makes it a blank, thereby eliminating the asterisk. If the element is already a blank, Entry leaves it alone. Entry is concerned only with element number Indx of the array.

The first and last statements of Entry's loop are a wait/signal pair. This semaphore, 'SHOOT!' (named in structure descriptor block Onoff), is signaled by Setup in the main program. As long as the variable Asterctr is not equal to 0, 'SHOOT!' will be signaled at the end of Entry, and the process can continue. As soon as Asterctr equals 0, 'SHOOT!' will not be signaled within Entry, and the process will enter the wait-active state unless or until 'SHOOT!' is signaled again by another process. This arrangement is used to control the execution of Entry. Entry will be eligible to run in the target CPU only if 'SHOOT!' is signaled regularly.

Assuming that 'SHOOT!' is signaled and that Entry proceeds through the first statement of the loop, it next encounters a READ. Entry receives a character from the console terminal keyboard. As soon as you enter a character at the terminal, Entry will proceed to its next statement. The next statement in Entry is an order, WAIT, on the semaphore named 'SPIN!!'. This is the mutual-exclusion semaphore guarding the variable Indx. Entry will not proceed until three conditions have been met:

1. The controlling semaphore 'SHOOT!' allows execution.

2. A character is entered from the terminal keyboard.

3. The target element of the array is free for access.

If the asterisks have been eliminated, 'SHOOT!' is not signaled at the bottom of the loop, and Entry is forced to wait, relinquishing the CPU.

The Entry process follows:

```
[PRIORITY(205), STACK_SIZE(200)] PROCESS Entry;
BEGIN
   No_echo (%O'100000');  { Turn off character echo. }
   WHILE TRUE DO
     BEGIN
       WAIT (DESC := Onoff);
       READ (Bullet);      { Wait for keyboard input. }
       WAIT (DESC := Screen);
       WRITE (Esc, 'Y', CHR(32), CHR(35));
       WRITE ('X');
       SIGNAL (DESC := Screen);
       WAIT (DESC := Spinner);
       IF Ar[indx] = '*'     { If asterisk in target spot. }
        THEN BEGIN
                Ar[Indx] := ' ';      { then blankit. }
                SIGNAL (DESC := Spinner);
                Asterctr := Asterctr - 1;
                WAIT (DESC := Screen);
                WRITE (Esc, 'Y', CHR(34), CHR(42));
                WRITE ('HITS:    ', (11 - Asterctr):2);
                SIGNAL (DESC := Screen);
             END
        ELSE BEGIN
                SIGNAL (DESC := Spinner);
                Miss := Miss + 1;
                WAIT (DESC := Screen);
                WRITE (Esc, 'Y', CHR(36), CHR(42));
                WRITE ('MISSES: ', Miss:2);
                SIGNAL (DESC := Screen);
             END;
       IF Asterctr <> 0      { If all asterisks are gone. }
        THEN SIGNAL (DESC := Onoff);
     END;
END;  { Entry }
```

6.2.2 A Potential Problem: Character Echoes

The READ statement normally causes the character entered at the terminal to automatically echo, or print immediately on the screen. However, in Entry, we call the procedure $TTYST, which defeats character echoes during the read operation.

The READ statement is not (and cannot be) protected by the mutual-exclusion semaphore 'SCREEN'. Enclosing the READ inside a critical section would probably stop both processes at some point while Entry waited for a character and Example waited for access to the terminal screen. This action would interrupt the smooth flow of execution necessary for the game to work properly.

Nonetheless, all write operations in Entry must be protected through mutual exclusion; otherwise, the display would be affected adversely. Therefore, we cannot allow this READ statement to cause an automatic write operation. The special procedure $TTYST is necessary. After you have successfully created this sample application and have seen it run, you might want to rebuild the application without including the $TTYST procedure.

6.2.3 Escape Sequences

The constant Escapecode, the variable Esc, and all WRITE statements containing Esc control the position of the cursor on the terminal screen. The variable Esc is of type CHAR. The first statement in the main program block assigns Esc the ASCII character value corresponding to a decimal 155 (the value of our constant Escapecode). Whenever the statement WRITE (Esc) is sent to the terminal, the next letter sent is interpreted as a command to reposition the cursor. There are several of these letters, each interpreted as a different command when sent to the terminal after the escape character. Esc, 'H', for example, immediately moves the cursor to the upper left-hand corner of the screen.

6.3 Creating the Application Example from Source Code

Now we will set up the host and target systems and transform the source code into an executable, concurrent application. We will then down-line load the application from the host into the target and run it.

6.3.1 Configuring the Hardware

First, the host and target hardware must be configured for down-line loading the application example. We need a serial line interface on both systems. In addition, we need a second, optional, serial line on the target for a DIGITAL VT52 terminal or VT100 terminal set to VT52 mode. We will not need to debug this sample application, but we will use PASDBG to load the application over the host-to-target serial line.

The *MicroPower/Pascal Installation Guide* details how to configure the system's hardware for down-line loading using the PASDBG symbolic debugger. That manual also contains directions for down-line loading using the DLLOAD utility.

6.3.2 Completing the Configuration Worksheet

Once the hardware has been set up correctly, you must specify the characteristics of your target system to the MicroPower/Pascal utilities. You do this by editing an existing configuration file, which consists of a number of MACRO–11 instructions. You edit the file in order to supply configuration information and to override any of the built-in configuration defaults that do not suit your needs.

Refer to Figure 6–3, which contains the correct configuration information for the sample application. We will look at each item on the worksheet.

Configuration name ___*Ex conf*___

SYSTEM: Debugger support   (Y) N   [No]

               Optimize   Y   (N)   [No]

```
                    INCLUDE ONLY IF SYSTEM OPTIMIZE = Y

RESOURCES:     Kernel stack size _____  bytes RAM
                                       [predefined minimum]
               Number of system packets _____
                                              [20.]
               Kernel pool for system data structures _____  bytes RAM
                                                      [1000.]
               Free RAM table size _____  bytes
                                            [20.]


TRAPS:
               [All]‾‾‾ ‾‾ ‾‾ ‾‾ ‾‾ ‾‾ ‾‾

               All =   | LSI | SBC-11/21 |  T10  EMT  TRP |
                       |     |           |      TR4  BRK  |

               Other traps:              FIS  FPP  MMU  MPT
```

PROCESSOR:

Memory-management unit	Y	(N)		No
Floating-point unit	(None)	FP11	FIS	None
Type	T11	L112	L1123	L1123
Vector	_____			1000(octal) (400octal SBC-11/21)

KXT11
(for SBC-11/21)

nxm	break	syshalt	level7	
__HALT	TRAP	HANG__	ODT __	level7 only if
__TRAP	HALT	ODT __	TRAP __	nxm = HALT
[HALT]	[TRAP]	ODT ROM __	ODT ROM __	break = TRAP syshalt = HANG
		USER__	IGNORE__	
		[ODTROM]	[TRAP]	

MEMORY:	Base address	Size (64-byte segment)	Type	Parity	Parity controller CSR	Volatile
Segment 1	*O*	*512.*	RO (RW)	Y N		Y N
Segment 2			RO RW	Y N		Y N
Segment 3			RO RW	Y N		Y N
Segment 4			RO RW	Y N		Y N
			[RW]	[No]	[0]	[Yes]

DEVICES: *60 64 100 300 304* ___ ___ ___

Figure 6-3: Configuration Worksheet

Note the word "optimize" beside SYSTEM. Unless we optimize, the configuration file will default certain values in the RESOURCES and TRAPS macros. All the defaults in these two areas are suitable for our sample, so we do not need to optimize. Because we will use PASDBG to load the application into the target, we check "debugger support."

Our memory requirements are 512 64-byte segments, starting at a base address of 0. Many memory configurations of different types, sizes, and starting addresses are possible. You can also specify memory size with the expression n.*32., where n. and 32. are decimal numbers. In this expression, n is the number of kilo words of target memory. If the target has 16K words of memory, you can express the memory size as 16.*32 on the worksheet.

The directions in our example assume an unmapped target system without floating-point capability. Therefore, these options are not checked after the PROCESSOR macro.

Beside the DEVICES macro, we will specify five vectors: 60, 64, 100, 300, and 304. The terminal will use these vectors for I/O with the target processor.

Since we did not check "optimize" beside SYSTEM, we need not include RESOURCES and TRAPS. Therefore, the kernel's free data space sizes need not be tailored for our application; defaults will suffice.

6.3.3 Editing the Configuration File

Now that we have the proper configuration outlined on the worksheet, we can tailor the configuration file (at the host) to our needs by using one of the available editor programs to change it. There are three editor programs (text editors) available: EDIT, KED, and K52; they are summarized in the *MicroPower/Pascal System User's Guide* (Chapter 3). The RT–11 documentation set contains detailed instructions on the use of the editors.

NOTE

In the following sections, the symbol (RET) indicates that you should press the return key, and the symbol (CTRL/C) indicates that you should press the C key simultaneously with the CTRL key. Red type is used for information you enter from the keyboard; black type indicates MicroPower/Pascal and RT–11 responses and prompts.

To call up one of the editors, you type one of the following lines at the host system console terminal:

```
.EDIT/OUTPUT:EXCONF.MAC CONFIG.MAC

.EDIT/KED/OUTPUT:EXCONF.MAC CONFIG.MAC

.EDIT/K52/OUTPUT:EXCONF.MAC CONFIG.MAC
```

Note how to specify the KED and K52 editors. CONFIG.MAC is the name of the configuration file in the distribution kit. OUTPUT:EXCONF.MAC specifies that a copy of CONFIG.MAC is to be made and renamed EXCONF.MAC. This copy, not the original file, will be changed. EX-

CONF.MAC is the suggested name of your edited copy of the configuration file. If you do not include OUTPUT:EXCONF.MAC, your editing commands will affect the original file, CONFIG.MAC.

The following pages describe 12 actions you perform to fully develop the sample program into a usable memory image.

1. Using one of the editors, access EXCONF.MAC (CONFIG.MAC) and make the indicated changes (red type) in the following lines:

```
CONFIGURATION      Exconf

SYSTEM             debug=YES, optimize=NO

PROCESSOR          mmu=NO

MEMORY             base=0, size=512., type=RAM

;RESOURCES         STACK=500., PACKETS=20., STRUCTURES=8192,

;TRAPS             ALL

DEVICES            60,64,100,300,304
```

Note how these changes to the configuration file reflect our choices on the configuration worksheet. As you can see, several defaults — in resources and traps — occur because we did not optimize the kernel.

2a. Enter the following command line:

```
.MACRO EXCONF+COMU.SML/LIBRARY
```

This command runs the MACRO-11 assembler program to produce an object module called EXCONF.OBJ. COMU.SML is the name of a macro library file containing all the macros named in EXCONF.MAC. The configuration file is now sufficient for our example, having been edited and assembled to produce an object module.

We will use the prefix file XLPFXE.MAC to include an XL driver in the application. This driver is required for the target terminal.

2b. Create an object module from the prefix file XLPFXE.MAC:

```
.MACRO XLPFXE+COMU.SML/LIBRARY
```

We will now compile the Pascal source code of our sample program. Compiling the source code will produce another object module containing the program.

6.3.4 Compiling the Source Code

The MicroPower/Pascal compiler requires a minimum 128KB of memory to run in the host system. In addition, a variable amount of mass storage (disk, diskette) is needed, depending on the size of the compiling job and the output options you desire.

You compile your complete application in stages. In addition, each Pascal main program can be compiled in stages.

Input to the compiler is a maximum of six source files containing Pascal code. The MicroPower/Pascal compiler concatenates the input files and compiles the whole.

Output from the compiler consists of the following:

- One object module of default file type .OBJ or one optional MACRO–11 code module of default file type .MAC

- One optional listing of default file type .LST

In summary, the compiler translates Pascal source-program statements into machine language instructions, optimized for efficiency to minimize the amount of code generated.

3. To compile the program, enter the following command:

```
,R PASCAL (RET)
*Exampl=Exampl  (RET)
```

The MicroPower/Pascal compiler, called PASCAL, has located our sample program source code in the file EXAMPL.PAS and has created an object module called EXAMPL.OBJ. The next step is to take the three object modules, EXAMPL.OBJ, XLPFXE.OBJ, and EXCONF.OBJ, and use them as input to the MicroPower/Pascal utilities. The utilities will create a memory image of the application that can be loaded into the target system.

6.3.5 Building the Application

Now that our program has been written and successfully compiled, we can begin creating the memory image to be loaded into the target processor.

In order to create the application, we must construct a kernel to contain the various system services that may be called on from the object code of our program. We will construct a full-function kernel, one that provides the maximum number of services, whether they are needed or not. Note that the size of the application can be greatly reduced by excluding unneeded kernel services. Building a customized kernel is covered in the *MicroPower/Pascal System User's Guide.*

The major utilities for MicroPower/Pascal are MERGE, RELOC, and MIB. A description of each follows.

MERGE resolves references from one module to another by using symbol table information within each module. (References to the kernel are resolved by using the kernel symbol table file.) MERGE combines program sections of the same name.

The input to MERGE is as follows:

- Multiple object modules, of file type .OBJ and/or file type .MOB

- The kernel symbol table, of file type .STB

The output is as follows:

- One object module, of file type .MOB

- One link map, of file type .MAP

- One auxiliary file, of type .OBJ

RELOC sorts program sections alphabetically as well as by R/O and R/W attributes, assigns virtual base addresses for each section, and relocates the contents of each program section. RELOC also creates a symbol table, using the relocated addresses of all symbols present within the module.

The input to RELOC is as follows:

• One merged object module, of file type .MOB

• One optional memory image, of file type .MIM (unmapped only)

The output is as follows:

• One process image, of file type .PIM

• One memory map, of file type .MAP

• One symbol table, of file type .STB

MIB creates in one file the executable memory image of the full application, consisting of the kernel and one or more static processes. (MIB can also install a .PIM file in an existing memory image.) MIB creates a symbol table for use with the MicroPower/Pascal symbolic debugger, if specified.

The input to MIB is as follows:

• One process image, of file type .PIM

• One memory image, of file type .MIM

• One symbol table, of file type .STB

The output is as follows:

• One memory image, of file type .MIM

• One map of installed processes, of file type .MAP

• One debug symbol table, of file type .DBG

4. To merge the kernel, type:

```
.R MERGE
*KERN.MOB=EXCONF.OBJ,PAXU.OBJ  (RET)
*(CTRL/C)
```

The files to the right of the equals sign are input files. These input files are the configuration object module and PAXU.OBJ, the file containing a library of modules that can be combined to form our application's kernel.

5. Next, relocate the kernel with RELOC to create a symbol table (.STB file) and a program image (.PIM) file:

```
.R RELOC
*KERN.PIM,,KERN.STB=KERN.MOB  (RET)
*(CTRL/C)
```

The output of step 4, KERN.MOB, is the single input file for this step. There are two output files: KERN.STB contains a list of names to be used when calling on kernel services, and KERN.PIM contains the kernel's program image.

6. Now create a memory image (.MIM file), with the kernel in place:

```
.R MIB (RET)
*EXAPPL.MIM=KERN.PIM/k/s (RET)
*(CTRL/C)
```

At this point we have created four things:

- A memory image of the example application — including just the kernel — in the file EXAPPL.MIM

- An object module containing the Pascal source program

- A system process that interfaces with the target terminal

- A symbol table containing the addresses of named entities inside our kernel that could be referenced automatically from the source program

7. Next, install the driver process in the application. Enter the following:

```
.R MERGE (RET)
*XL.MOB=XLPFXE.OBJ,KERN.STB,DRVU.OBJ (RET)
*(CTRL/C)
```

8. Run RELOC:

```
.R RELOC (RET)
*XL.PIM=XL.MOB,EXAPPL.MIM (RET)
*(CTRL/C)
```

9. Run MIB:

```
.R MIB (RET)
*EXAPPL.MIM=XL.PIM,EXAPPL.MIM/s (RET)
*(CTRL/C)
```

10. Merge the compiled sample program with the system library and the kernel symbol table (.STB):

```
.R MERGE (RET)
*EXAMPL.MOB=EXAMPL.OBJ,KERN.STB,LIBNHD.OBJ (RET)
*(CTRL/C)
```

The inputs to this merge step are object module files. The sample program contains calls to routines contained in the library LIBNHD.OBJ, renamed SYSLIB. These routines in turn request services from components of the kernel (listed in the file KERN.STB). The file EXAMPL.MOB contains the sample program object module, with all calls to other module libraries resolved.

11. Relocate the merged example object module:

```
.R RELOC (RET)
*EXAMPL.PIM=EXAMPL.MOB,EXAPPL.MIM (RET)
*(CTRL/C)
```

EXAPPL.MIM is the name of our existing memory image file. EXAMPL.MOB will be relocated to fit into EXAPPL.MIM.

12. Finally, insert the sample program memory image into the existing application memory image:

```
.R MIB (RET)
*EXAPPL.MIM=EXAMPL.PIM,EXAPPL.MIM/s (RET)
*(CTRL/C)
```

We have built a memory image ready to be placed in the target system memory.

6.3.6 Loading and Running the Application

Because we specified DEBUG=YES in the configuration file, we can now use PASDBG to load the application into the target. We do this by taking two actions, as follows:

1. Load the TD driver into the host:

```
.LOAD TD
```

2. Turn on the target system power.

Next, we run the PASDBG program at the host by typing the following:

```
.R PASDBG (RET)
```

PASDBG will print out several messages and then prompt:

```
PASDBG> LOAD/TARGET EXAPPL (RET)
```

PASDBG will print out several messages as the application is loaded and then prompt:

```
PASDBG> GO (RET)
```

The application will begin to execute.

The application is now running. On the target system terminal screen, you should see the message:

```
Shall we begin - y,n - ?
```

Type the letter y to begin the game.

The letter n exits from the application. To play again after entering n, you will have to restart the application in the target by typing the following lines at the host:

```
(RET)
PASDBG> INIT/RESTART (RET)
```

PASDBG will print out several messages and then prompt:

```
PASDBG> GO (RET)
```

6.3.7 Using DLLOAD

By running DLLOAD in the host, you can boot down-line the application into a target system when power is turned on, as if from a DECtape II drive. DLLOAD is a means of placing the application into target system RAM via a communication link from the development system. This communication link consists of a port connected to the target by a DLV11 serial line interface.

When power turns on in the target LSI–11 processor, a bootstrap program already present in permanent read-only memory automatically accesses the device connected to the processor's DL11 serial line interface. Normally, this device holds a secondary bootstrap program that loads the application into main memory and starts execution. But in down-line loading with DLLOAD,

the device connected to the serial line interface is replaced by a communica-tion link leading to the host processor. DLLOAD, running in the host, emu-lates a DECtape II drive and sends the application over the serial line into target memory.

To run DLLOAD at the host, enter the following:
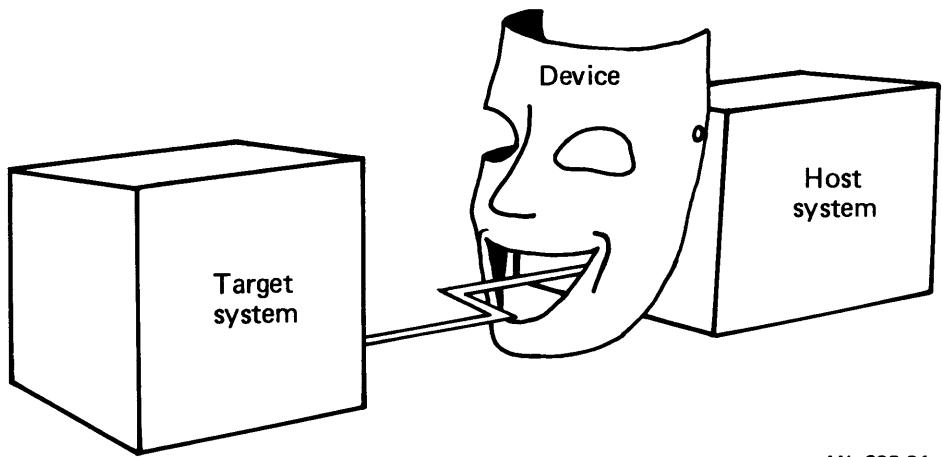
```
.R DLLOAD
```

You will be prompted to specify the following:

- The host-system device containing the application (memory image)

- The name and type of the file that holds the application — in this case, Exappl.mim (omit this if the device is not file-structured)

For DLLOAD to work, the XTX device driver must be loaded into the host memory, and the DD bootstrap must be included in the application memory image (using the /B switch to the MIB utility). Chapter 12 of the *MicroPower/Pascal System User's Guide* explains how to use DLLOAD.

When you turn on the target system power, Exappl.mim will be booted into target memory and will begin to run.

ML-098-81

6.3.8 Debugging the Application

PASDBG, the symbolic debugger supplied as part of the MicroPower/Pascal package, gives you a range of tools to use in testing your application for errors. There are more than 30 commands. You can use the debugger to do the following:

- Examine/modify memory locations

- Look at a process and its process control block, subordinate processes, stack contents, and state

- Find the location of certain data in memory

- Proceed through execution one statement/instruction/label at a time

- Halt execution at any point, according to breakpoints or flags set in the source code

PASDBG can treat the contents of memory locations like any of the following data:

ASCII	Octal
Binary	PDP–11 instruction
Decimal	RAD50
Hexadecimal	

You can enter and/or display memory contents in any of these representations. This determines the mode of the debugger's presentation of memory contents. In addition, PASDBG can interpret data in word or byte lengths.

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, each instance of a symbol must be distinguished from all other references to that name. This is done by using a pathname, which looks like the following:

system \ process \ procedure \ function \ symbol

Each part of the pathname leads to the next part, narrowing the scope, until the symbol has been identified. (Pathnames may contain more than one process or procedure.)

When the scope of symbols and the mode of data interpretation are clear, you can easily use the other PASDBG commands to check on the behavior of a misbehaving application. To use PASDBG, /D options must be included in the command lines when you run MERGE, RELOC, MIB, and the compiler. See the *MicroPower/Pascal System User's Guide* for the correct use of /D. The *MicroPower/Pascal Debugger User's Guide* contains a complete description of the PASDBG symbolic debugger, as well as some sample debugging sessions.

Appendix A
Source Program for Application Example

```
[ SYSTEM(MICROPOWER), PRIORITY(200),
  DATA_SPACE(1000), STACK_SIZE (200)] PROGRAM Example;

CONST
  Escapecode = 155;

VAR
  Spinner, { Semaphore for }
  Onoff, { Semaphore for }
  Screen { Semaphore for serializing access to the screen }
        : STRUCTURE_DESC;
  Ar : ARRAY [0..11] OF CHAR;
  Bullet,
  Play,
  Esc : CHAR;
  Indx,
  Miss,
  Asterctr : INTEGER;
  Firstime, Result : BOOLEAN; { for use with MicroPower/Pascal functions }

[EXTERNAL ($TTYST)] PROCEDURE No_echo (value : INTEGER); EXTERNAL;

[INITIALIZE] PROCEDURE Dofirst;
{ Create the binary semaphores. }
BEGIN
  Result := TRUE;
  IF NOT CREATE_BINARY_SEMAPHORE (NAME := 'SHOOT!', DESC := Onoff)
    THEN Result := FALSE;
  IF NOT CREATE_BINARY_SEMAPHORE (NAME := 'SCREEN', DESC := Screen)
    THEN Result := FALSE;
  IF NOT CREATE_BINARY_SEMAPHORE (NAME := 'SPIN!!', DESC := Spinner)
    THEN Result := FALSE;
  Esc := CHR(Escapecode);
  Firstime := TRUE;
END; { Dofirst }


PROCEDURE Setup;
VAR
  n : INTEGER;
BEGIN
{ Initialize the array, leave target spot blank. }
  FOR n := 1 TO 11 DO
    Ar[n] := '*';
```

```
      Ar[0] := ' ';
      Indx := -1;
   { Initialize the counters. }
      Miss := 0;
      Asterctr := 11;
   { Signal all the semaphores. }
      SIGNAL (DESC := Onoff);
      SIGNAL (DESC := Screen);
      SIGNAL (DESC := Spinner);
   END; { Setup }


   [PRIORITY(205), STACK_SIZE(200)] PROCESS Entry;
   BEGIN
      No_echo (%O'100000'); { Turn off character echo. }
      WHILE TRUE DO
         BEGIN
            WAIT (DESC := Onoff);
            READ (Bullet); { Wait for keyboard input. }
            WAIT (DESC := Screen);
            WRITE (Esc, 'Y', CHR(32), CHR(35));
            WRITE ('X');
            SIGNAL (DESC := Screen);
            WAIT (DESC := Spinner);
            IF Ar[indx] = '*' { If asterisk in target spot, }
             THEN BEGIN
                     Ar [Indx] := ' '; { then blank it. }
                     SIGNAL (DESC := Spinner);
                     Asterctr := Asterctr - 1;
                     WAIT (DESC := Screen);
                     WRITE (Esc, 'Y', CHR(34), CHR(42));
                     WRITE ('HITS : ', (11 - Asterctr):2);
                     SIGNAL (DESC := Screen);
                  END
             ELSE BEGIN
                     SIGNAL (DESC := Spinner);
                     Miss := Miss + 1;
                     WAIT (DESC := Screen);
                     WRITE (Esc, 'Y', CHR(36), CHR(42));
                      WRITE('MISSES : ', Miss:2);
                     SIGNAL (DESC := Screen);
                  END;
            IF Asterctr <> 0      { If all asterisks are gone, }
               THEN SIGNAL (DESC := Onoff);
         END;
   END; { Entry }


   BEGIN
      WRITE (Esc, 'H', Esc, 'J'); { Erase terminal screen. }
      WRITE (Esc, 'Y', CHR(40), CHR(32));
      WRITE ('Shall we begin (y/n) ? ');
      READ (Play);
      WRITELN;
      WHILE(Play = 'y') OR (Play = 'Y') DO
         BEGIN { Main loop of game. }
            Setup;
            IF Firstime
              THEN Entry (NAME := 'ENTRY!') { Create process }
              ELSE WRITE (Esc, 'Y', CHR(40), CHR(32));
            WRITELN ('Press any letter to shoot...Fire Away!');

            WHILE (Asterctr > 0) DO
               BEGIN { Update the asterisk wheel. }
                  WAIT (DESC := Spinner);
```

A-2   Source Program for Application Example

```
                Indx := Indx + 1;
                Indx := Indx MOD 12;
                SIGNAL (DESC := Spinner);       { Now print out the wheel. }
                WAIT (DESC := Screen);
                WRITE (Esc, 'H');
                WRITELN (' ->', ArCIndx], '              ');
                WRITELN ('   ',ArC(Indx + 11)MOD 12], '  ', ArC(Indx + 1) MOD 12]);
                WRITELN (' ',ArC(Indx + 10)MOD 12], '    ', ArC(Indx + 2) MOD 12]);
                WRITELN (ArC(Indx + 9)MOD 12], '       ', ArC(Indx + 3) MOD 12]);
                WRITELN (' ',ArC(Indx + 8)MOD 12], '    ', ArC(Indx + 4) MOD 12]);
                WRITELN ('   ',ArC(Indx + 7)MOD 12], '  ', ArC(Indx + 5) MOD 12]);
                WRITELN ('     ',ArC(Indx + 6)MOD 12]);
                WRITELN;
                SIGNAL (DESC := Screen);
             END;
        WAIT (DESC := Screen);
        WRITE (Esc, 'H', Esc, 'J');        { Clean the terminal screen. }
        WRITE (Esc, 'Y', CHR(40), CHR(32));
        WRITELN ('You shot out all the asterisks!');
        WRITELN ('...while missing ', miss:2, ' times...');
        CASE Miss OF
          0: WRITELN ('You are obviously a visitor from an alien race.');
          1, 2: WRITELN ('You must work out with Gary Cooper.');
          3, 4, 5: WRITELN ('Don''t go anywhere near Wyatt Earp.');
          6, 7, 8, 9: WRITELN ('Sugarfoot you ain''t.');
          12, 13, 14: WRITELN ('You missed more than you hit!');
          15, 16, 17: WRITELN ('You have poor..................timing.');
          10, 11, 20: WRITELN ('Take some advice--stay out of the O.K. corral.');
          21, 22, 23: WRITELN ('Crazy glue in the ol'' holster, hum?');
          24, 18, 19: WRITELN ('My horse can shoot better than that!');
          25, 26, 27: WRITELN ('Targets moving too fast for you, Cowboy?');
        END; { Case }
        IF Miss > 27
          THEN WRITELN ('You can do better than that, can''t you, Tenderfoot?');

        WRITELN;
        WRITELN ('Press <ret>');
        READLN;
        WRITE ('How about another round (y/n) ? ');
        READ (Play);
        IF (Play = 'y') OR (Play = 'Y')
          THEN Firstime := FALSE;
        WRITE (Esc, 'H', Esc, 'J'); { Erase terminal screen. }
        SIGNAL (DESC := Onoff);
      END;       { Main game loop. }

  WRITELN;
  WRITELN ('OK...Game cancelled due to lack of interest.');

{ Delete structures }
  DESTROY (DESC := Onoff);
  DESTROY (DESC := Screen);
  DESTROY (DESC := Spinner);
  STOP (NAME := 'ENTRY!');
  WRITELN;
  WRITE ('Thanks for the game, Sport.');

END.
```

Glossary

Terms in the Glossary are used throughout the documentation set.

Absolute section

A section of code that must reside in specific memory locations; it is not relocatable.

Actual parameter

A value or variable that is passed in a procedure or function call, used during execution of the subprogram.

Actual parameter list

In MicroPower/Pascal, the list of parameters specified in a subprogram call. The actual parameters must be in a specific sequence, separated by commas, and the list enclosed in parentheses.

Application program

In MicroPower/Pascal, a program developed on a host RT–11 operating system that runs on a stand-alone target system.

ASECT

An absolute-addressed section of a program that is not relocatable. ASECTs (usually used for symbols) must reside in specified memory locations.

Asynchronous

Not operating in exact time coincidence; asynchronous events occur unpredictably in relation to instruction execution.

Base type

1. For a subrange, the scalar type of which it is a subset. For example, the subrange 0...123 has the base type INTEGER.

2. For a set, the nonreal scalar type from which the elements of the set are chosen.

Binary semaphore

A global variable whose value varies from 0 to 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.

Block

1. 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 from a mass storage device involved in I/O.

2. In Pascal, a statement sequence delimited by the keywords BEGIN and END; can be used anywhere a statement is used.

3. As a verb in MicroPower/Pascal, to inhibit the execution of a process until a condition is met.

Blocked (process)

A process waiting for an event to occur (a specific semaphore 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 need 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 within 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.

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

Context

The set of data defining the environment, both hardware and software, in which a process executes. Hardware context includes the contents of the general registers, memory-mapping registers, floating-point registers, and 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 global variable whose value varies between 0 and some number greater than 1. Counting semaphores are managed by the kernel in response to requests from processes. Each counting semaphore can have a queue of waiting processes.

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 themselves, waiting for resources that will never be available.

Debugger

See Symbolic debugger.

Declaration (section/entry)

A program specification that lists one or more labels or associates an identifier with the class of data types it represents. In Pascal, labels and identifiers for constants, functions, procedures, types, and variables must be declared. The part of a program or subprogram block that contains the declarations is called the declaration section.

Device handler

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

Device register

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

Dispatcher

The kernel routines that allocate the processor resource to an eligible process and save and restore the process's context. (The system scheduler is part of the dispatcher.)

DLLOAD

A MicroPower/Pascal utility program that down-line loads a memory image from the host to the target system.

Down-line loading

Loading a program into the target processor's memory over a serial-line communications link from the host RT–11 operating system.

Driver

See Device handler.

Dynamic allocation

The granting to a process of a resource 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 start-up.

Dynamic process

A process not defined during target system initialization, but created by 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.

Exception condition

An event, detected by hardware or software, that causes a change in the flow of instruction execution (caused by a condition other than an interrupt or execution of a jump, branch, case, or call instruction). 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 trace traps.

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 address more than 64KB address space.

External symbol

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

Flag

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

2. Noting an error condition.

Fork process

A process on the fork queue. A fork process is created with the FORK request by an interrupt service routine. See Fork queue.

Fork queue

A queue set up in the kernel to allow processes special sequential access to the processor. Although lower than interrupt priorities, the fork queue priority is higher than any process priority, expediting the execution of processes placed on the fork queue. For example, interrupt-handling routines placed on the fork queue will execute before other processes in the application. This method ensures quick attention to interrupts.

Formal parameter

A name, declared in the heading of a procedure or function, that represents an actual parameter to be passed when that procedure or function is invoked. Variables declared as formal parameters are not stored.

General process

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

Global symbol

A link between independently compiled or assembled program modules. A global symbol is defined in one module and can be referenced from other modules. An identifier is an example of a global symbol.

Handler

See Device handler.

Heap

An area of memory in the MicroPower/Pascal application for dynamic allocation of pointer objects. Processes' stacks are allocated from the heap.

Host processor

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

Intermodule reference

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

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 when a particular device signals the processor with an interrupt. The processor locates ISRs in memory, using an address vector supplied by (or elicited from) the interrupting device. ISRs are also called interrupt-handling routines or interrupt handlers.

ISR

See Interrupt service routine.

Kernel

A set of software modules supplied by DIGITAL for inclusion in the MicroPower/Pascal target system. 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 a large number of service functions that can be invoked by the user. The kernal services fall into these catagories:

- 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 files are merged or linked with source program modules during MicroPower/Pascal development.

Library module

A program module from a library file.

Linking

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

1. Assigns absolute addresses

2. Produces a load map and creates a symbol table

3. Relocates the program sections within the object modules

4. Resolves global symbols that are defined in one module and referenced by external symbols in another

5. 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 ready for loading and executing (relocated, with references to labels and identifiers resolved). A completed memory image file is the load module for an application.

Mapped memory

Memory that is divided into segments, or pages, each located separately in (mapped into) physical storage. Mapping translates the 16-bit addresses used with LSI–11 processors into a physical memory space of 18- or 22-bit address size. Specifically in the LSI–11/23, up to the equivalent of 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 and containing the image of the application program as it will appear in the target system memory. This file can be down-line loaded or bootstrapped from DECtape II or RX02, or loaded 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 necessary)

• Kernel

• Relocated process image file (file containing an image of the program as it will appear 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.

Modular programming

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

Module

In the MicroPower/Pascal language, an attribute applied to a declaration section of statements.

Monitor

An RT-11 overseer program that controls and tracks system business. The monitor controls, observes, supervises, or verifies actions of the computer system. It is a collection of routines that control the operation of user and system programs, schedule operations, allocate resources, and perform I/O.

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 any one time.

Object module

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 an optional debug symbol table.

Object time system (OTS)

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

OTS

See Object time system.

Overlayed

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.

Page address registers (PAR)

Registers containing the base addresses of one to eight 8KB blocks of memory.

Page descriptor register (PDR)

A register containing access information about the 8KB pages of memory whose base is described by the corresponding PAR (length, R/O versus R/W, etc.).

PAR

See Page address registers.

PASCAL 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.

PASCAL procedure

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

PASDBG

The Pascal symbolic debugger program.

PCB

See Process control block.

PDR

See Page descriptor register.

Physical address

The hardware address of a specific main memory location. Physical addresses in the LSI–11/23 range from 0 to 4MB (in 22-bit mode with optional MSV11–L). Virtual addresses of up to 16 bits (64K bytes) can be relocated into the larger physical address space by memory mapping. (See 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 and by real-time extensions to the language.

Privileged process

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

Procedure

See PASCAL procedure.

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. More specifically, a process is an independent scheduling unit, representing 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. There are four types of processes in a mapped environment: general, device-access, driven, and privileged.

Process control block (PCB)

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

• Hardware context of the process (including general registers contents, FPU registers contents, and PSW contents)

• Process priority

• Software context of the process (contents of associated flags and pointers maintained by kernel operations)

• State code

• State queue pointers

Process index

A 16-bit value (identification number) that identifies a process to the kernel and is assigned by the kernel when the process is created.

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 given point in time. These states are:

• Exception wait $\begin{cases} \text{Active} \\ \text{Suspended} \end{cases}$

- Ready $\quad \begin{cases} \text{Active} \\ \text{Suspended} \end{cases}$

- Run

- Wait $\quad \begin{cases} \text{Active} \\ \text{Suspended} \end{cases}$

Process synchronization

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

Program

In the MicroPower/Pascal language, an attribute applied to a code unit composed of 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.

Program section

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 and global variables is dynamically allocated during execution and has the read/write (R/W) attribute. Others are read only (RO).

PROM (Programmable read-only memory)

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

P-sect

1. In MACRO-11, a contiguous section of code or data.

2. In MicroPower/Pascal, a compiler-generated structure containing program elements to be stored in one of four separate memory types. 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

A queue semaphore is an extension of counting semaphores. 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. Whenever a process signals the queue semaphore, it increments the semaphore by 1 and adds one element to the queue. Whenever 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.

Radial-serial protocol

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

RAM (Random-access memory)

A read/write memory device. Application programs that require storage space for variables and buffers can write data into RAM locations, as well as read the contents of the RAM locations.

Relocate

One step in the process of linking a MicroPower/Pascal application program. Relocation associates each program section in a merged object module to a specific set of virtual addresses. The RELOC utility performs this function and produces a process image file.

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 (Read-only memory)

A memory device manufactured with binary values already placed in each addressable location. The contents of ROM locations cannot be changed after manufacture. (ROM chips are purchased from an integrated circuit manufacturer, which places a program supplied by the purchaser on the chips.)

Runtime module

See Memory image.

Scheduling

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

Significant event

A change in the state of a running or ready-active process that affects its ability to take control of the CPU resource. 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

In other words, any change in a process's state involving either the run or ready-active queues is a significant event.

Statement

A line of Pascal code. A statement is delimited in Pascal by a semicolon (;). Note that a compound statement consists of more than one Pascal statement delimited by the Pascal-reserved words BEGIN and END.

Static allocation

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

Static process

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

Stepping

Stopping a process after each statement or instruction of the process executes.

Stopped

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

Structure (System data structure)

A process control block, semaphore, ring buffer, or packet.

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.

Structured type

A definition of a type that contains several data types. This structured type can be attributed to a structured variable. In this way, one structured variable may be composed of several types of data items.

Structured variable

A group of variables collected under one variable name. The parts of the structured variable may be different data types.

Suspend, suspended

A process is suspended when it is placed in suspend state by action of 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.

Symbol table

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

Symbolic debugger

A program residing in the RT-11 host system that allows the user to examine or deposit memory locations or Pascal variables, set breakpoints, and examine kernel structures in application storage. PASDBG is the symbolic debugger used with the MicroPower/Pascal application.

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 program is intended to run, once developed on the host (RT-11) processor.

Termination point

The location within 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 specific trap instruction. Trap instructions include 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 the process.

Unmapped memory

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

Virtual address

A value in the range octal 0 to 177777; a 16-bit address within 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 and 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

L

LIBR, 2-2
Load, down-line, 3-6, 6-17
LSI-11, 1-1, 2-4

M

MACRO-11 language, 5-1
Mapped memory processes, 4-8
Memory, 2-4
  macro in configuration file, 2-6, 6-11
  target system maximums, 2-4
MERGE, 3-5, 6-14
Message, 5-6
MIB (memory image builder), 3-6, 6-15
MXV11 interface, 2-4

N

Name (process), 4-3

O

Object module, 6-13

P

Packet, 5-6
Pascal language, 5-1
  main program, 6-3
PIP, 2-2
Prefix file, 6-13
Privileged process, 4-8
Process, 1-4, 1-6, 4-1
  control block, 4-2
  descriptor block, 4-3
  name, 4-2
  priority, 4-2, 5-8, 5-12
  state, 4-2, 4-5
Process scheduling, 4-4
Process type, 4-8
  device access, 4-8
  driver, 4-8
  general, 4-8
  privileged, 4-8
Processor macro, 2-6, 6-11
PROM, 2-1
PUT_ELEMENT, 5-11

Q

Queue semaphore, 1-4, 5-5, 5-10

R

Race condition, 5-7
Ready state, 4-5
Real time, 1-6
RECEIVE, 5-7
RELOC, 3-5, 6-15
Resource, shared, 5-3
Resources macro, 2-6, 6-11
RESUME, 5-8
Ring buffer, 1-5
Run state, 4-5
RX02, 2-1, 2-4, 4-11

S

SBC-11, 1-1, 2-4
Scope, 3-7
Semaphore, 1-4, 1-6
  binary, 1-4, 5-4, 6-5
  counting, 1-4, 5-5
  queue, 1-4, 5-5, 5-10
SEND, 5-7
Serial line, 2-1, 2-4
SIGNAL, 5-4, 5-7, 6-8
STACK_SIZE, 4-3, 6-3
State (process), 4-2, 4-5
Static process, 4-1, 4-7
SUSPEND, 5-8
Suspended state, 4-6
Symbolic debugger, 1-2, 3-7, 6-18
Synchronization (of processes), 1-6,
    4-4, 5-4
System processes, 4-11

T

Target, 1-6, 2-2
TERMINATE procedure, 4-9
Tools, development, 1-2
Tracepoint, 3-7
Traps macro, 2-6, 6-11
Type (of process), 4-8

U

Utility programs, 3–3
  MERGE, 3–5, 6–14
  MIB, 3–6, 6–15
  RELOC, 3–5, 6–15

V

Virtual address, 3–5, 6–15

W

WAIT, 5–4, 5–7, 6–8
Waiting (process state), 4–5
Watchpoint, 3–7

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

digital

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

POSTAGE WILL BE PAID BY ADDRESSEE

SMALL SYSTEMS SOFTWARE PUBLICATIONS ML5-5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MA 01754